Hydra HeadV2 Specification: Coordinated Head protocol

DRAFT

Sebastian Nagel sebastian.nagel@iohk.io

Sasha Bogicevic sasha.bogicevic@iohk.io

Franco Testagrossa franco.testagrossa@iohk.io

Daniel Firth daniel.firth@iohk.io

Noon van der Silk noon.vandersilk@iohk.io

Veronika Romashkina veronika.romashkina@iohk.io

July 9, 2026

Contents

1 Introduction **1**

2 Protocol Overview **2**
- 2.1 Opening the head . . . 2
- 2.2 The Coordinated Head protocol . . . 2
- 2.3 Closing the head . . . 4
- 2.4 Differences . . . 4

3 Preliminaries **5**
- 3.1 Notation . . . 5
- 3.2 Public key multi-signature scheme . . . 6
- 3.3 Extended UTxO . . . 6
- 3.4 KZG Accumulators . . . 8

4 Protocol Setup **10**

5 On-chain Protocol **11**
- 5.1 Init transaction . . . 11
- 5.2 Deposit Transaction . . . 13
- 5.3 Recover Transaction . . . 14
- 5.4 Increment Transaction . . . 15
- 5.5 Decrement Transaction . . . 17
- 5.6 Close Transaction . . . 18
- 5.7 Contest Transaction . . . 20
- 5.8 Fan-Out Transaction . . . 22

6 Off-Chain Protocol **26**
- 6.1 Assumptions . . . 27
- 6.2 Notation . . . 28
- 6.3 Variables . . . 28
- 6.4 Protocol flow . . . 29
- 6.5 Rollbacks and protocol changes . . . 32

7 Security (WIP — Iteration 1) **35**
- 7.1 Proofs . . . 36

1 Introduction

This document specifies the 'Coordinated Hydra Head' protocol to be implemented as the first version of Hydra Head on Cardano - **Hydra HeadV1**. The protocol is derived from variants described in the original paper [8], but was further simplified to make a first implementation on Cardano possible.

Note that the format and scope of this document is (currently) also inspired by the paper and hence does not include a definition of the networking protocols or concrete message formats. It

Add: network specification (message formats)

is structured similarly, but focuses on a single variant, and avoids indirections and unnecessary generalizations. The document is kept in sync with the reference implementation available on Github [4].

First, a high-level overview of the protocol and how it differs from legacy variants of the Head protocol is given in Section 2. Relevant definitions and notations are introduced in Section 3, while Section 4 describes protocol setup and assumptions. Then, the actual on-chain transactions of the protocol are defined in Section 5, before the off-chain protocol part specifies behavior of Hydra parties off-chain and ties the knot with on-chain transactions in Section 6. At last, Section 7 gives the security definition, properties and proofs for the Coordinated Head protocol.

2 Protocol Overview

The Hydra Head protocol provides functionality to lock a set of UTxOs on a blockchain, referred to as the *mainchain*, and evolve it inside a so-called off-chain *head*, independently of the mainchain. At any point, the head can be closed with the effect that the locked set of UTxOs on the mainchain is replaced by the latest set of UTxOs inside the head. The protocol guarantees full wealth preservation: no generation of funds can happen off-chain (inside a head) and no responsive honest party involved in a head can ever lose any funds other than by consenting to give them away. In exchange for decreased liveness guarantees (stop any time), it can essentially proceed at network speed under good conditions, thereby reducing latency and increasing throughput. At the same time, the head protocol provides the same capabilities as the mainchain by reusing the same ledger model and transaction formats — making the protocol "isomorphic".

2.1 Opening the head

To create a head-protocol instance, any party may take the role of an *initiator* and ask other parties, the *head members*, to participate in the head by exchanging public keys and agreeing on other protocol parameters. These public keys are used for both, the authentication of head-related on-chain transactions that are restricted to head members (e.g., a non-member is not allowed to close the head) and for signing off-chain transactions in the head.

The initiator then establishes and directly opens the head by submitting an *init* transaction to the mainchain that contains the Hydra protocol parameters and mints special *participation tokens (PT)* identifying the head members. The *init* transaction initializes a state machine (see Fig. 1) that manages the life-cycle of UTxOs in the head. The state machine comprises the three states: open, closed, and final. A *state thread token (ST)* minted in *init* marks the head output and ensures contract continuity [6].

The head is opened with an empty UTxO set. Participants can then add funds to the head by creating *deposit* transactions on the mainchain, which are subsequently incorporated into the head via *increment* transactions (see Section 5.2 and 5.4). Once the open state is confirmed, the head members start running the off-chain head protocol, which evolves the UTxO set independently of the mainchain.

2.2 The Coordinated Head protocol

The actual Head protocol starts once the head is open on-chain, with an initially empty UTxO set.

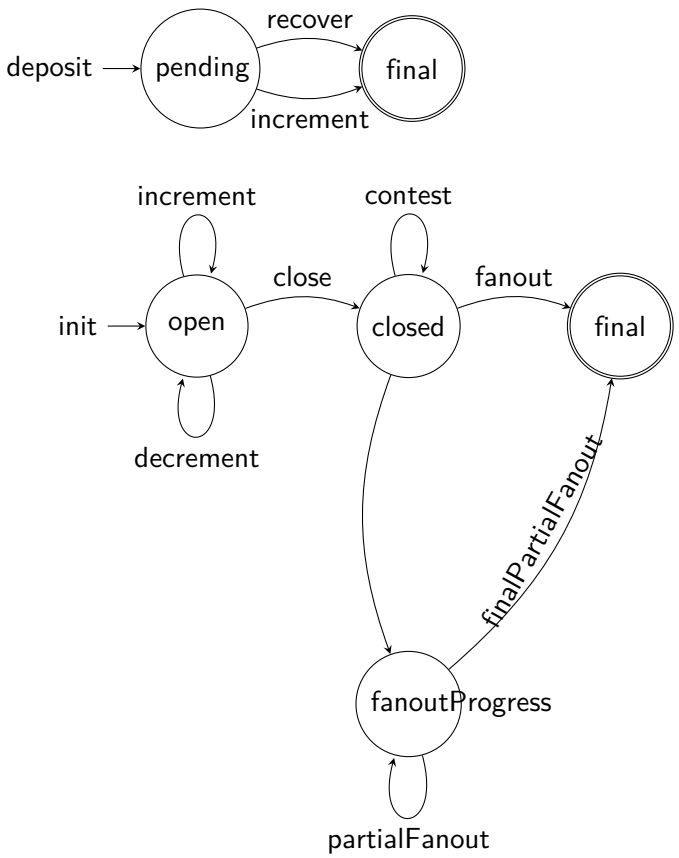

Figure 1: Mainchain state diagram for this version of the Hydra protocol.

The protocol processes off-chain transactions by distributing them between participants, while each party maintains their view of the local UTxO state. That is, the current set of UTxOs evolved from the empty initial state by applying transactions as they are received from the other parties.

To confirm transactions and allow for distribution of the resulting UTxO set without needing the whole transaction history, snapshots are created by the protocol participants. The initial snapshot U_0 corresponds to the initial (empty) UTxO set, while snapshots thereafter $U_1, U_2, ...$ are created with monotonically increasing snapshot numbers.

For this, the next snapshot leader (round-robin) requests his view of a new confirmed state to be signed by all participants as a new snapshot. The leader does not need to send his local state, but only indicate, by hashes, the set of transactions to be included in order to obtain the to-be-snapshotted UTxO set.

The other participants sign the snapshot as soon as they have (also) seen the transactions that are to be processed on top of its preceding snapshot: a party's local state is always ahead of the latest confirmed snapshot.

Signatures are broadcast and aggregated by each party. When all signature parts of the multi-signature are received and verified, a snapshot is considered confirmed. As a consequence, a participant can safely delete (if wished) all transactions that have been processed into it as the snapshot's multi-signature is now evidence that this state once existed during the head evolution.

2.2.1 Updating an open head

Besides processing "normal" transactions, participants can also request to withdraw some UTxO they can spend from the Head and make it available on main chain via a *decrement* 5.5 transaction — the overall process is also called "decommit".

To add UTxO to an open head, anyone may create a deposit of one or more UTxO using a *deposit* 5.2 transaction. The head participants will observe this deposit and, once settled, request off-chain agreement to include the deposited UTxO in the form of a snapshot. With that agreement, an *increment* 5.4 transaction can be created and used to update the head state on the mainchain. A deadline is associated with the deposit, which ensures that the UTxO is locked up long enough to be safely consumed into the head without risk of double spending. Should a deposit have not been picked up in time, a *recover* 5.3 transaction allows anyone to unlock the original UTxO.

2.3 Closing the head

The head protocol is designed to allow any head member at any point in time to produce, without interaction, a certificate to close the head. This certificate is created from the latest confirmed, multi-signed snapshot. Using this certificate, the head member may "force close" the head by advancing the mainchain state machine to the closed state.

Once in closed, the state machine grants parties a contestation period, during which parties may contest the closure by posting the certificate of a newer snapshot on-chain in a contest transaction. Contesting leads back to the state closed and each party can contest at most once. After the contestation period has elapsed, the state machine may proceed to the final state. The state machine enforces that the outputs of the transaction leading to final correspond exactly to the latest UTxO set seen during the contestation period.

2.4 Differences

In the Coordinated Head protocol, off-chain consensus is simplified by not having transactions confirmed concurrently to the snapshots (and to each other) but having the snapshot leader propose, in their snapshot, a set of transactions for explicit confirmation. The parties' views of confirmed transactions thus progress in sync with each other (once per confirmed snapshot), thus simplifying the close/contest procedure on the mainchain. Also, there is no need for conflict resolution as in Appendix B of [8]. In summary, the differences to the original Head protocol in [8] are:

- No hanging transactions due to 'coordination'.
- No acknowledgement nor confirmation of transactions.
- No confirmation message for snapshots (two-round local confirmation).
- Allow for deposits and decommits while head is open.
- No initialization phase: head opens directly with empty UTxO, funds added via deposits.

3 Preliminaries

This section introduces notation and other preliminaries used in the remainder of the specification.

3.1 Notation

The specification uses set-notation based approach while also inspired by [1] and [6]. Values a are in a set $a \in \mathcal{A}$, also indicated as being of some type $a : \mathcal{A}$, and multidimensional values are tuples drawn from a $\times$ product of multiple sets, e.g. $(a, b) \in (\mathcal{A} \times \mathcal{B})$. An empty set is indicated by $\emptyset$ and sets may be enumerated using $\{a_1 \ldots a_n\}$ notation. The $=$ operator means equality and $\leftarrow$ is explicit assignment of a variable or value to one or more variables. Projection is used to access the elements of a tuple, e.g. $(a, b)^{\downarrow 1} = a$. Functions are morphisms mapping from one set to another $x : \mathcal{A} \rightarrow f(x) : \mathcal{B}$, where function application of a function f to an argument x is written as $f(x)$.

Furthermore, given a set $\mathcal{A}$, let

- $\mathcal{A}^? = \mathcal{A} \cup \Diamond$ denotes an option: a value from $\mathcal{A}$ or no value at all indicated by $\perp$,
- $\mathcal{A}^n$ be the set of all n-sized sequences over $\mathcal{A}$,
- $\mathcal{A}^! = \bigcup_{i=1}^{n \in \mathbb{N}} \mathcal{A}^i$ be the set of non-empty sequences over $\mathcal{A}$, and
- $\mathcal{A}^* = \bigcup_{i=0}^{n \in \mathbb{N}} \mathcal{A}^i$ be the set of all sequences over $\mathcal{A}$.

With this, we further define:

- $\mathbb{B} = \{\texttt{false}, \texttt{true}\}$ are boolean values
- $\mathbb{N}$ are natural numbers $\{0, 1, 2, ...\}$
- $\mathbb{Z}$ are integer numbers $\{..., -2, -1, 0, 1, 2, ...\}$
- $\mathbb{H} = \bigcup_{n=0}^{\inf} \{0, 1\}^{8n}$ denotes a arbitrary string of bytes
- $\mathsf{concat} : \mathbb{H}^* \rightarrow \mathbb{H}$ is concatenating bytes, we also use operator $\bigoplus$ for this
- $\mathsf{hash} : x \rightarrow \mathbb{H}$ denotes a collision-resistant hashing function and $x^{\#}$ indicates the hash of x
- $\mathsf{bytes} : x \rightarrow \mathbb{H}$ denotes an invertible serialisation function mapping arbitrary data to bytes
- $a||b = \mathsf{concat}(\mathsf{bytes}(a), \mathsf{bytes}(b))$ is an operator which concatenates the $\mathsf{bytes}(b)$ to the $\mathsf{bytes}(a)$
- Lists of values $l \in \mathcal{A}^*$ are written as $l = [x_1, ..., x_n]$. Empty lists are denoted by [], the ith element x_i is also written $l[i]$ and the length of the list is $|l| = n$. An underscore is also used to indicate a list of values $\underline{x} = l$. Projection on lists are mapped to their elements, i.e. $\underline{x}^{\downarrow 1} = [x_1^{\downarrow 1}, ..., x_n^{\downarrow 1}]$.
- $\mathsf{sortOn} : i \rightarrow \mathcal{A}^* \rightarrow \mathcal{A}^*$ does sort a list of values on the ith projection.
- Data is a universal data type of nested sums and products built up recursively from the base types of $\mathbb{Z}$ and $\mathbb{H}$.

3.2 Public key multi-signature scheme

A multisignature scheme is a set of algorithms where

- MS-Setup generates public parameters Π, such that
- $(k^{ver}, k^{sig}) \leftarrow \mathsf{MS\text{-}KG}(\Pi)$ can be used to generate fresh key pairs,
- $\sigma \leftarrow \mathsf{MS\text{-}Sign}(\Pi, k^{sig}, m)$ signs a message m using key k^{sig},
- $\tilde{k} \leftarrow \mathsf{MS\text{-}AVK}(\Pi, \overline{k})$ aggregates a list of verification keys $\overline{k}$ into a single, aggregate key $\tilde{k}$,
- $\tilde{\sigma} \leftarrow \mathsf{MS\text{-}ASig}(\Pi, m, \overline{k}, \overline{\sigma})$ aggregates a list of signatures $\overline{\sigma}$ about message m into a single, aggregate signature $\tilde{\sigma}$.
- $\mathsf{MS\text{-}Verify}(\Pi, \tilde{k}, m, \tilde{\sigma}) \in \mathbb{B}$ verifies an aggregate signature $\tilde{\sigma}$ of message m under an aggregate verification key $\tilde{k}$.

The security definition of a multisignature scheme from [9, 11] guarantees that, if $\tilde{k}$ is produced from a tuple of verification keys $\overline{k}$ via MS-AVK, then no aggregate signature $\tilde{\sigma}$ can pass verification $\mathsf{MS\text{-}Verify}(\tilde{k}, m, \tilde{\sigma})$ unless all honest parties holding keys in $\overline{k}$ signed m.

Note that in the following, we make the parameter Π implicit and leave out the *ver* suffix for verification key such that $k = k^{ver}$ for better readability.

3.3 Extended UTxO

The Hydra Head protocol is specified to work on the so-called Extended UTxO (EUTxO) ledgers like Cardano.

The basis for EUTxO is Bitcoin's UTxO ledger model [5, 12]. Intuitively, it arranges transactions in a directed acyclic graph, such as the one in Figure 2, where boxes represent transactions with (red) inputs to the left and (black) outputs to the right. A dangling (unconnected) output is an *unspent transaction output (UTxO)* — there are two UTxOs in the figure.

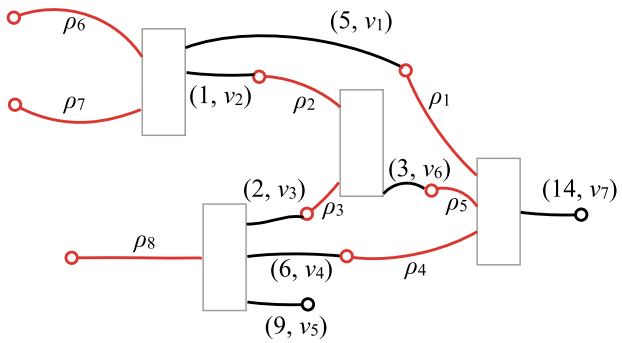

Figure 2: Example of a plain UTxO graph

The following paragraphs will give definitions of the UTxO model and it's extension to support scripting (EUTxO) suitable for this Hydra Head protocol specification. For a more detailed introduction to the EUTxO ledger model, see [6], [1] and [7].

3.3.1 Values

Definition 1 (Values). *Values are sets that keep track of how many units of which tokens of which currency are available. Given a finitely supported function $\mapsto$, that maps keys to monoids, a value is the set of such mappings over currencies (minting policy identifiers), over a mapping of token names t to quantities q:*

$$\mathsf{val} \in \mathsf{Val} = (c : \mathbb{H} \mapsto (t : \mathbb{H} \mapsto q : \mathbb{Z}))$$

where addition of values is defined as $+$ and $\emptyset$ is the empty value.

For example, the value $\{c_1 \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1\}\}$ contains tokens t_1 and t_2 of currency c_1 and addition merges currencies and token names naturally:

$$\begin{aligned} & \{c_1 \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1\}\} \\ + \ & \{c_1 \mapsto \{t_2 \mapsto 1, t_3 \mapsto 1\}, c_2 \mapsto \{t_1 \mapsto 2\}\} \\ = \ & \{c_1 \mapsto \{t_1 \mapsto 1, t_2 \mapsto 2, t_3 \mapsto 1\}, c_2 \mapsto \{t_1 \mapsto 2\}\} \ . \end{aligned}$$

While the above definition should be sufficient for the purpose of this specification, a full definition for finitely supported functions and values as used here can be found in [7]. To further improve readability, we define the following shorthands:

- $\{t_1, \ldots, t_n\} :: c$ for a set positive single quantity assets $\{c \mapsto \{t_1 \mapsto 1, \ldots, t_n \mapsto 1\}\}$,
- $\{t_1, \ldots, t_n\}^{-1} :: c$ for a set of negative single quantity assets $\{c \mapsto \{t_1 \mapsto -1, \ldots, t_n \mapsto -1\}\}$,
- $\{c \mapsto t \mapsto q\}$ for the value entry $\{c \mapsto \{t \mapsto q\}\}$,
- $\{c \mapsto \cdot \mapsto q\}$ for any asset with currency c and quantity q irrespective of token name.

3.3.2 Scripts

Validator scripts are called *phase-2* scripts in the Cardano Ledger specification (see [3] for a formal treatment of these). Scripts are used for multiple purposes, but most often (and sufficient for this specification) as a *spending* or *minting* policy script.

Definition 2 (Minting Policy Script). *A script $\mu \in \mathcal{M}$ governing whether a value can be minted (or burned), is a pure function with type*

$$\mu \in \mathcal{M} = (\rho : \mathsf{Data}) \to (\gamma : \Gamma) \to \mathbb{B},$$

where $\rho \in \mathsf{Data}$ is the redeemer provided as part of the transaction being validated and $\gamma \in \Gamma$ is the validation context.

Definition 3 (Spending Validator Script). *A validator script $\nu \in \mathcal{V}$ governing whether an output can be spent, is a pure function with type*

$$\nu \in \mathcal{V} = (\delta : \mathsf{Data}) \to (\rho : \mathsf{Data}) \to (\gamma : \Gamma) \to \mathbb{B},$$

where $\delta \in \mathsf{Data}$ is the datum available at the output to be spent, $\rho \in \mathsf{Data}$ is the redeemer data provided as part of the transaction being validated, and $\gamma \in \Gamma$ is the validation context.

3.3.3 Transactions

We define EUTxO inputs, outputs and transactions as they are available to scripts and just enough to specify the behavior of the Hydra validator scripts. For example outputs addresses and datums are much more complicated in the full ledger model [1, 2].

actual transactions $\mathcal{T}$ are not defined

Definition 4 (Outputs). *An output $o \in \mathcal{O}$ stores some value $\mathsf{val} \in \mathsf{Val}$ at some address, defined by the hash of a validator script $\nu^{\#} \in \mathbb{H} = \mathsf{hash}(\nu \in \mathcal{V})$, and may store (reference) some data $\delta \in \mathsf{Data}$:*

$$o \in \mathcal{O} = (\mathsf{val} : \mathsf{Val} \times \nu^{\#} : \mathbb{H} \times \delta : \mathsf{Data})$$

Definition 5 (Output references). *An output reference $\phi \in \Phi$ points to an output of a transaction, using a transaction id (that is, a hash of the transaction body) and the output index within that transaction.*

$$\phi \in \Phi = (\mathbb{H} \times \mathbb{N})$$

Definition 6 (Inputs). *A transaction input $i \in \mathcal{I}$ is an output reference $\phi \in \Phi$ with a corresponding redeemer $\rho \in \mathsf{Data}$:*

$$i \in \mathcal{I} = (\phi : \Phi \times \rho : \mathsf{Data})$$

Definition 7 (Validation Context). *A validation context $\gamma \in \Gamma$ is a view on the transaction to be validated:*

$$\gamma \in \Gamma = (\mathcal{I}^* \times \mathcal{O}^* \times \mathsf{Val} \times \mathcal{S}^{\leftrightarrow} \times \mathcal{K})$$

*where $\mathcal{I} \in \mathcal{I}^*$ is a **set** of inputs, $\mathcal{O} \in \mathcal{O}^*$ is a **list** of outputs, $\mathsf{mint} \in \mathsf{Val}$ is the minted (or burned) value, $(t_{\mathsf{min}}, t_{\mathsf{max}}) \in \mathcal{S}^{\leftrightarrow}$ are the lower and upper validity bounds where $t_{\mathsf{min}} <= t_{\mathsf{max}}$, and $\kappa \in \mathcal{K}$ is the set of verification keys which signed the transaction.*

Informally, scripts are evaluated by the ledger when it applies a transaction to its current state to yield a new ledger state (besides checking the transaction integrity, signatures and ledger rules). Each validator script referenced by an output is passed its arguments drawn from the output it locks and the transaction context it is executed in. The transaction is valid if and only if all scripts validate, i.e. $\mu(\rho, \gamma) = \texttt{true}$ and $\nu(\delta, \rho, \gamma) = \texttt{true}$.

3.3.4 State machines and graphical notation

State machines in the EUTxO ledger model are commonly described using the *constraint emitting machine (CEM)* formalism [6], e.g. the original paper describes the Hydra Head protocol using this notation [8]. Although inspired by CEMs, this specification uses a more direct representation of individual transactions to simplify description of non-state-machine transactions and help translation to concrete implementations on Cardano. The structure of the state machine is enforced on-chain through *scripts* which run as part of the ledger's validation of a transaction (see Section 3.3). For each protocol transaction, the specification defines the structure of the transaction and enumerates the transaction constraints enforced by the scripts ($\text{tx}^{\equiv}$ in the CEM formalism).

Add an example graph with a legend

3.4 KZG Accumulators

An accumulator scheme over a universe $\mathcal{U}$ is a set of algorithms where

- accSetup generates public parameters (the common reference string),

- $\eta \leftarrow \mathsf{accUTxO}(U)$ constructs an accumulator commitment from a UTxO set $U \subseteq \mathcal{U}$,
- $\eta' \leftarrow \mathsf{accCombine}(\eta_1, \eta_2)$ combines two accumulator commitments into one,
- $\pi \leftarrow \mathsf{accWitness}(\eta, S, U \setminus S)$ produces a membership witness π proving that $S \subseteq U$,
- $\mathsf{accVerify}(\eta, S, \pi) \in \mathbb{B}$ verifies a membership witness for subset S,
- $\eta' \leftarrow \mathsf{accExclude}(\eta, S)$ removes subset S from the accumulator, yielding the updated commitment η', which itself serves as the exclusion witness, and
- $\mathsf{accVerifyExclude}(\eta, S, \eta') \in \mathbb{B}$ verifies that η' is the correct accumulator after removing S from η.

The security property guarantees that a valid membership witness can only be produced for sets genuinely committed under η, and a valid exclusion witness can only be produced when the subset was genuinely removed.

The implementation uses KZG polynomial commitments [10] over the BLS12-381 elliptic curve.

4 Protocol Setup

In order to create a head-protocol instance, an initiator invites a set of participants (the initiator being one of them) to join by announcing to them the protocol parameters.

- For on-chain transaction authentication (Cardano) purposes, each party p_i generates a corresponding key pair (k_i^{ver}, k_i^{sig}) and sends their verification key k_i^{ver} to all other parties. In the case of Cardano, these are Ed25519 keys.
- For off-chain signing (Hydra) purposes, a very basic multisignature scheme (MS, as defined in Section 3.2) based on EdDSA using Ed25519 keys is used:
  - MS-KG is Ed25519 key generation (requires no parameters)
  - MS-Sign creates an EdDSA signature
  - MS-AVK is concatenation of verification keys into an ordered list
  - MS-ASig is concatenation of signatures into an ordered list
  - MS-Verify verifies the "aggregate" signature by verifying each individual EdDSA signature under the corresponding Ed25519 verification key

  To help distinguish on- and off-chain key sets, Cardano verification keys are written k_{C}, while Hydra verification keys are indicated as k_{H} for the remainder of this document.
- Each party p_i generates a hydra key pair and sends their hydra verification key to all other parties.
- Each party p_i computes the aggregate key from the received verification keys, stores the aggregate key, their signing key as well as the number of participants n.
- Each party establishes pairwise communication channels to all other parties. That is, every network message received from a specific party is checked for (channel) authentication. It is the implementer's duty to find a suitable authentication process for the communication channels.
- All parties agree on a contestation period T_{contest}.

If any of the above fails (or the party does not agree to join the head in the first place), the party aborts the initiation protocol and ignores any further action. Finally, at least one of the participants posts the *init* transaction onchain as described next in Section 5.

5 On-chain Protocol

Update figures

The following sections describe the *on-chain* protocol controlling the life-cycle of a Hydra head, which can be intuitively described as a state machine (see Figure 1). Each transition in this state machine is represented and caused by a corresponding Hydra protocol transaction on-chain: *init* 5.1, *increment* 5.4, *decrement* 5.5, *close* 5.6, *contest* 5.7, *fanout* 5.8, *partialFanout* 5.8.1, and *finalPartialFanout* 5.8.2.

The protocol uses KZG accumulators (see Section 3.4) to enable partial fanout when UTxO sets exceed transaction size limits. When all UTxOs fit in a single transaction, *fanout* distributes them all at once. When UTxO sets are too large, *partialFanout* distributes subsets across multiple transactions using membership witnesses, transitioning through an intermediate fanoutProgress state, until *finalPartialFanout* completes the distribution.

Besides the main state transitions of the head protocol, there is the related "deposit protocol" with two transactions in support of *increment*: *deposit* 5.2 and *recover* 5.3. There is also a *decrement* transaction 5.5 that allows for taking funds from the Head back to L1.

The head protocol defines one minting policy script and one validator script:

- μ_{head} governs minting of state and participation tokens in *init* and burning of these tokens in *fanout*.
- ν_{head} represents the main protocol state machine logic and ensures contract continuity throughout *increment*, *decrement*, *close*, *contest*, *fanout*, *partialFanout* and *finalPartialFanout*.

The deposit protocol defines one validator script:

- ν_{deposit} controls that *deposit* transaction output is claimed correctly into a head via *increment* or recovered after the deadline has passed in a *recover* transaction.

5.1 Init transaction

The *init* transaction creates a head instance and establishes the initial state of the protocol and is shown in Figure 3. The head instance is represented by the unique currency identifier cid created by minting tokens using the μ_{head} minting policy script which is parameterized by a single output reference parameter $\phi_{\mathsf{seed}} \in \Phi$:

$$\mathsf{cid} = \mathsf{hash}(\mu_{\mathsf{head}}(\phi_{\mathsf{seed}}))$$

Update initTx.svg to show direct Open output with all tokens.

Two kinds of tokens are minted:

- A single *State Thread (ST)* token marking the head output. This output contains the state of the protocol on-chain and the token ensures contract continuity. The token name is the well known string `HydraHeadV2`, i.e. $\mathsf{ST} = \{\mathsf{cid} \mapsto \texttt{HydraHeadV2} \mapsto 1\}$.
- One *Participation Token (PT)* per participant $i \in \{1 \dots |\underline{k}_{\mathsf{H}}|\}$ to be used for authenticating further transactions. The token name is the participant's verification key hash $k_i^{\#} = \mathsf{hash}(k_i^{ver})$ of the verification key as received during protocol setup, i.e. $\mathsf{PT}_i = \{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\}$.

All minted tokens (ST and all PT_i) are placed directly into the head output, which is created in the open state with an empty UTxO set. Consequently, the *init* transaction

- has at least input ϕ_{seed},
- mints the state thread token ST, and one PT for each of the $|\underline{k}_{\mathsf{H}}|$ participants with policy cid,
- has one head output o_{head}, which captures the open state of the protocol in the datum

$$\delta_{\mathsf{head}} = (\mathsf{open}, \mathsf{cid}', \phi'_{\mathsf{seed}}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}, v, \eta, \text{₳}_{\mathsf{o}})$$

  where

  - open is the state identifier,
  - cid' is the unique currency id of this instance,
  - ϕ'_{seed} is the output reference parameter of μ_{head},
  - $\underline{k}_{\mathsf{H}}$ are the off-chain multi-signature keys from the setup phase,
  - T_{contest} is the contestation period,
  - $v = 0$ is the initial snapshot version,
  - $\eta = \mathsf{hash}(\emptyset)$ is the hash of the (empty) initial UTxO set.

The $\mu_{\mathsf{head}}(\phi_{\mathsf{seed}})$ minting policy is the only script that verifies init transactions and can be redeemed with either a Mint or Burn redeemer:

- When evaluated with the Mint redeemer,
  1. The seed output is spent in this transaction. This guarantees uniqueness of the policy cid because the EUTxO ledger ensures that ϕ_{seed} cannot be spent twice in the same chain. $(\phi_{\mathsf{seed}}, \cdot) \in \mathcal{I}$
  2. All entries of mint are of this policy and of single quantity $\forall\{c \mapsto \cdot \mapsto q\} \in \mathsf{mint} : c = \mathsf{cid} \land q = 1$
  3. Right number of tokens are minted $|\mathsf{mint}| = |\underline{k}_{\mathsf{H}}| + 1$
  4. State token is sent to the head validator $\mathsf{ST} \in \mathsf{val}_{\mathsf{head}}$
  5. All participation tokens are sent to the head output alongside the state token $\forall i \in [1 \dots |\underline{k}_{\mathsf{H}}|] : \mathsf{PT}_i \in \mathsf{val}_{\mathsf{head}}$
  6. The δ_{head} contains own currency id $\mathsf{cid} = \mathsf{cid}'$ and the right seed reference $\phi_{\mathsf{seed}} = \phi'_{\mathsf{seed}}$
- When evaluated with the Burn redeemer,
  1. All tokens for this policy in mint need to be of negative quantity $\forall\{\mathsf{cid} \mapsto \cdot \mapsto q\} \in \mathsf{mint} : q < 0$.

Important: The μ_{head} minting policy only ensures uniqueness of cid and that the right amount of tokens have been minted and sent to ν_{head}, while ν_{head} in turn ensures continuity of the contract. However, it is **crucial** that all head members check:

- That the transaction mints exactly the correct tokens: one ST token and one PT for each head member (total $|\underline{k}_{\mathsf{H}}| + 1$ tokens). This distinguishes *init* from *increment* and *decrement* transactions, which only move tokens without minting.
- That the head output contains an ST token of policy cid which satisfies $\mathsf{cid} = \mathsf{hash}(\mu_{\mathsf{head}}(\phi_{\mathsf{seed}}))$. The ϕ_{seed} from a head datum can be used to determine this.
- That the correct verification key hashes are used in the PTs and the open state is consistent with parameters agreed during setup.

See the initialTx behavior in Figure 13 for details about these checks.

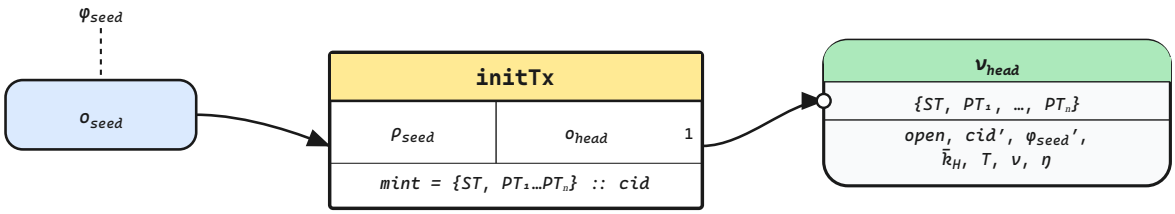

Figure 3: *init* transaction spending a seed UTxO and producing the head output directly in state open.

5.2 Deposit Transaction

The *deposit* transaction locks funds in ν_{deposit} for later collection into the head via an *increment* transaction. Any transaction paying to ν_{deposit} is a *deposit* transaction as there is no on-chain verification in *deposit* transactions. Consequently, protocol actors **must ensure off-chain** that a valid datum is used when paying to the ν_{deposit} validator.

explain why this is enough?

A valid deposit output is governed by ν_{deposit} with value $\mathsf{val}_{\mathsf{deposit}}$ and datum

$$\delta_{\mathsf{deposit}} = (\mathsf{cid}, t_{\mathsf{recover}}, C)$$

where

- cid is the currency id of the target head protocol instance (see 5.1),
- t_{recover} is a deadline after which the deposit can be recovered, and
- $C \in (\mathcal{J} \times \mathbb{H})^*$ is a list of transaction output references with along with serialized outputs that should become available in the head.

Head protocol participants determine **off-chain** whether a deposit output o_{deposit} is eligible for their head by checking

1. cid matches their head identifier,

2. t_{recover} is reasonably far in the future, and

3. $\mathsf{val}_{\mathsf{deposit}}$ contains the value of all decoded outputs of C from $\delta_{\mathsf{deposit}}$.

explain; relate to contestation period?

An example transaction which records all its spent inputs in a deposit output is shown in Figure 4. The $j \in \{1 \dots m\}$ inputs of this example with reference $\phi_{\mathsf{deposited}_j}$ each spend output $o_{\mathsf{deposited}_j}$ with $\mathsf{val}_{\mathsf{deposited}_j}$ would be recorded in the output datum as

$$C = \forall j \in \{1 \dots m\} : [(\phi_{\mathsf{deposited}_j}, \mathsf{bytes}(o_{\mathsf{deposited}_j}))]$$

and the value check would need to satisfy

$$\mathsf{val}_{\mathsf{deposit}} \supseteq \bigcup_{j=1}^{m} \mathsf{val}_{\mathsf{deposited}_j}$$

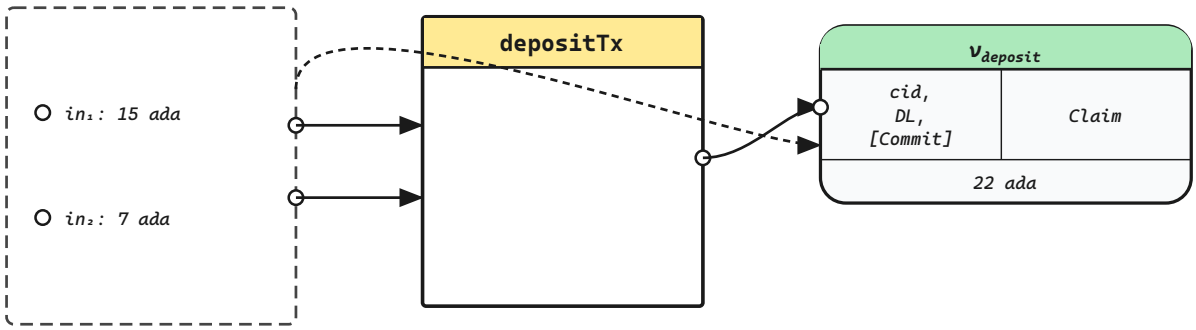

Figure 4: *deposit* transaction spending multiple UTxO into a deposit output.

5.3 Recover Transaction

If a *deposit* transaction output (see 5.2) was not collected into a head by an *increment* transaction 5.4, the *recover* transaction (Figure 5) allows for restoring the UTxO as recorded in the deposit after the deadline has passed. It consists of

- one input spending from ν_{deposit} with datum $\delta_{\mathsf{deposit}} = (\mathsf{cid}, t_{\mathsf{recover}}, C)$, and
- outputs $o_1 \dots o_m$ to recover UTxOs.

The script validator ν_{deposit} is spent with redeemer $\rho_{\mathsf{deposit}} = (\mathsf{Recover}, m)$, where m is the number of outputs to recover, and checks:

1. All UTxOs are recovered exactly as they were deposited. This is done by comparing hashes of serialised representations of the m recovering outputs $o_1 \dots o_m$ with the canonically combined deposited UTxOs in C:

$$\mathsf{hash}(\bigoplus_{j=1}^{m} \mathsf{bytes}(o_j)) = \mathsf{hash}(\mathsf{concat}(\mathsf{sortOn}(1, C)^{\downarrow 2}))$$

2. Transaction is posted after the deadline

$$t_{\mathsf{min}} > t_{\mathsf{recover}}$$

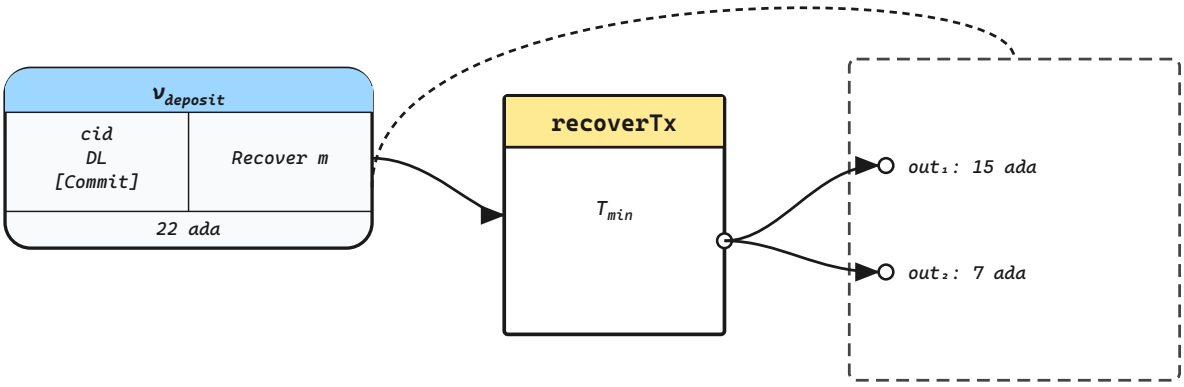

Figure 5: *recover* transaction restoring UTxO of a deposit output.

5.4 Increment Transaction

The *increment* transaction (Figure 6) allows a participant to add a *deposit* output 5.2 to an already open head using a snapshot that approves inclusion. Consequently this transaction consists of:

- one input spending from ν_{head} with value $\mathsf{val}_{\mathsf{head}}$ holding the ST and datum δ_{head},
- one input ϕ_{deposit} spending from ν_{deposit} with value $\mathsf{val}_{\mathsf{deposit}}$ and datum $\delta_{\mathsf{deposit}} = (\mathsf{cid}_{\mathsf{deposit}}, t_{\mathsf{recover}}, C)$,
- one output paying to ν_{head} with value $\mathsf{val}'_{\mathsf{head}}$ and datum δ'_{head}.

The deposit validator ν_{deposit} is spent with $\rho_{\mathsf{deposit}} = \mathsf{Claim}$ and ensures:

1. Claiming head id matches the deposit datum

$$\mathsf{cid} = \mathsf{cid}_{\mathsf{deposit}}$$

2. Transaction is posted before the deadline

$$t_{\mathsf{max}} <= t_{\mathsf{recover}}$$

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{increment}, \xi, s, \phi_{\mathsf{increment}})$, where ξ is a multi-signature of the increment snapshot which authorizes addition of deposited UTxO, s is the snapshot number and ϕ_{deposit} points to the claimed deposit. The validator checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim \mathsf{open}$ to $\delta'_{\mathsf{head}} \sim \mathsf{open}$, parameters $\mathsf{cid}, \tilde{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}$ stay unchanged and the new state is governed again by ν_{head}:

$$(\mathsf{open}, \mathsf{cid}, \tilde{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}, v, \eta, \mathbb{A}_{\mathsf{o}}) \xrightarrow[\xi, s, \phi_{\mathsf{increment}}]{\mathsf{increment}} (\mathsf{open}, \mathsf{cid}, \tilde{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}, v', \eta', \mathbb{A}_{\mathsf{o}})$$

2. New version v' is incremented correctly

$$v' = v + 1$$

3. Claimed deposit is spent

$$\phi_{\mathsf{increment}} = \phi_{\mathsf{deposit}}$$

4. ξ is a valid multi-signature of the new head state η'

$$\mathsf{MS\text{-}Verify}(\underline{k}_{\mathsf{H}}, (\mathsf{cid}||v||s||{\eta'}^{\#}), \xi) = \texttt{true}$$

where ${\eta'}^{\#} = (\eta')^{\#}$ is the hash of the new accumulator commitment η' stored in the output datum, reflecting the UTxO set after adding the deposited UTxOs.

5. The value in the head output is increased accordingly

$$\mathsf{val}_{\mathsf{head}} \cup \mathsf{val}_{\mathsf{deposit}} = \mathsf{val}'_{\mathsf{head}}$$

6. Transaction is signed by a participant

$$\exists\{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\} \in \mathsf{val}'_{\mathsf{head}} \Rightarrow k_i^{\#} \in \kappa$$

7. The ADA overhead $\mathbb{A}_{\mathsf{o}}$ is preserved across the state transition:

$$\mathbb{A}'_{\mathsf{o}} = \mathbb{A}_{\mathsf{o}}$$

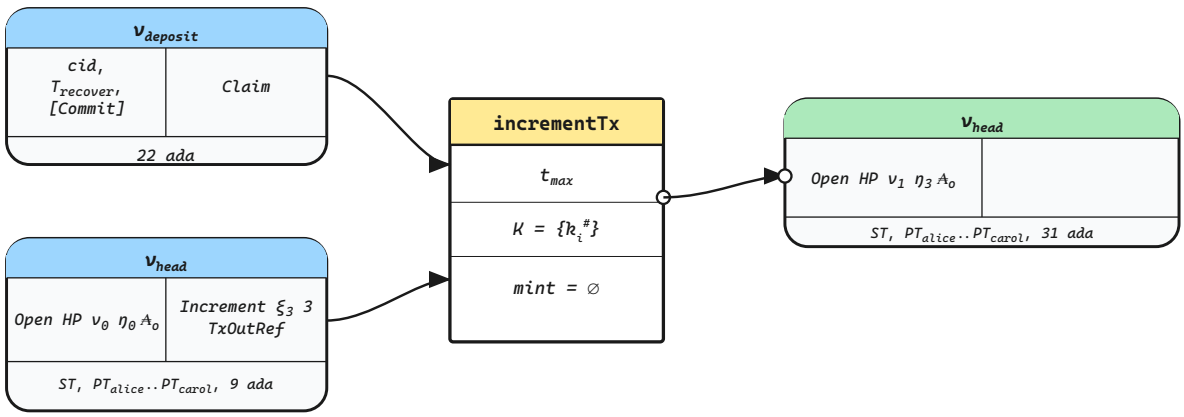

Figure 6: *increment* transaction spending an open head output, producing a new head output which includes the new UTxO.

5.5 Decrement Transaction

The *decrement* transaction (Figure 7) allows a party to remove a UTxO from an already open head and consists of:

- one input spending from ν_{head} holding the ST with δ_{head},
- one output paying to ν_{head} with value $\mathsf{val}'_{\mathsf{head}}$ and datum δ'_{head},
- one or more decommit outputs $o_2 \dots o_{m+1}$ with values $\mathsf{val}_2 \dots \mathsf{val}_{m+1}$.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{decrement}, \xi, s, m)$, where ξ is a multi-signature of the decrement snapshot which authorizes removal of some UTxO, s is the used snapshot number and m is the number of outputs to distribute. The validator checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim \mathsf{open}$ to $\delta'_{\mathsf{head}} \sim \mathsf{open}$, parameters $\mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}$ stay unchanged and the new state is governed again by ν_{head}

$$(\mathsf{open}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}, v, \eta, \mathtt{A}_{\mathsf{o}}) \xrightarrow[\xi,s,m]{\mathsf{decrement}} (\mathsf{open}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}, v', \eta', \mathtt{A}_{\mathsf{o}})$$

2. New version v' is incremented correctly

$$v' = v + 1$$

3. ξ is a valid multi-signature of the new snapshot state η'

$$\mathsf{MS\text{-}Verify}(\underline{k}_{\mathsf{H}}, (\mathsf{cid}||v||s||{\eta'}^{\#}), \xi) = \mathtt{true}$$

   where ${\eta'}^{\#} = (\eta')^{\#}$ is the hash of the new accumulator commitment η' stored in the output datum, reflecting the UTxO set after removing the decommitted UTxOs.

4. The value in the head output is decreased accordingly

$$\mathsf{val}'_{\mathsf{head}} \cup (\bigcup_{j=2}^{m+1} \mathsf{val}_j) = \mathsf{val}_{\mathsf{head}}$$

5. Transaction is signed by a participant

$$\exists \{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\} \in \mathsf{val}'_{\mathsf{head}} \Rightarrow k_i^{\#} \in \kappa$$

6. The ADA overhead $\mathtt{A}_{\mathsf{o}}$ is preserved across the state transition:

$$\mathtt{A}'_{\mathsf{o}} = \mathtt{A}_{\mathsf{o}}$$

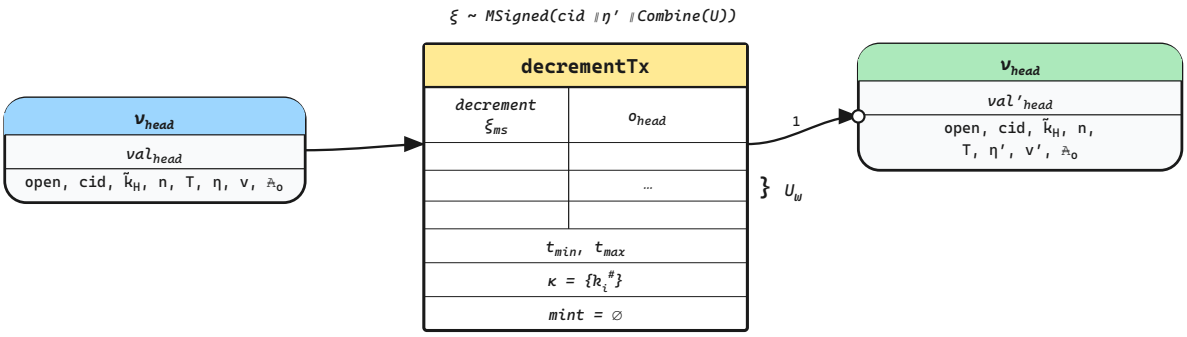

Figure 7: *decrement* transaction spending an open head output, producing a new head output and multiple decommitted outputs.

5.6 Close Transaction

In order to close a head, a head member may post the *close* transaction (see Figure 8). This transaction has

- one input spending from ν_{head} holding the ST with δ_{head},
- one output paying to ν_{head} with value $\mathsf{val}'_{\mathsf{head}}$ and datum δ'_{head}.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{close}, \mathsf{CloseType})$, where $\mathsf{CloseType}$ is a hint against which open state to close and checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim \mathsf{open}$ to $\delta'_{\mathsf{head}} \sim \mathsf{closed}$, parameters $\mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}$ stay unchanged and the new state is governed again by ν_{head}

$$(\mathsf{open}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}, v, \eta, \text{₳}_{\mathsf{o}}) \xrightarrow[\mathsf{CloseType}]{\mathsf{close}} (\mathsf{closed}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}, v', s', \eta', \mathcal{C}, t_{\mathsf{final}}, \text{₳}_{\mathsf{o}})$$

   The closed state carries a single unified accumulator η' that combines the snapshotted UTxO set with any pending increment or decrement UTxOs using $\mathsf{accCombine}$.

2. Last known open state version is recorded in closed state

$$v' = v$$

3. Based on the redeemer $\mathsf{CloseType} = \mathsf{Initial} \cup (\mathsf{Any}, \xi, \eta'^{\#}) \cup (\mathsf{Unused}, \xi, \eta'^{\#}) \cup (\mathsf{Used}, \xi, \eta'^{\#})$, where ξ is a multi-signature of the closing snapshot and $\eta'^{\#}$ is the hash of the unified accumulator commitment stored in the output datum, four cases are distinguished. In each case the closed state carries a single unified accumulator η':

   (a) $\mathsf{Initial}$: The initial snapshot is used to close the head and open state was not updated. No signatures are available and it suffices to check

$$v = 0$$

$$s' = 0$$

$$\eta' = \mathsf{accUTxO}(\emptyset)$$

(b) Any: Closing snapshot refers to current state version v with no pending increments or decrements, and $s' > 0$. The unified accumulator is simply the snapshotted state:

$$\eta' = \mathsf{accUTxO}(U')$$

$$\mathsf{MS\text{-}Verify}(\underline{k}_{\mathsf{H}}, (\mathsf{cid}||v||s'||{\eta'}^{\#}), \xi) = \texttt{true}$$

$${\eta'}^{\#} = (\eta')^{\#}$$

(c) Unused: Closing snapshot refers to current state version v and a pending increment or decrement is *not* applied in the snapshot. The unified accumulator is the snapshotted state only:

$$\eta' = \mathsf{accUTxO}(U')$$

$$\mathsf{MS\text{-}Verify}(\underline{k}_{\mathsf{H}}, (\mathsf{cid}||v||s'||{\eta'}^{\#}), \xi) = \texttt{true}$$

$${\eta'}^{\#} = (\eta')^{\#}$$

(d) Used: Closing snapshot refers to the previous state version $v - 1$ and a pending increment or decrement *is* applied in the snapshot. The unified accumulator combines the snapshotted UTxOs with the pending delta:

$$\eta' = \mathsf{accCombine}(\mathsf{accUTxO}(U'), \eta_{\Delta})$$

$$\mathsf{MS\text{-}Verify}(\underline{k}_{\mathsf{H}}, (\mathsf{cid}||v-1||s'||{\eta'}^{\#}), \xi) = \texttt{true}$$

$${\eta'}^{\#} = (\eta')^{\#}$$

where η_{Δ} is the accumulator commitment of the pending delta UTxOs.

4. Initializes the set of contesters

$$\mathcal{C} = \emptyset$$

This allows the closing party to also contest and is required for use cases where pre-signed, valid in the future, close transactions are used to delegate head closing.

5. Correct contestation deadline is set

$$t_{\mathsf{final}} = t_{\mathsf{max}} + T$$

6. Transaction validity range is bounded by

$$t_{\mathsf{max}} - t_{\mathsf{min}} \leq T$$

to ensure the contestation deadline t_{final} is at most $2 * T$ in the future.

7. Value in the head is preserved

$$\mathsf{val}'_{\mathsf{head}} \supseteq \mathsf{val}_{\mathsf{head}}$$

8. Transaction is signed by a participant

$$\exists\{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\} \in \mathsf{val}'_{\mathsf{head}} \Rightarrow k_i^{\#} \in \kappa$$

9. No minting or burning

$$\mathsf{mint} = \emptyset$$

10. The ADA overhead $₳_o$ is propagated unchanged from the open datum to the closed datum:

$$₳'_o = ₳_o$$

where $₳_o$ is the ADA in the head UTxO not belonging to any L2 UTxO (minimum-UTxO overhead), set at initialisation time and unchanged for the head's lifetime. On fanout, the on-chain value conservation check treats $₳_o$ as released from the head UTxO without requiring it in any distributed output, so it flows to whoever submits the fanout transaction as change (offsetting their transaction fee).

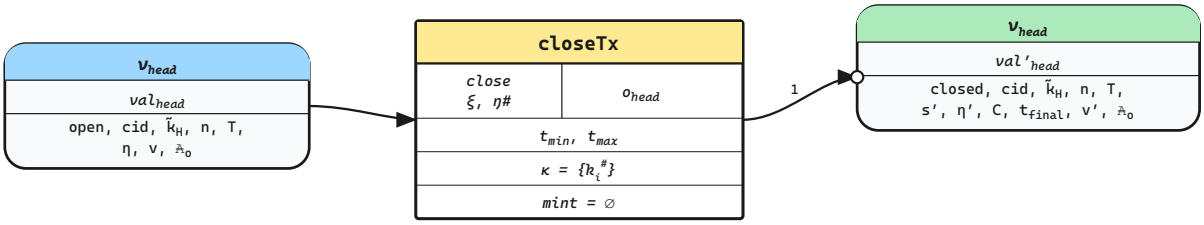

Figure 8: *close* transaction spending the open head output and producing a closed head output with unified accumulator η'.

5.7 Contest Transaction

The *contest* transaction (see Figure 9) is posted by a party to prove the currently closed state is not the latest one. This transaction has

- one input spending from ν_{head} holding the ST with δ_{head},
- one output paying to ν_{head} with value $\mathsf{val}'_{\mathsf{head}}$ and datum δ'_{head}.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{contest}, \mathsf{ContestType})$, where ContestType is a hint how to verify the snapshot and checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim \mathsf{open}$ to $\delta'_{\mathsf{head}} \sim \mathsf{closed}$, parameters $\mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}$ stay unchanged and the new state is governed again by ν_{head}

$$(\mathsf{closed}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}, v, s, \eta, \mathcal{C}, t_{\mathsf{final}}, ₳_o) \xrightarrow[\mathsf{ContestType}]{\mathsf{contest}} (\mathsf{closed}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}, v', s', \eta', \mathcal{C}', t'_{\mathsf{final}}, ₳_o)$$

   The closed state carries a single unified accumulator η' computed using accCombine based on the contest type.

2. Last known open state version stays recorded in closed state

$$v' = v$$

3. Contested snapshot number s' is higher than the currently stored snapshot number s

$$s' > s$$

4. Based on the redeemer $\mathsf{ContestType} = (\mathsf{Unused}, \xi, {\eta'}^{\#}) \cup (\mathsf{Used}, \xi, {\eta'}^{\#})$, where ξ is a multi-signature of the contesting snapshot and ${\eta'}^{\#} = (\eta')^{\#}$ is the hash of the unified accumulator commitment stored in the output datum, two cases are distinguished:

   (a) Unused: Contesting snapshot refers to current state version v (pending delta not applied in snapshot). The unified accumulator reflects only the snapshotted UTxOs:

   $$\eta' = \mathsf{accUTxO}(U')$$

   $$\mathsf{MS\text{-}Verify}(\underline{k}_{\mathsf{H}}, (\mathsf{cid}||v||s'||{\eta'}^{\#}), \xi) = \mathtt{true}$$

   $${\eta'}^{\#} = (\eta')^{\#}$$

   (b) Used: Contesting snapshot refers to the previous state version $v - 1$ (pending delta applied in snapshot). The unified accumulator combines the snapshotted UTxOs with the pending delta:

   $$\eta' = \mathsf{accCombine}(\mathsf{accUTxO}(U'), \eta_{\Delta})$$

   $$\mathsf{MS\text{-}Verify}(\underline{k}_{\mathsf{H}}, (\mathsf{cid}||v-1||s'||{\eta'}^{\#}), \xi) = \mathtt{true}$$

   $${\eta'}^{\#} = (\eta')^{\#}$$

   where η_{Δ} is the accumulator commitment of the pending delta UTxOs.

5. The single signer $\{k^{\#}\} = \kappa$ has not already contested and is added to the set of contesters

$$k^{\#} \notin \mathcal{C}$$

$$\mathcal{C}' = \mathcal{C} \cup k^{\#}$$

6. Transaction is posted before deadline

$$t_{\mathsf{max}} \leq t_{\mathsf{final}}$$

7. Contestation deadline is updated correctly to

$$t'_{\mathsf{final}} = \begin{cases} t_{\mathsf{final}} & \text{if } |\mathcal{C}'| = n, \\ t_{\mathsf{final}} + T & \text{otherwise.} \end{cases}$$

8. Value in the head is preserved

$$\mathsf{val}'_{\mathsf{head}} \supseteq \mathsf{val}_{\mathsf{head}}$$

9. Transaction is signed by a participant

$$\exists \{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\} \in \mathsf{val}'_{\mathsf{head}} \Rightarrow k_i^{\#} \in \kappa$$

10. No minting or burning

$$\mathsf{mint} = \emptyset$$

11. The ADA overhead $₳_o$ is preserved in the output closed datum:

$$₳'_o = ₳_o$$

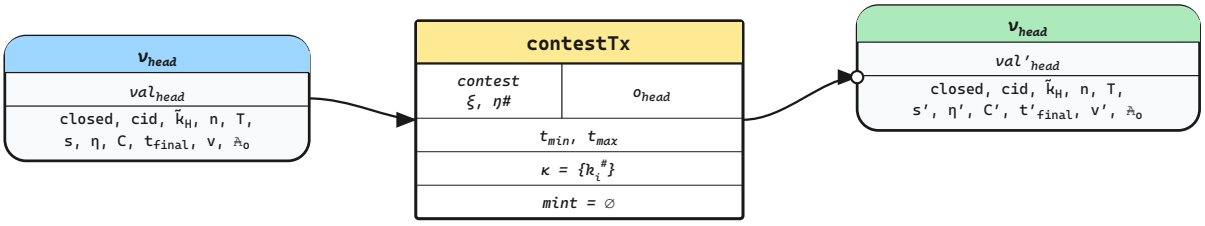

Figure 9: *contest* transaction spending the closed head output and producing a different closed head output.

5.8 Fan-Out Transaction

Once the contestation phase is over, a head may be finalized by posting a *fanout* transaction (see Figure 10), which distributes all UTxOs from the head according to the unified accumulator in the closed state. A fanout transaction consists of

- one input spending from ν_{head} holding the ST, and
- outputs $o_1 \dots o_m$ to distribute all UTxOs.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{fanout}, m, \pi, \mathsf{crsRef})$, where:

- m is the number of outputs to distribute from the closed state,
- π is the KZG membership witness,
- crsRef is the output reference of the reference input holding the Common Reference String (CRS).

The validator checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim$ closed to terminal state final:

$$(\mathsf{closed}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}, v, s, \eta, \mathcal{C}, t_{\mathsf{final}}) \xrightarrow[m,\pi]{\mathsf{fanout}} \mathsf{final}$$

2. All m outputs are verified as members of the unified accumulator η using the membership witness π:

$$\mathsf{accVerify}(\eta, \{o_1, \dots, o_m\}, \pi) = \texttt{true}$$

3. Transaction is posted after contestation deadline $t_{\mathsf{min}} > t_{\mathsf{final}}$.
4. All tokens are burnt $|\{\mathsf{cid} \mapsto \cdot \mapsto -1\} \in \mathsf{mint}| = n + 1$.
5. The head input value is fully conserved:

$$\mathsf{val}^{\mathsf{in}}_{\mathsf{head}} = \bigoplus_{i=1}^{m} \mathsf{val}(o_i) \oplus \mathsf{val}_{\mathsf{burned}} \oplus ₳_{\mathsf{o}}$$

where $₳_{\mathsf{o}}$ is the ADA overhead carried from the closed datum, and $\mathsf{val}_{\mathsf{burned}}$ is the value of all burned head tokens.

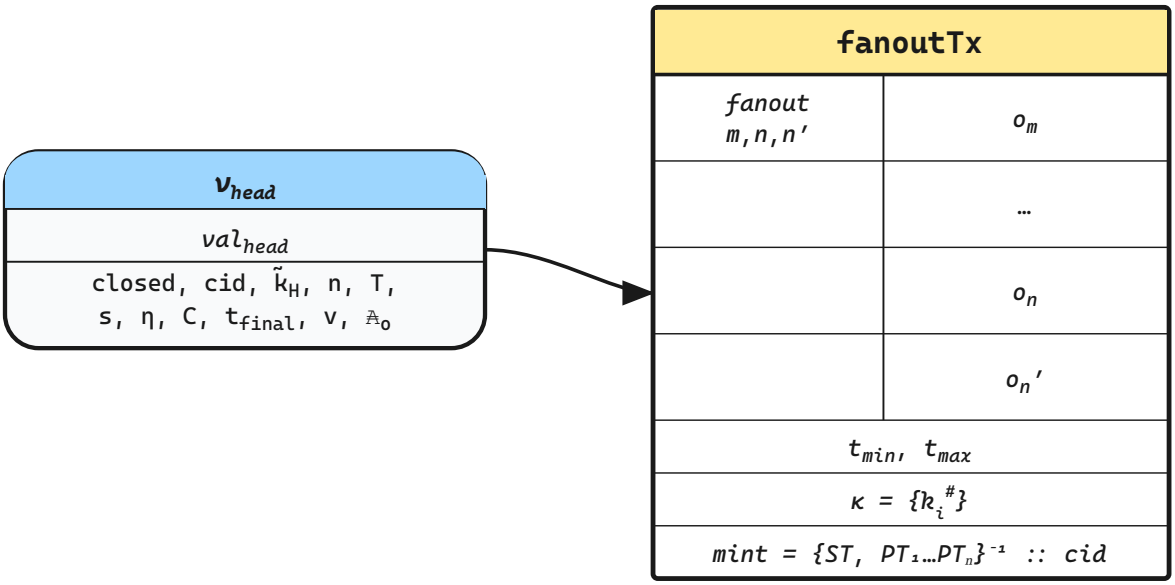

Figure 10: *fanout* transaction spending the closed head output with unified accumulator η and distributing funds with outputs $o_1 \ldots o_m$.

5.8.1 Intermediate Partial Fan-Out Transaction

When UTxO sets exceed transaction size limits, the protocol distributes UTxOs across multiple transactions using partial fanout. Each intermediate step distributes a subset of UTxOs and transitions the head into the fanoutProgress state, which carries only the fields needed for subsequent steps:

$$(\mathsf{fanoutProgress}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, t_{\mathsf{final}}, \eta, ₳_{\mathsf{o}})$$

where t_{final} is the contestation deadline (carried from closed), η is the current accumulator commitment, and $₳_{\mathsf{o}}$ is the ADA overhead in the head UTxO not belonging to any L2 UTxO (propagated from the closed datum).

An intermediate partial fanout transaction (see Figure 11) consists of:

- one input spending from ν_{head} in state closed or fanoutProgress, and
- a continuing head output at index 0 in state fanoutProgress with updated accumulator, and
- outputs $o_1 \ldots o_m$ at indices $1 \ldots m$ distributing a subset of UTxOs.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{partialFanout}, m, \mathsf{crsRef})$, where m is the number of UTxO outputs to distribute in this step and crsRef is the output reference of the reference input holding the CRS. The validator checks:

1. $m > 0$ (no zero-output batches).
2. State transitions into fanoutProgress with updated accumulator. For a closed input:

$$(\mathsf{closed}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, T_{\mathsf{contest}}, v, s, \eta, \mathcal{C}, t_{\mathsf{final}}) \xrightarrow[m]{\mathsf{partialFanout}} (\mathsf{fanoutProgress}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, t_{\mathsf{final}}, \eta', \mathbb{A}_{\mathsf{o}})$$

   For a fanoutProgress input:

$$(\mathsf{fanoutProgress}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, t_{\mathsf{final}}, \eta, \mathbb{A}_{\mathsf{o}}) \xrightarrow[m]{\mathsf{partialFanout}} (\mathsf{fanoutProgress}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, t_{\mathsf{final}}, \eta', \mathbb{A}_{\mathsf{o}})$$

3. The new accumulator η' (from the output datum) is not the G1 generator — all elements have *not* yet been removed (use finalPartialFanout for the last batch):

$$\eta' \neq G_1$$

4. No minting or burning $\mathsf{mint} = \emptyset$.
5. Transaction is posted after contestation deadline $t_{\mathsf{min}} > t_{\mathsf{final}}$.
6. Parameters cid, $\underline{k}_{\mathsf{H}}$, t_{final}, and $\mathbb{A}_{\mathsf{o}}$ are preserved in the output fanoutProgress datum.
7. Value is conserved: head input value equals head output value plus all distributed outputs:

$$\mathsf{val}^{\mathsf{in}}_{\mathsf{head}} = \mathsf{val}^{\mathsf{out}}_{\mathsf{head}} \oplus \bigoplus_{i=1}^{m} \mathsf{val}(o_i)$$

8. The new accumulator η' from the output datum correctly represents the remaining UTxOs after removing $S = \{o_1, \ldots, o_m\}$. The output datum value serves as the exclusion witness:

$$\mathsf{accVerifyExclude}(\eta, S, \eta') = \texttt{true}$$

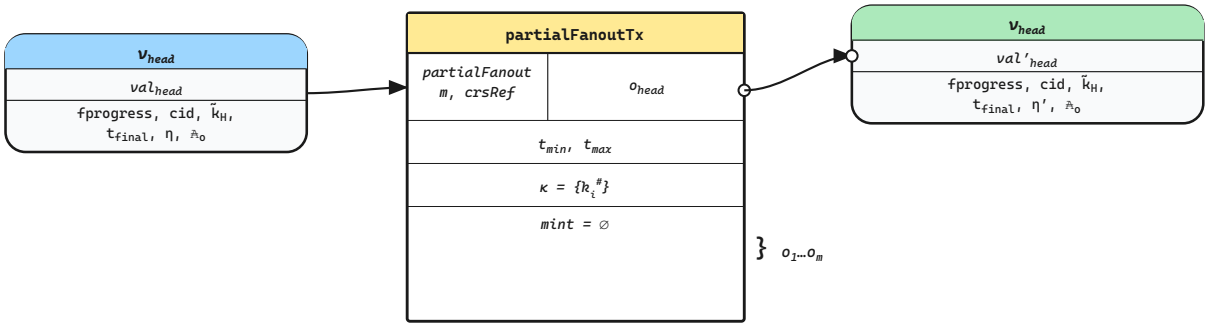

Figure 11: *partialFanout* transaction spending the fanoutProgress head output, distributing outputs $o_1 \ldots o_m$, and producing a new fanoutProgress head output with updated accumulator η'.

5.8.2 Final Partial Fan-Out Transaction

Once all UTxOs except the last batch have been distributed via partialFanout steps, the final step burns all head tokens and distributes the remaining UTxOs. A final partial fanout transaction (see Figure 12) consists of:

- one input spending from ν_{head} in state fanoutProgress, and
- outputs $o_1 \ldots o_m$ distributing the remaining UTxOs (no continuing head output).

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{finalPartialFanout}, m, \pi, \mathsf{crsRef})$, where m is the number of UTxO outputs, π is the KZG membership witness, and crsRef is the CRS reference. The validator checks:

1. $m > 0$ (prevents fund theft via a zero-output proof bypass).
2. State is advanced from fanoutProgress to terminal state final:

$$(\mathsf{fanoutProgress}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, t_{\mathsf{final}}, \eta, \mathbb{A}_{\mathsf{o}}) \xrightarrow[m,\pi]{\mathsf{finalPartialFanout}} \mathsf{final}$$

3. All head tokens are burnt $|\{\mathsf{cid} \mapsto \cdot \mapsto -1\} \in \mathsf{mint}| = n + 1$.
4. Transaction is posted after contestation deadline $t_{\mathsf{min}} > t_{\mathsf{final}}$.
5. The m distributed outputs are verified as members of the accumulator η using the membership witness π:

$$\mathsf{accVerify}(\eta, \{o_1, \ldots, o_m\}, \pi) = \texttt{true}$$

6. Value is conserved:

$$\mathsf{val}^{\mathsf{in}}_{\mathsf{head}} = \bigoplus_{i=1}^{m} \mathsf{val}(o_i) \oplus \mathsf{val}_{\mathsf{burned}} \oplus \mathbb{A}_{\mathsf{o}}$$

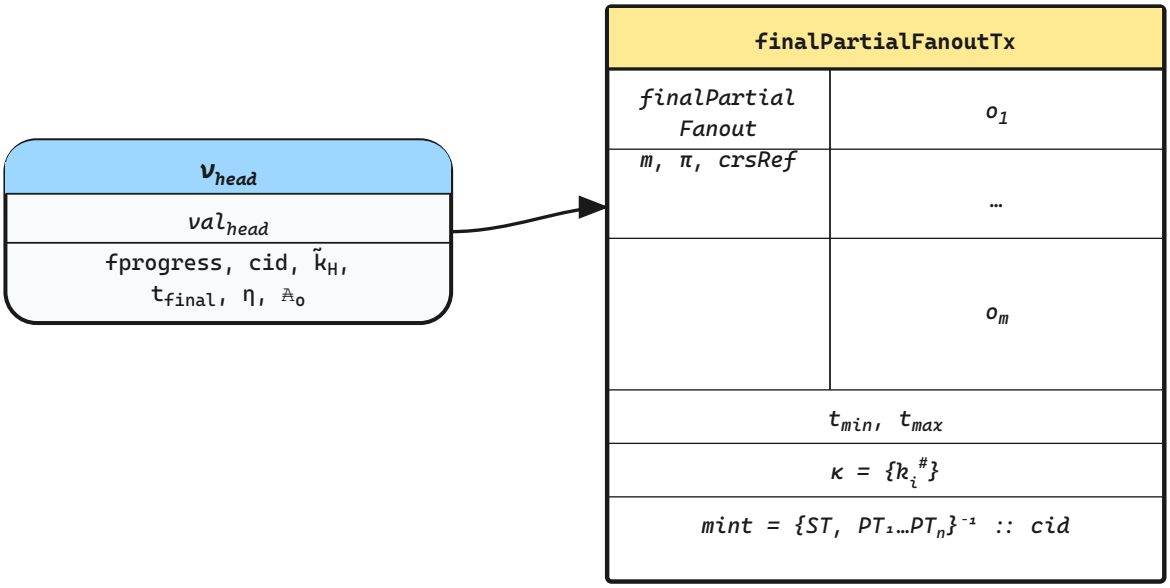

Figure 12: *finalPartialFanout* transaction spending the fanoutProgress head output, distributing the final batch of UTxOs $o_1 \ldots o_m$, and burning all head tokens to reach final.

The $\mu_{\mathsf{head}}(\phi_{\mathsf{seed}})$ minting policy governs the burning of tokens via redeemer burn that:

1. All tokens in mint need to be of negative quantity $\forall\{\mathsf{cid} \mapsto \cdot \mapsto q\} \in \mathsf{mint} : q < 0$.

6 Off-Chain Protocol

This section describes the actual Coordinated Hydra Head protocol, an even more simplified version of the original publication [8]. See the protocol overview in Section 2 for an introduction and notable changes to the original protocol. While the on-chain part already describes the full life-cycle of a Hydra head on-chain, this section completes the picture by defining how the protocol behaves off-chain and notably the relationship between on- and off-chain semantics. Participants of the protocol are also called Hydra head members, parties or simply protocol actors. The protocol is specified as a reactive system that processes three kinds of inputs:

1. On-chain protocol transactions as introduced in Section 5, which are posted to the mainchain and can be observed by all actors:
   - `initialTx`: opens a head
   - `depositTx`: some UTxO was deposited to be incremented
   - `recoverTx`: deposited UTxO was recovered
   - `incrementTx`: adds UTxO to an open head
   - `decrementTx`: removes UTxO from an open head
   - `closeTx`: closes a head

- `contestTx`: contests a closed head
- `fanoutTx`: distributes all UTxOs from a closed head in one step
- `partialFanoutTx`: distributes a subset of UTxOs, transitioning to fanoutProgress (intermediate)
- `finalPartialFanoutTx`: distributes the last subset and burns tokens (terminal from fanoutProgress)

Also, a special input when time advanced on chain may be used:

- `tick`: time advanced on chain

2. Off-chain network messages sent between protocol actors (parties):
   - `reqTx`: to request a transaction to be included in the next snapshot
   - `reqDec`: to request exclusion of UTxO via a decommit transaction
   - `reqSn`: to request a snapshot to be created & signed by every head member
   - `ackSn`: to acknowledge a snapshot by replying with their signatures
3. Commands issued by the participants themselves or on behalf of end-users and clients
   - `init`: to open a head
   - `close`: to request closure of an open head

The behavior is fully specified in Figure 13, while the following paragraphs introduce notation, explain variables and walk-through the protocol flow.

6.1 Assumptions

On top of the statements of the protocol setup in Section 4, the off-chain protocol logic relies on these assumptions:

move/merge with protocol setup?

- Every network message received from a specific party is checked for authentication. An implementation of the specification needs to find a suitable means of authentication, either on the communication channel or for individual messages. Unauthenticated messages must be dropped.
- The head protocol gets correctly (and with completeness) notified about observed transactions on-chain belonging to the respective head instance.
- All inputs are processed to completion, i.e. run-to-completion semantics and no preemption.
- Inputs are deduplicated. That is, any two identical inputs must not lead to multiple invocations of the handling semantics.
- Given the specification, inputs may pile up forever and implementations need to consider these situations (i.e. potential for DoS). A valid reaction to this would be to just drop these inputs. Note that, from a security standpoint, these situations are identical to a non-collaborative peer and closing the head is also a possible reaction.

- The lifecycle of a Hydra head on-chain does not cross (hard fork) protocol update boundaries. Note that these inputs are announced in advance hence it should be possible for implementations to react in such a way as to expedite closing of the head before such a protocol update. This further assumes that the contestation period parameter is picked accordingly.

6.2 Notation

missing:, apply tx

- **on** *event* specifies how the protocol reacts on a given input *event*. Further information may be available from the constituents of *event* and origin of the input.
- **require** p means that boolean expression $p \in \mathbb{B}$ must be satisfied for the further execution of a routine, while discontinued on $\neg p$. A conservative protocol actor could interpret this as a reason to close the head.
- **wait** p is a non-blocking wait for boolean predicate $p \in \mathbb{B}$ to be satisfied. On $\neg p$, the execution of the routine is stopped, queued, and reactivated at latest when p is satisfied.
- **multicast** msg means that a message msg is (channel-) authenticated and sent to all participants of this head, including the sender.
- **postTx** tx has a party create transaction tx, potentially from some data, and submit it on-chain. See Section 5 for individual transaction details.
- **output** *event* signals an observation of *event*, which is used in the security definition and proofs of Section 7. This keyword can be ignored when implementing the protocol.

6.3 Variables

Besides parameters agreed in the protocol setup (see Section 4), a party's local state consists of the following variables:

- $\hat{v}$: Last seen open state version.
- $\hat{s}$: Sequence number of latest seen snapshot.
- $\hat{\Sigma} \in (\mathbb{N} \times \mathbb{H})^*$: Accumulator of signatures of the latest seen snapshot, indexed by parties.
- $\hat{\mathcal{L}}$: UTxO set representing the local ledger state resulting from applying $\hat{\mathcal{T}}$ to $\bar{S}.U$ to validate requested transactions.
- $\hat{\mathcal{T}} \in \mathcal{T}^*$: List of transactions applied locally and pending inclusion in a snapshot (if this party is the next leader).
- $\text{tx}_\alpha \in \mathcal{T}$: Pending deposit transaction[1] to be used for incrementing the head state.
- $\text{tx}_\omega \in \mathcal{T}$: Pending decrement transaction, whose outputs are to be withdrawn from the head.
- $\mathcal{D}$: Set of deposit objects tracking deposit transactions with their status.

[1] In fact this would only need to be a transaction id to look up the corresponding deposit in $\mathcal{D}$

- $\bar{\mathcal{S}}$: Snapshot object of the latest confirmed snapshot which contains:

$\bar{\mathcal{S}}.v$	snapshot version
$\bar{\mathcal{S}}.s$	snapshot number
$\bar{\mathcal{S}}.\mathcal{T}$	list of transactions relating this snapshot to the previous
$\bar{\mathcal{S}}.U$	snapshotted UTxO set
$\bar{\mathcal{S}}.U_\alpha$	pending UTxO to increment
$\bar{\mathcal{S}}.U_\omega$	pending UTxO to decrement
$\bar{\mathcal{S}}.\eta'^{\#}$	hash of the accumulator commitment over the snapshotted UTxO set

where constructor $\text{snObj}(v, n, T, U, U_\alpha, U_\omega)$ initializes a new snapshot object with $\bar{\mathcal{S}}.\eta'^{\#} = \perp$.

Additionally, deposit objects are created using $\text{depositObj}(U, t_{\text{created}}, t_{\text{deadline}}, \text{status})$ where status can be Inactive, Active, or Expired.

6.4 Protocol flow

6.4.1 Initializing the head

`init.` Before a head can be initialized, all parties need to exchange and agree on protocol parameters during the protocol setup phase (see Section 4), so we can assume the public Cardano keys $\underline{k}_{\mathsf{C}}^{setup}$, Hydra keys $\tilde{k}_{\mathsf{H}}^{setup}$, as well as the contestation period $T_{\mathsf{contest}}^{setup}$ are available. One of the clients then can start head initialization using the `init` command, which will result in an *init* transaction being posted. Not strictly a protocol parameter, but the deposit period T_{deposit} would also be made available from configuration.

`initialTx.` All parties will receive this *init* transaction and validate announced parameters against the pre-agreed *setup* parameters, as well as the structure of the transaction and the minting policy used. This is a vital step to ensure the initialized Head is valid, which cannot be checked completely on-chain (see also Section 5.1). Importantly, parties must verify that the transaction mints the correct tokens (one state thread token and one participation token per head member). This also helps in to distinguish *init* transactions from *increment* or *decrement* transactions, which may produce similar outputs without minting.

Since the *init* transaction directly opens the head with an empty UTxO set, the parties initialize their local state upon observing it: the local ledger state $\hat{\mathcal{L}} = \emptyset$, the seen transaction set $\hat{\mathcal{T}} = \emptyset$, the seen head state version $\hat{v} = 0$, and the snapshot number $\hat{s} = 0$. No deposit transaction $\text{tx}_\alpha = \perp$ and no decrement transaction $\text{tx}_\omega = \perp$ are pending, and the last confirmed snapshot is initialized accordingly $\bar{\mathcal{S}} \leftarrow \mathsf{snObj}(0, 0, [], \emptyset, \emptyset, \emptyset)$.

6.4.2 Processing transactions off-chain

Transactions are announced and captured in so-called snapshots. Parties generate snapshots in a strictly sequential round-robin manner. The party responsible for issuing the i^{th} snapshot is the *leader* of the i^{th} snapshot. Leader selection is round-robin per the $\underline{k}_{\mathsf{H}}$ from the protocol setup.

While the frequency of snapshots in the general Head protocol [8] was configurable, the Coordinated Head protocol does specify a snapshot to be created after each transaction.

`reqTx`. Upon receiving request $(\texttt{reqTx}, \text{tx})$, the transaction is applied to the *local* ledger state $\hat{\mathcal{L}} \circ \text{tx}$. If not applicable yet, the protocol does **wait** to retry later or eventually marks this transaction as invalid (see assumption about events piling up). After applying and if there is no current snapshot in flight ($\hat{s} = \bar{\mathcal{S}}.s$) and the receiving party p_i is the next snapshot leader, a message to request snapshot signatures `reqSn` is sent.

`reqDec`. Upon receiving request $(\texttt{reqDec}, \text{tx}_\omega)$, the transaction is checked against the *local* ledger state and if it is not applicable yet or another deposit or decommit is pending still, the protocol does **wait** to retry later or eventually marks the decommit as invalid. After applying tx, its outputs are removed from *local* ledger state $\hat{\mathcal{L}}$ so that they are not available any more and the decommit transaction is kept in the local state (tx_ω). If there is no current snapshot in flight ($\hat{s} = \bar{\mathcal{S}}.s$) and the receiving party p_i is the next snapshot leader, a message to request snapshot signatures `reqSn` containing the decrement transaction tx_ω is sent.

`depositTx`. Upon observing a deposit transaction, each party records the deposit index by deposit transaction id to their local deposit registry $\mathcal{D}$ with status Inactive. The deposit contains deposited UTxO U, creation time t_{created}, and deadline t_{deadline}.

`recoverTx`. Upon observing a recover transaction, each party drops the corresponding entry from its deposit registry $\mathcal{D}$.

`tick`. Whenever time advances (on-chain) to point t, parties update the status of deposits in $\mathcal{D}$ accordingly using the configured deposit period T_{deposit}:

- Expired when deadline passed (or too soon): $t > t_{\text{deadline}} - T_{\mathsf{deposit}}$
- Active when deposit settled enough: $t > t_{\text{created}} + T_{\mathsf{deposit}}$

When deposits become Active and no other deposit / decommit is pending, and the party is the next snapshot leader, it may request a new snapshot including the deposit transaction tx_α.

`reqSn`. Upon receiving request $(\texttt{reqSn}, v, s, \underline{\text{tx}}_{\mathsf{req}}, \text{tx}_\alpha, \text{tx}_\omega)$[2] from party p_j, the receiving p_i **require**s that only a deposit or decommit may be pending, and that v refers to the current open state version, s is the next snapshot number and that party p_j is responsible for leading its creation. Party p_i may have to wait until the previous snapshot is confirmed ($\bar{\mathcal{S}}.s = \hat{s}$). Furthermore, the protocol validates the snapshot request by:

1. If a decommit is requested: verify the transaction is applicable to the last confirmed UTxO set and update the active utxo set with it

[2]Snapshot requests with only transaction identifiers and output references are possible if all parties keep an index of previously seen transactions and their identifiers.

2. If a deposit is requested: verify the corresponding deposit is Active and update the active utxo set with it

3. If we are on the same version as the last snapshot, any requested decommit or deposit must match the last snapshot.

4. Verify all requested transactions $\underline{\mathsf{tx}}_{\mathsf{req}}$ are applicable to the active UTxO set

Only then, p_i increments their seen-snapshot counter $\hat{s}$, resets the signature accumulator $\hat{\Sigma}$, and computes the UTxO set of the new local snapshot as $U \leftarrow U_{\mathsf{active}} \circ \underline{\mathsf{tx}}_{\mathsf{req}}$. Then, p_i creates a signature σ_i using their signing key k_{H}^{sig} on a message comprised by the cid, the new snapshot number $\hat{s}$, the new η resulting from canonically combining U (see Section 5.6 for details), and either η_α or η_ω derived from deposited U_α or decommit transaction tx_ω respectively. The signature is sent to all head members via message $(\texttt{ackSn}, \hat{s}, \sigma_i)$. Finally, the local ledger state $\hat{\mathcal{L}}$ and pending transaction set $\hat{\mathcal{T}}$ get pruned by re-applying all locally pending transactions $\hat{\mathcal{T}}$ to the just requested snapshot's UTxO set iteratively and ultimately yielding a "pruned" version of $\hat{\mathcal{T}}$ and $\hat{\mathcal{L}}$.

`ackSn`. Upon receiving acknowledgment $(\texttt{ackSn}, s, \sigma_j)$, all participants **require** that it is from an expected snapshot (either the last seen $\hat{s}$ or $+\ 1$), potentially **wait** for the corresponding `reqSn` such that $\hat{s} = s$ and **require** that the signature is not yet included in $\hat{\Sigma}$. They store the received signature in the signature accumulator $\hat{\Sigma}$, and if the signature from each party has been collected, p_i aggregates the multisignature $\tilde{\sigma}$ and **require** it to be valid (constructing the signed message as in `reqSn`). If everything is fine, the snapshot can be considered confirmed by creating the snapshot object $\bar{\mathcal{S}} \leftarrow \mathsf{snObj}(\hat{v}, \hat{s}, \hat{\mathcal{T}}, \hat{\mathcal{U}}, U_\alpha, \mathsf{outputs}(\mathrm{tx}_\omega))$ and storing the multi-signature $\tilde{\sigma}$ in it for later reference. In case there is a pending decommit, any participant can now submit a *decrement* transaction by providing the just confirmed snapshot with its accumulator commitment η' for the updated UTxO set. If, however, there was a pending deposit, any participant can now submit an `incrementTx` by providing the confirmed snapshot with its accumulator commitment η' for the updated UTxO set. Lastly, if p_i is the next snapshot leader and there are already transactions to snapshot in $\hat{\mathcal{T}}$, a corresponding `reqSn` is distributed.

`decrementTx`. Upon observing the *decrement* transaction, which removed outputs U from the head, the corresponding pending decrement transaction is cleared and the observed version v is used for future snapshots by setting $\hat{v} \leftarrow v$. Note that the version of the open head state is incremented on each *decrement* transaction as described in Section 5.5.

`incrementTx`. Upon observing the *increment* transaction, which added outputs U to the head, the local ledger state $\hat{\mathcal{L}}$ is extended with the newly added UTxO while the pending increment state U_α is cleared. Also the observed version v is used for future snapshots by setting $\hat{v} = v$. Note that the version of the open head state is incremented on each *increment* transaction as described in Section 5.4

6.4.3 Closing the head

`close`. In order to close a head, a client issues the `close` input which uses the latest confirmed snapshot $\bar{\mathcal{S}}$ to construct the accumulator commitment η' for the closing UTxO state and the certifi-

cate ξ using the corresponding multi-signature. With these, the *close* transaction can be constructed and posted. See Section 5.6 for details about this transaction.

`closeTx/contestTx.` When a party observes the head getting closed or contested, the η-state extracted from the *close* or *contest* transaction represents the latest head status that has been aggregated on-chain so far (by a sequence of *close* and *contest* transactions). If the last confirmed (off-chain) snapshot is newer than the observed (on-chain) snapshot number s_c, an updated accumulator commitment η' and certificate ξ are constructed and posted in a *contest* transaction (see Section 5.7).

`fanoutTx.` Upon observing a *fanout* transaction, all UTxOs are distributed and the head transitions to final state.

`partialFanoutTx.` When a head needs to be finalized, the node first attempts to post a single *fanout* transaction distributing all UTxOs at once. If the full UTxO set does not fit within the transaction size or script execution budget, the node falls back to partial fanout: it performs a binary search over batch sizes to find the largest batch that fits within protocol limits, minimising the total number of fanout steps. Concretely, for a remaining set of N UTxOs, it searches $[1, N-1]$ using ceiling-division midpoints (biasing toward larger batches) and selects the largest n for which the transaction satisfies both the size limit and the script execution budget.

A batch of size $n < N$ is posted as an intermediate *partialFanout* transaction, which distributes n UTxOs, updates the accumulator commitment to cover only the remaining $N - n$ UTxOs, and transitions the head to the fanoutProgress state. The procedure repeats: after each *partialFanout* observation the node again attempts to finalize all remaining UTxOs in one step; if they still do not fit, another binary search determines the next batch size. *finalPartialFanout* is used for the last batch when all remaining UTxOs fit in a single transaction, burning all head tokens and transitioning to final.

6.5 Rollbacks and protocol changes

Explain why rollbacks are no problem to increment/decrem

Write about contestation deadline vs. rollbacks

The overall life-cycle of the Head protocol is driven by on-chain inputs (see introduction of Section 6) which stem from observing transactions on the mainchain. Most blockchains, however, do only provide *eventual* consistency. The consensus algorithm ensures a consistent view of the history of blocks and transactions between all parties, but this so-called *finality* is only achieved after some time and the local view of the blockchain history may change until that point.

On Cardano with it's Ouroboros consensus algorithm, this means that any local view of the mainchain may not be the longest chain and a node may switch to a longer chain, onto another fork. This other version of the history may not include what was previously observed and hence, any tracking state needs to be updated to this "new reality". Practically, this means that an observer of the blockchain sees a *rollback* followed by rollforwards.

For the Head protocol, this means that chain events like `closeTx` may be observed a second time. Hence, it is crucial, that the local state of the Hydra protocol is kept in sync and also rolled back accordingly to be able to observe and react to these events the right way, e.g. correctly contesting this `closeTx` if need be.

The rollback handling can be specified fully orthogonal on top of the nominal protocol behavior, if the chain provides strictly monotonically increasing points p on each chain event via a new or

wrapped `rollforward` event and `rollback` event with the point to which a rollback happened:

`rollforward`. On every chain event that is paired or wrapped in a rollforward event $(\texttt{rollback}, p)$ with point p, protocol participants store their head state indexed by this point in a history Ω of states $\Delta \leftarrow (\hat{v}, \hat{s}, \hat{\mathcal{U}}, \hat{\Sigma}, \hat{\mathcal{L}}, \hat{\mathcal{T}}, \bar{s})$ and $\Omega' = (p, \Delta) \cup \Omega$.

`rollback`. On a rollback $(\texttt{rollback}, p_{rb})$ to point p_{rb}, the corresponding head state Δ need to be retrieved from Ω, with the maximal point $p \leq p_{rb}$, and all entries in Ω with $p > p_{rb}$ get removed.

This will essentially reset the local head state to the right point and allow the protocol to progress through the life-cycle normally. Most stages of the life-cycle are unproblematic if they are rolled back, as long as the protocol logic behaves as in the nominal case.

Since the head is directly opened by the *init* transaction, every rollback of on-chain state is effectively a rollback "past open". However, because the head opens with an empty UTxO set and funds are only added via deposits, the critical concern becomes rollbacks of deposit transactions. The deposit settlement period T_{deposit} (see Section 6) ensures that a deposit is only considered Active — and thus eligible for incrementing into the head — after sufficient time has elapsed since its creation on-chain. This means that by the time a deposit is incremented, its on-chain transaction is sufficiently settled and unlikely to be rolled back, preventing inconsistency between the on-chain and off-chain state.

Coordinated Hydra Head

on (`init`) *from client*
- $n \leftarrow |\underline{k}_{\mathsf{H}}^{setup}|$
- $\tilde{k}_{\mathsf{H}} \leftarrow \mathsf{MS\text{-}AVK}(\underline{k}_{\mathsf{H}}^{setup})$
- $\underline{k}_{\mathsf{C}} \leftarrow \underline{k}_{\mathsf{C}}^{setup}$
- $T_{\mathsf{contest}} \leftarrow T_{\mathsf{contest}}^{setup}$
- $T_{\mathsf{deposit}} \leftarrow T_{\mathsf{deposit}}^{setup}$
- **postTx** $(init, n, \tilde{k}_{\mathsf{H}}, \underline{k}_{\mathsf{C}}, T_{\mathsf{contest}})$

on $(\mathtt{initialTx}, \mathsf{cid}, \phi_{\mathsf{seed}}, n, \tilde{k}_{\mathsf{H}}, \underline{k}_{\mathsf{C}}^{\#}, T_{\mathsf{contest}})$ *from chain*
- **require** $\tilde{k}_{\mathsf{H}} = \mathsf{MS\text{-}AVK}(\underline{k}_{\mathsf{H}}^{setup})$
- **require** $\underline{k}_{\mathsf{C}}^{\#} = [\mathsf{hash}(k) \mid \forall k \in \underline{k}_{\mathsf{C}}^{setup}]$
- **require** $T_{\mathsf{contest}} = T_{\mathsf{contest}}^{setup}$
- **require** $\mathsf{cid} = \mathsf{hash}(\mu_{\mathsf{head}}(\phi_{\mathsf{seed}}))$
- $\hat{\mathcal{L}} \leftarrow \emptyset$
- $\bar{\mathcal{S}} \leftarrow \mathsf{snObj}(0, 0, [], \emptyset, \emptyset, \emptyset)$
- $\hat{v}, \hat{s} \leftarrow 0$
- $\hat{\mathcal{T}} \leftarrow \emptyset$
- $\mathrm{tx}_\omega \leftarrow \bot$
- $\mathrm{tx}_\alpha \leftarrow \bot$

---

on $(\mathtt{reqTx}, \mathrm{tx})$ *from* p_j
- **wait** $\hat{\mathcal{L}} \circ \mathrm{tx} \neq \bot$
  - $\hat{\mathcal{L}} \leftarrow \hat{\mathcal{L}} \circ \mathrm{tx}$
  - $\hat{\mathcal{T}} \leftarrow \hat{\mathcal{T}} \cup \{\mathrm{tx}\}$
  - **if** $\hat{s} = \bar{\mathcal{S}}.s \wedge \mathsf{leader}(\bar{\mathcal{S}}.s + 1) = i$
    - **if** $\mathrm{tx}_\alpha = \bot \wedge \mathrm{tx}_\omega = \bot \wedge \bar{\mathcal{S}}.U_\alpha = \emptyset$
      - $\mathrm{tx}_\alpha \leftarrow$ oldest $D \in \mathcal{D}$ with $D.\mathsf{status} = \mathsf{Active}$
    - **multicast** $(\mathtt{reqSn}, \hat{v}, \bar{\mathcal{S}}.s + 1, \hat{\mathcal{T}}, \mathrm{tx}_\alpha, \mathrm{tx}_\omega)$

on $(\mathtt{reqDec}, \mathrm{tx})$ *from* p_j
- **wait** $U_\alpha = \emptyset \ \wedge \ \mathrm{tx}_\omega = \bot \ \wedge \ \hat{\mathcal{L}} \circ \mathrm{tx} \neq \bot$
  - $\hat{\mathcal{L}} \leftarrow \hat{\mathcal{L}} \circ \mathrm{tx} \setminus \mathsf{outputs}(\mathrm{tx})$
  - $\mathrm{tx}_\omega \leftarrow \mathrm{tx}$
  - **if** $\hat{s} = \bar{\mathcal{S}}.s \wedge \mathsf{leader}(\bar{\mathcal{S}}.s + 1) = i$
    - **multicast** $(\mathtt{reqSn}, \hat{v}, \bar{\mathcal{S}}.s + 1, \hat{\mathcal{T}}, \bot, \mathrm{tx}_\omega)$

on $(\mathtt{reqSn}, v, s, \underline{\mathrm{tx}}_{\mathsf{req}}, \mathrm{tx}_\alpha, \mathrm{tx}_\omega)$ *from* p_j
- **require** $v = \hat{v} \ \wedge \ s = \hat{s} + 1 \ \wedge \ \mathsf{leader}(s) = j$
- **wait** $\hat{s} = \bar{\mathcal{S}}.s \ \wedge \ v = \hat{v}$
  - **require** $\mathrm{tx}_\omega = \bot \ \vee \ \mathrm{tx}_\alpha = \bot$
  - **if** $\mathrm{tx}_\omega \neq \bot$
    - **if** $v = \bar{\mathcal{S}}.v \wedge \bar{\mathcal{S}}.U_\omega \neq \bot$
      - **require** $\bar{\mathcal{S}}.U_\omega = \mathsf{outputs}(\mathrm{tx}_\omega)$
    - **else**
      - **require** $\bar{\mathcal{S}}.U \circ \mathrm{tx}_\omega \neq \bot$
      - $U_{\mathsf{active}} \leftarrow \bar{\mathcal{S}}.U \circ \mathrm{tx}_\omega \setminus \mathsf{outputs}(\mathrm{tx}_\omega)$
  - **if** $\mathrm{tx}_\alpha \neq \bot$
    - $\mathcal{D} \leftarrow \mathcal{D}[\mathrm{tx}_\alpha]$
    - **require** $\mathcal{D}.\mathsf{status} \neq \mathsf{Expired}$
    - **wait** $\mathcal{D}.\mathsf{status} = \mathsf{Active}$
      - **if** $v = \bar{\mathcal{S}}.v \wedge \bar{\mathcal{S}}.U_\alpha \neq \bot$
        - **require** $\bar{\mathcal{S}}.U_\alpha = \mathcal{D}.U$
      - **else**
        - $U_\alpha \leftarrow \mathcal{D}.U$
        - $U_{\mathsf{active}} \leftarrow U_{\mathsf{active}} \cup U_\alpha$
    - **require** $U_{\mathsf{active}} \circ \underline{\mathrm{tx}}_{\mathsf{req}} \neq \bot$
    - $U \leftarrow U_{\mathsf{active}} \circ \underline{\mathrm{tx}}_{\mathsf{req}}$
    - $\hat{s} \leftarrow s$
    - $\eta' \leftarrow \mathsf{accUTxO}(U)$
    - $\eta'^{\#} \leftarrow (\eta')^{\#}$
    - $\sigma_i \leftarrow \mathsf{MS\text{-}Sign}(k_{\mathsf{H}}^{sig}, (\mathsf{cid}||v||\hat{s}||\eta'^{\#}))$
    - $\hat{\Sigma} \leftarrow \emptyset$
    - **multicast** $(\mathtt{ackSn}, \hat{s}, \sigma_i)$
    - $\forall \mathrm{tx} \in \underline{\mathrm{tx}}_{\mathsf{req}}$: **output** $(\mathtt{seen}, \mathrm{tx})$
    - $\hat{\mathcal{L}} \leftarrow U$
    - $X \leftarrow \hat{\mathcal{T}}$
    - $\hat{\mathcal{T}} \leftarrow \emptyset$
    - **for** $\mathrm{tx} \in X : \hat{\mathcal{L}} \circ \mathrm{tx} \neq \bot$
      - $\hat{\mathcal{T}} \leftarrow \hat{\mathcal{T}} \cup \{\mathrm{tx}\}$
      - $\hat{\mathcal{L}} \leftarrow \hat{\mathcal{L}} \circ \mathrm{tx}$

on $(\mathtt{ackSn}, s, \sigma_j)$ *from* p_j
- **require** $s \in \{\hat{s}, \hat{s} + 1\}$
- **wait** $\hat{s} = s$
  - **require** $(j, \cdot) \notin \hat{\Sigma}$
  - $\hat{\Sigma}[j] \leftarrow \sigma_j$
  - **if** $\forall k \in [1..n] : (k, \cdot) \in \hat{\Sigma}$
    - $\tilde{\sigma} \leftarrow \mathsf{MS\text{-}ASig}(\underline{k}_{\mathsf{H}}^{setup}, \hat{\Sigma})$
    - $\eta' \leftarrow \mathsf{accUTxO}(\hat{\mathcal{U}})$
    - $\eta'^{\#} \leftarrow (\eta')^{\#}$
    - **require** $\mathsf{MS\text{-}Verify}(\tilde{k}_{\mathsf{H}}, (\mathsf{cid}||\hat{v}||\hat{s}||\eta'^{\#}), \tilde{\sigma})$
    - $\bar{\mathcal{S}} \leftarrow \mathsf{snObj}(\hat{v}, \hat{s}, \hat{\mathcal{T}}, \hat{\mathcal{U}}, U_\alpha, U_\omega)$
    - $\bar{\mathcal{S}}.\sigma \leftarrow \tilde{\sigma}$
    - $\forall \mathrm{tx} \in \mathcal{T}_{\mathsf{req}}$: **output**$(\mathtt{conf}, \mathrm{tx})$
    - **if** $\bar{\mathcal{S}}.U_\omega \neq \bot$
      - **postTx** $(\mathtt{decrementTx}, \hat{v}, \hat{s}, \eta'^{\#}, \bar{\mathcal{S}}.\sigma)$
    - **if** $\bar{\mathcal{S}}.U_\alpha \neq \bot$
      - **postTx** $(\mathtt{incrementTx}, \hat{v}, \hat{s}, \eta'^{\#}, \bar{\mathcal{S}}.\sigma)$
    - **if** $\mathsf{leader}(s + 1) = i \wedge \hat{\mathcal{T}} \neq \emptyset$
      - **if** $\mathrm{tx}_\alpha = \bot \wedge \mathrm{tx}_\omega = \bot \wedge \bar{\mathcal{S}}.U_\alpha = \emptyset$
        - $\mathrm{tx}_\alpha \leftarrow$ oldest $D \in \mathcal{D}$ with $D.\mathsf{status} = \mathsf{Active}$
      - **multicast** $(\mathtt{reqSn}, \hat{v}, \bar{\mathcal{S}}.s + 1, \hat{\mathcal{T}}, \mathrm{tx}_\alpha, \mathrm{tx}_\omega)$

on $(\mathtt{depositTx}, \mathrm{tx}_\alpha, U, t_{\mathsf{created}}, t_{\mathsf{deadline}})$ *from chain*
- $\mathcal{D} \leftarrow \mathcal{D} \cup (\mathrm{tx}_\alpha, \mathsf{depositObj}(U, t_{\mathsf{created}}, t_{\mathsf{deadline}}, \mathsf{Inactive}))$

on $(\mathtt{recoverTx}, \mathrm{tx}_\alpha)$ *from chain*
- $\mathcal{D} \leftarrow \mathcal{D} \setminus (\mathrm{tx}_\alpha, \cdot)$

on $(\mathtt{decrementTx}, U, v)$ *from chain*
- $\hat{s} \leftarrow \bar{\mathcal{S}}.s$
- $\hat{v} \leftarrow v$
- $\mathrm{tx}_\omega \leftarrow \bot$
- **if** $\mathsf{leader}(\bar{\mathcal{S}}.s + 1) = i \wedge \hat{\mathcal{T}} \neq \emptyset$
  - **multicast** $(\mathtt{reqSn}, \hat{v}, \bar{\mathcal{S}}.s + 1, \hat{\mathcal{T}}, \mathrm{tx}_\alpha, \bot)$

on $(\mathtt{incrementTx}, U, v)$ *from chain*
- $\hat{s} \leftarrow \bar{\mathcal{S}}.s$
- $\hat{v} \leftarrow v$
- $\mathrm{tx}_\alpha \leftarrow \bot$
- $\hat{\mathcal{L}} \leftarrow \hat{\mathcal{L}} \cup U$
- **if** $\mathsf{leader}(\bar{\mathcal{S}}.s + 1) = i \wedge \hat{\mathcal{T}} \neq \emptyset$
  - **multicast** $(\mathtt{reqSn}, \hat{v}, \bar{\mathcal{S}}.s + 1, \hat{\mathcal{T}}, \bot, \bot)$

on $(\mathtt{tick}, t)$ *from chain*
- **for** $D \in \mathcal{D}$
  - **if** $t > D.\mathsf{deadline} - T_{\mathsf{deposit}}$
    - $D.\mathsf{status} \leftarrow \mathsf{Expired}$
  - **else if** $t > D.\mathsf{created} + T_{\mathsf{deposit}}$
    - $D.\mathsf{status} \leftarrow \mathsf{Active}$ **if** $\mathrm{tx}_\alpha = \bot \wedge \mathrm{tx}_\omega = \bot$
      - $\mathrm{tx}_\alpha \leftarrow D$
- **if** $\exists D \in \mathcal{D} : D.\mathsf{status} = \mathsf{Active}$
  - **if** $\mathrm{tx}_\alpha \neq \bot \wedge \mathrm{tx}_\omega = \bot \wedge \hat{s} = \bar{\mathcal{S}}.s \wedge \mathsf{leader}(\bar{\mathcal{S}}.s + 1) = i$
    - **multicast** $(\mathtt{reqSn}, \hat{v}, \bar{\mathcal{S}}.s + 1, \hat{\mathcal{T}}, \mathrm{tx}_\alpha, \bot)$

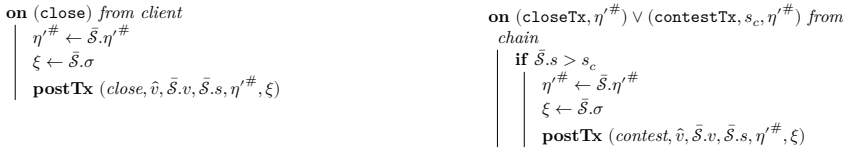

Figure 13: Head-protocol machine for the *coordinated head* from the perspective of party $\mathtt{p}_i$.

7 Security (WIP — Iteration 1)

The security analysis is still **sketchy**, with the goal to make it more formal in upcoming iterations

Add security experiment

above this section there is no security game

Adversaries:

Active Adversary. An *active adversary* $\mathcal{A}$ has full control over the protocol, i.e., he is fully unrestricted in the above security game.

Network Adversary. A *network adversary* $\mathcal{A}_\emptyset$ does not corrupt any head parties, eventually delivers all sent network messages (i.e., does not drop any messages), and does not cause the `close` event. Apart from this restriction, the adversary can act arbitrarily in the above experiment.

Random variables:

- $\hat{S}_i$: the set of transactions tx for which party $\mathtt{p}_i$, *while uncorrupted*, output $(\mathtt{seen}, \mathrm{tx})$;
- $\bar{C}_i$: the set of transactions tx for which party $\mathtt{p}_i$, *while uncorrupted*, output $(\mathtt{conf}, \mathrm{tx})$;
- $\bar{\Sigma}_i$: latest snapshot (s, U) that party $\mathtt{p}_i$ performed *while uncorrupted*: output $(\mathtt{snap}, (s, U))$;
- H_{cont}: the set of (at the time) uncorrupted parties who produced ξ upon close/contest request and ξ was applied to correct η; and
- $\mathcal{H}$: the set of parties that remain uncorrupted.

Security conditions / events:

- Consistency (Head): In presence of an active adversary, the following condition holds at any point in time: For all i, j, $U_0 \circ (\bar{C}_i \cup \bar{C}_j) \neq \bot$, i.e., no two uncorrupted parties see conflicting transactions confirmed.
- Oblivious Liveness (Head): Consider any protocol execution in presence of a network adversary wherein the head does not get closed for a sufficiently long period of time, and consider an honest party p_i who enters transaction tx by executing $(\mathtt{newTx}, \mathrm{tx})$ *each time after having finished a snapshot.*

  Then the following eventually holds: $\mathrm{tx} \in \bigcap_{i \in [n]} \bar{C}_i \ \vee \ \forall i : U_0 \circ (\bar{C}_i \cup \{\mathrm{tx}\}) = \bot$, i.e., every party will observe the transaction confirmed or every party will observe the transaction in conflict with their confirmed transactions.[3]

[3] In particular, *liveness* expresses that the protocol makes progress under reasonable network conditions if no head parties get corrupted.

- Soundness (Chain): In presence of an active adversary, the following condition is satisfied: $\exists \tilde{S} \subseteq \bigcap_{i \in \mathcal{H}} \hat{S}_i : U_{\mathsf{final}} = U_0 \circ \tilde{S} \neq \bot$, i.e., the final UTxO set results from applying a set of transactions to U_0 that have been seen by all honest parties (whereas each such transaction applies conforming to the ledger rules).

- Completeness (Chain): In presence of an active adversary, the following condition holds: For $\tilde{S}$ as above, $\bigcup_{p_i \in H_{\mathsf{cont}}} \bar{C}_i \subseteq \tilde{S}$, i.e., all transactions seen as confirmed by an honest party at the end of the protocol are considered.

Note that the original version of the coordinated head satisfies a stronger version of liveness which is important for the 'user experience' in the protocol:

- Liveness (Head): Consider any protocol execution in presence of a network adversary wherein the head does not get closed for a sufficiently long period of time, and consider an honest party p_i who enters transaction tx by executing $(\texttt{newTx}, \mathrm{tx})$.

  Then the following eventually holds: $\mathrm{tx} \in \bigcap_{i \in [n]} \bar{C}_i \ \vee \ \forall i : U_0 \circ (\bar{C}_i \cup \{\mathrm{tx}\}) = \bot$, i.e., every party will observe the transaction confirmed or every party will observe the transaction in conflict with their confirmed transactions.[4]

7.1 Proofs

Consistency.

Lemma 1 (Consistency). *The coordinated head protocol satisfies the* Consistency *property.*

Proof. Observe that $\bar{C}_i \cup \bar{C}_j \subseteq \hat{S}_i$ since no transaction can be confirmed without every honest party signing off on it. Since parties do not sign conflicting transactions (see `reqSn`, 'wait'), we have $U_0 \circ \bar{C}_i \neq \bot$, $U_0 \circ \bar{C}_j \neq \bot$, and $U_0 \circ \hat{S}_i \neq \bot$. Thus, since $\bar{C}_i \cup \bar{C}_j \subseteq \hat{S}_i$ it follows that $U_0 \circ (\bar{C}_i \cup \bar{C}_j) \neq \bot$ □

Oblivious Liveness. For all lemmas towards oblivious liveness, we assume the presence of a network adversary, and that the head does not get closed for a sufficiently long period of time. We call this the *liveness condition*.

Lemma 2. *Under the liveness condition, any snapshot issued as* $(\texttt{reqSn}, s, T)$ *will eventually be confirmed in the sense that every party holds a valid mulisignature on it.*

Proof. Consider a party p_i receiving message $(\texttt{reqSn}, s, T)$. We demonstrate that p_i executes the code past the 'wait' instruction of the `reqSn` routine.

- Passing the 'require' guard: Note that the snapshot leader sends the request only if $\hat{s} = \bar{s}$, and for $s = \hat{s} + 1$. Thus, $\hat{s}_i = \hat{s}$ since p_i has already signed the snapshot for $\hat{s}$. The 'require' guard is thus satisfied for p_i.

- Passing the 'wait' guard: Since the snapshot leader sees $\hat{s} = \bar{s}$, also p_i will eventually see $\hat{s}_i = \bar{s}_i$. Furthermore, since all leaders are honest, it holds that $\hat{\mathcal{U}} \circ \mathcal{T}_{res} \neq \bot$ by construction.

[4] In particular, *liveness* expresses that the protocol makes progress under reasonable network conditions if no head parties get corrupted.

This implies that every party will eventually sign and acknowledge the newly created snapshot. Finally, the 'require' and 'wait' guards of the `ackSn` code will be passed by every party since an `ackSn` for snapshot number s can only be received for $s \in \{\hat{s}, \hat{s}+1\}$ as an acknowledgement can only be received for the current snapshot being worked on by p_i or a snapshot that is one step ahead—implying that everybody will hold a valid multisignature on the snapshot in consideration. □

Lemma 3 (Eternal snapshot confirmation). *Under the liveness condition, as long as new transactions are issued, for any $k > 0$, every party eventually confirms a snapshot with sequence number $s = k$.*

Proof. By Lemma 2, any requested snapshot eventually gets confirmed, implying that the next leader observes $\hat{s} = \bar{s}$ and thus, in turn, issues a new snapshot. Thus, for any k, a snapshot is eventually confirmed. □

Lemma 4 (Oblivious Liveness). *The coordinated head protocol satisfies the* Oblivious Liveness *property.*

Proof. Consider the first point in time where a transaction tx enters the system by some party p_i issuing $(\texttt{newTx}, \text{tx})$, and consider the next point in time t when p_i issues a snapshot.

By Lemma 3, this snapshot will eventually be issued and confirmed by all parties.

Let $\hat{\mathcal{T}}$ be the transactions to be considered by p_i's snapshot: $\hat{\mathcal{L}} = \bar{\mathcal{U}} \circ \hat{\mathcal{T}}$ where $\bar{\mathcal{U}}$ is the snapshot prior to p_i's. Since p_i issues $(\texttt{reqTx}, \text{tx})$ after each snapshot, we have that, either,

- $\text{tx} \in \hat{\mathcal{T}}$, in which case $\text{tx} \in \bigcap_{i \in [n]} \bar{C}_i$ after everybody has completed this snapshot, or,
- $\text{tx} \notin \hat{\mathcal{T}}$, in which case $\hat{\mathcal{L}} \circ \text{tx} = \bot$ (tx is still in the wait queue of $(\texttt{reqTx}, \text{tx})$. After everybody has completed this snapshot, it thus holds that $\forall i : U_0 \circ \bar{C}_i = \hat{\mathcal{L}}$, and thus, that $\forall i : U_0 \circ (\bar{C}_i \cup \{\text{tx}\}) = \bot$.

In both cases, the lemma follows. □

Soundness and completeness.

Lemma 5 (Soundness). *The basic head protocol satisfies the* Soundness *property.*

Proof. Let T be the set of transactions such that $U_{\mathsf{final}} = U_0 \circ T$. Since U_{final} is multi-signed, it holds that $T \subseteq \hat{S}_i$ (T is *seen*) by every honest party in the head. Furthermore, since honest signatures are only issued for valid transaction, $U_{\mathsf{final}} \neq \bot$ (i.e., U_{final} is a valid state), and soundness follows. □

Lemma 6 (Completeness). *The basic head protocol satisfies the* Completeness *property.*

Proof. Consider all parties $p_i \in H_{\mathsf{cont}}$. Since the close/contest process finally accepts the latest multi-signed snapshot, it holds that $U_{\mathsf{final}}.s \geq \max_{p_i \in H_{\mathsf{cont}}}(\bar{s}_i)$, and thus that $\bigcup_{p_i \in H_{\mathsf{cont}}} \bar{C}_i \subseteq \bigcap_{p_i \in \mathcal{H}} \hat{S}_i$, and completeness follows. □

References

[1] Extended UTXO-2 model. https://github.com/hydra-supplementary-material/eutxo-spec/blob/master/extended-utxo-specification.pdf.

[2] A formal specification of the cardano ledger. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/shelley-ledger.pdf.

[3] A formal specification of the cardano ledger integrating plutus core. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/alonzo-ledger.pdf.

[4] Hydra repository. https://github.com/input-output-hk/hydra.

[5] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of Bitcoin transactions. In *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, pages 541–560, 2018.

[6] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended UTxO model. In *4th Workshop on Trusted Smart Contracts*, 2020. http://fc20.ifca.ai/wtsc/WTSC2020/WTSC20_paper_25.pdf.

[7] Manuel M. T. Chakravarty, James Chapman, Kenneth M. Mackenzie, Orestis Melkonian, Jann, Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, Joachim, and Zahnentferner. Utxoma: Utxo with multi-asset support. 2020.

[8] Manuel MT Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. Hydra: Fast isomorphic state channels. *Cryptology ePrint Archive*, 2020.

[9] Kazuharu Itakura and Katsuhiro Nakamura. A public-key cryptosystem suitable for digital multisignatures. *NEC Research & Development*, (71):1–8, 1983.

[10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology – ASIACRYPT 2010*, pages 177–194, 2010.

[11] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures: Extended abstract. pages 245–254, 2001.

[12] Joachim Zahnentferner. An abstract model of UTxO-based cryptocurrencies with scripts. *IACR Cryptology ePrint Archive*, 2018:469, 2018.