Hydra: Fast Isomorphic State Channels

Manuel M. T. Chakravarty[1], Sandro Coretti[1], Matthias Fitzi[1], Peter Gaži[1], Philipp Kant[1], Aggelos Kiayias[2], and Alexander Russell[3]

[1]IOHK. `firstname.lastname@iohk.io`.
[2]University of Edinburgh and IOHK. `akiayias@inf.ed.ac.uk`.
[3]University of Connecticut and IOHK. `acr@cse.uconn.edu`.

Abstract

State channels are an attractive layer-two solution for improving the throughput and latency of blockchains. They offer optimistic offchain settlement of payments and expedient offchain evolution of smart contracts between multiple parties without imposing any additional assumptions beyond those of the underlying blockchain. In the case of disputes, or if a party fails to respond, cryptographic evidence collected in the offchain channel is used to settle the last confirmed state onchain, such that in-progress contracts can be continued under mainchain consensus.

A serious disadvantage present in current layer-two state channel protocols is that existing layer-one smart contract infrastructure and contract code cannot be reused offchain without change.

In this paper, we introduce *Hydra*, an *isomorphic* multi-party state channel. Hydra simplifies offchain protocol and smart contract development by directly adopting the layer-one smart contract system, in this way allowing the same code to be used both on- and off-chain. Taking advantage of the *extended UTxO model*, we develop a fast off-chain protocol for evolution of Hydra *heads* (our isomorphic state channels) that has smaller round complexity than all previous proposals and enables the state channel processing to advance on-demand, concurrently and asynchronously.

We establish strong security properties for the protocol, and we present and evaluate extensive simulation results that demonstrate that Hydra approaches the physical limits of the network in terms of transaction confirmation time and throughput while keeping storage requirements at the lowest possible. Finally, our experimental methodology may be of independent interest in the general context of evaluating consensus protocols.

1 Introduction

Permissionless distributed ledger protocols suffer from serious scalability limitations, including high transaction latency (the time required to settle a transaction), low throughput (the number of transactions that can be settled per unit of time), and excessive storage required to maintain the state of the system and its transaction history, which can be ever growing.

Several solutions have been proposed to mitigate these problems by adapting the details of the underlying ledger protocols. Such direct adaptations for scalability are often referred to as *layer-one* solutions.

Layer-one solutions face an inherent limitation, however, as settlement remains a cumbersome process that involves the participation of a large, dynamic set of participants and requires exchange of significant amounts of data. An alternative approach to improve scalability, which is our emphasis in this work, is *layer-two* (sometimes referred to also as *offchain*) solutions that overlay a new protocol on top of the (layer-one) blockchain. Layer-two solutions allow parties to securely transfer funds from the blockchain into an offchain protocol instance, settle transactions in this instance (quasi) independently of the underlying chain, and safely transfer funds back to the underlying chain as needed.

Offchain solutions have the advantage that they do not require additional trust assumptions about the honesty of parties beyond those of the underlying blockchain, and that they can be very efficient in the optimistic case where all participants in the offchain protocol instance behave as expected. In particular, such an instance operates among a small number of parties that communicate with each other directly, and in a way that allows them to forget about recent transactions as soon as they respectively update (and secure) their local states.

The most prominent offchain scalability solution is the concept of payment channels [9, 33, 16]. A payment channel is established among two parties, allowing them to pay funds back and forth on this channel; in the optimistic case, this can take place without notifying the layer-one protocol. Payment channels have been extended to payment-channel networks, e.g., the Bitcoin Lightning Network [33]. Such networks, in principle, allow for offchain fund transfers among any two parties that are connected via a path of payment channels.

As a drawback, in a traditional payment-channel network a transaction between two parties that do not share a direct payment channel requires interaction among all parties on a payment-channel path between them (so-called *intermediaries*), even in the optimistic case. *Virtual* payment channels, e.g., Perun [19], address this and do not require interaction with intermediate parties (in the optimistic case).

State channels [5] extend the concept of payment channels to states in order to support smart contracts. State-channel networks [21, 15, 29] likewise extend the concept of state channels to networks (analogously to the network extension discussed above). Still, these networks only allow for the establishment of pairwise state channels over the network.

Multi-party state channels were introduced in [31] together with a high-level description of a respective protocol. A multi-party state channel allows a set of parties to maintain a "common" state whereon they can compute without interacting with the blockchain (in the optimistic case).

In [18], the notion of multi-party *virtual* state channels was introduced, state channels among multiple parties that can be setup without blockchain interaction (given that a connected graph of pairwise state channels among the parties already exists); and a respective protocol was presented.

Despite the above significant advances, important challenges remain, both in terms of establishing high offchain processing performance that approximates the physical limits of the underlying network as well as in the sense of imposing significant conceptual and engineering overhead over layer-one as the offchain contract state must be verified in a non-native representation; the reason is that the state of the contracts evolved in a specific state channel needs to be isolated and represented in a form that permits it to be manipulated both offchain and by the onchain smart contract scripting system in case of an offchain dispute. This lead to designs where the computations performed offchain are no longer in the representation used by the ledger itself; i.e., they are non-native. For example, the sample Solidity contract of [31] serializes the state into a `bytes32` array. The smart contracts themselves need to be adapted correspondingly. In other words, the scripting system of the ledger and of state channels attached to the ledger diverge in a substantial way, effectively imposing two distinct scripting systems.

Hydra. In Hydra, we tackle both problems, offchain processing performance and state representation, with the introduction of *isomorphic* multi-party state channels. These are state channels that are capable of expediently reusing the exact state representation of the underlying ledger and, hence, inherit the ledger's scripting system as is. Thus, state channels effectively yield parallel, offchain ledger siblings, which we call *heads*—the ledger becomes multi-headed. The creation of a new head follows a similar commitment scheme as is common in state channels. However, once a state channel is closed, either cooperatively or due to a dispute, the head state is seamlessly absorbed into the underlying ledger state and the same smart contract code as used offchain is now used onchain. This is possible, even without a priori registration of the contracts used in a head, because one and the same state representation and contract (binary) code is used offchain and onchain.

Not every blockchain scripting system is conducive to isomorphic state channels. Building them requires to efficiently carve out arbitrary chunks of blockchain state, process them independently, and be able at any time to efficiently merge them back in. We observe that the Bitcoin-style UTxO ledger model [6, 34] is particularly well suited as a uniform representation of onchain and offchain state, while simultaneously promising increased parallelism in transaction processing inside multi-party state channels. While the main restriction of the plain UTxO model has traditionally been its limited scripting capabilities, the introduction of the *Extended UTxO model (EUTxO)* [13] has lifted this restriction and enabled support for general state machines. Extended UTxO models form the basis for the smart contract platforms of existing blockchains, such as Cardano [14] and Ergo [17]; hence, the work presented in this paper would also be of immediate practical relevance.

Just like the UTxO ledger representation, the EUTxO ledger representation makes all data dependencies explicitly without introducing *false dependencies* — in other words, two transactions do only directly or indirectly depend on each other if there is an actual data dependency between them. This avoids the over-sequentialization of systems depending on a global state. Hence, the length of the longest path through the EUTxO graph coincides with the *depth complexity* of the workload entailed by transaction processing and validation. This is the optimum as far as parallel transaction processing is concerned [10].

Exploiting the EUTxO ledger representation, we are able to design an offchain protocol with unparalleled performance. In particular, the Hydra head protocol is capable of offchain processing asynchronously and concurrently between different members of the head, utilizing merely 3 rounds of interaction for updates. In contrast previous works in multiparty state channels either required a synchronous operation or imposed 4 rounds to facilitate sequentializing inputs and organizing the offchain state.

In more detail, in Hydra, a set of parties coordinates to *commit* a set of UTxOs (owned by the parties) into an offchain protocol, called the *head protocol.* That UTxO set constitutes the initial head state, which the parties can then evolve by handling smart contracts and transactions among themselves without blockchain interaction—in the optimistic case.

Due to the isomorphic nature of Hydra heads, transaction validation, including script execution, proceeds according to the exact same rules as onchain. In fact, the exact same validation code can be used. This guarantees that onchain and offchain semantics coincide, leading to significant engineering simplifications. In case of disputes or in case some party wishes to terminate the offchain protocol, the parties *decommit* the current state of the head back to the blockchain. Ultimately, a decommit will result in an updated blockchain state that is consistent with the offchain protocol evolution on the initially committed UTxO set. To reduce mainchain overhead, the mainchain is oblivious of the detailed transaction history of the head protocol that lead to the updated state. Crucially, the time required to decommit is independent of the number of parties participating in a head or the size of the head state. Moreover, the decommit process is designed such that, when the latest state in the head is very large, the head state can be decommitted in small (but parallel) chunks. Finally, Hydra allows incremental commits and decommits, i.e., UTxOs can be added to and removed from a running head without closing it.

Cross-head networking. In this paper, we focus solely on the analysis of the Hydra head protocol; nevertheless, the existence of multiple, partially overlapping heads off the mainchain can give rise to cross-head communication (as in the Lightning Network [33]), using similar techniques to [21, 18].

Online participation requirements. The Hydra head protocol is geared towards the scenario where the participants who are required to validate transactions are online and responsive. As in e.g. [33], being offline will prevent progress, and also participation in a potential onchain dispute resolution. The scenario where a number of parties are regularly offline is also of interest but not in scope for the current version.

Performance evaluation methodology and experimental results. As transaction-processing performance is the fundamental motivation for layer-two protocols, these properties of the Hydra protocol are particularly important to establish. While transactions-per-second (TPS) is an immediate figure of merit for deployed systems, it is sensitive to changes in the underlying hardware or network; in particular, it is an unreliable means for experimentally comparing various algorithmic proposals unless the experiments precisely duplicate the computing environment which is also sensitive to user inputs. To avoid these difficulties and second-guessing specific usage scenarios, we adopt a "baseline relative" approach to establish performance guarantees, which demonstrates that Hydra achieves performance that approaches the theoretical optimum for *any consensus protocol.* Our experimental results are obtained by simulation, which additionally permits a high-precision exploration of the specific design choices adopted by the Hydra protocol. We consider two major types of baselines elaborated below.

The universal baseline. As mentioned above, we begin by considering a baseline reflecting the weakest obligations of any consensus algorithm. Specifically, the universal baseline merely considers the cost of processing each transaction and disseminating the transactions across the network; observe that any iterated consensus algorithm that yields full state at each node must necessarily carry out both operations. We demonstrate that Hydra achieves efficiency that rivals even this ideal for most scenarios. As this protocol-independent baseline is one against which any iterated consensus algorithm can be compared, near optimality with respect to this baseline implicitly demonstrates that Hydra is competitive with any other consensus layer. In our experiments we compare Hydra with the universal baseline for a number of different scenarios that reflect user behavior.

The unlimited baseline. The second baseline focuses on the characteristics of the protocol itself. In particular it asks how does the protocol implementation compare to an idealized execution of the protocol by a set of nodes that experience no local contention for resources. This baseline comparison is meant to be complementary to the universal baseline and helps answer the following question. Whenever there is divergence between the universal baseline and the actual consensus protocol execution in the experiment, how much of this divergence is to be attributed to the inherent cost of running the consensus protocol vs. the costs arising due to contention for resources within each node. Even good consensus protocol designs are expected to diverge from the universal baseline: after all, consensus is a difficult problem to solve. However good protocol designs should always approximate their unlimited baseline. In our experiments we demonstrate that this is the case for Hydra in all the different scenarios of our experimental setup.

Experimental results. We conducted detailed simulations of head performance under a variety of load and networking scenarios, including both geographically localized heads and heads with participants spread over multiple continents, incurring large network delays. We found that our head protocol, in the optimistic case, achieves progress that rivals the speed and throughput of the network in all configurations; this is aided by the concurrency afforded by the partial-only transaction ordering permitted by the graph-structure underlying UTxO ledgers.

Comparison to previous work. A number of previous works study state channel protocols. The protocol by Miller et al. [31] allows a set of parties to initiate a smart contract instance (state) onchain and take it offchain. The state can then be evolved offchain without chain interaction in the all-honest case. By concurrently handling disputes in a shared contract, dispute resolution remains in $O(\Delta)$ time, where Δ is the settlement time for an onchain transaction. The offchain protocol proceeds in phases of 4 asynchronous rounds where a leader coordinates the confirmation of new transactions among the participants in the offchain protocol. Similarly to Hydra, the protocol allows to add/remove funds from the offchain contract while it is running.

The protocol by Dziembowski et al. [18] is based on pairwise state channels and allows the instantiation of a multi-party state channel among any set of parties that are connected by paths of pairwise state channels—the instantiation of the multi-party channel does not require any interaction with the mainchain. The offchain protocol proceeds in phases of 4 synchronous rounds to confirm new transactions without the need for a coordinating leader.

The Hydra offchain protocol is fully asynchronous; in the optimistic case, transactions are confirmed in 3 (asynchronous) rounds independently of each other, and without having to involve a leader. A leader is only required for the resolution of transaction conflicts and for periodic "garbage collection" that allows the protocol to maintain state size independent of the size of the transaction history.

In comparison to prior solutions cited above, Hydra provides faster confirmation times in the offchain protocol; this is an advantage enabled by the structural organization of transactions in the EUTxO model, whereas prior protocols are hindered by a monolithic state organization. An additional advantage over [31] and [18] is that those fix the set of contracts that can be evolved in a given state channel at channel creation time; Hydra does not require such an a priori commitment: new contracts can be introduced in a head after creation in the native EUTxO language of the underlying blockchain. Another significant difference to [18] is that their protocol calls for parties to lock funds on the mainchain on behalf of other parties—caused by asymmetries induced by the composition along paths of pairwise state channels, whereas in Hydra as well as in [31], the parties only need to lock funds on behalf of themselves. Finally, Hydra is isomorphic and thus reuses the existing smart contract system and code for offchain computations. This is not the case for [31] and [18]. For example, if we consider the sample Solidity contract of [31], it would have to implement a state machine capable of executing EVM bytecode to achieve contract (system) reuse—and hence, isomorphic state channels.

We note that there is also a large number of non-peer reviewed proposals for state-channel-based solutions such as [28, 15, 29, 3]. These proposals come with various degrees of formal specification and provable security guarantees and their systematization is outside of our current scope; it suffices to observe that none of them provides the isomorphism property or comes with a complete formal security analysis and an experimental evaluation.

Two concepts related, but distinct, from state channels are *sidechains* (e.g., [7, 24, 26]) and *non-custodial chains* (e.g., [32, 27, 20, 4]), including plasma and rollups. Sidechains enable the transfer of assets between a mainchain and a sidechain via a pegging mechanism, with the mainchain protected from sidechain security failures by a "firewall property"; the sidechain has its own consensus rules and, contrary to a state channel, funds may be lost in case of a sidechain security collapse. Non-custodial chains, on the other hand, delegate mainchain transaction processing to an untrusted aggregator and are capable, as in state channels, to protect against a security failure. Nevertheless, the aggregator is a single-point-of-failure and its corruption, in a setting where a large number users are served by the same non-custodial chain, gives rise to the "mass-exit" problem (see e.g., [20]); note that state channels, in contrast, can scale to a large number of users via state channel networks [21] without requiring many users per channel. We note finally that work in progress on optimistic rollups, reported in [4], claims a feature similar to our isomorphic property, nevertheless without the latency benefits of our approach as their settlement still advances with the underlying mainchain.

2 Preliminaries

2.1 Multisignatures

A multisignature scheme [25, 30] is a tuple of algorithms $\mathsf{MS} = (\mathsf{MS\text{-}Setup}, \mathsf{MS\text{-}KG}, \mathsf{MS\text{-}AVK}, \mathsf{MS\text{-}Sign}, \mathsf{MS\text{-}ASig}, \mathsf{MS\text{-}Verify})$ such that $\Pi \leftarrow \mathsf{MS\text{-}Setup}(1^k)$ generates public parameters; with these in place, $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{MS\text{-}KG}(\Pi)$ can be used to generate fresh key pairs. Then

- $\sigma \leftarrow \mathsf{MS\text{-}Sign}(\Pi, \mathsf{sk}, m)$ signs a message m using key sk;
- $\tilde{\sigma} \leftarrow \mathsf{MS\text{-}ASig}(\Pi, m, \mathcal{V}, \mathcal{S})$ aggregates a set $\mathcal{S}$ of signatures into a single, aggregate signature $\tilde{\sigma}$.

The algorithm $\mathsf{avk} \leftarrow \mathsf{MS\text{-}AVK}(\Pi, \mathcal{V})$ aggregates a tuple $\mathcal{V}$ of verification keys vk into a single, aggregate verification key avk which can be used for verification: $\mathsf{MS\text{-}Verify}(\Pi, \mathsf{avk}, m, \tilde{\sigma}) \in \{\mathtt{true}, \mathtt{false}\}$ verifies an aggregate signature under an aggregate verification key. In the following, we often make the parameter Π implicit in the function calls for better readability.

Intuitively, the security of a multisignature scheme guarantees that, if avk is produced from a tuple of verification keys $\mathcal{V}$ via $\mathsf{MS\text{-}AVK}$, then no aggregate signature $\tilde{\sigma}$ can pass verification $\mathsf{MS\text{-}Verify}(\mathsf{avk}, m, \tilde{\sigma})$ unless all honest parties holding keys in $\mathcal{V}$ signed m. A full treatment appears in Appendix A.

2.2 Extended UTxO model & state machines

The basis for our fast isomorphic state channels is Bitcoin's UTxO ledger model [6, 34]. It arranges transactions in a directed acyclic graph structure, thus making the available parallelism explicit: any two transactions that are not directly or indirectly dependent on each other can be processed independently.

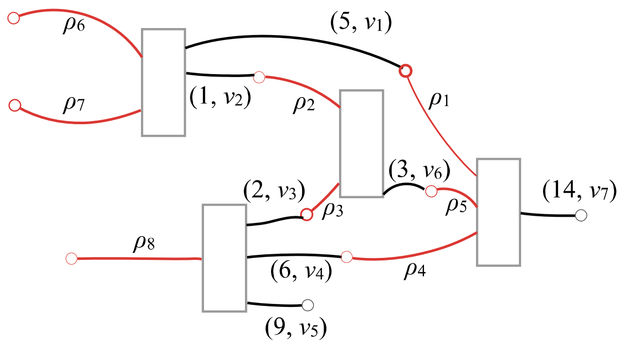


Figure 1: Example of a plain UTxO graph

UTxO. Transactions in an UTxO ledger contain a set of inputs and outputs, where outputs lock an amount of cryptocurrency, such that only authorized inputs of subsequent transactions can connect and consume those funds. This arrangement results in graphs, such as the one in Figure 1, where the boxes represent transactions with (red) inputs to the left and (black) outputs to the right.

Each output locks some cryptocurrency, which can be transferred via a subsequent transaction by consuming that output with a new input. The set of dangling (unconnected) outputs are the *unspent transaction outputs (UTxOs)* — there are two of those in Figure 1. In addition to the locked currency, each output also comes with a predicate ν, called its *validator*. In Figure 1, we use pairs (n, ν) to indicate that a given output locks n cryptocurrency with validator predicate ν.

Where outputs carry validators, each input comes with a *redeemer* value ρ. To determine whether a given input of the currently validated transaction *tx* is permitted to connect to a, as of yet, unspent output, we determine whether the validator predicate ν of that output applies for the redeemer ρ; or more formally, we check that $\nu(\rho, \sigma) = \texttt{true}$, where the *validation context* σ represents some properties of the transaction that the spending input belongs to, such as the transaction's cryptographic hash value. For example, the validator may require the redeemer to be a signature on the transaction hash contained in the context σ for a specific key pair, such that only the owner of the private key can spend an output locked by that validator.

Extended UTxO. The *Extended UTxO Model (EUTxO)* [13] preserves this structure, while adding support for more expressive smart contracts and, in particular, for multi-transaction state machines, which serve as the basis for the mainchain portion of the work presented here. This additional expressiveness is achieved by two changes to the plain UTxO scheme outlined before:

- Outputs carry, in addition to a cryptocurrency value n and a validator ν, now also a *datum* δ, which can, among other things, be used to maintain the state of long running smart contracts.

- The validation context σ is extended to contain the entire validated transaction *tx* as well as the UTxOs consumed by the inputs of that transaction.

In this extended model, evaluation of the validator predicate implies checking $\nu(\rho, \delta, \sigma) = \texttt{true}$. Besides maintaining contract state in δ, the fact that the validator can inspect the entire validated transaction *tx* through σ enables validators to enforce that contract invariants are maintained across entire chains of transactions.

Although formal results about EUTxO are rather recent, extended UTxO models already form the basis for the smart-contract platforms of existing blockchains — in particular, Cardano [14] and Ergo [17]. Consequently, the Hydra head protocol as presented in this paper is of immediate practical relevance to these existing systems.

User-defined tokens. In addition to the basic EUTxO extension, we generalize the currency *values* recorded on the ledger from integral numbers to *generalized user-defined tokens* [1]. Put simply (sufficient to understand the concepts in this paper), values are sets that keep track how many units of which tokens of which currency are available. For example, the value $\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 3\}, c \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1\}\}$ contains 3 Coin coins (there is only one (fungible) token Coin for a payment currency Coin), as well as (non-fungible) tokens t_1 and t_2, which are both of currency c. Values can be added naturally, e.g.,

$$\begin{aligned}
&\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 3\}, c \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1\}\} \\
+\ &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 1\}, c \mapsto \{t_3 \mapsto 1\}\} \\
=\ &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 4\}, c \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1, t_3 \mapsto 1\}\} \ .
\end{aligned}$$

In the following, $\varnothing$ is the empty value, and $\{t_1, \ldots, t_n\} :: c$ is used as a shorthand for $\{c \mapsto \{t_1 \mapsto 1, \ldots, t_n \mapsto 1\}\}$.

The EUTxO ledger consists of *transactions*: Transactions are quintuples $\mathrm{tx} = (I, O, \mathsf{val}_{\mathsf{Forge}}, r, \mathcal{K})$ comprising a set of *inputs* I, a list of *outputs* O, values of *forged/burned tokens* $\mathsf{val}_{\mathsf{Forge}}$, a *slot range* $r = (r_{\mathsf{min}}, r_{\mathsf{max}})$, and a set of public keys $\mathcal{K}$. Each input $i \in I$ is a pair consisting of an *output reference* $\mathsf{out\text{-}ref}$ (consisting of a transaction ID and an index identifying an output in the transaction) and a *redeemer* ρ (used to supply data for validation). Each output $o \in O$ is a triple $(\mathsf{val}, \nu, \delta)$ consisting of a value val, a validator script ν, and a datum δ. The slot range r indicates the slots within which tx may be confirmed and, finally, $\mathcal{K}$ are the public keys under which tx is signed.

In order to validate a transaction tx with input set I, for each output $o = (\mathsf{val}, \nu, \delta)$ referenced by an $i = (\mathsf{out\text{-}ref}, \rho) \in I$, the corresponding validator ν is run on the following inputs: $\nu(\mathsf{val}, \delta, \rho, \sigma)$, where the validation context σ consists of tx and *all* outputs referenced by some $i \in I$ (not just o). Ultimately, tx is valid if and only if all validators return `true`.

State Machines. A convenient abstraction for EUTxO smart contracts spanning a sequence of related transactions are state machines. Specifically, we adopt *constraint emitting machines (CEMs)* [13]. These are based on Mealy machines and consist of a set of states S_{CEM}, a set of inputs I_{CEM}, a predicate $\mathit{final}_{\text{CEM}} : S_{\text{CEM}} \to \mathit{Bool}$ identifying final states, and a step relation $s \xrightarrow{i} (s', \mathrm{tx}^{\equiv})$, which takes a state s on an input i to a successor state s' under the requirements that the constraints $\mathrm{tx}^{\equiv}$ are satisfied.

We implement CEMs on a EUTxO ledger (the mainchain) by representing a sequence of CEM states as a sequence of transactions. Each of these transactions has got a *state-machine input* i_{CEM} and a *state-machine output* o_{CEM}, where the latter is locked by a validator ν_{CEM}, implementing the step relation. The only exceptions are the initial and final state, which have got no state-machine input and output, respectively.

More specifically, given two transactions tx and tx$'$, they represent successive states under $s \xrightarrow{i} (s', \mathrm{tx}^{\equiv})$ iff

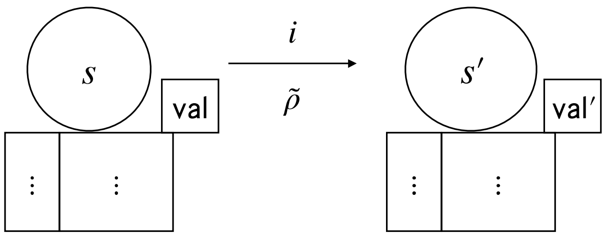


Figure 2: Transactions representing successive states in a CEM transition relation $s \xrightarrow{i} (s', \mathrm{tx}^{\equiv})$. Fields val and val' are the value fields of the state-machine outputs and $\tilde{\rho}$ is the additional data.

- state-machine output $o_{\mathrm{CEM}} = (\mathsf{val}, \nu_{\mathrm{CEM}}, s)$ of tx is consumed by the state-machine input $i'_{\mathrm{CEM}} = (\mathsf{out\text{-}ref}, \rho)$ of tx', whose redeemer is $\rho = i$ (i.e., the redeemer provides the state-machine input) and
- either $\mathit{final}_{\mathrm{CEM}}(s') = \mathtt{true}$ and tx' has no state-machine output, or $o'_{\mathrm{CEM}} = (\mathsf{val}', \nu_{\mathrm{CEM}}, s')$ and tx' meets all constraints imposed by $\mathrm{tx}^{\equiv}$.

Sometimes it is useful to have additional data $\tilde{\rho}$ provided as part of the redeemer, i.e., $\rho = (i, \tilde{\rho})$. A state transition of the described type is represented by two connected transactions as shown in Fig. 2. For simplicity, state-machine inputs and outputs are not shown, with the exception of the value fields val and val' of the state-machine output.

3 Protocol Overview

The Hydra protocol provides functionality to lock a set of UTxOs on a blockchain, referred to as the *mainchain*, and evolve it inside a so-called offchain *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 offchain, and no responsive honest party involved in a head can ever lose any funds other than by consenting to give them away.

The advantage of head evolution from a liveness viewpoint is that, under good conditions, it can essentially proceed at network speed, thereby reducing latency and increasing throughput in an optimal way. At the same time, the head protocol provides the same smart-contract capabilities as the mainchain.

To avoid overloading with technical details, the main body of the paper presents a simplified version of Hydra to convey the basic concepts and ideas of the new protocol. Also in the overview, we focus on the simplified protocol and outline the differences of the full protocol in Section 3.4. A detailed description of the simplified protocol is given in Sections 4– 6, and Appendix B. The full protocol is described in Appendix C.

3.1 The big picture

To create a head-protocol instance, any party may take the role of an *initiator* and ask a set parties (including himself), the *head members*, to participate in the head by announcing the identities of the parties.

Each party then establishes pairwise authenticated channels to all other parties or—if this is not possible—aborts the protocol setup.[1]

The parties then exchange, via the pairwise authenticated channels, some public-key material. This public-key material is used both for the authentication of head-related onchain transactions that are restricted to head members (e.g., a non-member is not allowed to close the head) and for multisignature-based event confirmation in the head.

The initiator then establishes the head by submitting an *initial* transaction to the mainchain that contains the head parameters and forges special *participation tokens* identifying the head members by assigning each token to the public key distributed by the respective party during the the setup phase. The initial transaction also initializes a state machine (see Fig. 3) for the head instance that manages the "transfer" of UTxOs between mainchain and head.

Once the initial transaction appears on the mainchain, establishing the initial state initial, each head member can attach a *commit* transaction, which locks (on the mainchain) the UTxOs that the party wants to commit to the head.

The commit transactions are subsequently collected by the *collectCom* transaction causing a transition from initial to open. Once the open state is confirmed, the head members start running the offchain *head protocol*, which evolves the initial UTxO set (the union over all UTxOs committed by all head members) independently of the mainchain. For the case where some head members fail to post a *commit* transaction, the head can be aborted by going directly from initial to final.

The head protocol is designed to allow any head member at any point in time to produce, without interaction, a certificate for the current head UTxO set. Using this certificate, the head member may advance the state machine to the closed state.

Once in closed, the state machine grants parties a *contestation period*, during which each party may (one single time) contest the closure by providing the certificate for a newer head UTxO set. Contesting leads back to the state closed. 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.

3.2 The mainchain state machine

The mainchain part of the Hydra protocol fulfills two principal functions: (1) it locks the mainchain UTxOs committed to the head while the head is active and (2) it facilitates the settlement of the final head state back to the mainchain after the head is closed. In combination, these two functions effectively result in replacing the initial head UTxO set by the final head UTxO set on the mainchain in a manner that respects but does not persist the complete set of head transactions.

The state machine (Fig. 3) implementing the mainchain protocol comprises the four states initial, open, closed, and final, where the first two realize the first function (locking the initial UTxO set) and the second two realize the second function (settling the final UTxO set on the mainchain).

State machines inherently sequentialize all actions that involve the machine state. This simplifies both reasoning about and implementing the protocol. However, steps that could otherwise be taken in parallel now need to be sequentialized, which might hurt performance. For the cases where this sequentialization would severely affect protocol performance, we employ a (to our knowledge) novel

[1]We generally assume that mechanisms for establishing pairwise authenticated channels are in place, e.g., by means of a public-key infrastructure.

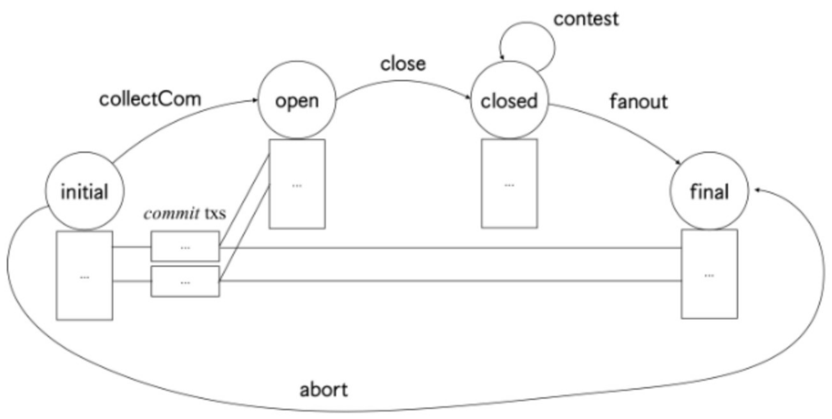


Figure 3: Mainchain state diagram for the simple version of the Hydra protocol.

technique to parallelize the progression of the state machine on the mainchain.

We use this technique to parallelize the construction of the initial UTxO set of the head. Without parallelization, all n head members would have to post their commit transactions (their portion of the initial UTxO set) in sequence, requiring a linear chain of n transactions, each for one state transition at a time. Instead, we make the state machine consume all n commit transactions in a single state transition. In Fig. 3, we represent this in the following way: the transaction representing state initial connects to the transaction representing state open not just via the *collectCom* state transition, but also via a set of commit transactions (one for each head member).

This requires some extra care. We want to ensure that each head member posts exactly one commit transaction and that the open transaction faithfully collects all commit transactions. We gain this assurance by issuing a single non-fungible token to each head member—we call this the *participation token*. This token must flow through the commit transaction of the respective head member and the open transaction, to be valid, must collect the full set of participation tokens. We may regard the participation token as representing a *capability* and *obligation* to participate in the head protocol.

3.3 The head protocol

The head protocol starts with an initial set U_0 of UTxOs that is identical to the UTxOs locked onchain.

Transactions and local UTxO state. The protocol *confirms* individual transactions in full concurrency by collecting and distributing multisignatures on each issued transaction separately. As soon as such a transaction is confirmed, it irreversibly becomes part of the head UTxO state evolution—the transaction's outputs are immediately spendable in the head, or can be safely transferred back onchain in case of a head closure.

Each party maintains their view of the local UTxO state $\overline{\mathcal{L}}$, which represents the current set of UTxOs evolved from the initial UTxO set U_0 by applying all transactions that have been confirmed so far in the head. As the protocol is asynchronous the parties' views of the local UTxO state generally differ.

Snapshots. The above transaction handling would be enough to evolve the head state. However, an eventual onchain decommit would have to transfer the full transaction history onchain as there would be no other way to evidence the correctness of the UTxO set to be restored onchain.

To minimize local storage requirements and allow for an onchain decommit that is independent of the transaction history, UTxO snapshots $U_1, U_2, \ldots$ are continuously generated. For this, a *snapshot leader* requests his view of the confirmed state $\overline{\mathcal{L}}$ to be multisigned as a new snapshot—the first head snapshot corresponding to the initial state U_0. A snapshot is considered *confirmed* if it is associated with a valid multisignature.

In contrast to transactions, the snapshots are generated sequentially. To have the new snapshot $U_{i+1} = \overline{\mathcal{L}}$ multisigned, the leader does not need to send his local state U_{i+1}, but only indicate, by hashes, the set of (confirmed) transactions to be applied to U_i in order to obtain U_{i+1}.

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

As soon as a snapshot is seen confirmed, a participant can safely delete all transactions that have already been processed into it as the snapshot's multisignature is now evidence that this state once existed during the head evolution.

Closing the head. A party that wants to close the head decommits his confirmed state $\overline{\mathcal{L}}$ by posting, onchain, the latest seen confirmed snapshot U_ℓ together with those confirmed transactions that have not yet been processed by this snapshot. During the subsequent contestation period, other head members can post their own local confirmed states onchain.

3.4 The full protocol and further aspects

To improve on the basic protocol, we change the mainchain state machine (as described in Appendix C) to include

- incremental commits and decommits (adding UTxOs to or removing them from the head without closing),
- optimistic one-step head closure without the need for onchain contestation,
- pessimistic two-step head closure with an $O(\Delta)$ contestation period, independent of n, where Δ is the onchain settlement time of a transaction, and
- split onchain decommit of the final UTxO set (in case it is too large to fit into a single transaction).

These further protocol aspects are summarized in Appendix D:

- The handling of fees incentivizing parties to advance the head's state machine on the mainchain.
- The handling of time and timing issues in the (asynchronous) head protocol.
- Transaction throttling in the head to avoid the head's state becoming too large under pessimistic conditions.

4 Protocol Setup

In order to create a head-protocol instance, an initiator invites a set of participants $\{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$ (himself being one of them) to join by announcing to them the protocol parameters: the list of participants, the parameters of the (multi-)signature scheme to be used, etc.

Each party then establishes pairwise authenticated channels to all other parties.

For some digital-signature scheme, each party p_i generates a key pair $(k_{i,\mathsf{ver}}, k_{i,\mathsf{sig}})$ and sends his respective verification key $k_{i,\mathsf{ver}}$ to all other parties. This "standard" digital-signature scheme will be used to authenticate mainchain transactions that are restricted to members of the head-protocol instance.

For the multisignature scheme (MS)—see Section 2.1—each party p_i generates a key pair

$$(K_{i,\mathsf{ver}}, K_{i,\mathsf{sig}}) \;\leftarrow\; \mathsf{MS\text{-}KG}(\Pi)$$

and sends his verification key $K_{i,\mathsf{ver}}$ to all other parties.

Each party then computes his aggregate key from the received verification keys:

$$K_{\mathsf{agg}} \;:=\leftarrow\; \mathsf{MS\text{-}AVK}(\Pi, (K_{j,\mathsf{ver}})_{j\in[n]})\,.$$

The multisignature scheme will be used for the offchain confirmation (and offchain and onchain verification) of head-protocol events.

At the end of this initiation, each party p_i stores his signing key and all received verification keys for the signature scheme,

$$\big(k_{i,\mathsf{sig}},\; \underline{k}_{\mathsf{ver}} := (k_{j,\mathsf{ver}})_{j\in[n]}\big)\;,$$

and his signing key, the verification keys, and the aggregate verification key for the multisignature scheme,

$$\big(K_{i,\mathsf{sig}},\; \underline{K}_{\mathsf{ver}} := (K_{j,\mathsf{ver}})_{j\in[n]},\; K_{\mathsf{agg}}\big)\;.$$

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.[2]

The initiator now posts the *initial* transaction onchain as described in Section 5.

5 Mainchain

Here we describe the details of the mainchain state machine (SM) controlling a Hydra head (see Fig. 3). For state transitions, a formal description of the conditions in $\mathrm{tx}^{\equiv}$ is foregone in favor of the intuitive explanations in the text and the figures.

Onchain verification algorithms. The status of the head is maintained in a variable η, which is part of the SM state and updated by so-called *onchain verification (OCV) algorithms* Initial, Close, Contest, and Final. In the context of the mainchain protocol, these OCV algorithms are intentionally kept as generic as possible; this keeps the mainchain SM compatible with many potential head-protocol variants. The concrete OCV algorithms for the head protocol specified in this paper are given in context of the head protocol itself as they depend on the specific head-protocol internals:

[2]Of course, aborting the initiation can be achieved more gracefully by explicitly notifying the initiator about one's non-participation. Techniques are even known to finish such an initiation in agreement among all parties [23].

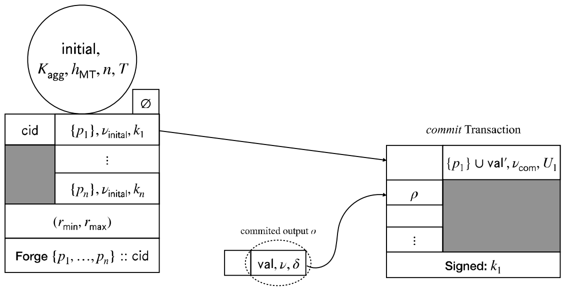

initial, $K_{\mathsf{agg}}, h_{\mathsf{MT}}, n, T$
∅
cid
$\{p_1\}, \nu_{\mathsf{inital}}, k_1$
⋮
$\{p_n\}, \nu_{\mathsf{inital}}, k_n$
$(r_{\mathsf{min}}, r_{\mathsf{max}})$
Forge $\{p_1, \ldots, p_n\}$ :: cid
commited output o
val, ν, δ
commit Transaction
$\{p_1\} \cup \mathsf{val}', \nu_{\mathsf{com}}, U_1$
ρ
⋮
Signed: k_1

Figure 4: *initial* transaction (left) with *commit* transaction (right) attached and one of the locked outputs (center).

verification of head-protocol certificates and related onchain state updates. As such, the OCV algorithms can be seen as abstract mainchain algorithms implemented by the specific head protocol. Consequently, the OCV implementation for our head protocol is described in Section 6.3.1.

Initial state. After the setup phase of Section 4, the head initiator posts an *initial* transaction (see Fig. 4). The *initial* transaction establishes the SM's initial state $(\mathsf{initial}, K_{\mathsf{agg}}, h_{\mathsf{MT}}, n, T)$, where $\mathsf{initial}$ is a state identifier, K_{agg} is the aggregated multisignature key established during the setup phase, h_{MT} is the root of a Merkle tree for the signature verification keys $\underline{k}_{\mathsf{ver}} = (k_1, \ldots, k_n)$ exchanged during the setup phase (identifying the head members), n is the number of head members, and T is the length of the contestation period. The *initial* transaction also forges n participation tokens $\{p_1, \ldots, p_n\} :: \mathsf{cid}$, where the currency ID cid is given by the unique *monetary-policy script* consumed by the cid input. The script is unique as it is bound to an output and the ledger prevents double spending. Consequently, we can use cid as a unique identifier for the newly initialized head.

Crucially, the *initial* transaction has n outputs, where each output is locked by a validator ν_{initial} and the i^{th} output has k_i in its data field. Validator ν_{initial} ensures the following: either the output is consumed by

1. an SM abort transaction (see below) or
2. a commit transaction (identified by having validator ν_{com} in its only output), and
   (a) the transaction is signed and the signature verifies as valid with verification key k_i,
   (b) the data field of the output of the commit transaction is $U_i = \mathsf{makeUTxO}(o_1, \ldots, o_m)$, where the o_j are the outputs referenced by the commit transaction's inputs and $\mathsf{makeUTxO}$ stores pairs $(\mathsf{out\text{-}ref}_j, o_j)$ of outputs o_j with the corresponding output reference $\mathsf{out\text{-}ref}_j$.

The general well-formedness and validity of the *initial* transaction is checked on the mainchain. The head members additionally check whether the head parameters match the parameters agreed on during the setup phase. In case of a mismatch the head opening is considered as failed.

Committing outputs to a head. To lock outputs for a Hydra head, the i^{th} head member will attach a *commit* transaction (see Fig. 4) to the i^{th} output of the *initial* transaction. Validator ν_{com} ensures that the *commit* transaction correctly records the partial UTxO set U_i committed by the party.

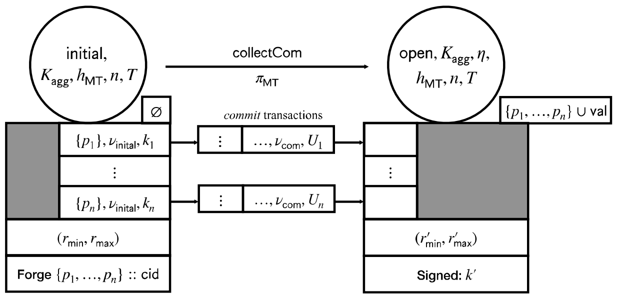

initial, $K_{\text{agg}}, h_{\text{MT}}, n, T$
collectCom
π_{MT}
open, $K_{\text{agg}}, \eta, h_{\text{MT}}, n, T$
∅
commit transactions
$\{p_1, \dots, p_n\} \cup$ val
$\{p_1\}, \nu_{\text{inital}}, k_1$
$\dots, \nu_{\text{com}}, U_1$
$\{p_n\}, \nu_{\text{inital}}, k_n$
$\dots, \nu_{\text{com}}, U_n$
$(r_{\text{min}}, r_{\text{max}})$
$(r'_{\text{min}}, r'_{\text{max}})$
Forge $\{p_1, \dots, p_n\}$ :: cid
Signed: k'

Figure 5: *initial* transaction (left) with *collectCom* transaction (right) and *commit* transactions (center).

All *commit* transactions will in turn be collected by an SM transaction—either *collectCom* or *abort* (see below).

Collecting commits. The SM transition from initial to open is achieved by posting the *collectCom* transaction (see Fig. 5). All parameters K_{agg}, h_{MT}, n, and T remain part of the state, but in addition, a value $\eta \leftarrow \mathsf{Initial}(U_1, \dots, U_n)$ is stored in the state. The idea is that η stores information about the initial UTxO set, which is made up of the individual UTxO sets U_i collected from the commit transactions, in order to verify head-status information later (see below).

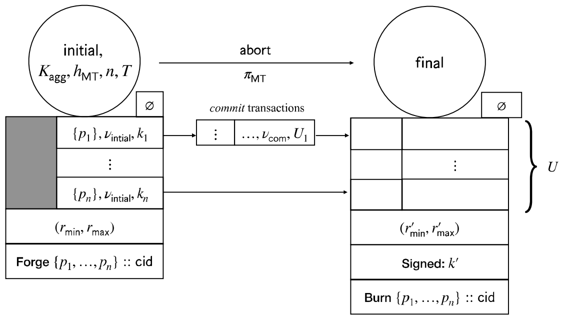
It is also required that all n participation tokens be present in the SM output of the *collectCom* transaction. This ensures that the *collectCom* transaction collects all n commit transactions. Note that since ν_{initial} does not allow an SM commit transaction to consume the outputs of the initial transaction, the only way to post the *collectCom* transaction is if each head member has posted a commit transaction.

Finally, note that the transition requires a proof π_{MT} that the signer k' is in the Merkle Tree belonging to h_{MT}, which ensures that only head members can post SM transactions. This will be the case for all transitions considered in this paper (and will not be pointed out any further).

Aborting a head. The *abort* transaction (see Fig. 6) allows a party to abort the creation of a head in case some parties fail to post a commit transaction. The final state does not contain any information (beyond its identifier), but it is ensured that (1) the outputs U correspond to the union of all committed UTxO sets U_i and (2) all participation tokens are burned.

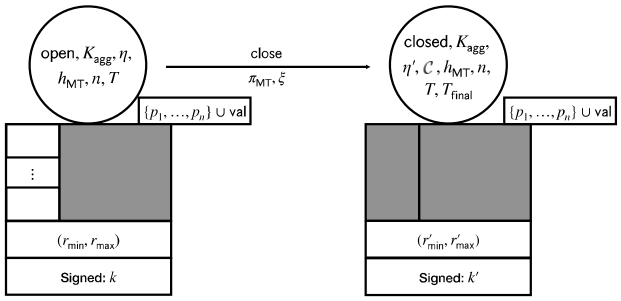
Close transaction. In order to close a head, a head member may post the *close* transaction (see Fig. 7), which results in a state transition from the open state to the closed state. For a successful close, a head member must provide valid information ξ about (their view of) the current head state. This information is passed through OCV algorithm Close, resulting in a new OCV status $\eta' \leftarrow \mathsf{Close}(K_{\mathsf{agg}}, \eta, \xi)$. OCV algorithm Close uses the previous OCV status η and K_{agg} to check the head information ξ. Note that if a check fails, Close may output $\perp$, but in order for a *close* transaction to be valid, $\eta' \neq \perp$ is required.

Once a *close* transaction has been posted, a *contestation period* begins which should last at least T slots. Hence, the last slot T_{final} of the contestation period is recorded in the state, and it is ensured that $T_{\mathsf{final}} \geq r'_{\mathsf{max}} + T$.



initial, K_{agg}, h_{MT}, n, T
abort
π_{MT}
final
∅
commit transactions
$\{p_1\}, \nu_{intial}, k_1$
$\ldots, \nu_{com}, U_1$
$\{p_n\}, \nu_{intial}, k_n$
(r_{min}, r_{max})
Forge $\{p_1, \ldots, p_n\}$ :: cid
(r'_{min}, r'_{max})
Signed: k'
Burn $\{p_1, \ldots, p_n\}$ :: cid
U

Figure 6: *initial* transaction (left) with *abort* transaction (right) and *commit* transactions (center).



open, $K_{agg}, \eta, h_{MT}, n, T$
close
π_{MT}, ξ
closed, $K_{agg}, \eta', \mathcal{C}, h_{MT}, n, T, T_{final}$
$\{p_1, \ldots, p_n\} \cup$ val
(r_{min}, r_{max})
Signed: k
(r'_{min}, r'_{max})
Signed: k'

Figure 7: *collectCom* transaction (left) with *close* transaction (right).

Finally, the SM state is extended by a set $\mathcal{C}$ initialized to the poster's signing key, i.e., $\mathcal{C} \leftarrow \{k'\}$. $\mathcal{C}$ is used to ensure that no party posts more than once during the contestation period.

Contestation. If the party first closing a head posts outdated/incomplete information about the current state of the head, any other party may post a *contest* transaction (see Fig. 8), which causes a state transition from the closed state to itself. The transition handles update information ξ by passing it through OCV algorithm Contest, resulting in a new OCV status $\eta' \leftarrow \mathsf{Contest}(K_{\mathsf{agg}}, \eta, \xi)$. OCV algorithm Contest uses the previous OCV status η and K_{agg} to check the update information ξ. Similarly to Close, Contest may output $\perp$, but in order for a *contest* transaction to be valid $\eta' \neq \perp$ is required.

The *contest* transaction is only valid if the old set $\mathcal{C}$ of parties who have contested (or closed) so far does not yet include the poster, i.e., $k' \notin \mathcal{C}$. If this check passes, the set is extended to include the poster of the *contest* transaction, i.e., $\mathcal{C}' \leftarrow \mathcal{C} \cup \{k'\}$. Furthermore, *contest* transactions may only be posted up until T_{final}, i.e., it is required that $r'_{\mathsf{max}} \leq T_{\mathsf{final}}$.

Observe that during the contestation period, up to $n-1$ *contest* transactions may be posted (of course, the parameter T has to be chosen large enough as to allow each head member to potentially post a *close*/*contest* transaction).

Final state. Once the contestation phase is over, a head may be finalized by posting a *fanout* transaction, taking the SM from closed to final. The *fanout* transaction must have outputs that correspond to the most recent head state. To that end, OCV predicate Final checks the transaction's

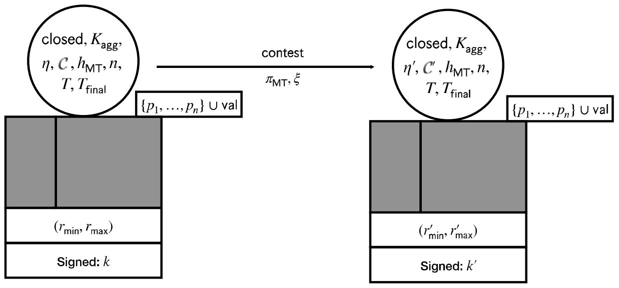

closed, K_{agg}, η, $\mathcal{C}$, h_{MT}, n, T, T_{final}
$\{p_1,\dots,p_n\} \cup$ val
(r_{min}, r_{max})
Signed: k
contest
π_{MT}, ξ
closed, K_{agg}, η', $\mathcal{C}'$, h_{MT}, n, T, T_{final}
$\{p_1,\dots,p_n\} \cup$ val
(r'_{min}, r'_{max})
Signed: k'

Figure 8: *close*/*contest* transaction (left); *contest* transaction (right)

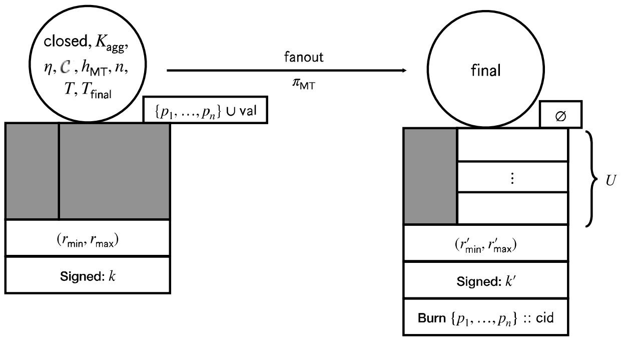

closed, K_{agg}, η, $\mathcal{C}$, h_{MT}, n, T, T_{final}
$\{p_1,\dots,p_n\} \cup$ val
(r_{min}, r_{max})
Signed: k
fanout
π_{MT}
final
$\varnothing$
$\vdots$
U
(r'_{min}, r'_{max})
Signed: k'
Burn $\{p_1,\dots,p_n\}$ :: cid

Figure 9: *close*/*contest* transaction (left); *fanout* transaction (right)

output set U against the information recorded in η. The *fanout* transaction is only valid if Final outputs `true`. Moreover, to ensure that the *fanout* transaction is not posted too early, $r'_{\mathsf{min}} > T_{\mathsf{final}}$ is required. Finally, all participation tokens must be burned.

6 Simple Head Protocol Without Conflict Resolution

This section describes the simplified version of the head protocol, and without conflict resolution, with the goal to demonstrate the protocol basics without overloading the presentation with too many details. Conflict resolution is added to the protocol in Appendix B, and the full protocol is sketched in Appendix C.

We first introduce a security definition for the head protocol in Section 6.1. The protocol machine is described in Section 6.2, the head-specific mainchain code in Section 6.3, and a security proof for the head protocol is given in Section 6.4.

6.1 Security definition

6.1.1 Protocol syntax

The head-protocol syntax is $\mathsf{HP} = (\mathsf{Prot}, \mathsf{Initial}, \mathsf{Close}, \mathsf{Contest}, \mathsf{Final})$. The main component is the protocol machine Prot, an instance of which is run by every head member. The other algorithms are used for setup and onchain verification and form the interface to the mainchain. In particular,

- $\Sigma \leftarrow$ generates global parameters,

- $(K_{\mathsf{ver}}, K_{\mathsf{sig}}) \leftarrow \mathsf{MS\text{-}KG}(\Sigma)$ allows every head member to generate fresh public/private key material based on the global parameters,
- $K_{\mathsf{agg}} \leftarrow \mathsf{MS\text{-}AVK}(\Sigma, (K_{\mathsf{ver},i})_i)$ aggregates public keys, and
- $\mathsf{Initial}$, Close, $\mathsf{Contest}$, and Final are onchain verification algorithms (cf. Section 5).

The head-protocol machine Prot has the following interface to the environment:

- input $(\mathtt{init}, i, \underline{K}_{\mathsf{ver}}, K_{\mathsf{sig}}, U_0)$ is used to initialize the head protocol, for the party with index i, with a vector of public-key material $\underline{K}_{\mathsf{ver}}$, private-key material K_{sig}, and an initial UTxO set U_0;
- input $(\mathtt{new}, \mathrm{tx})$ is used to submit a new transaction tx;
- output $(\mathtt{seen}, \mathrm{tx})$ announces that transaction tx has been seen (by the party outputting the message);
- output $(\mathtt{conf}, \mathrm{tx})$ announces that transaction tx has been confirmed (in the view of the party outputting the message);
- input $(\mathtt{close})$ is used to initiate head closure (produces a certificate ξ); and
- input $(\mathtt{cont}, \eta)$ is used to contest (produces a certificate ξ).

6.1.2 Protocol security

The security definition for the head protocol guarantees the following four, intuitively stated properties:

- CONSISTENCY: No two uncorrupted parties see conflicting transactions confirmed.
- LIVENESS: If all parties remain uncorrupted and the adversary delivers all messages, then every transaction becomes confirmed at some point.
- SOUNDNESS: The final UTxO set accepted on the mainchain results from a set of seen transactions.
- COMPLETENESS: All transactions observed as confirmed by an honest party at the end of the protocol are considered on the mainchain.

Experiment for security definition. The security properties above are captured by considering a random experiment that involves

- an adversary $\mathcal{A}$,
- a network under full scheduling control of $\mathcal{A}$, able to drop messages or delay them arbitrarily,
- a setup phase,
- n parties p_i, corruptible by $\mathcal{A}$, running the head protocol with the parameters from the setup phase and an initial UTxO set U_0 chosen by $\mathcal{A}$, and
- an abstract mainchain (mostly) controlled by $\mathcal{A}$.

The experiment ends once the mainchain state machine arrives in the final state, and the adversary wins if certain conditions are not satisfied at the end of the experiment.

In more detail, the experiment proceeds as follows:

1. Global parameters $\Sigma \leftarrow \mathsf{MS\text{-}Setup}$ are generated, and Σ is passed to $\mathcal{A}$.
2. For each party p_i, key material $(K_{\mathsf{ver},i}, K_{\mathsf{sig},i}) \leftarrow \mathsf{MS\text{-}KG}(\Sigma)$ is generated, and the vector $\underline{K}_{\mathsf{ver}}$ of all public keys and $K_{\mathsf{agg}} \leftarrow \mathsf{MS\text{-}AVK}(\Sigma, \underline{K}_{\mathsf{ver}})$ are passed to $\mathcal{A}$.
3. Each party p_i's protocol machine is initialized with $(\mathtt{init}, i, \underline{K}_{\mathsf{ver}}, K_{\mathsf{sig},i}, U_0)$, where U_0 is chosen by $\mathcal{A}$.
4. The adversary now gets to control inputs to parties (e.g., new transactions, close/contest requests) and sees outputs (e.g., seen and confirmed transactions). The following bookkeeping takes place:

   - when an uncorrupted party p_i outputs ξ upon $\mathtt{close}$ command, record $(\mathtt{close}, i, \xi)$;
   - when uncorrupted party p_i outputs ξ upon $(\mathtt{cont}, \eta)$ command, record $(\mathtt{cont}, i, \eta, \xi)$.

   In "parallel" to the above, the experiment sets $\mathcal{C}, H_{\mathsf{cont}} \leftarrow \emptyset$ and does the following to simulate the mainchain:

   (a) Initialize $\eta \leftarrow (U_0, 0, \emptyset)$.

   (b) When $\mathcal{A}$ supplies (i, ξ): if i is uncorrupted, ξ gets replaced by the ξ recorded in $(\mathtt{close}, i, \xi)$ and $H_{\mathsf{cont}} \leftarrow H_{\mathsf{cont}} \cup \{i\}$. Then, $\eta \leftarrow \mathsf{Close}(K_{\mathsf{agg}}, \eta, \xi)$ and $\mathcal{C} \leftarrow \mathcal{C} \cup \{i\}$ is computed. If Close rejects, everything in this step is discarded and the step repeated.

   (c) The adversary gets to repeatedly supply (i, ξ) for $i \notin \mathcal{C}$; if i is uncorrupted, ξ gets replaced by the ξ recorded in $(\mathtt{cont}, i, \xi)$ and $H_{\mathsf{cont}} \leftarrow H_{\mathsf{cont}} \cup \{i\}$. Then, $\eta \leftarrow \mathsf{Contest}(K_{\mathsf{agg}}, \eta, \xi)$ and $\mathcal{C} \leftarrow \mathcal{C} \cup \{i\}$ is computed. If $\mathsf{Contest}$ rejects, everything in this step is discarded.

   (d) When the adversary supplies U_{final}, $b \leftarrow \mathsf{Final}(\eta, U_{\mathsf{final}})$ is computed, and the experiment ends.

Our protocol gives different security guarantees depending on the level of adversarial corruption. It provides correctness independently of both, the number of corrupted head parties and the network conditions. But the guarantee that the protocol makes progress (i.e., that new transactions get confirmed in the head) is only provided in the case that no head parties are corrupted and that the network conditions are good.

To capture this difference, we distinguish:

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 $\mathtt{close}$ event. Apart from this restriction, the adversary can act arbitrarily in the above experiment.

Security events. Consider the following random variables:

- $\hat{S}_i$: the set of transactions tx for which party p_i, *while uncorrupted*, output $(\mathtt{seen}, \text{tx})$;
- $\overline{C}_i$: the set of transactions tx for which party p_i, *while uncorrupted*, output $(\mathtt{conf}, \text{tx})$;
- H_{cont}: the set of (at the time) uncorrupted parties who produced ξ upon close/contest request and ξ was applied to correct η (see above); and
- $\mathcal{H}$: the set of parties that remained uncorrupted.

The security of the head protocol is captured by considering the following events, each corresponding to one of the security properties introduced above:

- Consistency (Head): In presence of an active adversary, the following condition holds: For all i, j, $U_0 \circ (\overline{C}_i \cup \overline{C}_j) \neq \bot$, i.e., no two uncorrupted parties see conflicting transactions confirmed.
- Liveness (Head): In presence of a network adversary the following condition holds: For any transaction tx input via $(\mathtt{new}, \text{tx})$, the following eventually holds: $\text{tx} \in \bigcap_{i\in[n]} \overline{C}_i \ \vee\ \forall i : U_0 \circ (\overline{C}_i \cup \{\text{tx}\}) = \bot$, i.e., every party will observe the transaction confirmed or every party will observe the transaction in conflict with his confirmed transactions.[3]
- Soundness (Chain): In presence of an active adversary, the following condition is satisfied: $\exists \tilde{S} \subseteq \bigcap_{i\in\mathcal{H}} \hat{S}_i : U_{\mathsf{final}} = U_0 \circ \tilde{S}$, i.e., the final UTxO set results from a set of seen transactions.
- Completeness (Chain): In presence of an active adversary, the following condition holds: For $\tilde{S}$ as above, $\bigcup_{p_i\in H_{\mathsf{cont}}} \overline{C}_i \subseteq \tilde{S}$, i.e., all transactions seen as confirmed by an honest party at the end of the protocol are considered.

Note that our simplified protocol *with conflict resolution* and our full protocol in Appendices B and C achieve liveness in the above sense, but that our simplified protocol *without conflict resolution* in Section 6 only achieves a weaker notion of liveness, namely liveness in a

Conflict-Free Execution: Let $\mathcal{N} = \{\text{tx} \mid (\mathtt{new}, \text{tx})\}$ the set of all transactions input to a new event during the execution of the head protocol. A head-protocol execution is *conflict-free* iff $U_0 \circ \mathcal{N} \neq \bot$.

Respectively, the liveness aspect of the simplified protocol without conflict resolution is captured by the following event, instead:

- Conflict-Free Liveness (Head): In presence of a network adversary, a conflict-free execution satisfies the following condition: For any transaction tx input via $(\mathtt{new}, \text{tx})$, $\text{tx} \in \bigcap_{i\in[n]} \overline{C}_i$ eventually holds.

6.2 Protocol machine

The protocol machine Prot consists of a number of subroutines that handle inputs from the environment (e.g., the client command to issue a new transaction for confirmation, or the arrival of another party's confirmation request). The protocol is depicted in Figure 10. All relevant non-obvious notation is explained in the following paragraphs.

[3]In particular, *liveness* expresses that the protocol makes progress under reasonable network conditions if no head parties get corrupted – implying that, given any guaranteed upper bound δ on message delivery delay, the worst-case transaction-confirmation time is bounded in function of δ.

6.2.1 Local state representation

Every party maintains local objects to represent transactions, snapshots, and his local head-UTxO set. These objects exist in two versions, a *seen* object has been signed by the party (the party has seen and approved the event); and a *confirmed* object is associated with a valid multisignature (the party has received a valid multisignature on the object). A seen object X is denoted by $\hat{X}$ and a confirmed object by $\overline{X}$.

A party's local protocol state consists of the multisignature verification keys and its own signing key, of snapshot counters $\hat{s}$ and $\overline{s}$, and of variables

- $\hat{\mathcal{U}}$ and $\overline{\mathcal{U}}$, keeping track of the most recent seen resp. confirmed, snapshots,
- $\hat{\mathcal{L}}$ and $\overline{\mathcal{L}}$, keeping track of recent seen resp. confirmed UTxO sets, and
- $\hat{\mathcal{T}}$ and $\overline{\mathcal{T}}$, the sets of seen resp. confirmed, transactions that have not been considered by a snapshot yet.

Variables $\hat{\mathcal{U}}$ and $\overline{\mathcal{U}}$ store so-called *snapshot objects*, which are data structures keeping information about a snapshot. Specifically, a snapshot object $\mathcal{U}$ has the following structure:

$\mathcal{U}.s$	snapshot number
$\mathcal{U}.U$	corresponding UTxO set
$\mathcal{U}.h$	hash of U
$\mathcal{U}.T$	set of transactions relating this snapshot to its predecessor
$\mathcal{U}.S$	signature accumulator (array of signatures)
$\mathcal{U}.\tilde{\sigma}$	multisignature

The function $\mathsf{snObj}(s, U, T)$ initializes a snapshot object and is explained later.

Similarly, $\hat{\mathcal{T}}$ and $\overline{\mathcal{T}}$ store sets of *transaction objects*. A transaction object $\mathtt{tx}$ has the following structure:

$\mathtt{tx}.i$	index of the party issuing transaction for certification
$\mathtt{tx}.\mathrm{tx}$	transaction
$\mathtt{tx}.h$	hash of tx
$\mathtt{tx}.S$	signature accumulator (array of signatures)
$\mathtt{tx}.\tilde{\sigma}$	multisignature.

The function $\mathsf{txObj}(i, \mathrm{tx})$ initializes a transaction object by setting the appropriate fields to the passed values (including computing the hash of tx) and the remaining fields to $\emptyset$ resp. $\bot$.

6.2.2 Three-round entity confirmation

Transactions and snapshots are confirmed in an asynchronous 3-round process:[4]

- `req`: The issuer of a transaction or snapshot requests the entity to be signed by sending the entity description to every head member.

[4]Note that, as a variant, this 3-round process (with linear communication in n) can be condensed to 2 rounds (with quadratic communication in n) by combining the last two rounds into an "all-to-all" signature notification. This variant may be preferable for small n.

- `ack`: The head members acknowledge the entity be replying their signatures on the entity to the issuer.
- `conf`: The issuer collects all signatures, combines the multisignature, and sends the multisignature to all head members.

6.2.3 Code notation

Code is depicted by view of a generic head party p_i. We assume that a party only accepts messages authenticated by its claimed sender (by use of the authenticated channels established during the setup phase)—unauthenticated messages are simply treated as unseen by the recipient. For simplicity, whenever a party p_i sends a message to all head parties, it also sends the message to itself.

For the transaction set $\hat{\mathcal{T}}$ (and similarly $\overline{\mathcal{T}}$), $\hat{\mathcal{T}}[h]$ denotes $\mathtt{tx} \in \hat{\mathcal{T}}$ such that $\mathtt{tx}.h = h$, i.e., the transaction object corresponding to the transaction with hash $H(\mathrm{tx}) = h$.

The $\downarrow$ operator indicates the projection of an object onto a subset of its fields. For example, $\hat{\mathcal{T}}^{\downarrow(h)}$ denotes the set of hashes corresponding to the transactions in $\hat{\mathcal{T}}$.

The following notation is used to describe the application of transactions to a given UTxO set.

- $U' = U \circ \mathrm{tx}$ assigns to U' the UTxO set resulting from applying transaction tx to UTxO set U. In case that the validation fails it returns $U' = \bot$.
- $U' = U \circ T$ assigns to U' the UTxO set resulting from applying all transaction in the transaction set T to UTxO set U. In case that not all transactions can be applied it returns $U' = \bot$.

In the protocol routines of Fig. 10, by **require**(P), we express that predicate P must be satisfied for the further execution of a routine—while immediately terminated on $\neg P$. By **wait**(P) we express a non-blocking wait for predicate P to be satisfied. On $\neg P$, the execution of the routine is stopped, queued, and reactivated as soon as P is satisfied. Finally, we assume the code executions of each routine to be atomic—excluding the blocks of code that may be put into the wait queue for later execution, in which case we assume the wait block to be atomic.

6.2.4 Protocol flow

Initializing the head. Initially, by activation via the (`init`) event, the parties store their multisignature key material form the setup phase, ad set $\overline{\mathcal{L}} = \hat{\mathcal{L}} = \overline{\mathcal{U}} = \hat{\mathcal{U}} = U_0$ where U_0 is the initial UTxO set extracted from the η-state of the *collectCom* transaction (see Fig. 5). The initial transaction sets are empty, $\overline{\mathcal{T}} = \hat{\mathcal{T}} = \emptyset$, and $\bar{s} = \hat{s} = 0$.

Confirming new transactions.

(`new`). At any time, by calling ($\mathtt{new}, \mathrm{tx}$), a head party can (asynchronously) inject a new transaction tx to the head protocol—initiating a 3-round confirmation process for tx as described in Section 6.2.2. For this, the transaction must be well-formed ($\mathsf{valid\text{-}tx}$) and applicable to the current confirmed local UTxO state: $\overline{\mathcal{L}} \circ \mathrm{tx} \neq \bot$. If the checks pass, a ($\mathtt{reqTx}, \mathrm{tx}$) request is sent out to all parties.

Simplified Hydra Head Protocol Without Conflict Resolution

```
on $(\texttt{init}, i, \underline{K}_{\mathsf{ver}}, K_{\mathsf{sig}}, U_0)$ from client
    $\mathcal{V} \leftarrow \underline{K}_{\mathsf{ver}}$
    $\mathsf{avk} \leftarrow \mathsf{MS\text{-}AVK}(\mathcal{V})$
    $\mathsf{sk} \leftarrow K_{\mathsf{sig}}$
    $\hat{s}, \overline{s} \leftarrow 0$
    $\hat{\mathcal{U}}, \overline{\mathcal{U}} \leftarrow \mathsf{snObj}(0, U_0, \emptyset)$
    $\hat{\mathcal{L}}, \overline{\mathcal{L}} \leftarrow U_0$
    $\hat{\mathcal{T}}, \overline{\mathcal{T}} \leftarrow \emptyset$

on $(\texttt{new}, \mathrm{tx})$ from client
    require $\mathsf{valid\text{-}tx}(\mathrm{tx})$ and $\overline{\mathcal{L}} \circ \mathrm{tx} \neq \bot$
    multicast $(\texttt{reqTx}, \mathrm{tx})$

on $(\texttt{newSn})$ for $\mathsf{p}_i$
    require $\mathsf{leader}(\overline{s}+1) = i$ and $\hat{\mathcal{U}} = \overline{\mathcal{U}}$
    $T \leftarrow \big(\mathsf{maxTxos}(\overline{\mathcal{T}})\big)^{\downarrow(h)}$
    multicast $(\texttt{reqSn}, \overline{s}+1, T)$

on $(\texttt{close})$ from client
    return $(\overline{\mathcal{U}}.U, \overline{\mathcal{U}}.s, \overline{\mathcal{U}}.\tilde{\sigma}, \overline{\mathcal{T}}^{\downarrow(\mathrm{tx},\tilde{\sigma})})$

on $(\texttt{cont}, \eta)$ from client
    $(U_\eta, s_\eta, T_\eta) \leftarrow \eta$
    if $\overline{s} \leq s$
        $U \leftarrow U_\eta$
        $s \leftarrow s_\eta$
        $\tilde{\sigma} \leftarrow \varepsilon$
    else
        $U \leftarrow \overline{\mathcal{U}}.U$
        $s \leftarrow \overline{s}$
        $\tilde{\sigma} \leftarrow \overline{\mathcal{U}}.\tilde{\sigma}$
    $T' \leftarrow \mathsf{applicable}(U, \overline{\mathcal{T}}^{\downarrow(\mathrm{tx})} \cup T_\eta) \ \setminus \ T_\eta$
    if $U = U_\eta$
        $U \leftarrow \varepsilon$
    return
     $(U, s, \tilde{\sigma}, \{t \in \overline{\mathcal{T}}^{\downarrow(\mathrm{tx},\tilde{\sigma})} \mid t.\mathrm{tx} \in T'\})$
```

```
on $(\texttt{reqTx}, \mathrm{tx})$ from $\mathsf{p}_j$
    require $\mathsf{valid\text{-}tx}(\mathrm{tx}) \ \wedge \ \hat{\mathcal{L}} \circ \mathrm{tx} \neq \bot$
    wait $\overline{\mathcal{L}} \circ \mathrm{tx} \neq \bot$
        $h \leftarrow H(\mathrm{tx})$
        $\hat{\mathcal{T}}[h] \leftarrow \mathsf{txObj}(j, \mathrm{tx})$
        $\hat{\mathcal{L}} \leftarrow \hat{\mathcal{L}} \circ \mathrm{tx}$
        output $(\texttt{seen}, \mathrm{tx})$
        $\sigma_i \leftarrow \mathsf{MS\text{-}Sign}(\mathsf{sk}, h)$
        send $(\texttt{ackTx}, h, \sigma_i)$ to $\mathsf{p}_j$

on $(\texttt{ackTx}, h, \sigma_j)$ from $\mathsf{p}_j$
    require $\hat{\mathcal{T}}[h].i = i$
    require $\hat{\mathcal{T}}[h].S[j] = \varepsilon$
    $\hat{\mathcal{T}}[h].S[j] \leftarrow \sigma_j$
    if $\forall k : \hat{\mathcal{T}}[h].S[k] \neq \varepsilon$
        $\tilde{\sigma} \leftarrow \mathsf{MS\text{-}ASig}(h, \mathcal{V}, \hat{\mathcal{T}}[h].S)$
        if $\tilde{\sigma} \neq \bot$
            multicast $(\texttt{confTx}, h, \tilde{\sigma})$

on $(\texttt{confTx}, h, \tilde{\sigma})$ from $\mathsf{p}_j$
    if $\mathsf{MS\text{-}Verify}(\mathsf{avk}, h, \tilde{\sigma})$
        $\mathrm{tx} \leftarrow \hat{\mathcal{T}}[h].\mathrm{tx}$
        $\overline{\mathcal{L}} \leftarrow \overline{\mathcal{L}} \circ \mathrm{tx}$
        $\hat{\mathcal{T}}[h].\tilde{\sigma} \leftarrow \tilde{\sigma}$
        $\overline{\mathcal{T}}[h] \leftarrow \hat{\mathcal{T}}[h]$
        $\hat{\mathcal{T}} \leftarrow \hat{\mathcal{T}} \setminus \hat{\mathcal{T}}[h]$
        output $(\texttt{conf}, \mathrm{tx})$

on $(\texttt{reqSn}, s, T)$ from $\mathsf{p}_j$
    require $s = \overline{s}+1$ and $\mathsf{leader}(s) = j$
    wait $\overline{s} = \hat{s}$ and $T \subseteq \overline{\mathcal{T}}^{\downarrow(h)}$
        $\hat{s} \leftarrow \hat{s}+1$
        $\hat{\mathcal{U}} \leftarrow \mathsf{snObj}(\hat{s}, \overline{\mathcal{U}}.U, T)$
        $\sigma_i \leftarrow \mathsf{MS\text{-}Sign}(\mathsf{sk}, \hat{\mathcal{U}}.h \| \hat{s})$
        send $(\texttt{ackSn}, \hat{s}, \sigma_i)$ to $\mathsf{p}_j$

on $(\texttt{ackSn}, s, \sigma_j)$ from $\mathsf{p}_j$
    require $s = \hat{s}$ and $\mathsf{leader}(s) = i$
    require $\hat{\mathcal{U}}.S[j] = \varepsilon$
    $\hat{\mathcal{U}}.S[j] \leftarrow \sigma_j$
    if $\forall k : \hat{\mathcal{U}}.S[k] \neq \varepsilon$
        $\tilde{\sigma} \leftarrow \mathsf{MS\text{-}ASig}(\hat{\mathcal{U}}.h \| s, \mathcal{V}, \hat{\mathcal{U}}.S)$
        if $\tilde{\sigma} \neq \bot$
            multicast $(\texttt{confSn}, s, \tilde{\sigma})$

on $(\texttt{confSn}, s, \tilde{\sigma})$ from $\mathsf{p}_j$
    require $s = \hat{s} \neq \overline{s}$
    if $\mathsf{MS\text{-}Verify}(\mathsf{avk}, \hat{\mathcal{U}}.h \| \hat{s}, \tilde{\sigma})$
        $\overline{s} \leftarrow s$
        $\hat{\mathcal{U}}.\tilde{\sigma} \leftarrow \tilde{\sigma}$
        $\overline{\mathcal{U}} \leftarrow \hat{\mathcal{U}}$
        $\overline{\mathcal{T}} \leftarrow \overline{\mathcal{T}} \setminus \mathsf{Reach}^{\overline{\mathcal{T}}}(\overline{\mathcal{U}}.T)$
```

Figure 10: Head-protocol machine for the *simple protocol without conflict resolution* from the perspective of party p_i.

$(\texttt{reqTx})$. Upon receiving request $(\texttt{reqTx}, \mathrm{tx})$, a signature is only issued by a party p_i if tx applies to his local *seen* UTxO state: $\hat{\mathcal{L}} \circ \mathrm{tx} \neq \bot$. If this is the case, the party waits until his *confirmed* UTxO state $\overline{\mathcal{L}}$ has "caught up": the signature is only delivered as soon as $\overline{\mathcal{L}} \circ \mathrm{tx} \neq \bot$, i.e., a transaction is only signed once it is applicable to the local confirmed state.

In case the preconditions are satisfied, a respective transaction object is allocated, initialized, and added to $\hat{\mathcal{T}}$; $\hat{\mathcal{L}}$ is updated by applying tx, and $(\texttt{seen}, \mathrm{tx})$ is output; and, finally, a signature on the hash of tx, $\sigma = \mathsf{MS\text{-}Sign}(H(\mathrm{tx}))$, is delivered back to the transaction issuer by replying with an $(\texttt{ackTx}, H(\mathrm{tx}), \sigma)$.

(ackTx). Upon receiving acknowledgment $(\texttt{ackTx}, h, \sigma_j)$, the transaction issuer stores the received signature in the respective transaction object. If a signature from each party has been collected, p_i computes the multisignature $\tilde{\sigma}$ and, if valid, sends it to all parties in a $(\texttt{confTx}, h, \tilde{\sigma})$ message.

(confTx). Upon receiving confirmation $(\texttt{confTx}, h, \tilde{\sigma})$ from the transaction issuer, containing a valid multisignature, the multisignature is stored in the respective transaction object, $\overline{\mathcal{L}}$ is updated by applying tx, and the transaction object is moved from $\hat{\mathcal{T}}$ to $\overline{\mathcal{T}}$. Finally, $(\texttt{conf}, \mathrm{tx})$ is output.

Creating snapshots. In parallel to confirming transactions, parties generate snapshots in a strictly sequential round-robin manner. We call the party responsible for issuing the i^{th} snapshot the *leader* of the i^{th} snapshot. The issuance frequency of the snapshots tunes a tradeoff between the transaction space that has to maintained by the parties for storing confirmed but snapshot-unprocessed transactions against the snapshot-communication overhead in the head protocol. As the information to be exchanged among the parties for snapshot confirmation is small, such snapshots can in principle be greedily issued as soon as the next snapshot leader sees a new confirmed transaction.

(newSn). On activation by $(\texttt{newSn})$, the snapshot leader verifies whether $\hat{\mathcal{U}} = \overline{\mathcal{U}}$ to ensure that he is not already in the process of snapshot creation. The leader p_i then announces the transaction set $\overline{\mathcal{T}}$, the not yet snapshot-processed confirmed transactions to be applied to compute the new snapshot. However, to reduce communication overhead, only the hashes of the *maximal* transactions of $\overline{\mathcal{T}}$ are announced which are the transactions of $\overline{\mathcal{T}}$ not referenced by another transaction in $\overline{\mathcal{T}}$. This maximal set is computed by function $T = \mathsf{maxTxos}(\overline{\mathcal{T}})^{\downarrow(h)}$. Finally the leader sends $(\texttt{reqSn}, \bar{s}+1, T)$ to all parties.

(reqSn). Upon receiving request $(\texttt{reqSn}, s, T)$, party p_i checks that s is the next snapshot number and that p_j is responsible for leading its creation. Party p_i then waits until the previous snapshot is confirmed ($\bar{s} = \hat{s}$) and all transactions referred in T are confirmed.

Only then, p_i increments his seen-snapshot counter $\hat{s}$, and allocates a new snapshot object calling function snObj that performs the following steps:

1. It reconstructs the transaction set to be applied to the latest confirmed snapshot by calling function $\mathsf{Reach}^{\overline{\mathcal{T}}}(T)$ that computes all transactions in $\overline{\mathcal{T}}$ reachable from the transactions (with hashes) in T by following the output references (the inverse of $\mathsf{maxTxos}$); and

2. computes the UTxO set of the new snapshot as $\hat{\mathcal{U}}.U \leftarrow \overline{\mathcal{U}}.U \circ \mathsf{Reach}^{\overline{\mathcal{T}}}(T)$, and

3. computes the hash of $\hat{\mathcal{U}}.U$ and sets the fields for the snapshot number and the maximal transactions applied.

Finally, p_i computes a signature $\sigma_i = \mathsf{MS\text{-}Sign}(\mathsf{sk}, H(\hat{\mathcal{U}}) \| \hat{s})$, and replies to p_j the message $(\mathtt{ackSn}, \overline{s}, \sigma_i)$.[5]

$(\mathtt{ackSn})$. Upon receiving acknowledgment $(\mathtt{ackSn}, s, \sigma_j)$, the snapshot leader stores the received signature in the respective snapshot object. If a signature from each party has been collected, p_i computes the multisignature $\tilde{\sigma}$ and, if valid, sends it to all parties in a $(\mathtt{confSn}, h, \tilde{\sigma})$ message.

$(\mathtt{confSn})$. Upon receiving confirmation $(\mathtt{confSn}, s, \tilde{\sigma})$ from the snapshot leader, containing a valid multisignature, p_i stores the multisignature and updates $\overline{s} = s$ and $\overline{\mathcal{U}} = \hat{\mathcal{U}}$. Finally, the set of confirmed transactions is reduced by excluding the transactions that have been processed by $\overline{\mathcal{U}}$: $\overline{\mathcal{T}} \leftarrow \overline{\mathcal{T}} \setminus \mathsf{Reach}^{\overline{\mathcal{T}}}(\overline{\mathcal{U}}.T)$.

Closing the head.

$(\mathtt{close})$. In order to close a head, a party causes the $(\mathtt{close})$ event which returns the latest confirmed snapshot $\overline{\mathcal{U}}.U$, snapshot number $\overline{\mathcal{U}}.s$ and the respective multisignature $\overline{\mathcal{U}}.\tilde{\sigma}$, together with the remaining confirmed transactions $\overline{\mathcal{T}}^{\downarrow(\mathrm{tx},\tilde{\sigma})}$ (multisigned). These items form the certificate ξ to be posted onchain (see Section 6.3.2).

$(\mathtt{cont})$. In order to contest the current state closed on the mainchain, a party causes the $(\mathtt{cont}, \eta)$ event with input η being the latest observed head status that has been aggregated onchain for this head so far (by a sequence of *close* and *contest* transactions).

The algorithm then computes "differential" data between the current onchain head status and the contester's confirmed view: the latest confirmed snapshot (if newer than seen onchain) and the set of confirmed transactions (in his view) not yet considered by the current state η. These items form the certificate ξ to be posted onchain (see Section 6.3.2).

We only want to pass on the (multisigned) transactions in $\overline{\mathcal{T}}^{\downarrow(\mathrm{tx})} \setminus T_\eta$ that have not yet been processed by the latest snapshot U. This is achieved by applying function $\mathsf{applicable}$ that tests, for each transaction in $\mathrm{tx} \in \overline{\mathcal{T}}^{\downarrow(\mathrm{tx})} \cup T_\eta$ in appropriate order, whether $U \circ \mathrm{tx} \neq \perp$ is still applicable. Note that the transactions in T_η have to be considered in this process as some transactions in $\overline{\mathcal{T}}$ may directly depend on them, and would otherwise not be detected to be applicable. As we only want to extract "differential" data, the transactions in T_η are finally removed again as they are already recorded in the (accumulative) η state.

6.3 Head-specific mainchain functionality

On an abstract level, as described in Section 5, mainchain and head functionality are clearly separated into events that happen onchain and events that happen in the head. In particular, network participants that are not members of the head protocol only observe mainchain events.

Still, depending on the concrete implementation of the head certification process (which our abstract description of the mainchain functionality is agnostic of), some mainchain functionality has to be adapted to the specific choice made for the head protocol. This concerns two aspects:

[5]Note that no UTxO sets have to be exchanged in this process as the parties can locally compute a new snapshot by the given transaction hashes.

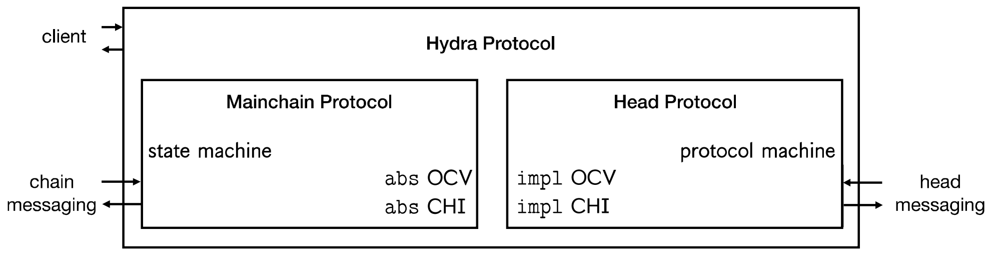


Figure 11: Hydra protocol components.

Onchain verification (OCV). The confirmation of head events by means of the multisignature scheme must be verifiable onchain; and thus, the exact workings of head certification must be known to the mainchain protocol. Now, given that the onchain portion of the mainchain protocol (i.e., state machine transition validation) is realized by EUTxO validator scripts, these scripts utilize the abstract interface of the OCV. Hence, we interpret the head OCV code as implementing the abstract mainchain OCV specification for all network participants (Fig. 11).

Chain/head interaction (CHI). Upon observing certain onchain events, a head member's mainchain functionality must interact with the head protocol. For instance, this is the case, when a head member observes the closing of the head on the mainchain. The mainchain functionality must then query the head protocol to know whether a *contest* transaction must be posted.

6.3.1 Onchain verification (OCV)

Recall that the mainchain functionality was generically described it terms of η, the latest head state as known onchain, and ξ, a certificate posted by a head member to update η by delivering head-confirmed information.

We shortly recapitulate the abstract workings of OCV. After the processing of the *collectCom* transaction, the initial UTxO set is stored as η in association of the open state. Later, a party p_i can

- produce a certificate ξ accepted by $\mathsf{Close}(\cdot)$ to close out the head and make (his view of) the current head-UTxO set available on the mainchain, and,
- given the current state η on the mainchain, produce a certificate ξ accepted by $\mathsf{Contest}(\eta, \cdot)$ to contest a closure and supply an updated view of the head-UTxO set to the mainchain.

Finally, the function Final checks the UTxO set in the transaction that moves the state machine into its final state against the information stored in η.

We now instantiate the respective onchain verification (OCV) functionality for the head protocol given in this section—with its specific way of certifying head states (see Fig. 12).

$\mathsf{Initial}$. The entire initial UTxO set U_0 is composed from the n committed UTxO sets $U_{\mathsf{p}_1}, \ldots, U_{\mathsf{p}_n}$, and returned as $\eta = (U_0, 0, \emptyset)$.

Algorithms for Onchain Verification

```
Initial $(U_{p_1}, \dots, U_{p_n})$
    return $(U_{p_1} \cup \dots \cup U_{p_n}, 0, \emptyset)$

Close $(K_{\mathsf{agg}}, \eta, \xi)$
    $(U, s, \tilde{\sigma}, T) \leftarrow \xi$
    if $\exists(\mathrm{tx}_i, \tilde{\sigma}_i) \in T :$
      $\neg\mathsf{MS\text{-}AVerify}(K_{\mathsf{agg}}, H(\mathrm{tx}_i), \tilde{\sigma}_i)$
        return $\bot$
    if $s = 0$
        $(U, \cdot, \cdot) \leftarrow \eta$
    else if $\neg\mathsf{MS\text{-}AVerify}(K_{\mathsf{agg}}, H(U)\|s, \tilde{\sigma})$
        return $\bot$
    if $U \circ T^{\downarrow(\mathrm{tx})} = \bot$
        return $\bot$
    return $(U, s, T^{\downarrow(\mathrm{tx})})$

Contest $(K_{\mathsf{agg}}, \eta, \xi)$
    $(U_\eta, s_\eta, T_\eta) \leftarrow \eta$
    $(U, s, \tilde{\sigma}, T) \leftarrow \xi$
    if $\exists(\mathrm{tx}_i, \tilde{\sigma}_i) \in T : \neg\mathsf{MS\text{-}AVerify}(K_{\mathsf{agg}}, H(\mathrm{tx}_i), \tilde{\sigma}_i)$
        return $\bot$
    if $s \leq s_\eta$
        $U_N \leftarrow U_\eta$
    else
        $U_N \leftarrow U$
        if $\neg\mathsf{MS\text{-}AVerify}(K_{\mathsf{agg}}, H(U)\|s, \tilde{\sigma})$ return $\bot$
        $T_\eta \leftarrow \mathsf{applicable}(U_N, T_\eta)$
    if $U_N \circ (T_\eta \cup T^{\downarrow(\mathrm{tx})}) = \bot$
        return $\bot$
    return $(U_N, s, T_\eta \cup T^{\downarrow(\mathrm{tx})})$

Final $(\eta, U)$
    $(U_\eta, s_\eta, T_\eta) \leftarrow \eta$
    return $(U = U_\eta \circ T_\eta)$
```

Figure 12: The algorithms used by the state machine for onchain verification.

Close. The state machine uses the onchain verification (OCV) algorithm Close to verify the information submitted by the party.

Recall that, when a p_i receives the `close` command, it simply outputs as certificate the snapshot number, the UTxO set, and the multisignatures corresponding to the most recent confirmed snapshot $\overline{\mathcal{U}}$ as well as all confirmed transactions in $\overline{\mathcal{T}}$ which have not yet been considered in $\overline{\mathcal{U}}$, along with the corresponding multisignatures.

OCV function Close (see Figure 12) verifies all multisignatures in $\xi = (U, s, \tilde{\sigma}, T)$, i.e., those of $H(U)\|s$ and the transactions in T, and ensures that the transactions in T can be applied to U (or, in case of $s = 0$, to U_0). The algorithm then outputs the new state $\eta' = (U, s, T^{\downarrow(\mathrm{tx})})$.

Contest. The state machine uses the OCV algorithm Contest to verify the "differential" data submitted by a contesting party.

Recall that, when a p_i receives the command $(\mathtt{cont}, \eta)$ for $\eta = (U_\eta, s_\eta, T_\eta)$, he supplies his latest snapshot $\overline{\mathcal{U}}.U$ if it is newer than U_η, and those confirmed transactions that have not yet been considered by the latest snapshot. In case that U_η is newer than the own snapshot, the transactions yet to be delivered can be found by trying to apply them (together with T_η) to U_η—as those already considered by U_η can no longer be applied; this computation is performed by function applicable.

Similarly to the close case, OCV function Contest, given $\xi = (U, s, \tilde{\sigma}, T)$, first checks all signatures.

In case that the provided snapshot U is newer than the snapshot U_η from the onchain state η, the set T_η is reduced to those transactions that are still applicable to the newer of both snapshots, U_N.

Finally, it is ensured that $T_\eta \cup T^{\downarrow(\mathrm{tx})}$ can be applied to the newest of both snapshots, and the

Chain/Head Interaction

```
on $(\texttt{clientTx}, \mathrm{tx})$
    $\mathsf{head}.(\texttt{new}, \mathrm{tx})$

on $(\texttt{clientClose})$
    $\xi \leftarrow \mathsf{head}.(\texttt{close})$
    $\mathsf{chain.postTx}(\mathit{close}, \xi)$

on $(\texttt{chainInitial})$
    require $K_{\mathsf{agg}}^{\mathsf{chain}} = K_{\mathsf{agg}}^{\mathsf{setup}}$
    require $h_{\mathsf{MT}} = H_{\mathsf{Merkle}}(\underline{k}_{\mathsf{ver}})$
    $\mathsf{chain.postTx}(\mathit{commit}, U)$

on $(\texttt{chainInitialTimeOut})$
    if (all members committed)
        $\mathsf{chain.postTx}(\mathit{collectCom})$
    else
        $\mathsf{chain.postTx}(\mathit{abort})$

on $(\texttt{chainCollectCom})$
    $(U_0, \cdot, \cdot) \leftarrow \mathsf{Initial}(U_{p_1}, \dots, U_{p_n})$
    $\mathsf{head}.(\texttt{init}, i, \underline{K}_{\mathsf{ver}}, K_{\mathsf{sig},i}, U_0)$

on $(\texttt{chainClose})$
    $\eta' = (U', s', T') \leftarrow \mathsf{chain.Close}(K_{\mathsf{agg}}, \eta, \xi)$
    $\xi = (U, s, \tilde{\sigma}, T) \leftarrow \mathsf{head}.(\texttt{cont}, \eta')$
    if $s > s' \ \vee \ T \neq \emptyset$
        $\mathsf{chain.postTx}(\mathit{contest}, \xi)$

on $(\texttt{chainContest})$
    $\eta' = (U', s', T') \leftarrow \mathsf{chain.Contest}(K_{\mathsf{agg}}, \eta, \xi)$
    $\xi = (U, s, \tilde{\sigma}, T) \leftarrow \mathsf{head}.(\texttt{cont}, \eta')$
    if $s > s' \ \vee \ T \neq \emptyset$
        $\mathsf{chain.postTx}(\mathit{contest}, \xi)$

on $(\texttt{chainClosedTimeOut})$
    $\mathsf{chain.postTx}(\mathit{fanout})$
```

Figure 13: Chain/head interaction: Additional mainchain actions for head members.

new (aggregate) state $\eta' = (U_N, s, T_\eta \cup T^{\downarrow(\mathrm{tx})})$ is output.

Final. Given $\eta = (U_\eta, s_\eta, T_\eta)$ and U, Final checks that $U = U_\eta \circ T_\eta$.

6.3.2 Chain/head interaction (CHI)

In Fig. 13, we summarize that part of the Hydra mainchain functionality that interacts with the head member (client) and the head protocol.

Routine `clientTx` handles the client's request to issue a head transaction by delegating the request to the head protocol. Routine `clientClose` handles the client's request to close the head. It gathers a certificate for the current local state from the head protocol, and posts this certificate onchain.

Routine `chainInitial` gets triggered on seeing the head's *initial* transaction onchain. It verifies the parameters recorded in the *initial* transaction against the parameters gathered during the setup phase described in Section 4: in particular, the aggregate multisignature key must match, and h_{MT} must be the Merkle-tree hash of the gathered verification keys $\underline{k}_{\mathsf{ver}}$. If successful, the client's UTxO set is committed onchain.

Routine `chainInitialTimeOut` gets triggered once the initial commit period has expired. It then either posts a *collectCom* transaction containing all committed UTxO sets—in case that all head members committed a UTxO set—or an *abort* transaction otherwise.

Routine `chainCollectCom` gets triggered on seeing the head's *collectCom* transaction onchain. It computes, into U_0, the set of committed UTxOs, and initializes the head protocol.

Routines `chainClose` and `chainContest` get triggered by observing the head's *close* and *contest* transactions, respectively. They compare the latest onchain state η to the party's own head state by calling the head protocol's `cont` function to obtain a certificate ξ for a differential onchain update to represent the portions of the local state not yet considered by η. If necessary, a corresponding *contest* transaction is posted onchain.

Routine `chainClosedTimeOut` gets triggered once the contestation period has expired. It then posts a *fanout* transaction containing the final UTxO set.

6.4 Security proof

This section proves that the head protocol presented in Section 6 satisfies CONSISTENCY, CONFLICT-FREE LIVENESS, SOUNDNESS, and COMPLETENESS. The proof proceeds by establishing several invariants that facilitate proving these properties. Throughout the proof, the assumption is made that at most $n-1$ head members are corrupted. Moreover, assume no signatures are forged and no hashes collide; these events occur with negligible probability only. Consider the following random variables:

- SN_j: the UTxO set corresponding to jth snapshot, i.e., the set that gets the jth multisignature on snapshots ($\mathrm{SN}_0 = U_0$);
- $\tilde{T}_j$: the transaction set corresponding to SN_j, formally defined via $\tilde{T}_0 = \emptyset$, and $\tilde{T}_j := \tilde{T}_{j-1} \circ \mathsf{Reach}^{\overline{T}}(T)$ where T is the set proposed in $(\texttt{reqSn}, j, T)$;
- C_{chain}: keeps track of "transactions on chain" and is defined as follows: upon (successful) close resp. contest with ξ for η, let $C_{\mathsf{chain}} \leftarrow \tilde{T}_s \cup T$, where $(\cdot, s, T)$ is the output of $\mathsf{Close}(K_{\mathsf{agg}}, \eta, \xi)$ resp. $\mathsf{Contest}(K_{\mathsf{agg}}, \eta, \xi)$;
- $\mathrm{SN}_{\mathsf{cur},i}$: latest confirmed snapshot as seen by party p_i.

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

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

Invariant 1. *Consider a conflict-free execution of the basic head protocol in presence of a network adversary. Then, for any transaction* tx *input to the protocol via* (`new`) *the following holds with respect to any parties p_i and p_j:*

$$\forall t_0 : \ \overline{\mathcal{L}}_i^{(t_0)} \circ \mathrm{tx} \neq \bot \Rightarrow \exists T \geq t_0 \forall t \geq T : \ \overline{\mathcal{L}}_j^{(t)} \circ \mathrm{tx} \neq \bot \ \vee \ \mathrm{tx} \in \overline{C}_j^{(t)}$$

where the superscript $\cdot^{(t)}$ indicates the time when the respective variable is evaluated.

Proof. Assume that party p_i sees tx at time t_0 and $\overline{\mathcal{L}}_i^{(t_0)} \circ \mathrm{tx} \neq \bot$. By conflict-freeness and full delivery we get that, eventually, each party p_j holds $\overline{C}_j^{(t)} \supseteq \overline{C}_i^{(t_0)}$. By this time t, either $\overline{\mathcal{L}}_j^{(t)} \circ \mathrm{tx} \neq \bot$ or $\mathrm{tx} \in \overline{C}_j^{(t)}$ (as we have conflict-freeness, and $\overline{C}_j^{(t)} \subseteq \mathcal{N}$). □

Lemma 2 (Conflict-Free Liveness). *The basic head protocol achieves* CONFLICT-FREE LIVENESS.

Proof. We demonstrate that a transaction tx issued by a player p_i will eventually be confirmed by every player p_j. By conflict-freeness, in $(\texttt{new}, \mathrm{tx})$ we have that $\overline{\mathcal{L}}_i \circ \mathrm{tx} \neq \bot$.

Assume that $\mathrm{tx} \notin \overline{C}_j$, i.e., that p_j has not seen tx confirmed yet. As soon as p_j enters (or gets reactivated from the wait queue) $(\texttt{reqTx}, \mathrm{tx})$ under the condition $\overline{\mathcal{L}}_j \circ \mathrm{tx} \neq \bot$ (eventually guaranteed by Invariant 1), by conflict-freeness, also $\hat{\mathcal{L}}_j \circ \mathrm{tx} \neq \bot$ holds, and p_j acknowledges the transaction. Thus, every p_j eventually acknowledges the transaction, and $\mathrm{tx} \in \cap_{i \in [n]} \overline{C}_i$. □

Invariant 2. *Consider an arbitrary uncorrupted party* p_i*. Let* $\tilde{T}$ *be the set corresponding to* $\mathrm{SN}_{\mathsf{cur},i}$*. Then,* $\tilde{T} \cup \overline{\mathcal{T}}_i = \overline{C}_i$*, where* $\overline{\mathcal{T}}_i$ *is the set* $\overline{\mathcal{T}}$ *of* p_i*.*

Proof. Observe that the invariant is trivially satisfied at the onset of the protocol's execution. Furthermore, each time a new transaction is confirmed via $\texttt{confTx}$, both $\overline{\mathcal{T}}_i$ and $\overline{C}_i$ grow by the newly confirmed transaction, while $\tilde{T}$ remains unchanged.

The only other time one of the sets $\tilde{T}$, $\overline{\mathcal{T}}_i$, or $\overline{C}_i$ change is when a new snapshot is confirmed via $\texttt{confSn}$. In such a case, note that $\overline{C}_i$ stays the same while any transaction removed from $\overline{\mathcal{T}}_i$ is considered by the new snapshot and thus added to $\tilde{T}$. Hence, the invariant is still satisfied. □

Invariant 3. $\tilde{T}_0 \subseteq \tilde{T}_1 \subseteq \tilde{T}_2 \subseteq \ldots$*.*

Proof. Let p_i be an honest party. It is easily seen that the set of transactions considered by a new snapshot always includes the set considered by the previous snapshot since the set of transactions T in a $\texttt{reqSn}$ satisfies that $\mathrm{SN}_{\mathsf{cur},i} \circ \mathsf{Reach}^{\overline{\mathcal{T}}_i}(T) \neq \bot$, (this is implied by Invariant 2). □

Invariant 4. C_{chain} *grows monotonically (w.r.t.* $\subseteq$*).*

Proof. Consider operation $\mathsf{Contest}(K_{\mathsf{agg}}, \eta, \xi)$ and let $\eta = (U_\eta, s_\eta, T_\eta)$ and $\xi = (U, s, \tilde{\sigma}, T)$. Note that before the operation $C_{\mathsf{chain}} = \tilde{T}_{s_\eta} \cup T_\eta$. Consider now the set T^* in the output $(\cdot, \cdot, T^*)$ of $\mathsf{Contest}$. Note that after the operation $C_{\mathsf{chain}} = \tilde{T}_s \cup T^*$. Observe that:

- Since $s \geq s_\eta$, Invariant 3 implies that a transaction $\mathrm{tx} \in \tilde{T}_{s_\eta}$ is also in $\tilde{T}_s$.
- If a transaction $\mathrm{tx} \in T_\eta$ is not in T^*, then $s > s_\eta$ and the transaction is consumed by the snapshot with number s, i.e., $\mathrm{tx} \in \tilde{T}_s$.

Hence, C_{chain} grows monotonically. □

Invariant 5. *For all* $i \in H_{\mathsf{cont}}$*,* $\overline{C}_i \subseteq C_{\mathsf{chain}}$*.*

Proof. Take any honest party p_i and let $\tilde{s}$ be the current snapshot number at p_i, i.e., $\mathrm{SN}_{\mathsf{cur},i} = \tilde{T}_{\tilde{s}}$. Recall that, by Invariant 2, $\overline{C}_i = \tilde{T}_{\tilde{s}} \cup \overline{\mathcal{T}}_i$. Consider a close or contest operation by p_i as well as the output (U, s, T^*) of $\mathsf{Contest}$, and observe that after the operation $C_{\mathsf{chain}} = \tilde{T}_s \cup T^*$. By Invariant 3, $\tilde{T}_{\tilde{s}} \subseteq \tilde{T}_s$ and, by a similar argument as in the proof of Invariant 4, if $\mathrm{tx} \in \overline{\mathcal{T}}_i$ is not in T^*, it must be in $\tilde{T}_s$. Hence, $\overline{C}_i \subseteq C_{\mathsf{chain}}$. Furthermore, since C_{chain} grows monotonically (Invariant 4), the invariant remains satisfied. □

Invariant 6. *For all uncorrupted parties* p_i*,* $\bigcup_{j \in [n]} \overline{C}_j \subseteq \hat{S}_i$*.*

Proof. Honest parties will only output $(\texttt{conf}, \mathrm{tx})$ if there exists a valid multisignature for tx, which implies that each honest party output $(\texttt{seen}, \mathrm{tx})$ just before they signed tx. □

Invariant 7. *For any* j*,* $\tilde{T}_j \subseteq \bigcap_{i \in \mathcal{H}} \overline{C}_i$*.*

Proof. Only transactions that have been seen as confirmed by all honest parties can ever be included in a confirmed snapshot. □

Invariant 8. $C_{\mathsf{chain}} \subseteq \bigcap_{i\in\mathcal{H}} \hat{S}_i$.

Proof. Let $\eta = (U, s, T)$. Observe that $C_{\mathsf{chain}} = \tilde{T}_s \cup T$. Consider a transaction $\text{tx} \in C_{\mathsf{chain}}$.

- If $\text{tx} \in \tilde{T}_s$, then $\text{tx} \in \bigcap_{i\in\mathcal{H}} \overline{C}_i \subseteq \bigcap_{i\in\mathcal{H}} \hat{S}_i$ by Invariants 7 and 6.
- If $\text{tx} \in T$, then $\text{tx} \in \bigcap_{i\in\mathcal{H}} \hat{S}_i$ since no transaction can be confirmed without being seen by all honest parties.

□

Lemma 3 (Soundness). *The basic head protocol satisfies the* Soundness *property.*

Proof. Let $\eta = (U, s, T)$ be the value of η just before applying $\mathsf{Final}(\eta, U_{\mathsf{final}})$. Clearly, the only set U_{final} that will be accepted by Final is $U_0 \circ (\tilde{T}_s \cup T)$. By definition $\tilde{T}_s \cup T = C_{\mathsf{chain}}$. Soundness now follows from Invariant 8. □

Lemma 4 (Completeness). *The basic head protocol satisfies the* Completeness *property.*

Proof. Follows from Invariant 5 and an argument similar to that in the proof of Lemma 3. □

7 Experimental Evaluation

We will now investigate the performance of the Hydra protocol in terms of both latency (transaction settlement time) and throughput (rate of transaction processing, TPS), using timing-accurate simulations. The simulations will demonstrate that Hydra is optimal in achieving fast transaction settlement, and we employ *baselines* to systematically gain insight into the transaction-rate performance characteristics of the protocol.

In order to determine how quickly transactions settle in Hydra, and at which rate they can be processed, we have to consider the following factors:

Opening and closing of a head. This consists of creating and submitting the commit/decommit transactions, and waiting until they are confirmed to be in the chain.

The performance of the head protocol. Given a geographical distribution and CPU/network capacity of the head nodes, how long does it take to exchange the messages that lead to transactions and snapshots being confirmed?

Limitations on in-flight transactions. When a player wants to send two transactions, where one uses the change from the other, they have to defer sending the second transaction until they have confirmation for the first. Furthermore, players may want to prevent an excessive number of confirmed, but not snapshotted transactions to keep decommits smaller. Together, this limits the number of *in-flight* (submitted but not yet confirmed) transactions that any one node can have.

The value at risk. To minimize this, players may wait for some transactions to be confirmed before sending more transactions, further limiting the number of in-flight transactions.

Since the time for opening and closing a head is largely dependent on the underlying layer-one protocol and can be amortized over the head's lifetime, we do not cover this aspect in our simulations. Furthermore, to simplify the simulations, we model the effect of a finite UTxO by directly limiting the number of in-flight transactions per node. Thus, we focus the simulations on the execution of the head protocol, as specified in Fig. 10.

7.1 Methodology

The experimental setup involves a fixed set of nodes, with a specified network bandwidth per node and geographic location of each node that determines the network latency between each pair of nodes. Each node submits transactions with a specified *transaction concurrency* c: it sends c transactions as fast as its resources allow, and then sends another one whenever one of the transactions it sent previously gets confirmed. This controls the number of inflight transactions to be c per node. *Snapshots* are performed regularly: nodes take turns to produce snapshots, and whenever the current leader has at least one confirmed transaction, it will create a snapshot with all the confirmed transactions it knows about.

In order to properly gauge the simulation results, we compare it to baseline scenarios that are sufficiently simple to facilitate optimistic performance limits exactly. We derive those limits by considering each sequence of events that has to happen in order for a number of transactions to be confirmed, and summing up the time for each event in those sequences. In particular, we have three resources that potentially limit the transaction rate:

1. The *CPU capacity* at each node determines how fast transactions can be validated, and signatures be created or verified;
2. The inbound and outbound *network bandwidth* limits how many message bytes can be received and sent by each node in a given time;
3. Each message between two nodes is delayed by the *network latency* between those nodes.

Depending on the configuration of the system, the most utilized of these resources will limit the transaction rate. This is an idealization: in a real execution of a protocol, contention effects will cause even the scarcest resource to be blocked and idle occasionally. We thus expect experimental results to be bounded by the baselines, and interpret the difference as the impact of contention effects. We consider the following baselines:

Universal Baseline: Full Trust. To quantify the price we pay for consensus in Hydra, we compare our simulations with a scenario where we assume perfect trust between all participants; i.e., we only distribute the knowledge on transactions, without trying to achieve consensus. In this scenario, nodes submit transactions (after checking that they are valid). Other nodes just acknowledge that they have seen them (without validating or signing them). We still consider the effect of having a finite transaction concurrency.

This baseline sets an upper limit for the transaction throughput of *any* protocol that distributes and validates transactions in a distributed system. Furthermore, for any *consensus* protocol, we should expect some additional overhead (which might or might not reduce the actual throughput in different regions of the parameter space).

Hydra Unlimited. This scenario resembles the head protocol, but executed under ideal circumstances, ignoring contention effects as described above. In contrast to a real execution of the protocol, where the snapshot size is an emergent property depending on how fast transactions are confirmed, in the baseline, we can directly control how many transactions are contained in a snapshot.

Sprites Unlimited. In order to compare to prior work, we also include a baseline according to an optimal execution of the off-chain protocol from [31]. A deciding difference to the head protocol is that in Sprites, all nodes send their inputs to a leader, which collates them and collects signatures for a whole batch of transactions. Compared to Hydra, this batching reduces the demand on CPU time and number of messages, since less signatures have to be performed and shared, at the expense of additional network roundtrips and higher network bandwidth usage at the current leader node, which has to send the batch of all transactions to every other node.

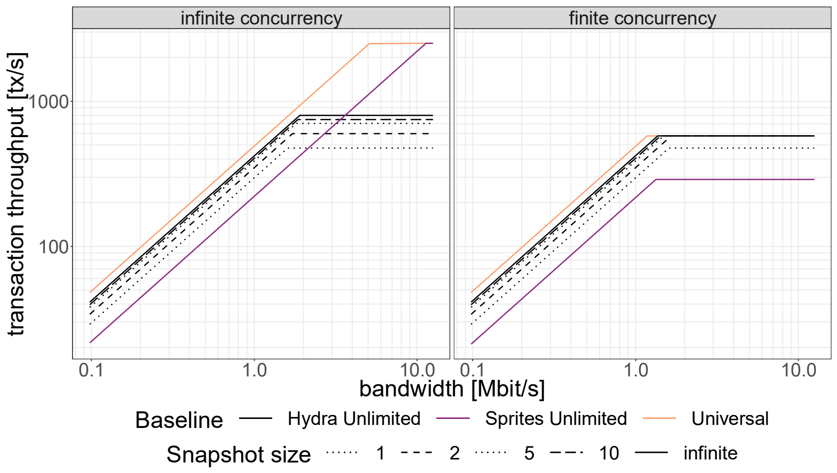


Figure 14: Example baselines scenarios, for finite and infinite transactions concurrency.

We show examples of baselines in Fig. 14. We draw the different baseline scenarios using different colours. For the Hydra Unlimited case, we have multiple lines, depending on the number of transactions in each snapshot; the more transactions are bundled in any one snapshot, the lower the overhead *per transaction*.

The left panel shows the limit of infinite transaction concurrency. In that case, the network roundtrip time can be perfectly amortised and is not a limiting factor. The resulting transaction rate has a knee shape: it is linear in the network bandwidth as long as that is the limiting factor, and turns constant once the limit from CPU time dominates. Comparing the Hydra Unlimited and Universal baselines, we see that there is some difference in the low bandwidth region, which is due to the multisignatures being sent in Hydra. In the region where CPU time is relevant, the difference is more pronounced, due to the computational cost of multisignatures. Looking at the Sprites Unlimited baseline, we see the tradeoff in batching transactions: the computational work is significantly reduced, by signing just a single large batch of transactions[6]. This comes at the cost of increasing the network traffic at the leader node, which has to send every transaction to every

[6]In the limit of infinite transaction concurrency, we take the batch size in Sprites to be unlimited as well.

other node. Note that in this picture, we only used a cluster of three nodes; for larger clusters, the demand on the leader node's network bandwidth would be even higher.

To get a more realistic picture, let us turn to the right panel. Here, we have a finite transaction concurrency, and the network roundtrip time is large enough to become the limiting factor (instead of CPU time) once we have enough bandwidth. Comparing Hydra Unlimited to the Universal line, we see that both flatten at about 580 TPS. The limit from network latency is the same for both baselines, since the number of roundtrips to confirm a transaction is the same (the messages are larger for Hydra Unlimited, but this only places a higher demand on the bandwidth). Interestingly, if we make a snapshot for each single transaction, we are still limited by CPU power, but as soon as we only make a snapshot every other transaction, the overhead from producing snapshots is small enough to not matter, compared to the limit from network latency. In this picture, the Sprites Unlimited baseline is well below the others. The demands on bandwidth – particularly, the network bandwidth of the leader node – is much larger than for the other protocols, and bundling transactions centrally before sending them to each node requires an additional roundtrip.

Note that devising an unlimited baseline for a given protocol, and comparing it to a universal baseline or those of other protocols, is not only valuable for evaluating an implementation, but also as a tool to predict possible performance during the protocol design phase.

7.2 Implementation

In the following, we will describe how we implemented the simulations for the head protocol. The implementation is available at `https://github.com/input-output-hk/hydra-sim`.

We model the head nodes using concurrent threads which exchange the protocol messages from Fig. 10 via channels. We use the `io-sim` library [2], which allows us to write concurrent code, and then *either* execute it directly as threads in the Haskell runtime system, *or* run the same code in a simulation of the runtime system. The latter yields an execution trace of the code very quickly, as it delays a thread by just increasing a number representing the thread's clock, instead of *actually* pausing the thread. As we describe below, the simulations make heavy use of thread delays, so this allows us to perform simulations much more quickly. We can also manually insert trace points at relevant points in the protocol (such as when a transaction is confirmed). Measuring, for instance, the confirmation time for a message, can then be done by simply subtracting timestamps of the events “transaction is submitted” (`new`) and “transaction is confirmed” (`confTx`).

Cryptographic Operations. Instead of using real cryptographic functions for multisignatures, we use mock functions that do not perform any calculations, but instead allow for a tunable delay of the thread that is performing the operation.

Message Propagation. Before being sent across the network, each message has to be serialized and pass the networking interface, which takes time linear in the message size. So the event of a message being sent by a node does not correspond to a single point in time, but rather to a time interval. We take that into account by modeling each message by its *leading* and *trailing edge*. The time distance between leading and trailing edge—the *serialization delay*—of a message is determined by its size and the bandwidth of the node's networking interface. We capture this with a parameter S, giving the delay per byte. Furthermore, we take into account that the networking interface can only start sending the next message *after* the trailing edge of the previous message has been sent. When the network is sufficiently busy, this can be a point of contention.

We model the network by a delay G between each message edge leaving the sending node and its arrival at the target node. The parameter G is determined by the distance between the two nodes and is independent of message size.[7] We use real data measured between Amazon Web Services data centers.

Once the leading edge of a message reaches the receiving node, we put its incoming networking interface into a busy state, for a time given by the size of the message and the bandwidth of this node. Finally, when the trailing edge is received, the message contents is placed into the local inbox, so that the node can start acting on the message.

If we only consider a single message, this model will just lead to a delay of the whole message determined by G, the message size, and S of the slower node. But once we have multiple messages in the system, it also correctly accounts for the contention at the outgoing and incoming connection points. The contention introduces variance, since messages may or may not have to wait at either end of the network.

Simulation optimizations. We applied two refinements that optimize the performance without changing the security of the protocol. First, when submitting a new transaction via `new`, a node will validate the transaction, and then send `reqTx` to every party, including itself. Every party, upon receiving `reqTx`, will then validate the transaction again. For the sending node, this is not necessary (it just validated the same transaction), so we skip the second validation on the same node. Second, the specification of the protocol states that handlers are executed strictly one after the other. Avoiding concurrency in this way simplifies the analysis of the protocol. But there is one case where we can safely perform actions in parallel: upon receiving `reqTx` (and similarly `reqSn`), a node will validate the transaction or snapshot against its local state, and, if appropriate, sign it and reply. The action of signing does not access the state of the node, so we can safely perform it concurrently with handling subsequent events.

These are fairly trivial changes, that any concrete implementation would apply, so we felt it was appropriate to reflect them in our simulations, as well as in the baselines.

7.3 Experimental Results

We performed experiments for three clusters with different geographic distributions of nodes: a *local* deployment of three nodes within the same AWS region, a *continental* deployment across multiple AWS regions on the same continent (Ireland, London, and Frankfurt), and a *global* deployment (Oregon, Frankfurt, and Tokyo). For each of those clusters, we measured the dependency of confirmation time and transaction throughput on bandwidth and transaction concurrency, and compare with the baselines described above. The numerical results depend on a number of parameters that we set, representing the time that elementary operations within the protocol take. We use the settings described below.

Transaction size. We use two representative transaction types: (1) *simple* UTxO transactions with two inputs and two outputs, whose size is 265 bytes, and (2) script transactions containing larger scripts of 10 kbytes. We use transaction references of 32 bytes. For each message, we allow for a protocol-level overhead of 2 bytes.

[7]The messages in the Hydra protocol are small enough to ignore TCP window effects that would introduce a dependency on the message size.

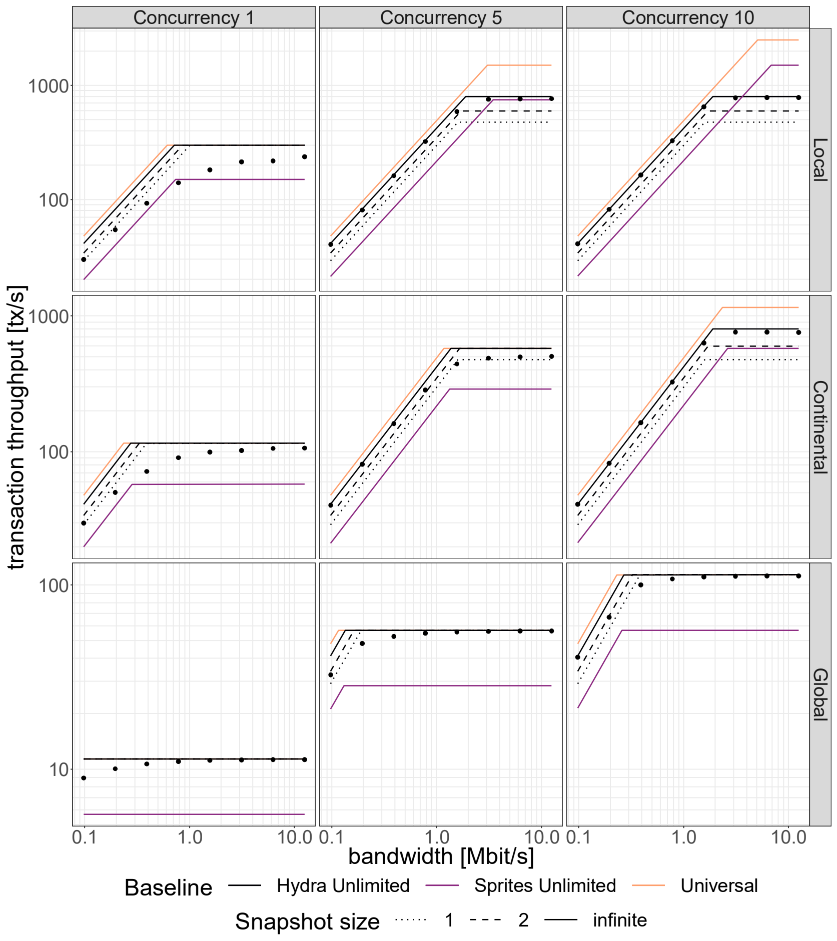


Figure 15: Transaction rates for the Hydra head protocol, compared with the baseline scenarios. Simple UTxO transactions with 2 inputs and 2 outputs.

Transaction validation time. This is the CPU time that a single node will expend in order to check the validity of a transaction. We use conservative values here: 0.4 ms for simple transactions, and 3 ms for script transactions.

Time for multisignature operations. We performed benchmarks for the multisignature scheme [11] resulting in the following estimates: 0.15 ms for MS-Sign, 0.01 ms for MS-ASig, and 0.85 ms for MS-AVerify.

Transaction throughput. Figs. 15 and 16 display results for ordinary UTxO and script transactions, respectively. The different rows correspond to the different geographical setups of the clusters, while the columns differ in transaction concurrency.

As expected, the Universal baseline consistently gives the highest transaction rate. For Hydra Unlimited, we see three baselines, for different snapshot sizes (depicted by dotted, dashed, and solid lines). In some cases, they coincide. Those are the configurations where we are limited by the network latency: performing snapshots increases the demand on CPU time and bandwidth (for

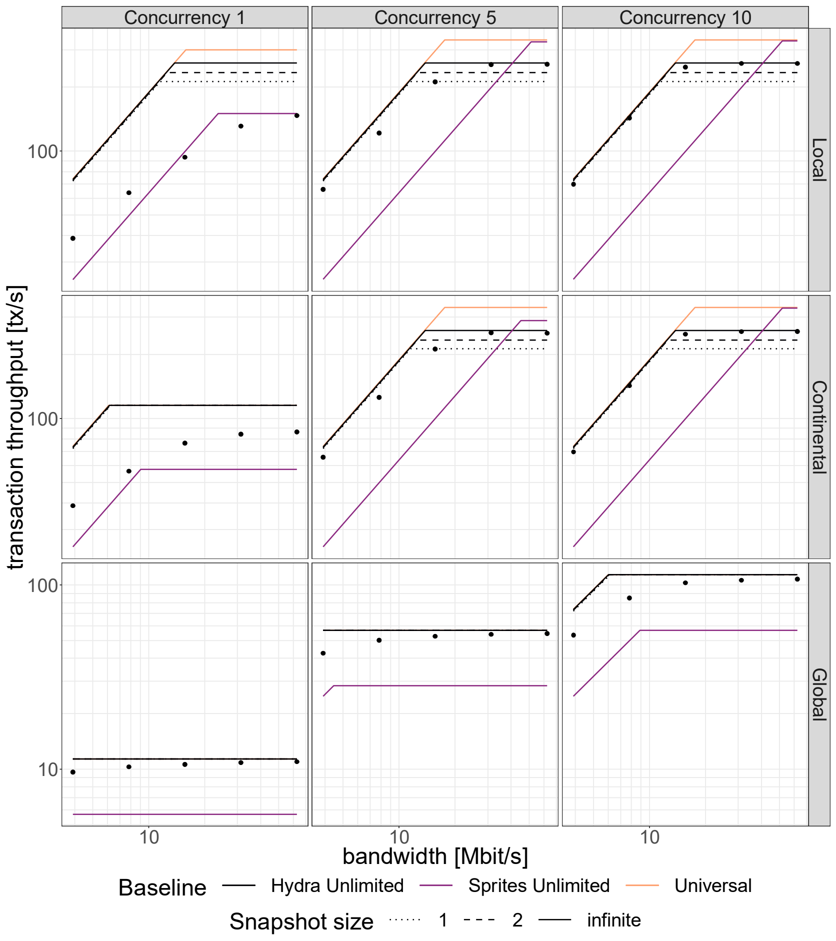


Figure 16: Transaction rates for the Hydra head protocol, compared with the baseline scenarios. Script transactions.

the additional signatures and messages), but it does not increase the number of sequential network roundtrips that have to be performed to confirm transactions (the messages for snapshots and for transactions propagate through the network concurrently).

Comparing the Universal and Hydra Unlimited baselines, we see that they are identical whenever the transaction rate is limited by the network latency. That can be explained since the difference between the two baselines differ only in their demand for CPU time (for creating and validating signatures) and bandwidth (for sending signatures). Note that for script transactions (Fig. 16), the demands on CPU are higher anyway, so that the additional cost for the multisignatures generally has a much lower impact on the transaction rate.

Looking at the Sprites Unlimited baseline, we observe the effect of batching via a central leader: the leader needs to send all transactions to every other node, and so its networking interface is frequently a bottleneck. Also, we see the additional roundtrip between the leader and every other node reducing the TPS whenever the network latency is the limiting resource. But when we have enough concurrency to form large batches, and get to the region where we are limited by CPU time, the savings by signing batches instead of individual transactions become apparent, and the

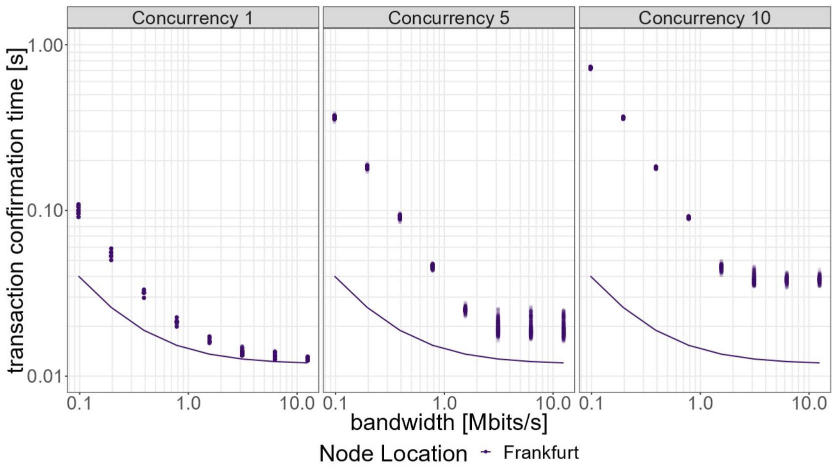


Figure 17: Confirmation times for simple UTxO transactions, in a cluster located in one AWS region. From panel to panel, we increase the transaction concurrency. The theoretically minimal confirmation time is represented by a dashed line.

Sprite baseline nearly reaches the Universal one.

Comparing the experimental results with the Hydra Unlimited baseline, we see that in most cases, the simulation of the protocol approximates the optimal curve quite well. We only get sizeable differences for low concurrency and insufficient bandwidth.

Regarding snapshots, the figures reveal that performing snapshots has a negligible impact on the transaction rate: apart from the regions where bandwidth is the limit, the baselines for different snapshot sizes only differ when we are CPU bound, which requires enough transaction concurrency to amortize the network latency. But for large concurrency, we also get large snapshots, so the overhead from snapshots *per transaction* is small.

Transaction confirmation times. One aspect where Hydra really shines is fast settlement: as soon as all parties have signed a transaction, and the sending node has aggregated a valid multisignature, this multisignature provides a guarantee that the transaction can be included into the ledger of the layer-one system. We can derive a minimal confirmation time by adding up the times for validating a transaction two times (once at the issuing node, once at every other node), sending the `reqTx` and `ackTx` messages across the longest path in the network, and creating and validating the aggregate signature.

Fig. 17 illustrates the conditions under which we achieve minimal confirmation time. In the first panel, we have a transaction concurrency of one. We see that, with enough bandwidth, we get very close to the minimal validation time, indicated by the line. In the other panels, we increase the concurrency. While this increases the total transaction throughput by sending transactions in parallel, individual transactions are more likely to be slowed down by congestion in the networking interfaces. Hence, confirmation time and its spread increase.

In clusters across different regions, the confirmation time generally depends on which node sent the transaction. For example, in Fig. 18, we see that the transactions from Oregon tend to get confirmed faster than those from Frankfurt or Tokyo. This is because confirmation requires a roundtrip to the farthest peer, and Frankfurt and Tokyo are farther away from one another than

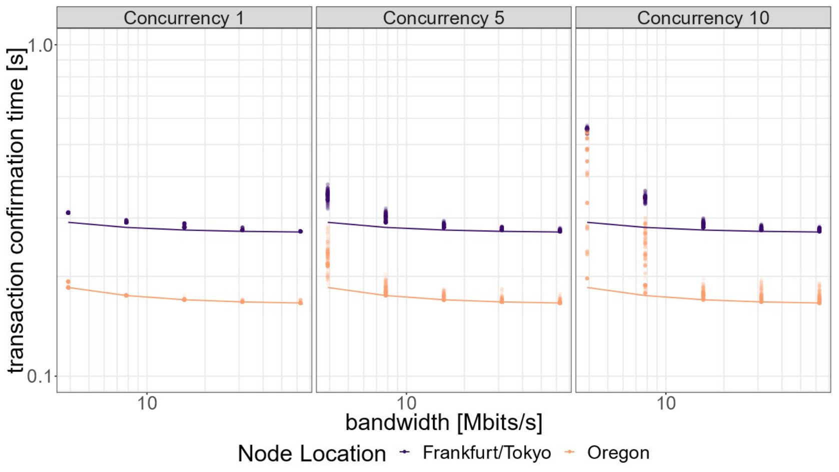


Figure 18: As Fig. 17, but for script transactions in a cluster spanning the AWS regions Oregon, Frankfurt, and Tokyo. Here, the minimal confirmation time depends on which node is sending the transaction, so we have two optimal lines.

either of them is from Oregon.

We see that even for script transactions and a globally distributed network, we consistently achieve settlement well below half a second if we provide enough bandwidth.

Larger clusters. In addition to three node clusters, we have also evaluated how the results depend on cluster size by running simulations with clusters of up to 100 nodes (located in the same AWS region):

- The transaction rates of a larger cluster are close to those for a three-node cluster. This is due to the fact that the amount of computation per node per transaction does not depend on the number of participants[8].
- The bandwidth needed at each node to reach the maximal transaction rate *does* depend on the cluster size. This is not surprising, since each node needs to communicate with more peers.
- For the same reason, the confirmation time of transactions increases with the cluster size.

Note that these simulations still use a communication pattern where everyone sends messages to everyone, which is not optimal for large clusters. Instead, we ought to construct a graph to broadcast messages, keeping the number of peers for direct communication small for each participant. An advantage of the Hydra approach is that we can easily have different versions of the head protocol, or different implementations of the same head protocol, optimized for different cluster sizes.

[8]Note that aggregating signatures and verifying an aggregate signature do depend on the number of participants. However, this does not impact the transaction rates in our simulations, for three reasons: i) we assume that we aggregate the verification keys once at the beginning of the head protocol, and only perform verification against the already computed aggregate verification key during the protocol, ii) even for 100 participants, combining the signatures is quicker than producing a single signature, iii) combining signatures is performed concurrently with the rest of the protocol (see Section 7.2).