

AsyncLS: an iUC Framework for Wallet-Based Asynchronous Ledger Services

Keyang Liu¹ and Li Duan²

¹ Talus Labs Inc, Tokyo, Japan

`stephen@taluslabs.xyz`

² University of Wuppertal, Wuppertal, Germany

`li.duan@uni-wuppertal.de`

Abstract. We define an asynchronous ledger service (AsyncLS) in the IITM-based universal composability (iUC) framework. The wallet-based service exposes a general state machine for a decentralized ledger-based application through registration, reads of committed application state, authenticated submission, and status queries with immutable final results. Each logical request binds the user’s authorization to a finalized checkpoint and the applicable execution context. A confirmed outcome identifies the first eligible transaction and records application success or rejection. Strict expiry requires complete processing beyond the eligible height window with no recorded outcome, and certifies that the request has no application effect.

We construct a protocol Π_{AsyncLS} that realizes the ideal service using four components: **Wallet**, **Relay**, **Adapter**, and **Replay**. The proof separates the service construction from the implementations of **Relay** and **Adapter** while preserving ledger, clock, authentication and setup interactions. We prove realization for the trusted and local implementations under their respective execution and trust assumptions.

We use the service to construct an ORDI interactive payment and a parameterized execution layer above transaction ordering. These examples distinguish the ledger interaction protocol from the application state and effects exposed by the service. Each use requires the stated authorization, setup, ledger, clock, authentication, and ledger-effect assumptions.

Keywords: Universal Composability · Wallets · Transaction Ledgers · Asynchronous services

1 Introduction

Distributed ledgers [31] support cryptocurrencies [22,33,30] and applications in banking [11] and supply chain management [32]. These applications depend on the service that wallet infrastructure exposes above the ledger.

Badertscher et al. [3] formalized a general ideal ledger $\mathcal{G}_{\text{LEDGER}}$ and related Bitcoin to it in the universal composability (UC) framework. Their functionality specifies transaction submission, ledger reads and consensus-backed updates. We ask: *given a secure ledger, what application-facing functionality can it provide through a wallet-mediated access path?*

1.1 Our contributions

An external protocol modeled directly above a ledger must connect authorization to the complete request, interpret transactions as application transitions, track repeated or competing requests, and relate later observations to the approved action. Submission or a ledger occurrence alone does not supply this application-level contract. We define an ideal service that derives request results from ordered ledger transactions and construct a protocol that realizes it. A client using **AsyncLS** invokes registration, read, submit and status operations and can use the reconstruction, terminal-agreement and conditional-expiry results proved below. The client still supplies its application transition and proves admissibility and any correspondence to ledger effects. Funding, authorization and progress assumptions also remain explicit. The benefit is a reusable statement and proof boundary for request processing and outcomes.

1. **An ideal service for applications.** We define $\mathcal{F}_{\text{AsyncLS}}$ for ledger-backed state machines, with registration, reads of committed application state, authenticated submission, and immutable final request results. Each request is bound to its checkpoint. For a request with a finite eligible window, strict expiry rules out application effects once the service has completely processed the ledger prefix beyond that window without recording an outcome for the request.
2. **A protocol and its realization proof.** We construct Π_{AsyncLS} from **Wallet**, **Relay**, **Adapter**, and **Replay** and prove that it realizes $\mathcal{F}_{\text{AsyncLS}}$ under the stated assumptions.

We give trusted and local implementations of **Relay** and **Adapter**. Both preserve the ledger, clock, authentication and setup connections and use the same shared authentication functionality $\mathcal{G}_{\text{auth}}$, realized by signatures and authenticated registration. They differ in where computation runs and which parties must perform that computation honestly. Section 3.3 proves their correctness and states the additional obligations of remote deployments.

The adversary may delay and reorder messages. The stated realization uses the responsive environment class of Section 2.6 and its request-arrival clock condition, under which the first clock-read result equals the clock value at public-input arrival. The service reads history under the selected ledger's rules and verifies requests under the selected authentication functionality's rules. We assume protected signing endpoints, correct service computation and permanent finalized history. Rollback requires a fork-aware checkpoint specification and reconstruction proof, since reverting history need not reverse external application effects.

The service distinguishes confirmed application success, confirmed rejection and strict **Failure(Expired, ...)**, which certifies that the request has no application effect. An external protocol can then wait, use an outcome or authorize a successor request as its own rules permit.

The application state machine can also model an execution layer above $\mathcal{G}_{\text{LEDGER}}$'s transaction ordering (Section 5.2).

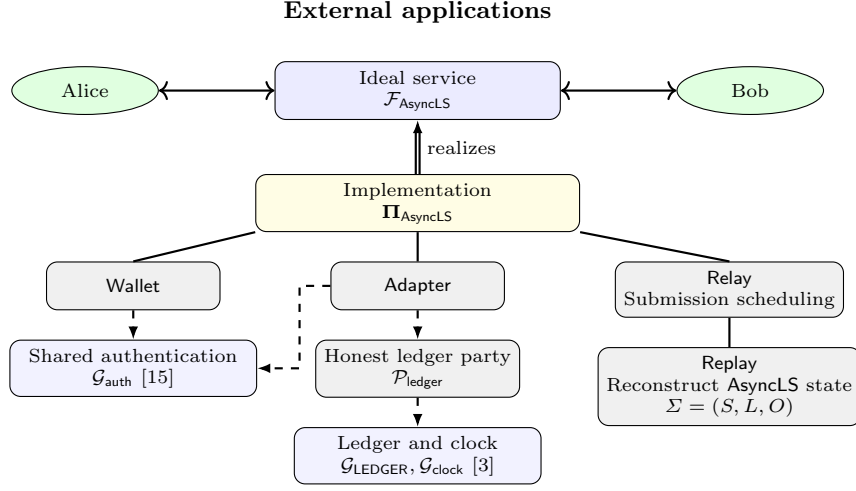


Fig. 1. External applications use $\mathcal{F}_{\text{AsyncLS}}$, realized by Wallet, Relay, Adapter and Replay with shared authentication $\mathcal{G}_{\text{auth}}$ and the declared ledger and clock.

1.2 Related work

Prior work provides complementary wallet, payment, and application interfaces. Formal wallet models study key derivation, authorization to spend, and ledger-facing operations [2,26]. Badertscher et al. prove an address management protocol above their transaction ledger [3, Sec. 9]. OCash binds recipient observation to the payment identifier, parties, and amount [25]. Ledger-based applications specify insured computation and private exchange under their own custody, privacy, and timing assumptions [5,6]. Our request and outcome contract separates the reusable AsyncLS reconstruction and agreement proof from the application-specific obligations in Section 2.14. Each application’s other guarantees require separate proofs.

General ledger models parameterize the underlying service [3,24]. Contract frameworks and state channels support programmable execution and composition over a ledger, under their respective model and delay assumptions [27,4,17]. Related implementation techniques keep results available across failures and delegate retries: Lee et al. record the result of a remote procedure call (RPC) atomically with the RPC’s changes to stored application data [29], while Etherless Ethereum permits competing submissions with one sender identifier [1]. Above the selected ledger’s ordering, read and submission contract, our construction realizes the request/outcome contract and proves immutable outcomes and strict expiry

under the stated finite-window and effect-correspondence conditions. Section 4.3 proves composition with the chosen implementations.

Sections 2–4 define the ideal, construct the protocol and prove realization. Sections 5 and 6 give applications and limitations. Section 2.5 lists notation.

2 Ideal AsyncLS

The asynchronous ledger service (**AsyncLS**) exposes a deterministic application through registration, committed-state reads, authenticated submission and status queries. Its ideal functionality owns application data and request outcomes and uses the declared authentication, ledger and clock.

2.1 Primitives

Fix a ledger $\mathcal{G}_{\text{LEDGER}}$, its clock $\mathcal{G}_{\text{clock}}$, authentication functionality $\mathcal{G}_{\text{auth}}$, its registration board and application $\mathcal{M}$. The configured transaction-construction and $\mathcal{G}_{\text{LEDGER}}$ setup procedures declare additional interfaces, including transaction signing when required. The ideal functionality, protocol and theorem use this fixed configuration of instances, connections and application interpretation. Changing the configuration requires a fresh service session.

Ideal ledger interface. The parameterized ledger and clock of [3] use `gid` through distinct interfaces. A party invokes `(register, gid)`, `(read, gid)` or `(submit, gid, tx)` under its actual identity. Registration returns that identity. A synchronized honest read returns an authentic permanent prefix `stateP`. Each submission creates an occurrence recording its sender and clock time, distinct from `tx.id`, and notifies the adversary regardless of admission. The caller receives no admission acknowledgment. Ledger maintenance, clock operations `clock-read`, `clock-update` and adversarial operations retain their interfaces (Appendices B.1 and B.2).

Authentication. Fix the security parameter λ and a global authentication functionality $\mathcal{G}_{\text{auth}}$ with the operations

$$\text{Register}(p), \quad \text{Sign}(p, x), \quad \text{Verify}(p, x, w).$$

Registration binds p to its assigned owner’s key. `Verify( $p, x, w$ )` checks the exact request and certificate on the verifier’s connection, returning a Boolean or `NoBinding`. For all callers, $\mathcal{G}_{\text{auth}}$ reports each `Register`, `Sign` or `Verify` invocation and its result on its adversarial interface. Appendix A.1 specifies the shared registration board, connections and observations. Appendix A.2 proves the signature realization.

2.2 Application state

A typed application $\mathcal{M}$ fixes the party, request and query domains $\mathcal{P}, \mathcal{R}, \mathcal{Q}$ and has a deterministic, polynomially bounded core $(\mathcal{S}, S_0, \delta, \text{Query})$. For $p \in \mathcal{P}$, $r \in \mathcal{R}$ and $S \in \mathcal{S}$, its transition returns

$$\delta(p, S, r) = (S', \text{Applied}(o)) \quad \text{or} \quad (S, \text{Rejected}(f)).$$

Success records effect o . Rejection preserves S and returns $f \in \text{Errors}$. The read-only $\text{Query}(S, q)$ returns the query answer computed from S (complete types appear in Section 2.11). **AsyncLS** checks authentication, request identity, checkpoint and validity window before invoking δ . The application enforces balance and smart-contract rules and may reject an included request. Ledger-derived nonce and outcome tables prevent repeated evaluation.

The configuration supplies the transaction constructor $\text{TransactionCreation}(x)$ and parser $\text{TransactionParsing}(tx)$. Reconstruction uses $\text{MatchesRequest}(tx, x)$ to check that the complete transaction and its authentic ledger context match the approved request. Section 2.11 specifies these application procedures.

Consider the same approved request x with a finite eligible window. A ledger read containing no matching transaction does not by itself certify that x can have no later application effect. A protocol modeled directly above the ledger can establish that conclusion by proving the interpretation, reconstruction and expiry properties of its request-processing rules. Through **AsyncLS**, a completed authentic scan beyond x 's last eligible height, with all required Boolean authentication decisions completed and no recorded outcome, yields the existing strict-expiry result (Lemma 1). The client can use that result in its own next-step proof under the stated correspondence between requests and represented ledger effects (Condition 5). It must still justify that next step, and finite completion still requires the separate progress assumptions of Section 2.16.

2.3 Ledger and AsyncLS

The ledger maintains a permanent block sequence **state**. A synchronized honest party reads a prefix $\text{state}_{\mathcal{P}_{\text{ledger}}} \preceq \text{state}$, with lag governed by `windowSize` [3, Secs. 4.2–4.3].

Reconstruction starts from S_0 and maps successive ledger prefixes to application states:

$$P_0 \preceq P_1 \preceq \cdots \preceq P_H \quad \mapsto \quad (S_0, S_1, \dots, S_H),$$

where P_i extends P_0 by i blocks. Version zero is relative to P_0 . **AsyncLS** processes only later blocks. Every reconstruction uses that same anchor.

The reserved field $S.\text{ver}$ starts at zero. Before processing block $i > 0$ after P_0 , **AsyncLS** sets only this field to i , including for empty blocks, without a ledger transaction, user request, nonce entry or receipt. Ordinary transitions preserve it. Rejection and duplication preserve the immediately preceding application state S . Hence, $\mathcal{M}$ can inspect the current relative height, $S_i.\text{ver} = i$, while queries use only completely processed prefixes.

Let **Encode** be the fixed injective tagged-tuple codec and **Decode** its partial inverse. Fix lifetimes $W, W_L \in \mathbb{N} \cup \{\infty\}$ measured in ledger blocks and release delays $0 \leq d_{\min} \leq d_{\max} \leq \infty$ in ledger-clock units, with finite $d_{\min}$.

Definition 1 (Service parameters). *For the fixed ledger, clock, authentication and application configuration, the shared numerical parameters are*

$$\zeta = (W, W_L, d_{\min}, d_{\max}).$$

Well-formedness requires the conditions in Section 2.11. The notation $\mathcal{F}_{\text{AsyncLS}}^\zeta$ suppresses the fixed configuration. Equal numerical parameters need not identify the same application or functionality instances.

Definition 2 (Service session). *A logical AsyncLS session has a unique address $\text{sid} \in \{0, 1\}^*$ bound to the configuration of $\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{clock}}, \mathcal{G}_{\text{auth}}, \mathcal{M}, \zeta$ and the immutable anchor prefix P_0 . Replicas of this session share sid and its authenticated anchor, while each retains its already configured execution identity, local state and actual interface connections. A distinct replica is not a new logical session.*

The checkpoint token and request encodings are

$$\begin{aligned} c_h &= \text{Encode}(\text{ALS/checkpoint}, \text{sid}, h), \\ x &= \text{Encode}(\text{ALS/request}, \text{sid}, p, \nu, r, c_h). \end{aligned}$$

Under the authenticated setup, checkpoint token c_h identifies the ledger prefix P_h and reconstructed application state S_h . Request x binds caller p , nonce ν , operation r and checkpoint token c_h for transaction construction

$$tx \leftarrow \text{TransactionCreation}(x), \quad \text{txid} = tx.\text{id}.$$

TransactionParsing recovers the complete request and certificate and recomputes the identifier from the complete transaction bytes. Both systems use the configured hash procedure and public index specified in Section 2.11. The constructor uses the caller-approved protected owner-signing route and authenticated data. It aborts when it detects a violation of its configured construction requirements. Section 3.4 gives the implementation requirements.

The service and transaction-ledger lifetimes both start at checkpoint h , with

$$W_{\text{eff}} = \min\{W, W_L\}, \quad D(x) = h + W_{\text{eff}}.$$

Here $D(x)$ is the last eligible relative height. A finite W_L must be enforced by the selected transaction format, validation and effect requirements. All represented effects respect both bounds (the effect-correspondence requirement in Section 2.11). Changing the request, checkpoint or lifetimes requires new authorization.

2.4 AsyncLS state and request records

The AsyncLS state is $\Sigma = (S, L, O)$, where S is the application state. The ledger-derived nonce table $L[(\text{sid}, p, \nu)]$ stores the first eligible request using that nonce.

The outcome table O stores each request's first result unchanged. $\perp$ denotes absent entries in both tables. Initially $\Sigma_0 = (S_0, \emptyset, \emptyset)$. Ordered processing of a complete prefix P_H determines Σ_H . The AsyncLS committed checkpoint $C = (P_H, \Sigma_H, c_H)$ is updated atomically, with $H = S_H.\text{ver}$ and checkpoint token c_H .

Each instance constructs at most one completed transaction per x . Repeated Submit invocations share preparation and reuse the exact retained transaction (the submission-preparation procedure in Section 2.13). Transaction equality means equality of the complete encoding. The transaction table stores $E[x] = (tx, t_0, T, s)$, or $\perp$ for absent entries. The first save requires an identifier unused in E . An identical transaction preserves the record, while a request, transaction or identifier conflict aborts the operation. In particular, equal hashes of different complete transactions trigger the existing conflict branch at saving, not transaction reuse. This guard gives distinct identifiers among records saved in this instance's E , even on collision executions. It asserts no global hash injectivity across ledger records or other instances. Reconstruction also handles transactions from other instances and senders.

Request-arrival clock sampling. Each correctly encoded Submit starts with $t_0 = \perp$ and issues its first actual clock-read on the configured connection, through Adapter.Time in the reference. The invocation saves that operation's returned value once as t_0 , including the source Round-Update's effect. The invocation cannot enqueue the transaction, save its entry in the transaction table, set its release target or emit PendingTx until it has saved the returned clock value. Its other work follows Section 2.13's saved-operation and terminal-priority rules. Delayed consumption never resamples or replaces the original value. Let τ_{arr} denote the clock time at public-input arrival as an execution-trace quantity. The *request-arrival clock condition* requires the first read's returned value to equal τ_{arr} , hence $t_0 = \tau_{\text{arr}}$ whenever that value is saved. This condition restricts admitted schedules. Keeping one registered honest party's update bit at zero from arrival through sampling suffices. The invocation that first saves $E[x]$ supplies its own t_0 , including when it joined construction. Reuse preserves the complete record and never refreshes that timestamp.

Initially $T = t_0 + d_{\text{min}}$ and $s = \text{Waiting}$. At release, the request's pending status gives $s = \text{Released}$ and its terminal status gives $s = \text{Suppressed}$. Sending permanently sets $s = \text{Submitted}$.

A request result receipt records its ledger occurrence and application result:

$$\text{receipt} = (\text{gid}, x, \text{txid}, (i, j), z), \quad z \in \{\text{Applied}(o), \text{Rejected}(f)\}.$$

Here $\text{txid} = tx.\text{id}$, (i, j) is the first eligible occurrence's relative height and position, and z is its application result or nonce-conflict rejection. Expiry creates no receipt.

Define

$$\text{Status}_H(x) = \begin{cases} \text{Confirmed}(O_H[x]), & x \in \text{dom}(O_H), \\ \text{Failure}(\text{Expired}, x, c_{D(x)+1}), & D(x) < \infty, H > D(x), \\ & x \notin \text{dom}(O_H), \\ \text{Pending}, & \text{otherwise.} \end{cases}$$

Both **Confirmed** and **Failure** are immutable. Expiry requires complete processing beyond $D(x)$ with no outcome and rules out an application effect. This no-effect guarantee covers represented ledger effects under the effect-correspondence requirement in Section 2.11, excluding independent ledger fees. Checking a request requires no local submission record.

2.5 Notation

Table 1: Core notation. Symbols used only within one proof or example are defined there.

Symbol or function	Meaning
$\mathcal{F}_{\text{AsyncLS}}, \mathbf{\Pi}_{\text{AsyncLS}}$	Ideal asynchronous ledger service and its reference construction.
$\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{clock}}, \mathcal{G}_{\text{auth}}$	Configured transaction ledger, clock and shared authentication functionality.
gid, sid	Ledger/clock identifier and logical service-session identifier.
$\mathcal{M}$	Typed application with its fixed ledger interpretation.
Encode, Decode	Fixed injective tagged-tuple codec and partial decoder.
$\mathcal{P}, \mathcal{R}, \mathcal{Q}$	Application caller, request and query domains. Elements are p, r, q .
$p, \omega_p, \mathcal{P}_{\text{ledger}}$	Application caller, actual signing owner and the service's registered ledger submitting party. Their connection permissions are distinct.
$(\mathcal{S}, S_0, \delta, \text{Query})$	Application state domain, initial application state, transition and read-only query.
Applied(o), Rejected(f)	Application success with output o , or rejection with reason f and unchanged application state S .
$P_0, P_h, \preceq$	Immutable setup prefix, its extension by h finalized blocks, and the block-prefix relation.
b_0, i, H	Absolute height of the anchor, relative block position after it, and completely processed relative height.

Symbol or function	Meaning
$S.ver, c_h$	Reserved relative version and checkpoint token identifying (P_h, S_h) under authenticated setup.
$\Sigma = (S, L, O), C$	AsyncLS state with application state S , nonce table L and outcome table O . The AsyncLS committed checkpoint is $C = (P_H, \Sigma_H, c_H)$.
$L[(sid, p, \nu)]$	First eligible request using the caller's nonce, or $\perp$ before consumption.
x, ν, w	Encoded request, its nonce, and the certificate authenticating that request.
$W, W_L, W_{eff}, D(x)$	Lifetimes in blocks, effective lifetime $\min(W, W_L)$ and last eligible relative height $h + W_{eff}$.
T, d_{min}, d_{max}	Absolute release target and allowed ledger-clock delays from the saved transaction's immutable first sample t_0 .
ζ	Shared AsyncLS parameters $(W, W_L, d_{min}, d_{max})$. $\mathcal{M}$, $\mathcal{G}_{LEDGER}$, $\mathcal{G}_{clock}$, $\mathcal{G}_{auth}$ and the setup/constructor interfaces are fixed separately.
TransactionCreation	Configured resumable construction with approved signing and applicable transaction checks.
TransactionParsing, decodeRecords	Transaction parser and resumable decoder of authentic ledger blocks.
MatchesRequest	Checks whether the complete transaction and its authentic ledger context match the request.
$tx, txid, (i, j)$	Transaction, its hash identifier with probabilistic complete-byte binding, and its occurrence at position j in relative block i .
receipt, $O[x]$	First finalized outcome record $(gid, x, txid, (i, j), z)$. Ledger occurrence metadata remains separate.
Status _H (x)	Request status from one AsyncLS committed checkpoint: confirmed outcome, strict expiry or pending.
Confirmed, Failure, Pending	Immutable recorded outcome, expiry ruling out the defined application effect, or unresolved status.
Abort	An operation error, also returned for NoBinding . A later call may retry. It certifies neither rejection nor expiry.
Register, Read, Submit, Check	The four AsyncLS operations: registration, read of committed application state, authenticated submission and status query.
Wallet	Public caller invocations, approvals and protected signing.
Relay	Transaction table E , release decisions and submission scheduling.

Symbol or function	Meaning
Adapter	Ledger operations (registration, reads and submissions), clock reads and $\mathcal{G}_{\text{auth}}.\text{Verify}$.
Replay	Reconstruction of the AsyncLS state $\Sigma = (S, L, O)$ from authenticated ledger prefixes.
$\Pi_{\mathcal{G}_{\text{auth}}}$	Signature implementation of the shared authentication interface.
AuthInput, AuthOutput	Uncorrupted service-input and result observations, correlated by actual caller route and invocation index. Corrupted-signer I/O follows framework forwarding.
NoBinding, $\text{Cert}_\omega[m, w]$	Missing-binding result (no Boolean decision), and the global certificate decision fixed after positive binding retrieval.
$\mathcal{E}, \mathcal{S}_{\text{sim}}, \lambda$	Admitted iUC environment, fixed simulator and security parameter.
$\epsilon_{\text{RA}}, \epsilon_{\text{auth}}$	Errors for implementing the code of Relay and Adapter , and for replacing authentication.
$\text{View}_{\mathcal{E}, P}$	Environment-visible I/O, network messages and output of the complete system.
$E[x], s$	Sole saved transaction record (tx, t_0, T, s) for request x , or $\perp$. s is Waiting , Released , Submitted or Suppressed .
$T_\zeta, t_0, \tau_{\text{arr}}$	Conditional terminal bound, first clock sample (initially $t_0 = \perp$), and trace-only arrival time. Sampling under the request-arrival clock condition gives $t_0 = \tau_{\text{arr}}$.
$\text{Env}, \text{Env}_{\text{RA}},$ $\text{Env}_{\mathcal{G}_{\text{auth}}}$	Classes of complete iUC environments for service, component and authentication comparisons, respectively.
Exec_ζ	Execution class satisfying the fixed uniform progress assumptions for the profile ζ .
$D(X)$	Probability distribution of a random variable X .

2.6 iUC environments and comparison

For a random variable X , write $D(X)$ for its probability distribution.

Environment and iUC realization. Composition $\parallel$ uses the declared role map: the declared public roles of the ledger, clock, authentication, board and constructor-specific interfaces remain public when used internally, unlike the role-hiding shorthand of [13, Sec. 3.4]. We quantify over every universally bounded, compatible, responsive PPT environment $\mathcal{E}(1^\lambda)$, including its adversarial scheduling and permitted independent ledger, clock, authentication, board and constructor-specific requests, whose connected protocols are environmentally bounded. For

service realization, its schedule also satisfies the request-arrival clock condition of Section 2.4. Universal boundedness gives the environment one polynomial computation bound across connectable systems. Environmental boundedness gives the protocol a polynomial bound for each such environment, except on negligible runs. Responsiveness requires the next environment message back to the protocol, if any, after a designated restricting message to answer the same instance with an allowed response. It does not guarantee completion [13, Sec. 2 and App. H]. The simulator is fixed before $\mathcal{E}$. Negligibility is required separately for each such environment, without one common budget over all PPT environments. The operational proof constructs an exact coupling for the core comparison. The authentication proof constructs a coupling outside its explicitly monitored bad event. For these two comparisons only, write $(R, I, \epsilon) = (\Pi_{\text{AsyncLS}}^\zeta, \mathcal{F}_{\text{AsyncLS}}^\zeta, 0)$ or $(\Pi_{\mathcal{G}_{\text{auth}}}, \mathcal{G}_{\text{auth}}, \epsilon_{\text{auth}})$, respectively, with the surrounding unchanged interfaces and corresponding simulator included. These two couplings satisfy

$$\Pr[\text{View}_{\mathcal{E}, R}(1^\lambda) \neq \text{View}_{\mathcal{E}, \mathcal{S}_{\text{sim}}, I}(1^\lambda)] \leq \epsilon(\lambda).$$

For a complete view v , let $\text{out}(v) = 1$ if the environment outputs 1 in v , and let $\text{out}(v) = 0$ otherwise. Define

$$\begin{aligned} \text{Adv}_{\mathcal{E}}(R, I, \mathcal{S}_{\text{sim}}; \lambda) &= |\Pr[\text{out}(\text{View}_{\mathcal{E}, R}(1^\lambda)) = 1] \\ &\quad - \Pr[\text{out}(\text{View}_{\mathcal{E}, \mathcal{S}_{\text{sim}}, I}(1^\lambda)) = 1]|. \end{aligned}$$

Write $\leq_{\text{iUC}}^{\epsilon(\lambda)}$ for distinguishing advantage at most $\epsilon(\lambda)$ in every environment of the specified class, with its polynomial budgets suppressed. An arbitrary **Relay/Adapter** realization supplies only the stated computational iUC bound on the difference of environment acceptance probabilities. Section 4.3 combines that bound with authentication by a triangle inequality, without asserting a coupling for arbitrary implementations. An error is negligible if it is eventually below $1/q(\lambda)$ for every positive polynomial q .

2.7 Domains and encodings

The fixed application and transaction formats specify bounded domains and encodings for application parties $\mathcal{P}$, application accounts, requests $\mathcal{R}$, queries $\mathcal{Q}$, application states $\mathcal{S}$, certificates, transaction payloads, and error codes. Environmental boundedness limits aggregate execution data polynomially while allowing storage to grow over the system's lifetime. Each encoded field has an explicit length and tag. A deterministic partial decoder rejects strings that do not use the specified encoding. For every accepted tuple z , $\text{Decode}(\text{Encode}(z)) = z$. For every accepted encoded string b , $\text{Encode}(\text{Decode}(b)) = b$. Equality of such tuple encodings therefore means exact equality of the full tuples. Transaction hash identifiers have the probabilistic binding guarantee in the transaction-identifiers paragraph of Section 2.11.

2.8 Authenticated AsyncLS setup and ledger access

The service’s honest ledger party registers using $(\text{register}, \text{gid})$ under its actual ledger-party identity. Before relying on ledger reads, the party obtains a clock value after its registration response and waits until the clock exceeds that value plus Delay . This enforces the strict synchronization condition in Appendix B.1. Both compared systems perform the same clock operations and ledger reads with the same delays. The honesty assumption keeps this party registered and uncorrupted. An arbitrary desynchronized view is never treated as permanent ledger history.

After this condition holds, setup issues an actual ledger read on the party’s saved connection. Setup sets P_0 only to the matching, well-formed finalized prefix returned on that connection. This is the current view obtained by that read, not necessarily the global finalized tip when its response is delivered. Malformed data aborts setup without initializing S_0 or enabling service. On success, the service records (sid, P_0) once in its public setup configuration, returns that record on the actual setup/configuration connection, and enables its service ports. The session identifier sid and its fixed configuration remain unchanged. Setup associates the returned prefix with that session once. At this anchored ledger prefix, the configured S_0 is the initial application state with version zero, and the service initializes empty nonce and outcome tables L, O . AsyncLS processing begins strictly after P_0 . Any required correspondence between S_0 and ledger application state that already exists remains a deployment obligation.

Each logical session has one completed setup. Replicas import its authenticated (sid, P_0) configuration through the declared service interface and retain the original sid and anchor, each using its own configured execution identity, local state and actual connections with the existing registration and synchronization requirements. Independent setups use distinct identifiers under Definition 2. A checkpoint string or unauthenticated announcement does not supply this configuration. The service’s setup output adds no trusted registry or global dissemination guarantee, and repeated or stale responses cannot reset the anchor.

Interpreting ledger records. Reconstruction uses Appendix B.1’s ledger-assigned submitter and occurrence metadata. The scanner assigns (i, j) by block sequence and order after P_0 . Let b_0 be the ledger height at the P_0 boundary, using the configured height convention for an empty prefix. Relative position i has ledger height $b_0 + i$. The transaction format distinguishes protected AsyncLS payloads from ordinary data. Application interpretation follows ledger validation: a transaction refused by the ledger yields no application outcome.

The generic safety theorems impose no inclusion schedule. **Adapter** serializes its read connection, checks complete block formatting and compares each returned prefix with committed history: equality needs no update, an extension is scanned, and a malformed, older or incompatible view causes the affected read or recovery operation to abort without changing the AsyncLS committed checkpoint C . The synchronized ledger-party connection authenticates these views. All permitted

independent reads, submissions, maintenance, view controls and corruptions remain available under the stated honesty assumptions.

The service ledger party accepts submission only on its protected private connection under the submission-permission procedure in Section 2.13. Its published access role excludes submission. Other registered parties retain direct $\mathcal{G}_{\text{LEDGER}}$ access under their own identities. Both complete protocol systems expose the same four AsyncLS operations and the same submission-disclosure and scheduling interfaces, with identical private-send restrictions.

The AsyncLS scheduler issues **clock-read** through its declared actual connection. It issues neither a **clock-update** nor a maintenance input merely because a transaction became due. Ledger maintenance and the selected policy still govern progression. Honest reserved AsyncLS identities remain uncorrupted under the stated honesty assumptions. The simulator cannot turn adversarial traffic into a send on an honest protected connection.

2.9 Identifiers and information read from ledger records

The ledger/clock identifier gid and service identifier sid are distinct. The request and checkpoint token encodings are defined in Section 2.3. The checkpoint-token encoding requires a nonnegative relative height for the session. The token identifies the prefix/application pair (P_h, S_h) only under the authenticated setup. The session-configuration paragraph in Section 2.11 specifies the immutable logical-session binding to the common interpretation and owner identities, while retaining each replica's configured ledger/clock/authentication/setup connections.

The data-only AsyncLS carrier is the complete transaction

$$tx = \text{Pack}(x, w) = \text{Encode}(\text{ALS/authorized-carrier}, x, w).$$

The serialized envelope contains exactly its tag, x and w . Its parser recovers x, w and computes $\text{txid} = tx.\text{id}$ from these complete bytes under the transaction-identifier rule in Section 2.11. Other formats preserve the parsed request and certificate and compute the same kind of complete-byte hash accessor, subject to Section 2.11's effect and context requirements. The certificate is opaque to the carrier. Ledger-assigned occurrence metadata and recorded submitting party are separate from its bytes, and public block order determines (i, j) . The selected procedure for checking request authentication uses this authentic ledger record under the historical-authentication requirement in Section 2.11. The trusted and local implementations are proved in Section 3.3. Different static formats have distinct session bindings.

2.10 Using authentication in AsyncLS

The AsyncLS configuration uses Appendix A.1's owner binding and invokes $\text{Register}(p)$, $\text{Sign}(p, x)$ and $\text{Verify}(p, x, w)$ directly. The complete message $x = \text{Encode}(\text{ALS/request}, \text{sid}, p, \nu, r, c_h)$ binds the session and owner. Only actual caller approval of this complete x enables its owner's protected signing route.

An AsyncLS signing invocation comes only from its saved caller-authorized `TransactionCreation(x)` operation, with $m = x$. The certificate-production event precedes any delayed outer `Submit` reply.

The global authentication setup is retained as a separate interface. AsyncLS rejects malformed service inputs before invoking authentication. Observation correlations use Appendix A.1’s actual routes.

Missing registration and consistent reconstruction checks Under the historical-authentication requirement in Section 2.11, when reconstruction consumes a delivered `NoBinding`, it aborts the current recovery with the existing `Abort` result. The abort preserves the AsyncLS committed checkpoint C without skipping the record. A later public call starts a new recovery from C and may obtain a Boolean result after registration. Section 3.3 proves that the direct shared- $\mathcal{G}_{\text{auth}}$ procedure gives consistent completed decisions across scans without a private signing log.

2.11 Ideal-service and implementation requirements

The following requirements define well-formed ideal configurations. Section 2.13 specifies their reference operations.

Application and transaction types The configuration fixes the creation, parsing, bootstrap and reconstruction procedures, with bounded encodings and local work. Their resumable programs retain declared interface requests and responses under the saved-invocation procedure in Section 2.13.

The application core $\mathcal{M} := (\mathcal{S}, S_0, \delta, \text{Query})$ has $S_0 \in \mathcal{S}$ and the complete signatures

$$\begin{aligned} \delta : \mathcal{P} \times \mathcal{S} \times \mathcal{R} &\longrightarrow \mathcal{S} \times (\text{Applied}(\text{Outputs}) \sqcup \text{Rejected}(\text{Errors})), \\ \text{Query} : \mathcal{S} \times \mathcal{Q} &\longrightarrow \text{Answers}. \end{aligned}$$

`Outputs` and `Answers` are the application’s encoded success-output and query-answer domains, and rejection reasons lie in `Errors`. Rejection preserves the complete input application state S . Both programs are total and deterministic. Bootstrap performs its actual registration, synchronization, prefix read and checks of the initial application state S_0 , initializing the application to S_0 with version zero only on success. P_0 is the runtime result.

Configured transaction construction. `TransactionCreation(x)` is resumable and may use its configured randomness. It uses the saved approved owner route, reuses a retained usable AsyncLS certificate or obtains one through $\mathcal{G}_{\text{auth}}.\text{Sign}(p, x)$, and performs the configured transaction-format, authorization, funding, fee, size and ledger-nonce checks where applicable. It uses only the authenticated data returned by its declared constructor interfaces or saved from earlier preparation. If the constructor detects a violation, including insufficient funding or failure of its

selected fee policy or budget, it returns an operation error. Checks dependent on final signed size complete before construction returns a usable transaction. Issued constructor requests, including authentication and transaction signing, produced signatures and disclosed results remain saved on failure. Reuse preserves the completed transaction. Fee replacement is not an automatic retry.

Construction neither reserves funds nor guarantees admission (Section 3.4). An operation error proves neither permanent ledger invalidity nor AsyncLS expiry. Stricter relay or fee-selection preferences are not ledger-validity rules (Appendix B.1). The selected ledger transition determines fees, including whether any are charged.

TransactionParsing(tx) deterministically recovers $(x, w, txid)$ from complete bytes or rejects them. It recomputes the hash identifier specified below rather than trusting a supplied identifier. $\prec_{id}$ is the configured decidable total order on the hash-output domain. Complete transaction equality remains byte equality. Hash equality supplies only the probabilistic binding guarantee below and never replaces an exact byte comparison in a save or reuse check.

Transaction identifiers. Let λ be the security parameter. The authenticated transaction-format configuration uses an efficient collision-resistant hash family $(HGen, H)$ with output length $\ell_{id}(\lambda)$ and a public index $hk \leftarrow HGen(1^\lambda)$ sampled independently of the environment's coins before transaction processing. Replicas and both compared systems share the index, serialization and hash program. Define

$$\begin{aligned} id_{hk}(b) &= H_{hk}(\text{Encode}(\text{ALS}/txid, b)), \\ tx.id &= id_{hk}(\text{ser}_{full}(tx)). \end{aligned}$$

Here $\text{ser}_{full}(tx)$ contains the complete canonical bytes, including every encoded field, signature and witness. The derived accessor is not itself serialized. Constructors compute it from finished transactions, and parsers recompute it from received bytes without normalizing distinct accepted encodings. The hash is defined even on rejected bytes. This public local computation adds no runtime interface or coordinate of ζ .

For each fixed complete context $\mathcal{C}$ of Section 2.6, choose a polynomial cutoff for its computation and encoded data, with overrun event $\text{Tail}_{\mathcal{C}}$ of probability at most the negligible $\rho_{\mathcal{C}}(\lambda)$. Let $t_{\mathcal{C}}$ be a polynomial budget that also covers identifying a collision in that bounded execution. Collision resistance supplies a negligible bound $\beta_{id, \mathcal{C}}(\lambda)$ on collision-finding success for any algorithm running in time at most $t_{\mathcal{C}}$, over honest index generation and the algorithm's coins. This bound permits adaptive inputs after the index is disclosed. Write $\text{Bad}_{id, \mathcal{C}}$ for a collision between distinct complete transaction byte strings processed in $\mathcal{C}$. Then

$$\begin{aligned} \Pr[\text{Bad}_{id, \mathcal{C}} \wedge \neg \text{Tail}_{\mathcal{C}}] &\leq \beta_{id, \mathcal{C}}(\lambda), \\ \Pr[\text{Bad}_{id, \mathcal{C}}] &\leq \epsilon_{id, \mathcal{C}}(\lambda) := \min\{1, \rho_{\mathcal{C}}(\lambda) + \beta_{id, \mathcal{C}}(\lambda)\}. \end{aligned} \tag{1}$$

The budgets depend on the complete context, including all instances sharing the index. This bound does not assert global injectivity. If a joint bound already

charges the same tail, add only $\beta_{\text{id}, \mathcal{C}}$. Exact comparisons of the common programs retain zero error, including on collision executions (Lemma 2).

The record decoder consumes a complete authentic ledger prefix P obtained by the existing saved read. Well-formedness requires a deterministic causal interpretation of its ordered block boundaries and complete records. Write $\text{Items}(P)$ for that sequence before application decoding, retaining every boundary, record and its position. The output has the following type:

$$\begin{aligned} \text{decodeRecords} &: \text{Prefixes} \rightsquigarrow \text{DecodedItems}^*, \\ \text{DecodedItems} &= \{\text{Boundary}(\ell)\} \sqcup \{\text{Record}(\ell, j, \rho, \kappa)\}, \\ \text{recordFields}(\rho) &= (tx, \text{rid}_L, P_{\text{sub}}), \quad \kappa = (d_{\text{before}}, d_{\text{after}}, \text{events}, \text{links}). \end{aligned}$$

Here ℓ, j are the ledger height and record position. Every block has a boundary, including empty blocks. ρ retains the complete selected **Blockify** record. Its fixed projection recovers the complete transaction, ledger occurrence meta-data and actual submitting party. The format determines the source-state and event domains: $d_{\text{before}}, d_{\text{after}} \in \text{SourceStates}$, $\text{events} \in \text{Events}^*$, and $\text{links} \subseteq \{1, \dots, |\text{events}|\} \times \{(\ell, j)\}$ relates events to this occurrence. Source event identifiers remain distinct from $tx.\text{id}$. These fields describe views of decoded data.

Causal decoding and resumption. For preceding decoder state d and next item a , the fixed transition F produces a new state and one decoded item, $(d', u) = F(d, a)$. It has bounded resumable local work and preserves the item's kind, position and complete record. A boundary may update source bookkeeping. At a record, d_{before} and d_{after} are the source-state projections before and after this transition, and its ordered events and links depend only on (d, a) . The state contains all earlier source information needed by the next transition. Neither F nor the emitted context may inspect a later item, the supplied prefix's final height or end marker, the chosen scan partition, or mutable data outside the declared authentic input. Any source preprocessing must satisfy this same causal condition. Required record data must be recoverable from the preceding state and current complete record.

Let $\text{Run}_F(d, A) = (d_A, U_A)$ be the iterated transition on a finite item sequence A , and let $\parallel$ denote concatenation. The empty sequence leaves d unchanged and emits nothing. Deterministic iteration gives the resume law

$$\begin{aligned} \text{Run}_F(d, A) &= (d_A, U_A), \quad \text{Run}_F(d_A, B) = (d_{AB}, U_B) \\ \implies \text{Run}_F(d, A \parallel B) &= (d_{AB}, U_A \parallel U_B). \end{aligned}$$

Induction on $|B|$ applies the same F to the same state and next item at each step. Source initialization fixes one state at the authenticated anchor, obtained by the configured causal scan of P_0 or authenticated initial source data. For reconstruction, $\text{decodeRecords}(P)$ denotes iteration on the suffix after this common anchor from that state. The same laws apply to a bootstrap scan from its fixed initial source state. Since $\text{Items}(P)$ is a prefix of $\text{Items}(P')$ whenever $P \preceq P'$, the same induction proves

$$\text{decodeRecords}(P') \upharpoonright_P = \text{decodeRecords}(P).$$

Restriction retains exactly the boundaries and records in P , including complete ρ , both source states, events and links. Each emitted context is immutable. The scan retains the decoder continuation and any emitted item whose **AsyncLS** processing is unfinished across activations, so resumption neither decodes that item twice nor changes its context. A **NoBinding** abort discards staging. A later recovery starts from committed decoder state and redecodes the suffix, obtaining the same contexts by the deterministic resume law. An end marker controls only complete-prefix commitment. A data-only format satisfies the law by echoing each item with singleton source state and empty events/links. Section 5.1 gives the selected ORDI fold and its external input assumptions.

For $u = \text{Record}(\ell, j, \rho, \kappa)$, let $\text{context}(u) = (\ell, j, \rho, \kappa)$ and **RecordContexts** be its image. Then

$$\text{MatchesRequest} : \text{Transactions} \times \text{ALSRequests} \times \text{RecordContexts} \longrightarrow \{0, 1\}.$$

ALSRequests is the fixed encoded type of x . The notation $\text{MatchesRequest}(tx, x)$ suppresses $\text{context}(u)$ for that same authentic record and its complete tx .

The resumable decoder processes every record, including those later skipped by **AsyncLS**, and retains its cursor, state and immutable pending item across activations. Invalid returned prefixes cause operation failure. Failed **AsyncLS** parsing/context checks skip the record. Delivered **NoBinding** aborts the recovery attempt under the recovery-ownership procedure in Section 2.13, preserving the **AsyncLS** committed checkpoint and decoder state. Decoding itself issues no ledger/authentication/setup request and deducts no external fee.

Operations and observations. The ideal applies the release and send checks in Section 2.13 and the submission-permission procedure in Section 2.13. Under the transaction-disclosure procedure in Section 2.13, the ideal emits the adversarial **PendingTx** observation once when it first saves a transaction-table entry. Authentication observations belong to $\mathcal{G}_{\text{auth}}$ (Appendix A.1). All other observations retain their declared interfaces, independently of the implementation.

Session identity and fixed configuration. Definition 2 uses the logical session identifier directly. The configuration fixes the ledger, clock, authentication, board and constructor-specific interfaces, application and interpretation, numerical parameters and service operations, including each replica's configured owner/verifier/ledger-party connections and permissions. This binding is immutable, but does not equate different replicas' execution identities or connections. Their common reconstruction data are the authenticated (sid, P_0) , selected ledger and ζ , application domains and initial state, decoder and its anchor state, parsing and context predicates, transition and query programs, transaction-identifier configuration, and historical-authentication interpretation with consistent completed decisions. The deterministic reconstruction programs use only their declared inputs, not the observing replica's identity or private local history. Ledger-recorded submitters and occurrence metadata remain inputs and are never replaced by that replica's identity. Each replica maintains its own state and saved operations.

Every real/ideal comparison preserves the corresponding replica’s actual identities and connections exactly. A checkpoint or asserted identifier alone does not establish this configuration. Setup checks the received session identifier and authenticates the anchor under that configuration (the saved-invocation procedure in Section 2.13). Its public output remains (sid, P_0) .

Authorization and representation. Section 2.10 specifies approval of complete x and its protected signing route. The submission-preparation procedure in Section 2.13 governs preparation ownership, certificate and transaction retention, retries after all constructor requests issued by the unsuccessful preparation have completed and that preparation has halted without a transaction, and independent nonce-conflicting requests. A returned signing error requires another authorized $\mathcal{G}_{\text{auth}}.\text{Sign}$ invocation.

For represented ledger effects, x authenticates the complete transaction/effect context, for example as a structured field χ_x of its nonce. A deterministic $\text{MatchesRequest}(tx, x)$ checks the complete included bytes and actual ledger data. A template factors out the embedded request/certificate slots to avoid recursive encoding. Every remaining field affecting authorization, effect or validity is bound. Transactions for the same x from other instances or senders may vary only in fields proved irrelevant to those semantics. Changing protected context requires new approval and a new request. At anchor height b_0 , every represented effect must obey the common first-outcome rule and the upper height $b_0 + D(x)$, where $D(x) = h + \min(W, W_L)$. A finite W_L must therefore be enforced by the selected format, validation and effect requirements. Locktimes that set only the earliest valid time or height, and mempool eviction, provide no such upper bound [9]. Data-only carriers have only the interpreted AsyncLS effect and may use $W_L = \infty$.

Request authentication during reconstruction. The configured application interpretation fixes a procedure for checking request authentication with bounded work per step. For each structurally eligible record, its inputs are its decoded (p, x, w) , its ledger occurrence, and the authenticated setup and history. It reads no private approval or submission log. Its local steps are deterministic. External results come only from its declared authentication/history queries and their saved response connections. Acceptance must establish a positive result from $\mathcal{G}_{\text{auth}}.\text{Verify}(p, x, w)$ for the actual owner, either by consuming such a result or through an immutable authenticated record proving an earlier check on the same request and certificate. A completed Boolean rejection skips the record. Consuming a delivered NoBinding ends the current recovery with Abort and discards its staging under the recovery-ownership procedure in Section 2.13. The no-binding result fixes no authentication decision, skips no record and changes no committed prefix. A later public call may start a new recovery from the AsyncLS committed checkpoint with new queries. That recovery may repeat local application computations from the aborted recovery, but publishes no discarded update and repeats no ledger submission. The ideal-service requirement is that completed decisions for the same record in an anchored history agree across scans, including scans on different replicas’ configured verifier connections, and remain unchanged under

extensions. An implementation must prove that its checking procedure satisfies this requirement. This consistency requirement concerns the authentication results or records and the declared authentication/history interfaces that supply them.

All implementations follow Figure 2's ordered update of (S, L, O) , including complete-prefix commitment. An aborted recovery discards its staged state. Queries read AsyncLS committed checkpoints. Section 2.15 proves the resulting reconstruction invariant.

2.12 Ideal functionality

Figure 2 specifies the monolithic ideal. The complete saved-operation rules that follow also govern the execution of the reference protocol in Section 3.

Description of the ideal functionality $\mathcal{F}_{\text{AsyncLS}}^\zeta$:

Participating roles: The monolithic AsyncLS service on its configured caller and interface connections.
Corruption model: Honest ledger party. Retained interface policies (Section 4).
Parameters: $\zeta = (W, W_L, d_{\min}, d_{\max})$ in the fixed configuration.

Operation-structure adaptation of the monolithic $\mathcal{F}_{\text{AsyncLS}}^\zeta$:

Implemented role(s): The public AsyncLS service, owning its state and saved operations.
Subroutines: $\mathcal{G}_{\text{LEDGER}}$, $\mathcal{G}_{\text{clock}}$, $\mathcal{G}_{\text{auth}}$, the registration board and declared constructor/setup interfaces, including transaction signing when used.
Initialization:
 1 Obtain validated (P_0, S_0) by authenticated setup (Section 2.8) and bind (sid, P_0) once. Initialize $H = 0$, $C = (P_0, (S_0, \emptyset, \emptyset), c_0)$, $E = \emptyset$, idle recovery and empty queues. Return (sid, P_0) and enable public ports.
Main: public operations:
 2 **on** Register() from p : invoke the owner's $\mathcal{G}_{\text{auth}}.\text{Register}(p)$ and return its result.
 3 **on** Read(q) from p : validate q , complete its recovery or abort, and save/return $(\text{Query}(S_H, q), c_H)$ from one AsyncLS committed checkpoint.
 4 **on** Submit(ν, r, c_h) from p : validate caller/encoding and start clock-read. Save its first result once as t_0 . Preparation may proceed meanwhile.
 (a) Form $x = \text{Encode}(\text{ALS/request}, \text{sid}, p, \nu, r, c_h)$. Return terminal status. Otherwise recover through h , abort on error, and obtain the owner's Registered result.
 (b) With fresh pending status, atomically reuse the retained transaction, join preparation for x , or own TransactionCreation(x). Retain completed tx before yielding. After this invocation's t_0 arrives, parse tx as (x, w, txid) and invoke $\mathcal{G}_{\text{auth}}.\text{Verify}(p, x, w)$. Continue on 1 and abort on 0 or NoBinding.
 (c) Recheck status. If pending and $E[x] = \perp$, require unused txid, save $E[x] = (tx, t_0, t_0 + d_{\min}, \text{Waiting})$, grant one initial send, emit PendingTx(sid, txid, tx, t_0 , $d_{\min}$, $d_{\max}$) once and yield. Identical reuse preserves $E[x]$ with Section 2.13's permissions. Conflicts abort. Return Pending with terminal priority.

- 5 **on** $\text{Check}(x)$ from p : validate x and return known terminal status. Otherwise start/join recovery and return committed status or a recovery error at the next runnable reply step, independently of outstanding recovery responses.

Adversarial interface:

- 6 **on** $\text{SetDelay}(\text{sid}', \text{txid}, d)$ from the adversary: locate $E[x]$ by $\text{tx.id} = \text{txid}$. Require valid encoding, correct session, known identifier, $s = \text{Waiting}$ and $d \in [d_{\min}, d_{\max}]$ before/after reading clock t . Reject failures. If $T \leq t$, reject and process due work. Otherwise, set $T \leftarrow t_0 + d$ only if $t_0 + d \geq t$.

Recovery and application update:

- 7 Obtain an authentic extension of P_H under Section 2.13's single recovery owner. Invalid views abort without changing C . Process the fixed target in order, setting only $S.\text{ver} = i$ at each block boundary.
- 8 For decoded record u at position j , parse tx as $(x, w, tx.\text{id})$. Skip unless x is well-formed, $0 \leq h < i \leq D(x)$, $O[x] = \perp$, and $\text{MatchesRequest}(tx, x, \text{context}(u)) = 1$. Establish $\mathcal{G}_{\text{auth}}.\text{Verify}(p, x, w) = 1$ by its result or an immutable authenticated record of that check. Skip on 0. **NoBinding** aborts recovery under Section 2.13.
- 9 Set $k = (\text{sid}, p, \nu)$. If $L[k] \notin \{\perp, x\}$, preserve S, L and set $z = \text{Rejected}(\text{NonceConflict})$. Otherwise set $L[k] = x$ and $(S, z) \leftarrow \delta(p, S, r)$. Record $O[x] = (\text{gid}, x, tx.\text{id}, (i, j), z)$ atomically. Commit only the complete prefix with unchanged base.

Release and ledger submission:

- 10 On a permitted release poll, process $s = \text{Waiting}$, $T \leq t$ in $(T, tx.\text{id})$ order. Terminal status sets **Suppressed**, otherwise set **Released** and queue the initial send.
- 11 For an authorized send under Section 2.13, output $(\text{submit}, \text{gid}, tx)$ as the configured party and permanently set **Submitted**.

Fig. 2. Operation structure of $\mathcal{F}_{\text{AsyncLS}}^{\zeta}$. Saved operations follow Section 2.13. Complete rules appear in Section 2.13.

2.13 Saved operations and transaction-submission rules

Description of the complete AsyncLS operation rules:

Participating roles: The ideal has one monolithic service role. The reference assigns public service/signing to **Wallet**, coordination to **Relay**, interface requests to **Adapter** and reconstruction to **Replay**, on their existing connections.

Corruption model: The honest service ledger identity and the retained interface corruption policies of Section 4. The following boxes adapt common procedures, without declaring a new machine/address partition.

Protocol parameters: $\zeta = (W, W_L, d_{\min}, d_{\max})$ and the fixed configuration of Section 2.

Subroutines and state: The ledger, clock, authentication, registration board and explicitly declared constructor/setup interfaces keep their permissions. The records below are local state.

Procedure adaptation for saved invocations and returns:

Implemented role(s): The monolithic ideal, or the existing **Wallet/Relay/Adapter/Replay** recipients of the saved invocation.

Subroutines: The specific ledger, clock, authentication, board and constructor/setup interfaces named by that invocation.

Initialization: The common authenticated setup of Section 2.8.

EntityInitialization: The reference uses **Adapter** for ledger-party setup, then the authenticated **Replay.Initialize** exchange. The monolithic ideal performs authenticated setup and state initialization directly. Both retain one completed setup for fixed (sid, P_0) , its actual setup connection and existing trigger, with no extra per-caller activation.

Internal state, saved responses and initialization:

- 1 Public AsyncLS inputs are checked at the monolithic ideal's entry points and at the reference **Wallet**'s entry points before dispatch to authentication or any other declared interface. Malformed inputs and inputs on unauthorized caller connections return **Abort**.
- 2 The saved-operation rules in this paragraph apply to authentication and public AsyncLS invocations, exchanges among **Wallet**, **Relay**, **Adapter** and **Replay**, and requests to the declared ledger, clock, registration-board and constructor-specific interfaces.
 - (a) Every invocation saves its input, actual caller/callee, return connection, expected stage and issued requests. This saved operation record is a continuation in the execution model. The issuing operation retains its continuation, and each outstanding request occupies its response connection until completion.
 - (b) A step requiring an interface result becomes runnable only after its matching response has been saved. A private request number is not added to an interface that does not return it. Only a response on that connection at the expected stage, with the fields repeated by the invoked operation, may be saved and consumed once. Responses from the wrong connection, stale or malformed responses, and results already used have no effect.
 - (c) An invocation returns at most once, closing its return connection without cancelling issued requests. A later input is a new invocation. Each admitted

activation advances one bounded runnable step in queue order. Issuing any of these requests yields wherever the execution model requires it.

- 3 For an **AsyncLS** invocation, authorized transaction submission and reconstruction that has not aborted may continue after its public return, with responses returning to their original saved operation records.
 - (a) Consuming delivered **NoBinding** produces the existing **Abort** result under the rules below and stops the affected work. A later public input can retry.
 - (b) For this error, the aborted operation retains completed transactions and the records needed to consume already issued responses once on their original connections. The abort transition occurs when the caller consumes the delivered result, not at its earlier production by $\mathcal{G}_{\text{auth}}$. Those responses cannot restart its preparation, verification, recovery or enqueue operation, and its closed return cannot reopen.
- 4 Before public ports are enabled, the setup driver uses **Adapter**'s ledger registration, synchronization and prefix reads to obtain P_0 . It passes (sid, P_0, S_0) through **Relay** to **Replay.Initialize**, which checks that the received **sid** is this session's identifier (Definition 2), validates the authenticated anchor and the application's configured initial state S_0 , and initializes $H = 0$, $C = (P_0, (S_0, \emptyset, \emptyset), c_0)$ once. It returns (sid, P_0, c_0) . An exact repeat preserves H, C and conflicting input aborts. Only completion enables **Wallet** and the public setup result (sid, P_0) .

Procedure adaptation for submission preparation:

Implemented role(s): **Wallet** owns public invocation, signing and preparation. **Relay** owns enqueue/scheduling. The ideal performs the same work directly.

Subroutines: **Relay**, **Replay**, **Adapter** and the declared owner authentication and constructor/signing connections, at the steps below.

Main:

- 1 **Wallet** owns **Register**, **Read**, **Submit** and **Check** and the actual owner connection. **Register** returns $\mathcal{G}_{\text{auth}}$. **Register**'s declared result.
 - (a) A correctly encoded **Submit** saves x with $t_0 = \perp$ and issues its first **Adapter.Time** request. Its actual **clock-read** response sets t_0 once on that invocation's saved connection. Preparation may proceed while this timestamp is unset, but **Enqueue** waits for it. A terminal reply has priority at its next runnable reply step and closes no issued clock request. After terminal return, a late response still belongs to its original saved operation and causes no new read or enqueue.
 - (b) **Relay**'s **PreSubmitCheck** (x, h) obtains **Replay**'s atomic $(H, c_H, \text{Status}_H(x))$. Terminal status returns immediately. Otherwise it starts or resumes recovery until $H \geq h$. An invalid view causes this recovery to abort. A **NoBinding** recovery abort ends this **PreSubmitCheck** and its **Submit** with **Abort**, subject to terminal priority. In particular, an earlier clock read survives an abort on its original connection, but its late result cannot enqueue this invocation.
 - (c) A later permitted activation may request another insufficient-prefix read.
 - (d) Each invocation obtains its own positive owner **Register** result before joining preparation. The monolithic ideal issues and retains the same first clock read directly.

- 2 **Wallet's** saved continuations coordinate construction by complete x .
 - (a) After a fresh status read, the following choice is atomic and precedes construction: return current terminal status. Otherwise reuse the permanently saved completed transaction, join the current preparation if one exists, or record this invocation as its sole owner and start **TransactionCreation**(x). Suspension before the choice requires a fresh status read.
 - (b) An owner keeps its approval, issued constructor requests and returned preparation results. Joining creates no constructor or additional ledger/authentication/setup/signing request. Constructor completion atomically retains its usable transaction in that continuation before any yield, endpoint verification or $E[x]$ save. That transaction remains immutable for the instance's lifetime.
 - (c) Completion makes waiters runnable at their existing queue positions, without creating an activation or a reply from the ledger, authentication, setup or signing interface. A joiner resumes from this internal object, rechecks status and continues on its own connections. A waiting invocation may return newly known terminal status at its next runnable reply step.
- 3 If a construction operation fails before producing a usable transaction, its owner retains ownership until the issued constructor requests have completed and the construction continuation has halted without a transaction. Only then may a later actual **Submit** own another construction operation, retaining usable certificates and other completed preparation results as data, never as responses to its own constructor requests. Waiters on the failed construction receive an operation error on their own return connections, subject to terminal priority.
- 4 Endpoint verification follows retention of the completed transaction, so its **NoBinding** error never releases ownership for another construction. Returning to the owner or a waiter never clears the preparation or cancels issued constructor requests: late responses stay with their originating continuation, and completed transactions are retained there before later status/reply checks. Once a completed transaction exists, no error in verification, saving or delivery and no terminal reply permits another construction.
- 5 Different x , including nonce-conflicting requests, remain independent. No local step reserves L . Ledger processing selects the first eligible outcome, and expired or late records claim no nonce.
- 6 **Enqueue** accepts only requests from an authorized saved **Wallet** operation with that request's completed transaction and its set t_0 . It validates the complete transaction and obtains that invocation's actual endpoint **Verify** result.
 - (a) Terminal status takes precedence before further preparation and when replying. While the request is pending, verification result 1 permits the atomic save rule of Section 2.4. Verification result 0 aborts this invocation. Delivered **NoBinding** marks this invocation aborted and returns **Abort** on its own open return connection, subject to terminal priority. It saves no $E[x]$, grants no permission and initiates no send. Existing $E[x]$ and other invocations' authorized work are unchanged.
 - (b) The same completed transaction remains retained, but this invocation has no verification continuation. A later actual **Submit** may reuse it after its own registration, first clock sample and fresh endpoint **Verify**.

- (c) First saving $E[x]$ uses the saving invocation's own first clock sample and checks identifier use throughout E . A construction joiner does not inherit the constructor owner's timestamp. An existing identical record preserves all fields.
- (d) Successful scheduling returns **Pending** while its authorized submission work remains active. Sharing construction does not waive any invocation's registration, verification or send-permission checks. The monolithic ideal performs these same saved-operation updates itself.

Procedure adaptation for ledger, clock, verification and reconstruction operations:

Implemented role(s): **Adapter** issues interface requests and **Replay** reconstructs state. The ideal owns the corresponding operations.

Subroutines: The actual ledger, clock and $\mathcal{G}_{\text{auth}}$ connections specified below.

Main:

- 1 **Adapter's** **Time()** and **ReadPrefix()** return the actual clock value or validated authentic prefix on their saved connections. Its **Verify**(p, x, w) returns the actual verification result of 0 or 1 or **NoBinding**.
- 2 **Replay.Recover**(P) stages the specified processing, returns (H, c_H) only after complete commitment, and returns **Abort** if that recovery consumes a delivered **NoBinding**.
- 3 **Query**(q) saves $(\text{Query}(S_H, q), c_H)$ at evaluation, and **Status**(x) returns $(H, c_H, \text{Status}_H(x))$ from one **AsyncLS** committed checkpoint. Later checkpoint updates cannot refresh a saved query pair.
- 4 The affected operation aborts on malformed inputs or invalid views without changing the **AsyncLS** committed checkpoint C .

Procedure adaptation for recovery ownership and monotone commitment:

Implemented role(s): **Replay** is the reference recovery owner. The monolithic ideal owns its own recovery.

Subroutines: The configured authentic-read and verifier connections, through **Adapter** in the reference.

Main:

- 1 Each service instance has one recovery owner: **Replay** in the reference and the monolithic ideal itself in the ideal execution.
 - (a) **Read**, **Check** and **PreSubmitCheck** create demands in the existing operation queue, not separate staged reconstructions. A demand retains its originating operation, actual response connection, kind and required target.
 - (b) A **PreSubmitCheck** demand requires committed height at least h unless its request becomes terminal. A **Read** demand waits for its assigned authentic read/recovery to complete. A nonterminal **Check** demands a bounded recovery step but retains an independently runnable status-reply step. Joining a demand uses no second read or verification response.
- 2 The owner is either idle, waiting for its one issued recovery read, or processing one fixed authentic target prefix P^* .
 - (a) Its active record retains the base **AsyncLS** committed checkpoint C_{base} and decoder state, immutable target/end marker, staged decoder state and **AsyncLS** state (S, L, O) , cursor, any emitted unfinished item, issued authentication/history requests and their occupied response connections, and the waiting demands. Joining callers use this record. They neither copy its staging state nor take ownership of its responses.

- (b) A new **Read** joins the active target or outstanding read, or requests one read when recovery is idle. **PreSubmitCheck** joins while its height requirement is unmet. **Check** starts such a read only when recovery is idle and otherwise advances or joins the existing recovery. No second recovery read is issued while that read or scan is active.
 - (c) An insufficient completed prefix leaves a height demand waiting for a later permitted activation, as in the existing insufficient-prefix rule. Completion alone starts no further read.
- 3 A supplied prefix is accepted only from its saved authentic read/**Recover** operation and is checked against the prefix P_H in the **AsyncLS** committed checkpoint C .
- (a) If the prefix is older, malformed or incompatible, the affected operation aborts without changing C . Equality with P_H completes without staging.
 - (b) While a scan is active, a supplied prefix contained in its target joins that scan, and a strict extension is saved with its originating operation in queue order for later processing. This also defines the case of a response to an already issued operation arriving during a scan.
 - (c) No later prefix replaces P^* , its end marker, pending item or occupied verification connection. Responses are consumed only by the operations that issued them, even if all public query returns have closed.
- 4 On consuming **NoBinding**, the owner terminates its active recovery attempt without skipping the record or committing any staged application or decoder state.
- (a) It discards the target, cursor, unfinished item, staging and queued extensions attached to that attempt, and preserves the **AsyncLS** committed checkpoint C and its decoder state. Every unresolved demand attached to this attempt, including one waiting for a later extension, is marked failed and detached at its existing queue position.
 - (b) An open **Read** returns **Abort**. An open **PreSubmitCheck** returns that error to its originating **Submit**, and an open **Check** returns that error, in both cases subject to freshly committed terminal status taking priority. Already satisfied demands keep their saved results. A closed **Check** has no further output.
 - (c) The owner returns to idle, with no suspended scan or automatic retry. Its issued requests still occupy their original connections until their actual responses are consumed once by the aborted operation, without restarting work.
 - (d) A later actual **Read**, nonterminal **Check** or height-demanding **Submit** may start or join a new attempt from current C , using a fresh authentic read when idle and new verification invocations on available connections. Every record after the prefix stored in C , including the record that returned **NoBinding**, is processed again. No failed demand or later response starts this retry by itself.
- 5 Only the recovery owner may commit.
- (a) After all items through P^* and all required checks finish, it checks that the current C is still C_{base} and that P^* extends its prefix, then atomically commits the complete **AsyncLS** and decoder states. Thus committed height never decreases.
 - (b) If the base check fails, the recovery owner leaves C unchanged and reports an operation error. In executions of these rules the check succeeds because there is only one writer.

- (c) Satisfied demands become runnable at their existing queue positions and keep their own saved return connections. A **Read** evaluates and saves its answer/checkpoint from one resulting **AsyncLS** committed checkpoint. A **PreSubmitCheck** rechecks its height and terminal condition. A closed **Check** return is never reopened.
- (d) At a later permitted activation, the next saved strict extension starts from the current committed decoder and **AsyncLS** states, and a saved target already covered by the prefix stored in C completes without another scan. None of these transitions creates an external response, activation or send permission.

Procedure adaptation for independent status queries:

Implemented role(s): **Wallet** and **Relay** retain the query connections. **Replay** owns the shared recovery. The ideal retains both directly.

Subroutines: The existing status, recovery, ledger-read and authentication operations below.

Main:

- 1 Each **Check** validates its request and returns known terminal status immediately. Otherwise it starts one recovery on its separate connection for checking status, or enables a bounded step of an existing one, and rechecks committed status at its next runnable return step.
- 2 It may return **Pending** before an issued ledger read or $\mathcal{G}_{\text{auth}}.\text{Verify}$ invocation completes. Its reply step is scheduled independently of recovery's response-delivery steps. Closing the query leaves recovery active if it has not aborted, with no right to answer the closed connection again.
- 3 If that recovery consumes **NoBinding**, it terminates under the preceding demand rules. An open query reports the error unless committed terminal status has priority, while a closed query keeps its earlier reply. A later poll creates a new query.
- 4 All three recovery-demand kinds share the single staged reconstruction of the recovery-ownership procedure in Section 2.13 without duplicating ledger reads or authentication verification. A recovery error may abort the operation but never certifies request expiry.
- 5 **Read** instead waits for its required recovery and returns its saved query/checkpoint pair. Owing or joining preparation, and waiting for transaction submission, do not block independent runnable query work.

Procedure adaptation for transaction disclosure:

Implemented role(s): **Relay** saves reference transactions. The monolithic ideal performs its own save and disclosure.

Subroutines: The saving invocation retains its existing interfaces. $\mathcal{G}_{\text{auth}}$ emits its own observations.

Main:

- 1 First saving $E[x]$ under an identifier unused throughout E emits **PendingTx**(**sid**, **txid**, tx , t_0 , $d_{\min}$, $d_{\max}$) once and yields. Reusing a transaction or joining preparation emits no further **PendingTx** and creates no $\mathcal{G}_{\text{auth}}.\text{Sign}$ invocation. The caller separately receives **Pending**. The default target needs no acknowledgment. Authentication observations come from $\mathcal{G}_{\text{auth}}$ itself (Appendix A.1). Independently visible ledger, clock and setup interactions retain their own interfaces.

Procedure adaptation for release and send checks:

Implemented role(s): **Relay** schedules releases and **Adapter** issues protected ledger sends. The ideal performs both directly.

Subroutines: The configured clock and ledger connections, with the previously obtained authentication result and send permission.

Main:

- 1 **Relay** keeps E and the queue. **Adapter** retains the saved requests, responses and connections, recorded directly by the ideal. The first save sets $T = t_0 + d_{\min}$ and $s = \text{Waiting}$ under Section 2.4's uniqueness/reuse rule. A public input, admitted scheduling input or saved non-poll result permits progress. With transactions awaiting release and no release poll outstanding, the next runnable release operation calls **Time** to test saved targets. Poll completion alone starts no new poll. The clock supplies no alarm.
- 2 **SetDelay**($\text{sid}', \text{txid}, d$) locates the unique record $E[x]$ with $\text{tx.id} = \text{txid}$ and applies Figure 2's checks before and after its actual clock read.
 - (a) A target with $T \leq t$ is due and cannot be revised. Otherwise an admissible d with $t_0 + d \geq t$ replaces only T .
 - (b) Process due records in global $(T, \text{tx.id})$ order, breaking ties by $\prec_{\text{id}}$. In one bounded step, check the target and the request's terminal status, then mark the record **Suppressed** if terminal, or mark it **Released** and queue the initial send otherwise. The submission-permission procedure in Section 2.13 governs this permission and later repeats. Every runnable send uses the checks below.
- 3 **Deliver** accepts requests only from an already authorized saved operation on its protected connection with a positive verification result on that endpoint's verifier connection, the saved complete transaction, $s \in \{\text{Released}, \text{Submitted}\}$, pending committed status and a free route, and that invocation's unused submission permission.
 - (a) Acceptance consumes that permission atomically before any yield and retains that accepted delivery until completion. Refusal after acceptance does not restore it.
 - (b) The operation emits $(\text{submit}, \text{gid}, \text{tx})$ through its configured ledger party and sets $s = \text{Submitted}$ permanently when the transaction is sent, then returns **Emitted** privately. The operation returns **Refused** for refusal before that ledger-submission output, without recording a transaction emission.
 - (c) Neither is an acknowledgment of ledger admission: the ledger allocates its own occurrence and notifies its adversary on each actual input regardless of admission. A terminal result learned later cannot cancel a ledger submission already accepted for execution.

Procedure adaptation for submission permissions:

Implemented role(s): **Relay** grants the reference permission and **Adapter** consumes it. The ideal performs the corresponding updates.

Subroutines: The original owner **Register**, endpoint **Verify** and protected **Deliver** connections specified below.

Main:

- 1 First saving $E[x]$ grants its invocation permission for one initial send, usable after release. A separate actual **Submit** that successfully reuses a released or submitted transaction gains one further send after its own positive owner **Register** and

endpoint **Verify**. Each invocation applies this rule once at successful save/reuse. Waiting reuse and joining alone grant none. Busy routes retain unused permission, consumed by **Deliver** as specified above. Resumption, queries and returned ledger prefixes, clock samples or authentication results grant no extra send.

The atomic owner/join choice prevents overlapping constructors for x , and permanent retention of the completed transaction prevents a second completed transaction. Only the first save of $E[x]$ writes its transaction and the saving invocation's sampled t_0 . That save and admissible **SetDelay** are the only target writes. Initially E is empty. A first save tests that its hash identifier is unused throughout this E , and reuse compares the complete bytes without changing the saved record. No later transition changes a saved transaction or its identifier. Induction therefore gives pairwise distinct identifiers among this instance's saved records, even if the hash collides on other transaction bytes. A colliding new transaction aborts at the existing save check and does not obtain a new send permission. This local invariant makes **SetDelay** lookup and the release order unambiguous. It supplies no global injectivity across ledger transactions or replicas. The same induction preserves the original interval, prevents reopening a due transaction or one no longer waiting, and makes every transaction output occur no earlier than $t_0 + d_{\min}$. Sending the same transaction again preserves these facts. Under the request-arrival clock condition this is also the lower bound $\tau_{\text{arr}} + d_{\min}$ for that saving invocation. These invariants are used in Lemma 2 and the submission-time bound in Section 3.2.

2.14 Application admissibility

The following five conditions collect the application obligations already imposed by Section 2.11's types and transaction procedures, Section 2.11's effect correspondence, and Figure 2's ordered update. They introduce no additional hypothesis for the general service results.

Fix a service parameter tuple $\zeta = (W, W_L, d_{\min}, d_{\max})$ and a session identifier sid . The application is specified by a domain configuration $\mathcal{D} = (\mathcal{P}, \mathcal{R}, \mathcal{Q}, \text{Outputs}, \text{Errors}, \text{Answers})$, the application core $\mathcal{M} = (\mathcal{S}, S_0, \delta, \text{Query})$ of Section 2.11, and the canonical transaction procedures

$$\text{TxOps} = (\text{TransactionCreation}, \text{TransactionParsing}, \text{MatchesRequest}).$$

The fixed configuration retains Section 2.11's bootstrap, causal decoder, constructor interfaces and authentication requirements.

Condition 1 (Core Determinism and Strict Rejection Invariance).

- **Totality and Boundedness:** *The transition function δ and query function Query are deterministic, polynomial-time bounded total functions with signatures:*

$$\delta: \mathcal{P} \times \mathcal{S} \times \mathcal{R} \rightarrow \mathcal{S} \times (\text{Applied}(\text{Outputs}) \sqcup \text{Rejected}(\text{Errors}))$$

$$\text{Query}: \mathcal{S} \times \mathcal{Q} \rightarrow \text{Answers}$$

- Rejection Invariance: *For all $p \in \mathcal{P}$, $S \in \mathcal{S}$, and $r \in \mathcal{R}$, if δ evaluates to a rejection, the output state remains strictly identical to the input state:*

$$\delta(p, S, r) = (S', \text{Rejected}(e)) \implies S' = S.$$

- Query Purity: *For all $S \in \mathcal{S}$ and $q \in \mathcal{Q}$, $\text{Query}(S, q)$ is read-only and produces no side effects on S .*

Rejection invariance concerns S alone: the service may still update L or O , and neither source-decoder transitions nor ledger fees are rolled back.

Condition 2 (Consensus Version Invariance).

- Every application state $S \in \mathcal{S}$ contains a designated integer field $S.ver \in \mathbb{N}$, initialized to $S_0.ver = 0$.
- Transition Invariance: *Application transitions strictly preserve the version field:*

$$\forall p \in \mathcal{P}, S \in \mathcal{S}, r \in \mathcal{R}, \quad (S', z) = \delta(p, S, r) \implies S'.ver = S.ver.$$

- Service-Controlled Progression: *Mutating $S.ver$ is the exclusive privilege of the ledger service, which enforces $S.ver \leftarrow i$ upon processing block boundary i .*

The version field records the relative ledger height processed by the service, independently of whether an application transition succeeds.

Condition 3 (Codec Invertibility and Probabilistic Identifier Binding).

- Deterministic Parsing: *Let $\mathcal{X} = \text{ALSRequests}$, and let $\mathcal{W}, \mathcal{I}$ be the configured certificate and transaction-identifier domains. The parsing algorithm $\text{TransactionParsing}: \{0, 1\}^* \rightarrow (\mathcal{X} \times \mathcal{W} \times \mathcal{I}) \cup \{\perp\}$ is deterministic. Every usable transaction tx completed by $\text{TransactionCreation}(x)$ satisfies $\text{TransactionParsing}(tx) = (x, w, tx.id)$ for its request certificate w .*
- Hash-Identifier Binding: *Apply Section 2.11's collision-resistance assumption to the common hash family, honestly generated public index and complete serialization. For every fixed admitted context $\mathcal{C}$, outside $\text{Bad}_{id, \mathcal{C}}$, any two encountered transactions tx, tx' with successful parses*

$$\begin{aligned} \text{TransactionParsing}(tx) &= (x, w, txid), \\ \text{TransactionParsing}(tx') &= (x', w', txid'), \end{aligned}$$

satisfy $txid = txid' \implies tx = tx'$. The unconditional failure probability is at most $\epsilon_{id, \mathcal{C}}$ from (1), over the shared index generation and execution coins. This computational requirement does not assert injectivity on all possible transactions or justify replacing exact byte comparisons by hash comparisons.

- Syntactic Request Matching: *The canonical predicate is deterministic on complete record contexts:*

$$\text{MatchesRequest}(tx, x, \chi) \in \{0, 1\}, \quad \chi \in \text{RecordContexts}.$$

It checks the complete transaction, request certificate and the full authentic record context $\chi = \text{context}(u)$ under Section 2.11's authorization/effect correspondence. The full context includes the occurrence height and position, source states, events and links.

Hash binding concerns complete transaction bytes processed in the fixed context, not injectivity from transactions to logical requests. Different identifiers and transactions may carry the same x subject to the protected-context requirements. The request-keyed $E[x]$ and $O[x]$ govern transaction retention and first outcomes separately, using complete bytes where transaction equality is required.

Condition 4 (Immediate-State Authorization).

- Let $u = (tx, \dots)$ denote the first eligible occurrence of request $x = \text{Encode}(\text{ALS/request}, \text{sid}, p, \nu, r, c_h)$ in a canonically finalized ledger prefix.
- Predecessor State Evaluation: Let $k = (\text{sid}, p, \nu)$. When $L[k] \in \{\perp, x\}$, application validity and output are determined at the complete application state S_{prev} immediately preceding the processing of u :

$$(S_{\text{next}}, z) = \delta(p, S_{\text{prev}}, r).$$

If $L[k] \notin \{\perp, x\}$, the service instead records $\text{Rejected}(\text{NonceConflict})$ without invoking δ or changing S_{prev} , as in Figure 2.

- Contextual Self-Containment: *Any authorization check executed by δ on behalf of caller p (e.g., account ownership, balance sufficiency, or access permissions) uses the supplied caller and request as inputs and must obtain all application-state information from S_{prev} , without anticipating later state changes.*

Application validity is evaluated at the complete state S immediately before the first eligible occurrence, after the nonce check. The application checks the authenticated caller's authority over its accounts and other fields in that immediate state, as in Section 2.2.

Condition 5 (Checkpoint Binding and Strict Expiry Soundness).

- Causal Checkpoint Injective Binding: *Checkpoint tokens*

$$c_h = \text{Encode}(\text{ALS/checkpoint}, \text{sid}, h)$$

injectively bind session identifier sid to relative ledger height $h \geq 0$. A request x referencing c_h is eligible for application processing only if $S_{\text{prev.ver}} > h$.

- **Hard Expiry Boundary:** *If $W_{\text{eff}} = \min\{W, W_L\} < \infty$, the last eligible relative height is $D(x) = h + W_{\text{eff}}$. For a record u at relative height i , the canonical eligibility tests require*

$$\begin{aligned} 0 \leq h < i \leq D(x), \quad O[x] = \perp, \\ \text{MatchesRequest}(tx, x, \text{context}(u)) = 1, \end{aligned}$$

together with the parsing, session and authentication checks. Thus $i > D(x)$ excludes application processing by the independent height test. Every separately represented ledger effect must also obey Section 2.11's first-outcome and upper-height correspondence. After complete processing to $H > D(x)$ with $O_H[x] = \perp$, the result $\text{Failure}(\text{Expired}, x, c_{D(x)+1})$ therefore certifies no represented application effect. Independent ledger fees are outside this conclusion.

These checks separate the request's checkpoint, its eligible height window and its first recorded outcome. The strict-expiry conclusion also depends on the stated correspondence for separately represented ledger effects.

Definition 3 (AsyncLS Admissible Application). *The complete application tuple*

$$\mathcal{A} = (\mathcal{D}, \mathcal{S}, S_0, \delta, \text{Query}, \text{TxOps})$$

includes the domains and transaction interpretation in addition to the core $\mathcal{M}$. We say that $\mathcal{A}$ is an admissible AsyncLS application with respect to (ζ, sid) , denoted by the predicate $\text{AdmissibleApp}(\mathcal{A}, \zeta, \text{sid}) = 1$, if and only if it satisfies Conditions 1–5.

2.15 Reconstruction and immutable outcomes

Lemma 1 (Reconstruction and immutable outcomes). *Fix a well-formed logical AsyncLS session with authenticated (sid, P_0) , the causal decoder of Section 2.11 and the historical-authentication procedure of Section 2.11. Reconstructions may run at distinct replicas sharing the session's common reconstruction data specified in the session-configuration paragraph of Section 2.11, on their own configured connections. For every complete authentic permanent $P_H \succeq P_0$, completion of the required Boolean authentication checks and sufficient permitted activations yield a unique reconstructed state $\Sigma_H = (S_H, L_H, O_H)$ and checkpoint c_H . If $\text{Rec}_a(P_H)$ denotes the committed pair produced by any completed reconstruction a , allowing scan partitions, resumption and preceding aborted attempts, then*

$$\forall a, b: \quad \text{Rec}_a(P_H) = \text{Rec}_b(P_H) = (\Sigma_H, c_H), \quad S_H.\text{ver} = H.$$

Every $q \in \mathcal{Q}$ evaluated at that checkpoint returns the same saved pair $(\text{Query}(S_H, q), c_H)$. Reconstruction uses no private signing or submission history. For all completely processed authentic extensions $P_{H'} \succeq P_H$ and all correctly encoded requests x , it satisfies

$$\begin{aligned} O_H[x] \neq \perp &\implies O_{H'}[x] = O_H[x], \\ D(x) < \infty \wedge H > D(x) \wedge O_H[x] = \perp &\implies O_{H'}[x] = \perp. \end{aligned}$$

Thus each request has at most one published first outcome, and both confirmed receipts and the strict-expiry tuple $\text{Failure}(\text{Expired}, x, c_{D(x)+1})$ are immutable.

Proof. Combined induction. We use the canonical requirements collected as Conditions 1–5, as required by Definition 3. Fix the authenticated anchor and its decoder state d_0 . Write the suffix's finite ordered item sequence as $A_H = (a_1, \dots, a_N)$, retaining every boundary and complete record, and let

$$(d_j, u_j) = F(d_{j-1}, a_j), \quad \Xi_j = (d_j, \Sigma^{(j)}), \quad \Xi_0 = (d_0, (S_0, \emptyset, \emptyset)).$$

Here $\Sigma^{(j)}$ is the staged AsyncLS state after processing decoded item u_j . For two scans a, b that finish the same first j items, the induction invariant is $\Xi_j^a = \Xi_j^b$. During a suspended item, pair the next input, emitted immutable context and logical processing stage, without equating the scans' actual response connections or delivery schedules. The case $j = 0$ includes the empty suffix and Condition 2's common initial $S_0.\text{ver} = 0$. Section 2.11's common transition and resume law give $F(d_{j-1}^a, a_j) = F(d_{j-1}^b, a_j)$, independently of the observing replica, end marker or previous pauses. In particular, comparison with a longer prefix preserves each earlier record's transaction, occurrence, ledger-recorded submitting party, source states, events and links. None of these inputs is replaced by the scanning replica's identity. The common parsing, context, application and query programs use the same declared inputs, and the shared identifier configuration gives the same receipt field from the same complete bytes.

A boundary at relative height i performs the prescribed source transition and sets only $S.\text{ver} = i$ on the AsyncLS side, as required by Condition 2. This

includes empty blocks. At a record, the induction hypothesis gives the same complete transaction, immutable context and preceding (S, L, O) . Condition 3's deterministic parsing and context predicate therefore agree, as do Condition 5's session/checkpoint, height and existing-outcome tests. A failing test skips **AsyncLS** processing without authentication, while retaining the decoder's source update. For a surviving candidate, the historical-authentication requirement in Section 2.11 gives equal completed authentication decisions on this same anchored record. A zero verification result skips the record. A result of one continues processing. Consuming a delivered **NoBinding** aborts the current recovery and discards staging without changing the **AsyncLS** committed checkpoint C or fixing a rejection. A later public call may start again from C . Determinism reproduces the same contexts, and immutable Boolean decisions reproduce all previously completed record checks. Thus aborted recoveries do not alter any completed reconstruction.

For a candidate that passes all structural tests and authentication, write its occurrence as (i, ℓ) and set $k = (\text{sid}, p, \nu)$. Condition 4 places the nonce check before evaluation at this immediate S . When $L[k] \in \{\perp, x\}$, write $(S_\delta, z_\delta) = \delta(p, S, r)$. The common update is

$$(S', L', z) = \begin{cases} (S, L, \text{Rejected}(\text{NonceConflict})) & L[k] \notin \{\perp, x\}, \\ (S_\delta, L[k \mapsto x], z_\delta) & L[k] \in \{\perp, x\}, \end{cases}$$

with the atomic outcome update $O' = O[x \mapsto (\text{gid}, x, \text{txid}, (i, \ell), z)]$. Condition 1 gives the same application result in both scans and, on application rejection, $S_\delta = S$. That identity concerns only S : this branch still assigns $L[k] = x$ and records $O[x]$. The nonce-conflict branch preserves S, L without invoking δ but also records $O[x]$. Condition 2 gives $S_\delta.\text{ver} = S.\text{ver}$ for every application transition. The two scans therefore have $\Sigma^{(j),a} = \Sigma^{(j),b}$. These cases exhaust the ordered update. A hash collision between distinct ledger transactions establishes neither equality of their complete records nor equality of their requests. The decoder retains both occurrences, L is nonce-keyed, and O is request-keyed. No history record is removed merely because its hash equals another record's hash. The induction is therefore exact for every fixed common hash index, without conditioning on identifier binding. The decoder advances even for records that **AsyncLS** skips, so later contexts include all preceding source activity. Application rejection reverses no source transition or ledger fee.

Completion and partition independence. The suffix has finitely many items. With the stated completion of required authentication and sufficient permitted activations, its bounded resumable steps finish. Condition 2 and the induction give $S_H.\text{ver} = H$, including $H = 0$ and suffixes of empty blocks. The resume law and the induction show that dividing this same input sequence into scans or activations changes no completed state. Aborted recoveries commit none of their staged application or decoder state, and the same induction applies to their recomputed suffixes. Completion here requires Boolean decisions for every eligible record in a successful recovery, not merely delivery of **NoBinding**. Atomic commitment publishes exactly the **AsyncLS** committed checkpoint (P_H, Σ_H, c_H) .

On this common S_H and the same q , Condition 1's total deterministic read-only query gives a unique saved $(\text{Query}(S_H, q), c_H)$, which later checkpoint updates cannot refresh.

Immutable terminal results. Every application evaluation or nonce-conflict branch records the request's first outcome in $O[x]$, and Condition 5's existing-outcome test skips every later copy. Thus no extension replaces a receipt or evaluates the same request again. If $D(x) < \infty$, $H > D(x)$ and $O_H[x] = \perp$, the completed scan has exhausted every eligible position $h < i \leq D(x)$. Prefix consistency preserves these decisions, and Condition 5's independent height guard excludes every later position, even if its `MatchesRequest` predicate returns 1. All extensions therefore return $\text{Failure}(\text{Expired}, x, c_{D(x)+1})$. When $W_{\text{eff}} = 0$ the eligible interval is empty, but its next boundary must still be processed. An existing outcome always takes precedence, including when first observed after $D(x)$. Infinite effective lifetime has no expiry branch. No-effect expiry for a separately represented ledger effect additionally uses the effect correspondence in Section 2.11.

2.16 Agreement and conditional liveness

Ordered processing fixes each eligible request's result. Extensions may change application state but preserve terminal results. A uniform resolution bound additionally requires progress assumptions.

Write $\text{Terminal}(y) = 1$ exactly when y is a correctly typed `Confirmed(receipt)` or `Failure(Expired, x, c_{D(x)+1})` result. In particular, `Pending` and operation-level `Abort` are not terminal request results.

Theorem 1 (Agreement on terminal request results). *Fix two honest replicas a, b of a well-formed logical AsyncLS session. They share authenticated (sid, P_0) , the selected ledger and ζ , application domains and initial state, decoder and its anchor state, parsing and context predicates, transition and query programs, transaction-identifier configuration, and historical-authentication interpretation with immutable consistent completed decisions. Each retains its own execution identity, local state and configured caller, verifier, ledger and response connections as specified in the session-configuration paragraph of Section 2.11. For every correctly encoded request x and every pair of delivered `Check(x)` replies y_a, y_b from these replicas,*

$$\text{Terminal}(y_a) = \text{Terminal}(y_b) = 1 \implies y_a = y_b.$$

The equality includes the complete receipt or expiry tuple and does not require simultaneous replies or equal nonterminal statuses.

Proof (Theorem 1). Let the two terminal replies be evaluated on committed authentic prefixes P_{H_1} and P_{H_2} read through the replicas' respective configured honest ledger connections. Permanence of their common ledger and the shared anchor make these prefixes comparable, so relabel a, b if necessary to obtain $H_1 \leq H_2$. This is only a choice of proof notation, not a change to either execution

identity. The stated common reconstruction data satisfy Lemma 1’s hypotheses. Its decoder-state induction identifies the processing of every boundary and record through H_1 in both scans, including the recorded submitter and occurrence metadata. In particular, Condition 3 fixes the parsed request and context checks, and Conditions 1 and 4 fix the result at the same immediate application state after the nonce check. Historical-authentication consistency supplies equal immutable completed decisions across their distinct verifier connections. The corresponding replies may arrive at different times, and a **NoBinding** abort may discard one replica’s unfinished staging without changing the other’s committed state.

There are exactly two terminal cases:

$$\begin{aligned} O_{H_1}[x] \neq \perp &\implies O_{H_2}[x] = O_{H_1}[x], \\ D(x) < \infty, H_1 > D(x), O_{H_1}[x] = \perp &\implies O_{H_2}[x] = \perp. \end{aligned}$$

In the second case $H_2 \geq H_1 > D(x)$, and Condition 5 fixes the expiry checkpoint from x . The longer prefix can neither reinterpret an earlier occurrence nor add a later eligible one. In the first case the existing outcome takes priority over expiry, even for $H_1 > D(x)$. That outcome may contain an applied result, an application rejection or a nonce-conflict rejection without any δ evaluation. Both instances therefore return the shorter prefix’s saved terminal tuple, so $y_a = y_b$. Cached or delayed delivery preserves that tuple. The argument applies whenever both terminal observations have completed.

The bounded-resolution condition Fix well-formed ideal settings ζ with finite $W_{\text{eff},\zeta} = \min(W_\zeta, W_{L,\zeta})$. Separately fix the following uniform progress bounds and the class of executions satisfying them before execution. Calling ζ bounded means bounded relative to these additional assumptions. Subscripts ζ and T_ζ below suppress these separately fixed progress bounds. For a considered **Submit** arriving at trace time τ_{arr} , let $b(\tau_{\text{arr}})$ be the permanent height relative to P_0 . Requests in this admitted execution class obey a uniform finite checkpoint bound $h \leq b(\tau_{\text{arr}}) + \kappa_\zeta$. Checkpoints already obtained from permanent history have $\kappa_\zeta = 0$. The selected ledger policy and admitted clock/maintenance schedule provide a finite nondecreasing $G_{\mathcal{G}_{\text{LEDGER},\zeta}}(k)$ bounding production of k further permanent heights from any time. $\Delta_{\text{party},\zeta}$ bounds the synchronized observer’s actual read of the required prefix. Production and visibility costs are counted once.

A uniform $\Delta_{\text{query},\zeta}$ covers subsequent public calls, remaining setup, complete prefix validation/processing, all required authentication checks and a fresh terminal **Check** return on independent runnable connections while submission waits or has aborted. For every eligible record encountered through the required target, the admitted class must bound the number of **NoBinding** aborts before a successful recovery, the delay to later public retry calls, fresh authentic reads, repeated decoding/application computations and eventual Boolean decisions. This includes binding retrieval for other owners whose records precede the queried request. Uniform encoded-history, workload and retry bounds give a total remaining-step

bound K_ζ , including work performed by aborted recoveries. If A_ζ bounds activation gaps, P_ζ the total wait for required public calls, E_ζ the total wait for authentication and repeated-read responses, and R_ζ return delivery, then

$$\Delta_{\text{query},\zeta} = P_\zeta + K_\zeta A_\zeta + E_\zeta + R_\zeta$$

is sufficient when those allowances partition the costs. An implementation using the same bound must include its actual additional internal and communication work in these allowances. These progress bounds are additional assumptions. Ideal settings with infinite effective lifetime remain well-formed but do not satisfy this finite-expiry condition.

Theorem 2 (Liveness of bounded AsyncLS). *Fix ζ and an execution class Exec_ζ satisfying all of Section 2.16's uniform progress assumptions, including $W_{\text{eff},\zeta} < \infty$. There is one finite bound, chosen before the execution, request and arrival time,*

$$T_\zeta = G_{\mathcal{G}_{\text{LEDGER},\zeta}}(\kappa_\zeta + W_{\text{eff},\zeta} + 1) + \Delta_{\text{party},\zeta} + \Delta_{\text{query},\zeta} < \infty,$$

such that for every $\rho \in \text{Exec}_\zeta$ and every admitted Submit invocation for a correctly encoded request x arriving at trace time τ_{arr} , a fresh Check(x) invocation delivers a reply y at a time τ_{ret} satisfying

$$\tau_{\text{arr}} \leq \tau_{\text{ret}} \leq \tau_{\text{arr}} + T_\zeta, \quad \text{Terminal}(y) = 1.$$

The request-arrival clock condition identifies the saved first sample with $t_0 = \tau_{\text{arr}}$. A confirmed rejection satisfies the conclusion, and an operation-level abort does not.

Proof (Theorem 2). Fix the execution class and all uniform allowances in Section 2.16 before choosing a request. For an admitted x received at τ_{arr} , the checkpoint-distance assumption gives

$$D(x) + 1 = h + W_{\text{eff},\zeta} + 1 \leq b(\tau_{\text{arr}}) + \kappa_\zeta + W_{\text{eff},\zeta} + 1.$$

The growth bound is nondecreasing and applies from every arrival time. Hence a permanent prefix through the decisive height $D(x) + 1$ exists within $G_{\mathcal{G}_{\text{LEDGER},\zeta}}(\kappa_\zeta + W_{\text{eff},\zeta} + 1)$ time units. The actual synchronized read supplies it within the further allowance $\Delta_{\text{party},\zeta}$.

By definition, $\Delta_{\text{query},\zeta}$ covers the remaining work through complete processing and delivery of a fresh Check reply, including authentication waits, a bounded number of recovery aborts, later public retries and their repeated reads and processing. Set

$$T_\zeta = G_{\mathcal{G}_{\text{LEDGER},\zeta}}(\kappa_\zeta + W_{\text{eff},\zeta} + 1) + \Delta_{\text{party},\zeta} + \Delta_{\text{query},\zeta}.$$

Every term is finite and uniform over the fixed admitted class, so T_ζ is independent of ρ, x and τ_{arr} . The bound's order is $\exists T_\zeta \forall \rho \in \text{Exec}_\zeta \forall (x, \tau_{\text{arr}})$. By $\tau_{\text{arr}} + T_\zeta$, the fresh reply uses a completely processed prefix with $H \geq D(x) + 1$, and hence

$$\text{Status}_H(x) = \begin{cases} \text{Confirmed}(O_H[x]) & O_H[x] \neq \perp, \\ \text{Failure}(\text{Expired}, x, c_{D(x)+1}) & O_H[x] = \perp. \end{cases}$$

Confirmation includes confirmed rejection. The extra height is necessary because expiry tests $H > D(x)$, including zero effective lifetime and empty blocks. This argument uses only trace time and independent query progress.

Construction and inclusion are unnecessary for the guarantee that a fresh query returns confirmation or strict expiry. A hash collision may cause the existing unused-identifier save check to abort a submission, but it does not alter the independent complete-prefix processing and fresh-query assumptions used here. A withheld required response, an owner binding that never becomes available for an eligible record, or unbounded abort/retry work falls outside the stated finite progress allowances. When a delivered **NoBinding** is consumed, the affected invocation may end without resolving the request. The resulting abort is neither **Confirmed** nor **Failure**. Infinite effective lifetime supplies no decisive finite expiry height and is excluded from this result. Transferring the deadline to a component implementation additionally requires its actual work to fit those same allowances and the deadline failure to be observable by an admitted environment. Under those conditions, the composition theorem bounds the implementation's excess failure probability by its distinguishing error.

3 Protocol AsyncLS

We construct $\Pi_{\text{AsyncLS}} = (\text{Wallet}, \text{Relay}, \text{Adapter}, \text{Replay})$ under the ideal's configuration and ζ . **Wallet** handles public invocations, approvals, and protected signing. **Relay** maintains E and schedules submission. **Adapter** issues and records ledger registration, read and submission requests, clock reads, and authentication-verification requests. **Replay** reconstructs the AsyncLS state $\Sigma_H = (S_H, L_H, O_H)$. Figure 3 specifies their operations.

Description of the protocol $\Pi_{\text{AsyncLS}}^\zeta$:

Participating roles: Wallet, Relay, Adapter and Replay on their configured connections.
Corruption model: Honest ledger party. Retained interface policies (Section 4).
Parameters: $\zeta = (W, W_L, d_{\min}, d_{\max})$ in the fixed configuration.

Component-structure adaptation of $\Pi_{\text{AsyncLS}}^\zeta$:

Implemented role(s): Wallet: public service/signing. Relay: coordination. Adapter: interface requests. Replay: reconstruction.
Subroutines: Wallet: Relay, Adapter, $\mathcal{G}_{\text{auth}}$, declared constructor/signing routes. Relay: Adapter, Replay. Replay: Adapter. Adapter: $\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{clock}}, \mathcal{G}_{\text{auth}}$. The registration board retains its existing users.
Initialization:
¹ Via **Adapter**, obtain authenticated P_0 as in Section 2.8. Relay initializes $E = \emptyset$ and empty queues, then invokes **Replay.Initialize**(sid, P_0, S_0). Its matching (sid, P_0, c_0) reply enables **Wallet** and returns (sid, P_0) on the setup connection.
EntityInitialization (Replay, fixed sid, P_0):
² Valid **Initialize**(sid, P_0, S_0): store $H = 0$, $C = (P_0, (S_0, \emptyset, \emptyset), c_0)$ once. Identical repeats preserve state, conflicts abort. Return (sid, P_0, c_0).

Wallet: public operations from p :

- 3 **on** Register(): invoke the owner's $\mathcal{G}_{\text{auth}}.\text{Register}(p)$ and return its result.
- 4 **on** Submit(ν, r, c_h): form valid x , set $t_0 = \perp$ and start Adapter.Time.
 - (a) Call Relay.PreSubmitCheck(x, h). Return terminal status or a recovery error. Otherwise complete recovery through h and this invocation's owner registration.
 - (b) With fresh pending status, atomically reuse the retained transaction, join preparation for x , or own TransactionCreation(x). Retain the completed transaction before yielding.
 - (c) Save the first clock result once as t_0 , then call Relay.Enqueue(x, tx, t_0) and return its result with terminal priority.
- 5 **on** Read(q)/Check(x): validate, forward to Relay and return its result.

Relay: coordination:

- 6 **on** PreSubmitCheck(x, h) from Wallet: obtain Replay's $(H, c_H, \text{Status}_H(x))$. Return terminal status or recover through h . Propagate recovery errors.
- 7 **on** Enqueue(x, tx, t_0) from an authorized saved Wallet operation: parse tx as (x, w, txid) or abort, and obtain Adapter.Verify(p, x, w). Continue on 1 and abort on 0 or NoBinding. Retain the transaction throughout. Recheck status, apply the ideal's atomic save/reuse rule and one-time PendingTx disclosure, and return Pending with terminal priority.
- 8 **on** Read(q)/Check(x) from Wallet: return known terminal Check status. Otherwise join recovery or issue one read when idle. Read waits and saves its query/checkpoint pair. Check returns committed status or a recovery error at its next runnable step.
- 9 On a permitted release poll, use Adapter.Time and apply the ideal's ordered release rule. Queue authorized Adapter.Deliver operations, retaining them while the protected route is busy.
- 10 **on** SetDelay($\text{sid}', \text{txid}, d$) from the adversary: locate $E[x]$ by $tx.\text{id} = \text{txid}$. Apply the ideal's checks before/after Adapter.Time, changing only an admissible target.

Adapter: declared interfaces:

- 11 **on** Time()/ReadPrefix(): issue clock-read to $\mathcal{G}_{\text{clock}}$ or read as the synchronized ledger party. Return t /validated P on the saved connection.
- 12 **on** Verify(p, x, w) from Relay/Replay: invoke $\mathcal{G}_{\text{auth}}$ on the saved verifier connection and return its result.
- 13 **on** Deliver(tx) from Relay: apply Section 2.13's acceptance and permission rules. Output (submit, gid, tx) as the configured party, permanently setting Submitted. Return Emitted after output or Refused before it.

Replay: requests from Relay:

- 14 **on** Recover(P'): validate and stage the ideal's ordered reconstruction as sole recovery owner. Join demands and queue later extensions. NoBinding aborts under Section 2.13. Commit complete prefixes only with unchanged base, and return (H, c_H) .
- 15 **on** Query(q)/Status(x): return (Query(S_H, q), c_H) or $(H, c_H, \text{Status}_H(x))$.

Fig. 3. Operation structure of $\Pi_{\text{AsyncLS}}^\zeta$. Section 2.13's saved-operation rules apply to all components. Complete code appears in Section 2.13.

The components execute Section 2.13’s saved-operation rules on their configured connections. We keep **Wallet** and **Replay** fixed when implementing **Relay** and **Adapter**.

3.1 Correctness of implementations of **Relay** and **Adapter**

The code of **Relay** and **Adapter** is specified by Figure 3 and the saved-operation rules above, at the shared parameters ζ . The iUC realization covers **Wallet**’s initial Time request and saved **PreSubmitCheck**, **Enqueue**, **Read** and **Check** invocations, **Replay**’s **Initialize**, **Recover**, **Query** and **Status** exchanges, **PendingTx** and **SetDelay**, and each ledger read/submission, clock read, authentication query and declared setup/constructor exchange. The configuration binds the owner, verifier, ledger-party and clock connections. Owner signing stays with **Wallet**. Exchanges among the four components carry only their existing saved inputs and return on their actual connections. Authentication observations remain on the retained global $\mathcal{G}_{\text{auth}}$ ’s own adversarial interface.

Procedure adaptation for **Adapter** operation records:

Implemented role(s): The configured **Adapter** requester and response recipient.

Subroutines: The original ledger, clock, authentication and declared setup interfaces, preserving caller identities and occupied response connections.

Internal state and main:

- ¹ For each operation, **Adapter** keeps the next local step, pending input and its ledger/clock/authentication request stage: unissued, issued with occupied connection, result produced, result delivered, or completed. These are fields of that machine’s ordinary private memory.
- (a) It accepts a local continuation once at its existing queue position and advances the fixed next step. It records the ledger, clock or authentication request as issued before yielding, retains its exact input and requester identity, and accepts its actual response once. A produced response remains distinct from a response delivered for consumption.
- (b) Retrying after **NoBinding** requires a later actual public input and a new invocation. Other invocations retain their own verification work unless they have aborted. An insufficient-prefix read follows its separate later-activation rule. Each retry invocation or insufficient-prefix read is a new operation on an available connection with its own observations.
- (c) Completion never manufactures another activation. The queue includes the real recovery work caused by **Check**. **Replay.Status** alone does not stand for the public operation.

To implement the code of **Relay** and **Adapter** with error ϵ_{RA} , we compare the complete systems obtained by attaching any compatible, responsive and environmentally bounded callers at this boundary, with the same $\mathcal{G}_{\text{LEDGER}}$, $\mathcal{G}_{\text{clock}}$, $\mathcal{G}_{\text{auth}}$, registration board and declared setup/constructor interfaces and all their public roles. One simulator fixed before these callers must match their complete views within ϵ_{RA} , including the caller identities used on those interfaces, occurrence multiplicity, authentication observations, clock effects, unfinished requests on them, private forwarding and delivery opportunities. These functionality instances

are retained once, with their public interfaces exposed, in both systems. This iUC emulation obligation compares complete systems against the independently specified reference code. Section 3.3 establishes particular cases.

3.2 Conditional bounds on transaction-submission time

For a first-saved $E[x]$ with finite $d_{\max}$ and an allowed first send that is not refused, let $a \leq t_0 + C_{\text{ready}}$ bound completion of the saving invocation's first clock read and response delivery, checkpoint recovery, registration, any shared-preparation wait and remaining construction, request verification at that endpoint and saving the transaction. Under the request-arrival clock condition, $t_0 = \tau_{\text{arr}}$, so this allowance begins at that invocation's arrival and includes all clock-request waiting. If T is its final target and Δ_{net} bounds all remaining actual activation, clock, release, routing and output work from $\max\{a, T\}$, the invariant above gives

$$t_0 + d_{\min} \leq t_{\text{emit}} \leq \max\{a, T\} + \Delta_{\text{net}} \leq t_0 + \max\{C_{\text{ready}}, d_{\max}\} + \Delta_{\text{net}}.$$

The bound $t_{\text{emit}} \leq t_0 + d_{\max}$ additionally requires $\max\{t_0 + C_{\text{ready}}, T\} + \Delta_{\text{net}} \leq t_0 + d_{\max}$ for every accepted final target. The allowance must include all actual authentication observations, recovery, queueing, ledger/clock/authentication/setup requests and their delivery steps of the implementation. Preparation and holding are not counted twice. Suppressed or refused work has no transaction-emission time, and later repeats have no new time allowance for the first send. These optional bounds do not imply inclusion or finite terminal resolution. Finite terminal resolution uses Section 2.16.

3.3 Implementing Relay and Adapter

Both implementations satisfy Section 2.11 for the fixed configuration, ζ and authenticated (sid, P_0) . **Adapter** supplies authentic finalized prefixes extending committed history. Submission and reconstruction enforce the verification rules in Figures 2 and 3. Reconstruction reads no private signing log. Its completed authentication decisions must agree across scans and remain unchanged under history extensions.

Write $R_{\text{RA}}^\zeta = (\text{Relay}, \text{Adapter})^\zeta$ for the reference component pair and Env_{RA} for the complete compatible, responsive, bounded contexts in Section 3.1. In the comparisons below, **View** includes the unchanged **Wallet**, **Replay** and application code, one copy of each retained interface, and all independent users under their actual role map. The operator **D** defined in Section 2.6 gives the probability distribution of a view.

Trusted Relay. An honest operator executes **Relay** with **Adapter** issuing requests and accepting matching responses on the configured connections. Both can run in one protected backend. Hosted node APIs illustrate provider-operated ledger access with client-held signing keys [21]. The operator receives the private request data needed for computation.

Condition 6 (Private worker execution). *The worker is honest and uncorrupted, and the coordinator is correct and retains the configured Relay/Adapter connections and actual ledger and verifier identities. Coordinator/worker I/O is authenticated and private. Internal messages and cache updates are bounded and contain no declared ledger, clock, authentication or setup request, visible output or observable yield. Such interface interactions and network exchanges remain visible boundaries. The compared system includes the private coordinator/worker roles, with no external worker-invocation or corruption interface under these honesty assumptions. This neither hides declared public roles nor exposes private inputs to the simulator.*

Snapshots may contain private requests, E , C and saved operations, so checking proposals would not protect the private data in those snapshots from a corrupted provider.

Condition 7 (Nonblocking worker jobs). *Local enqueue and cache checks are bounded and nonblocking. An unavailable job supplies no proposal, and callbacks only cache results. Local enqueueing, cache checks and callbacks do not consume the activation for a runnable reference step. On that activation, an absent or unusable proposal leaves the coordinator to perform the same bounded reference computation locally, without waiting for the worker.*

Proposition 1 (Trusted relay realization). *Fix the common requirements above and retain the same ideal $\mathcal{G}_{\text{auth}}$, ledger, clock, registration board, and declared setup and constructor interfaces. Under Conditions 6 and 7, an honest uncorrupted worker computes bounded deterministic local proposals, and a correct coordinator retains the configured Relay/Adapter connections and performs the matching reference steps. There exist an implementation $I_{\text{RA}}^{\text{tr}}$ and a PPT simulator $\mathcal{S}_{\text{tr}}$, both chosen independently of the context, such that*

$$\forall \mathcal{E} \in \text{Env}_{\text{RA}} \quad \forall \lambda: \\ \text{D}(\text{View}_{\mathcal{E}, I_{\text{RA}}^{\text{tr}}}(1^\lambda)) = \text{D}(\text{View}_{\mathcal{E}, \mathcal{S}_{\text{tr}}, R_{\text{RA}}^\zeta}(1^\lambda)).$$

Consequently $I_{\text{RA}}^{\text{tr}} \leq_{\text{iUC}}^0 R_{\text{RA}}^\zeta$, with $\epsilon_{\text{RA}} = 0$.

Local Relay. The owner's correct local endpoint executes all four components on their configured connections. A validating node illustrates local ledger access [20], with a separate signer retaining approval and key use [23]. Public node access alone supplies neither authenticated history nor a protected submission connection.

Proposition 2 (Local relay realization). *Fix the same common configuration and retained interfaces as in Proposition 1, with a correct local endpoint executing the reference Relay/Adapter operations. There exist an implementation $I_{\text{RA}}^{\text{loc}}$ and a PPT simulator $\mathcal{S}_{\text{loc}}$, both chosen independently of the context, such that*

$$\forall \mathcal{E} \in \text{Env}_{\text{RA}} \quad \forall \lambda: \\ \text{D}(\text{View}_{\mathcal{E}, I_{\text{RA}}^{\text{loc}}}(1^\lambda)) = \text{D}(\text{View}_{\mathcal{E}, \mathcal{S}_{\text{loc}}, R_{\text{RA}}^\zeta}(1^\lambda)).$$

Consequently $I_{\text{RA}}^{\text{loc}} \leq_{\text{iUC}}^0 R_{\text{RA}}^\zeta$, with $\epsilon_{\text{RA}} = 0$.

Replacing authentication by signatures adds ϵ_{auth} in Corollary 1. Public worker communication needs its own channel realization. Finite submission-time bounds require the additional assumptions in Section 3.2.

Constructions and correctness proofs The implementations below satisfy Section 3.1 at the shared parameters ζ , retaining **Wallet**, **Replay** and the configured interfaces. All use the following shared- $\mathcal{G}_{\text{auth}}$ procedure for historical requests, including those submitted by arbitrary ledger parties. The ledger-recorded submitting party remains observable in the occurrence metadata.

Fig. 4. Local and trusted implementations of **Relay** and **Adapter** under the stated execution conditions.

Description of the **Relay/Adapter** implementations:

Participating roles: Local implementation: the correct endpoint executes **Relay/Adapter** roles. Trusted implementation: the coordinator retains those roles and the worker performs only bounded private computation. **Wallet** and **Replay** retain their original roles.

Corruption model: The correct local endpoint or honest coordinator and worker of Propositions 2 and 1, retaining the declared interface corruption policies.

Parameters and connections: The fixed configuration and ζ of Section 3.1. Delegation preserves the original ledger and verifier identities.

Verifying requests read from ledger history.

Operation-structure adaptation of historical-request verification:

Implemented role(s): **Replay** retains its reconstruction role in both implementations and uses the original verifier connection.

Subroutines: $\mathcal{G}_{\text{auth}}$ on the configured verifier connection and authentic history from the original ledger-read route.

Initialization: The retained authenticated service setup and committed checkpoint of Section 2.8.

EntityInitialization: The original endpoint setup and saved-operation initialization of Section 2.13, on its existing ledger and verifier identities. Delegation adds no setup or signing authority.

Main:

- ¹ After the fixed structural, checkpoint, context, duplicate and height checks, **Replay** requests $\mathcal{G}_{\text{auth}}.\text{Verify}(p, x, w)$ on the configured verifier connection for each remaining candidate.
 - (a) A zero verification result skips the record, and a result of one accepts it. Consuming a delivered **NoBinding** terminates that recovery, fails its unresolved demands and discards staging under the recovery-ownership procedure in Section 2.13. A later public call starts a new recovery from the **AsyncLS** committed checkpoint C and uses fresh verification invocations with their observations.
 - (b) Every Boolean decision follows positive binding retrieval. The immutable global Cert_w table gives independent callers the same completed decision. A delayed no-binding result does not erase an established binding.

- (c) This procedure satisfies the reconstruction-authentication requirement in Section 2.11. The concrete signature replacement supplies the corresponding immutable-key and deterministic-verification facts through Lemma 5. Verification authenticates requests, while synchronization and actual ledger reads supply authentic history.

Saved operations and private computation.

Operation-structure adaptation of the saved operations and private worker:

Implemented role(s): Locally, the endpoint executes *Relay/Adapter*. In the trusted case, the coordinator owns their actual interface requests and commits proposals. The worker computes private proposals without acquiring those identities.

Subroutines: The unchanged reference component connections and existing ledger, clock and $\mathcal{G}_{\text{auth}}$ requests. The board and constructor/setup interfaces retain their original users. Private jobs use only the stated coordinator/worker connections.

Initialization: The retained authenticated service setup and committed checkpoint of Section 2.8.

EntityInitialization: The original endpoint setup and saved-operation initialization of Section 2.13, on its existing ledger and verifier identities. Delegation adds no setup or signing authority.

Internal state:

- ¹ The existing *Adapter* retains Section 3.1's operation table, queue, occupied connections and results in local memory, accessed by *Relay* through their configured private connections. No shared store or recovery oracle is added. Delegation preserves the original ledger and verifier identities.

Main: bounded private computation:

- ² Split each program step at its next declared-interface request, error, return or observable yield. A worker computes the bounded deterministic local portion from an immutable snapshot. The reference owner draws and saves configured random choices once, only when the reference computation calls for them. The snapshot grants no signing authority. Enqueue at most one private job for each reference local step.
 - (a) A private job identifier binds the operation and versions of every read dependency, including shared E , committed status and saved operation records. Any intervening change, including another invocation's, invalidates the proposal. These versions add no interface fields.
 - (b) A proposal is only a local result: it supplies no interface response, permission, activation or application-state update. At the original runnable step, the coordinator repeats the reference guards on current E , C and saved operations. A rejecting guard or an existing terminal result takes the reference error or return branch directly.
 - (c) Otherwise, use a ready proposal only if every dependency version and the step's saved reference choices match. If it is absent, stale or unusable, the coordinator performs that same bounded local computation on current dependencies, drawing only choices that the reference step requires and has not yet saved. On this same activation, it performs exactly the reference state transition and next declared-interface request, error, return or yield on the original connection.
 - (d) A consumed or obsolete job cannot commit again. Late or duplicate callbacks change only private job/cache data. Without a runnable reference step, a

cached proposal changes no operation table, queue, permission, saved result or interface state and creates no activation.

Private-worker requirements:

³ The coordinator and worker satisfy Conditions 6 and 7. Network exposure requires the separate channel realization discussed below.

Proof (Proposition 2). Choose $I_{\text{RA}}^{\text{loc}}$ to execute the reference **Relay/Adapter** code at the correct local endpoint, and let $\mathcal{S}_{\text{loc}}$ be the identity forwarder. Couple their tapes and all retained interfaces. Write $\sigma_n^{\text{loc}}, \sigma_n^{\text{ref}}$ for the operation tables, queues, occupied connections, permissions, saved results and retained-interface states after n matched transitions. If a_n includes the next action and its actual identity/connection, and ω_n contains the coupled coins, then

$$\begin{aligned} \sigma_0^{\text{loc}} &= \sigma_0^{\text{ref}}, \\ \sigma_n^{\text{loc}} = \sigma_n^{\text{ref}} &\implies \text{Step}(\sigma_n^{\text{loc}}, a_n, \omega_n) = \text{Step}(\sigma_n^{\text{ref}}, a_n, \omega_n). \end{aligned}$$

Each step returns its next state and emitted messages, so this induction preserves both state equality and the complete transcript.

For historical requests, the preceding direct verification procedure supplies the required consistent completed decisions from the immutable global authentication table. The well-formed causal interpretation satisfies the decoder obligation. The identity forwarder preserves independent ledger, clock, authentication, board and constructor/setup actions, including their observations and pending operations. Thus the complete joint-contract views coincide with zero error.

Proof (Proposition 1). Construct $I_{\text{RA}}^{\text{tr}}$ with the stated honest worker and correct coordinator under Conditions 6 and 7. For each job J , let D_J be the complete vector of read dependencies, v_J their versions, and ω_J the reference owner's saved coins. The honest worker returns $r_J = f_J(D_J, \omega_J)$ for the bounded deterministic local function f_J . At the original runnable queue position, the coordinator first repeats the current reference guards. If they permit computation, a ready proposal is accepted only when $v_{\text{current}} = v_J$ and the saved choices match. Because every dependency change increments its version, acceptance implies

$$\text{AcceptJob}(J) = 1 \implies D_{\text{current}} = D_J \implies r_J = f_J(D_{\text{current}}, \omega_J).$$

A valid proposal therefore supplies exactly the reference local result. If no proposal is ready, or another invocation has changed a dependency, the coordinator instead computes f_J on the current dependencies and the same legitimate reference choices. A proposal requiring a choice not yet drawn cannot be used. The coordinator draws that choice at its reference step and saves it once. Thus every runnable reference step has a matching implementation transition on its original activation, independently of worker completion. A rejecting guard or an existing terminal result gives the same error or return without accepting a proposal. In every case, the coordinator commits the same transition and reaches the same next declared-interface request, error, return or yield, retaining the original identity and connection.

Project away only private job/cache data. Late or duplicate callbacks leave all operation tables, queues, permissions, saved results and retained interfaces unchanged, and cannot repeat a committed step. When no reference step is runnable, private job activity changes none of this projected state and manufactures no activation. Each bounded private computation before the next visible boundary has no declared-interface request, visible output or observable yield inside it. Contract it to the corresponding reference local step. Starting with identical projected states and retained interfaces, induction over every runnable step and actual interface boundary preserves queued requests, identities, results, observations and response-delivery opportunities. It covers both accepted proposals and coordinator computation, including stale proposals after another invocation's update. Independent interface users see identical traffic under the same public role map.

Choose the fixed forwarder $\mathcal{S}_{\text{tr}}$. Private jobs produce no external transcript, and this simulator never reads a job to synthesize a protected input. For every fixed context and coupled tape, the induction gives $v_{\text{tr}}^{(n)} = v_{\text{ref}}^{(n)}$ for each finite visible-event prefix n . There is at most one bounded private job and one bounded coordinator computation per reference local step, with the bounded cache work of Conditions 6 and 7. The existing environmental bounds therefore retain polynomial total work. No worker fairness, extra reference randomness or additional observable waiting is used. Consequently

$$\forall \mathcal{E} \in \text{Env}_{\text{RA}} \quad \forall \lambda: \quad \Pr[\text{View}_{\mathcal{E}, I_{\text{RA}}^{\text{tr}}} (1^\lambda) \neq \text{View}_{\mathcal{E}, \mathcal{S}_{\text{tr}}, R_{\text{RA}}^\zeta} (1^\lambda)] = 0,$$

which gives the asserted complete-view equality and $\epsilon_{\text{RA}} = 0$.

Public-channel deployment. The propositions establish the stated realizations for implementations using private I/O. A public network deployment additionally needs a compatible realization of those private channels, including traffic leakage, authentication, scheduling and corruption behavior, with its own error.

3.4 Ledger access, keys and transaction construction

Selecting the ledger configuration. An implementation fixes $\mathcal{G}_{\text{LEDGER}}$'s `Validate`, `ExtendPolicy` and `Blockify` algorithms, visibility and synchronization parameters, and clock policy [3, Secs. 4.1–4.3]. Validation enforces the chosen transaction authorization, funding, fee, and conflict rules. The block format retains complete transaction bytes, block boundaries, ordered occurrences, and the ledger-recorded submitting party. These requirements allow reconstruction to check the actual included transaction and its context. A conventional node API need not expose this complete interface.

Setup uses the synchronized ledger party's returned prefix as P_0 . Replicas import authenticated (sid, P_0) and accept prefixes returned by later reads only if they equal or extend the committed prefix from that anchor, even if they lag behind the ledger tip. A malformed, older or incompatible returned prefix causes the affected operation to abort. Clock reads retain the clock's enabled

round-update behavior and provide no release alarm. Section 2.8 gives the full procedure.

Implementing request authentication. Choose a perfectly correct EUF-CMA signature scheme with deterministic verification, as specified in Appendix A.2. At the owner’s first registration, generate (sk_p, v_p) once, retain sk_p at the protected owner endpoint, and publish v_p under that owner’s identity on the authenticated first-write board. Readiness requires retrieval of the same published key by the owner. Each verifier independently retrieves the immutable binding on its own connection.

After approval, construction signs exactly x , producing $w \leftarrow \text{Sign}(sk_p, x)$. Verification computes $\text{Ver}(x, w, v_p)$ under the retrieved key. Authenticated registration associates that key with the owner. For all callers, the wrapper reports $\mathcal{G}_{\text{auth}}$ inputs and produced results, including errors, for **Register**, **Sign** and **Verify** on its adversarial interface. All permitted signing domains using a shared key count in the same security bound. Transaction signing retains its separate route when required by the ledger.

Constructing and retaining a transaction. The constructor combines the approved request, certificate, and transaction fields. Authenticated replies from its declared interfaces support funding, fee, size, authorization, and ledger-nonce checks. The constructor aborts when it detects a violation of these requirements. It checks the final signed size before returning a usable transaction, which **Wallet** retains before verification. Construction neither reserves funds nor guarantees admission. Changing a fee or another protected field requires new approval. Section 5.1 illustrates both signing stages.

4 Provably secure realization

We prove realization in iUC [13]. Fix a well-formed configuration of $\mathcal{G}_{\text{LEDGER}}$, $\mathcal{G}_{\text{clock}}$, ideal $\mathcal{G}_{\text{auth}}$, the authenticated registration board, $\mathcal{M}$ and ζ . Retain transaction signing when used and every additional constructor or setup interface. Each system includes one copy of the fixed ledger, clock, board and additional interfaces, with identical actual caller and response connections and public and adversarial roles.

For security parameter λ , let **Env** be the class of universally bounded, compatible, responsive probabilistic polynomial-time (PPT) environments whose connected protocols are environmentally bounded and whose schedules satisfy the request-arrival clock condition (Section 2.4). Each $\mathcal{E}(1^\lambda)$ in this class may independently use the declared public and adversarial ledger, clock, authentication and registration-board interfaces, the separate transaction-signing interface when used, and additional constructor and setup interfaces under their existing permissions. Section 2.6 gives the boundedness and responsiveness definitions.

4.1 Shared configuration of the reference construction and ideal

Configuration and initial correspondence. Fix the well-formed configuration in the session-configuration paragraph of Section 2.11, sid and ζ before choosing

the environment. Both systems retain $\mathcal{G}_{\text{LEDGER}}$, $\mathcal{G}_{\text{clock}}$, ideal $\mathcal{G}_{\text{auth}}$, the registration board, the separate transaction-signing interface when used and every additional construction/setup interface, with their actual owner, verifier, ledger-party, caller/callee and response connections and public roles. Independent users retain every permitted operation. Authentication and Relay/Adapter substitutions are separate steps proved in Section 4.3.

Couple the fixed creation, parsing, reconstruction, application/query and setup programs and constructor randomness. Use the same serialization, hash program and public index hk in both systems. The following transition comparison holds for every fixed common index, even on hash-collision executions. Averaging this pointwise equality over Section 2.11's honest index generation preserves zero mismatch probability. Section 2.11's checks, declared requests, disclosures and errors then agree. Both setup procedures perform the same registration, synchronization and prefix read and validate the returned anchor and S_0 . The reference Initialize operation and the corresponding ideal checks give equal (sid, P_0) outputs, preserve exact repeats and reject the same conflicts and invalid responses. Replicas import that authenticated configuration without reinitializing it. Definition 2's binding is logical, not serialized program equality, and a changed configuration requires a fresh identifier. This establishes the initial correspondence of the source-decoder state, (S, L, O) , E and saved operations.

4.2 Reference operational correspondence

The reference protocol consists of the four fixed components declared above. The ideal owns the AsyncLS committed checkpoint C , staged AsyncLS state (S, L, O) and scan cursor, E and all saved operations directly. Its four public operations arrive on their corresponding actual caller connections.

Lemma 2 (Operational correspondence). *Fix Section 4.1's well-formed configuration, ζ and actual interface connections. There is one PPT simulator $\mathcal{S}_{\text{sim}}$, chosen from this configuration alone, such that*

$$\forall \mathcal{E} \in \text{Env} \ \forall \lambda: \\ \mathsf{D}(\text{View}_{\mathcal{E}, \Pi_{\text{AsyncLS}}^{\zeta}}(1^\lambda)) = \mathsf{D}(\text{View}_{\mathcal{E}, \mathcal{S}_{\text{sim}}, \mathcal{F}_{\text{AsyncLS}}^{\zeta}}(1^\lambda)).$$

The proof constructs a coupling with mismatch probability zero for every fixed common hk , including hash-collision executions, and hence also under the specified shared index generation. The comparison retains the same declared interfaces and public/adversarial roles, contracting only bounded private forwarding with no declared-interface request, visible output or observable yield.

Proof. Fixed simulator. Choose the following forwarding machine from the configuration alone. On each declared adversarial connection it relays the received input or observation once to the counterpart connection with the same identity, arguments and order. The environment already supplies adversarial scheduling and permitted independent actions under Section 2.6. The simulator does not receive or instantiate an environment-dependent adversary program. Actual public

service inputs reach the ideal directly. The ideal issues its own prescribed protected signing and ledger submissions. The forwarder has neither an honest owner connection nor an honest ledger-submission connection and does not emulate a replacement clock.

State relation. Couple the environment tape, retained-interface tapes and configured constructor coins. Pair the session and authenticated setup, the AsyncLS committed checkpoint C , staged AsyncLS state (S, L, O) , decoder state and pending decoded item, scan cursor/end marker, and $E[x] = (tx, t_0, T, s)$. Pair each invocation's unset or saved t_0 and its first clock request, including the original response connection and any produced but undelivered value. Pair preparation owners, waiters, intermediate results and immutable completed transactions, including transactions saved before $E[x]$ exists. Pair each operation by input, actual caller/callee, occupied response connection, stage and produced/delivered/consumed result. Pair the single recovery owner, its committed base, immutable target, waiters and queued later prefixes under the recovery-ownership procedure in Section 2.13. Pair aborted-operation markers, failed demands, retained records for already issued responses, queue positions and unused or consumed send permissions. Corresponding public invocations have both returned or neither has returned, while their issued operations stay related. Contract only bounded private forwarding that has no declared-interface request, visible output or observable yield. Initialization and setup satisfy this relation by Section 4.1.

Formally, for complete paired configurations U, V , let $\pi_R(U)$ and $\pi_I(V)$ be the projections onto exactly these service records, and let $g_R(U), g_I(V)$ be the states of the retained interfaces and the environment. Define the proof relation

$$\mathcal{R}(U, V) \iff \pi_R(U) = \pi_I(V) \wedge g_R(U) = g_I(V).$$

For a transition $U \xrightarrow{a/o} U'$ with input/action a and visible output o , write $V \xRightarrow{a/o} V'$ for the matching transition with only the bounded private forwarding steps contracted. The obligations below are

$$\begin{aligned} & \mathcal{R}(U_0, V_0), \\ & \mathcal{R}(U, V) \wedge U \xrightarrow{a/o} U' \implies \exists V': V \xRightarrow{a/o} V' \wedge \mathcal{R}(U', V'). \end{aligned}$$

The same matching condition holds with the two executions interchanged. The projections and relation are proof notation, not additional protocol state or an interface available to the simulator. In particular, π_I is a mathematical projection of the ideal execution, not a query the forwarder can issue. The simulator neither runs a private copy of **Replay** to obtain ideal state nor receives a new observation of that state. The ideal performs its own canonical parsing, reconstruction and updates.

Parsing and record correspondence. Condition 3, required by Definition 3, collects the fixed configuration's parsing and identifier requirements. Both executions receive identical complete bytes and recompute the same hash under their

common index. Those bytes parse to the same (x, w, txid) or fail parsing in both executions. The parser need not succeed on malformed bytes, and distinct byte strings can encode the same logical x . The parsed transactions' protected context must still pass the canonical predicate, while $E[x]$ and $O[x]$ retain their separate request-keyed roles. Equal preceding decoder states and the same next complete record give the same immutable occurrence, source states, events and links by the causal-decoding rule in Section 2.11. Thus **MatchesRequest** and the independent height/outcome guards receive equal inputs. After the shared authentication decision, Conditions 1 and 4 give the same nonce branch and application transition at the immediate state. An application rejection preserves only S . The paired L, O updates and source transition remain as specified. These are exact transition equalities for every fixed common hash index. Distinct transactions with equal hashes remain distinct complete records and need not encode the same x . No step of this induction infers byte equality from hash equality. Section 2.11's probability bound is needed only for the separate assertion that an observed hash binds its encountered transaction bytes.

Submission and preparation. Equal inputs undergo the same encoding/caller checks, initialize $t_0 = \perp$, and issue the same first clock read under the same actual requester identity. The retained clock executes the same **Round-Update** and produces the same value on the paired response connections. Its delivery sets each invocation's t_0 once, with no resampling at a later continuation. The paired continuations apply the same timestamp guard before enqueue and endpoint verification. Other preparation and terminal-priority replies use the same runnable steps. Returning a terminal result cancels neither paired clock read, and the corresponding late responses authorize no further submission. Checkpoint recovery and each invocation's owner registration use the same saved connections and results. The atomic choice in the submission-preparation procedure of Section 2.13 returns terminal status, reuses completed transactions, joins active preparation or starts the sole constructor in both executions. Equal constructor data and coins yield identical declared requests, errors and usable transactions. Completion saves those transactions before yielding or verification, even if $E[x] = \perp$. A later endpoint **NoBinding** cannot release ownership of already completed transactions. A new preparation owner may start only after all requests issued by the unsuccessful preparation have completed and that preparation has halted without a completed transaction. Waiters retain their own connections, clock samples and queue positions. These rules allow independent nonce-conflicting requests without a local nonce reservation.

Authentication and saving. After its own t_0 is set, each invocation verifies its exact (p, x, w) on the paired verifier connection. The retained $\mathcal{G}_{\text{auth}}$ produces the same route counters, **AuthInput**, **AuthOutput**, saved results and delivery opportunities, including retries, errors and cached bindings. Verification result 0 aborts. Consumed **NoBinding** ends the paired invocation with the same **Abort** result used for malformed input, subject to terminal priority. Both executions retain completed transactions and existing E , but create no entry, permission or send. A retry of the aborted invocation requires a later actual **Submit** with its own

clock sample, registration and authentication observations. Other invocations keep their own verification work unless they have aborted. Verification result 1 permits the same fresh status and identifier tests. The first save installs equal $E[x]$ using that invocation's first sample, even if it joined another invocation's construction, and emits one matched **PendingTx**. Reuse compares complete bytes and preserves the record. A different transaction with an already saved hash takes the same identifier-conflict abort on both sides, as do the other request/transaction conflicts. Initially E is empty, every first save checks that its identifier is unused throughout E , and no later transition changes the saved transaction. Induction therefore establishes pairwise distinct identifiers within each paired E even when hashes collide outside that table. The caller's **Pending** return is separate from that observation. Equal first-save/reuse cases grant precisely the permissions of the submission-permission procedure in Section 2.13.

Release and ledger output. Every permitted release activation issues the same clock read through the original connection. The release guards compare the same clock samples with the same saved targets. The just-proved invariant on E ensures that a **SetDelay** input locates at most one record in each execution. For a known identifier it locates the same record and repeats the same checks after its saved clock response. Due entries are visited in $(T, tx.id)$ order. Terminal status suppresses the same entry, while pending status queues the same initial send. Dispatch checks equal current status, transactions, verification, permission and route availability. Acceptance consumes one permission before either execution yields. A busy route preserves the same wait. Actual output sends the same complete transaction under the same ledger party and permanently sets **Submitted**. The retained ledger consequently creates the same occurrence and notification even when it rejects admission. A pre-output **Refused** restores no consumed permission. Terminal status learned later cancels neither accepted submission.

Recovery, reads and checks. A returned authentic prefix selects the same equality, extension or invalid-view branch. The paired causal decoder visits the same item and preserves the same source transition even when **AsyncLS** skips it. Pre-authentication context tests and later authentication/history queries therefore match. Completed decisions and deterministic updates give the same first outcomes and complete-prefix commit, as established by Lemma 1. A cached terminal **Check** returns on both sides without a new ledger read. A nonterminal **Check** starts or advances the same independent recovery without duplicating occupied read/verification connections. At its next runnable reply step it returns the current committed status independently of recovery's response-delivery steps. If its recovery has consumed **NoBinding** first, it returns the same abort unless terminal status takes priority. That reply completes only its corresponding check invocation. **Read** instead waits for successful recovery and saves the same query/checkpoint pair, or returns the matched recovery error. Later recovery cannot answer a closed query or revise a saved answer.

Concurrent recovery. The paired recovery owners have the same idle/read/scan phase and demand queue. A new **Read**, **Check** or **PreSubmitCheck** therefore starts the same sole read, joins the same active target, or waits for the same unmet

height. Joining changes no staged data or response ownership. A returned prefix undergoes the same validation against the prefix P_H in C , completes on equality with P_H , joins a covering active target, or is queued as a later extension with its originating connection. The paired scans retain the same fixed end marker and current item across activations. On consuming **NoBinding**, both discard the same staged decoder/application data, cursor, target and dependent queued extensions, leave the **AsyncLS** committed checkpoint C unchanged, and fail the same unresolved demands. Still-open **Read**, **PreSubmitCheck/Submit** and **Check** invocations receive matching error replies in the two executions, with the specified terminal priority. Closed returns stay closed. Already issued requests keep paired occupied connections whose late responses cannot restart the failed scan. A later public input may start a new paired recovery from C , issuing the same new read and verification requests. Thus this branch leaves no suspended scan. Only a successful active owner commits, with equal base and extension checks. No other staged writer can change C first, so every successful commit extends the preceding committed prefix. Satisfied waiters preserve their own runnable positions and return connections. Later queued work starts from the same newly committed decoder and **AsyncLS** states without an invented activation. This closes the induction for concurrent demands, including a **Pending** reply that has closed while its shared scan remains active.

Auxiliary clock-trace invariant. The transition comparison pairs executable state without reading τ_{arr} . At a matched invocation both timestamps are unset. Issuance preserves equal clock-request identities and response connections. Every intervening context-originated clock-affecting action reaches the same retained ledger/clock code under the same identity and has the same effect. The first reads therefore perform equal **Round-Update** steps and return equal values. Paired delivery replaces $\perp$ by those values exactly once. **Release** and **SetDelay** use subsequent actual samples in identical guards, while **reuse** never refreshes the saved t_0 . Induction preserves this relation for any identical legal clock-input traces, even when an intervening update makes $t_0 \neq \tau_{\text{arr}}$. No program reads, compares or stores the trace-only τ_{arr} . On admitted traces the separately imposed sampling condition gives $t_0 = \tau_{\text{arr}}$ and preserves the advertised arrival-relative bounds. The auxiliary equality supplies no timing guarantee outside that class. It supports Section 4.3's combined simulation without assuming that an intermediate simulator preserves the sampling condition.

Remaining transitions and view equality. Each response is saved and consumed only on its original occupied connection at the expected stage. Late, malformed, wrong-route and already consumed responses have matching effects. Equal preceding views and the coupled environment tape choose the same next independent action. The unchanged interfaces process that action under the same actual identity, including public authentication, raw ledger submissions, proposals, maintenance, registration and permitted corruption. Thus every input, local step, interface interaction, yield and output satisfies the matching obligation or is one of the contracted private forwarding steps. If $v_R^{(n)}, v_I^{(n)}$ are the first n

visible events, induction yields

$$\begin{aligned} \forall n \text{ reached: } & \mathcal{R}(U_n, V_n) \wedge v_R^{(n)} = v_I^{(n)}, \\ \Pr[\text{View}_{\mathcal{E}, \Pi_{\text{AsyncLS}}^\zeta} & \neq \text{View}_{\mathcal{E}, \mathcal{S}_{\text{sim}}, \mathcal{F}_{\text{AsyncLS}}^\zeta}] = 0. \end{aligned}$$

Here U_n, V_n are matched configurations after those events, and the security parameter is suppressed. The core comparison has service-computation error zero.

Polynomial work. The simulator is the same fixed forwarding program for every admitted environment. It copies and dispatches each received interaction once on its prescribed connection, with work polynomial in the encoded message length and retained routing data. Section 2.6’s admitted interaction budgets therefore bound its total work. For the service programs, Condition 1’s total polynomial-time δ and **Query** are only part of the boundedness requirements. Section 2.11 also bounds encodings, type checks, parsing and public hash evaluation. Arbitrary malformed bytes are checked before a typed application or query evaluation. Section 2.11’s causal-decoder and transaction-construction rules require bounded resumable decoding and construction, while the saved-invocation procedure in Section 2.13 retains each issued operation, result and continuation on its actual connection. Scanning, saved-operation bookkeeping, retries and private forwarding all count toward the complete composition’s computation budget. The admitted environmental-boundedness requirement supplies the resulting bound for each environment, with the negligible-tail convention stated above. It does not supply one common polynomial for all environments. Waiting for a response supplies no unbounded local computation, and polynomial work supplies neither response delivery nor sufficient activations, ledger inclusion or progress.

Theorem 3 (Core realization of AsyncLS). *Fix the well-formed configuration of Section 4.1, including ζ , shared ideal $\mathcal{G}_{\text{auth}}$ and all actual interface connections and public/adversarial roles. There exists a PPT simulator $\mathcal{S}_{\text{sim}}$ such that*

$$\forall \mathcal{E} \in \text{Env} \ \forall \lambda: \quad \text{Adv}_{\mathcal{E}}(\Pi_{\text{AsyncLS}}^\zeta, \mathcal{F}_{\text{AsyncLS}}^\zeta, \mathcal{S}_{\text{sim}}; \lambda) = 0.$$

Thus $\Pi_{\text{AsyncLS}}^\zeta \leq_{\text{iUC}}^0 \mathcal{F}_{\text{AsyncLS}}^\zeta$, with quantifier order $\exists \mathcal{S}_{\text{sim}} \forall \mathcal{E}$. All other declared interfaces are retained once on both sides.

Assume the signature scheme and wrapper satisfy the signature, owner, registration, and corruption conditions stated in Lemma 5, giving error ϵ_{auth} . Fix an implementation of **Relay** and **Adapter** that iUC-realizes their reference code with error ϵ_{RA} while ideal $\mathcal{G}_{\text{auth}}$ and all named interfaces remain unchanged, as required by Section 3.1.

Corollary 1 (Implementation composition). *Let I_{RA} be the chosen trusted or local implementation under the stated authentication and component-realization hypotheses, and let $\hat{\Pi}^\zeta$ denote Π_{AsyncLS} with this component implementation and with $\Pi_{\mathcal{G}_{\text{auth}}}[\text{Sig}]$ replacing shared ideal $\mathcal{G}_{\text{auth}}$ on all authentication connections. For*

the same retained interfaces and actual role map, there is one PPT simulator $\mathcal{S}_{\text{comp}}$ such that

$$\forall \mathcal{E} \in \text{Env} \quad \forall \lambda: \quad \text{Adv}_{\mathcal{E}}(\widehat{\Pi}^{\zeta}, \mathcal{F}_{\text{AsyncLS}}^{\zeta}, \mathcal{S}_{\text{comp}}; \lambda) \leq \epsilon_{\text{auth}}(\lambda) + \epsilon_{\text{RA}}(\lambda).$$

Equivalently, $\widehat{\Pi}^{\zeta} \leq_{\text{iUC}}^{\epsilon_{\text{auth}} + \epsilon_{\text{RA}}} \mathcal{F}_{\text{AsyncLS}}^{\zeta}$. The simulator is chosen independently of the environment, while the error budgets are those of the corresponding complete composed contexts.

4.3 Proofs of core realization and composition

Proof (Theorem 3). Fix the theorem's configuration. Lemma 2 supplies a PPT forwarding simulator determined by that configuration and independent of the environment. Both compared systems retain one copy of each declared global interface with the same connections and public/adversarial roles. The lemma's state relation includes staged decoder state, completed transactions, pending operations and every actual interface observation. Only bounded invisible component forwarding is contracted. The equality holds for every fixed common hash index, including collision executions, and averaging over the shared setup preserves it.

For each $\mathcal{E} \in \text{Env}$ and security parameter λ , the lemma's coupling gives

$$\begin{aligned} \text{Adv}_{\mathcal{E}}(\Pi_{\text{AsyncLS}}^{\zeta}, \mathcal{F}_{\text{AsyncLS}}^{\zeta}, \mathcal{S}_{\text{sim}}; \lambda) &\leq \Pr[\text{View}_{\mathcal{E}, \Pi_{\text{AsyncLS}}^{\zeta}}(1^{\lambda}) \neq \text{View}_{\mathcal{E}, \mathcal{S}_{\text{sim}}, \mathcal{F}_{\text{AsyncLS}}^{\zeta}}(1^{\lambda})] \\ &= 0. \end{aligned}$$

The same simulator works for all these environments. This proves $\Pi_{\text{AsyncLS}}^{\zeta} \leq_{\text{iUC}}^0 \mathcal{F}_{\text{AsyncLS}}^{\zeta}$ with ideal authentication and all other declared interfaces retained.

Proof (Corollary 1). Fix the chosen implementations and the common session, configuration and ζ . Define three substitutions in order. In the initial system H_0 , all authentication connections use $\Pi_{\mathcal{G}_{\text{auth}}}[\text{Sig}]$ and Relay/Adapter use the chosen implementation. In H_1 , replace authentication by ideal $\mathcal{G}_{\text{auth}}$ with $\mathcal{S}_{\mathcal{G}_{\text{auth}}}$. In H_2 , additionally replace the implementation by reference Relay/Adapter with $\mathcal{S}_{\text{RA}}$. In H_3 , replace the resulting $\Pi_{\text{AsyncLS}}^{\zeta}$ by $\mathcal{F}_{\text{AsyncLS}}^{\zeta}$ with the core simulator. Each hybrid retains every other named interface once with its actual role map, and includes the simulators from earlier substitutions. The configured hash family, public index, full serialization and hash program are unchanged in every hybrid.

$$H_0 \xrightarrow[\epsilon_{\text{auth}}]{\mathcal{S}_{\mathcal{G}_{\text{auth}}}} H_1 \xrightarrow[\epsilon_{\text{RA}}]{\mathcal{S}_{\text{RA}}} H_2 \xrightarrow[0]{\mathcal{S}_{\text{sim}}} H_3.$$

These arrows denote comparisons of complete systems, not protocol messages. The composed simulator $\mathcal{S}_{\text{comp}}$ connects the three fixed simulators under the declared role map.

Lemma 5 bounds the first change by ϵ_{auth} in the complete context of the service, other applications and independent authentication users. All permitted

signing domains before corruption count once in each key’s budget. Appendix A.2 chooses one cutoff for this complete context, including connected protocols and preceding simulators. Its single negligible tail is included once in ϵ_{auth} .

With ideal authentication fixed, the assumed joint iUC contract bounds the second change by ϵ_{RA} . Its context includes **Wallet**, **Replay**, application code and all independent users, with their actual identities, pending requests and declared public/adversarial roles. This assumption is computational indistinguishability and requires no small complete-view coupling mismatch. For the third change, use Lemma 2’s auxiliary clock-trace invariant as follows.

The actual context around H_2 contains the already chosen authentication and **Relay/Adapter** simulators. Treat all their legal interface actions, including any clock-affecting ledger or clock input, as actions of this same context in the paired H_2, H_3 executions. Couple its tape and retain the same global interfaces once. A matched service invocation starts with $t_0 = \perp$ on both sides and issues the same actual clock request. An intervening context action reaches the same ledger/clock state under the same identity. The first reads therefore execute the same **Round-Update** and return equal values, saved once as t_0 on their original response connections. Pending and produced-but-undelivered samples remain paired. The auxiliary invariant does not use $t_0 = \tau_{\text{arr}}$, so it also pairs the complete transition sequence when the intermediate context violates the request-arrival condition. This directly simulates the executable programs in the combined context. The public composition claim retains its original sampling restriction and arrival-relative timing assumptions.

All three simulators are chosen before the environment. Compatibility, responsiveness and environmental boundedness are the stated hypotheses of complete iUC composition [13, Corollary 1], so the three simulators compose into one simulator. Fix any $\mathcal{E} \in \text{Env}$ and let $p_j(\lambda) = \Pr[\mathcal{E} \text{ outputs 1 in } H_j]$. The three comparisons give $|p_0 - p_1| \leq \epsilon_{\text{auth}}$, $|p_1 - p_2| \leq \epsilon_{\text{RA}}$ and $p_2 = p_3$. Therefore

$$|p_0 - p_3| \leq |p_0 - p_1| + |p_1 - p_2| + |p_2 - p_3| \leq \epsilon_{\text{auth}} + \epsilon_{\text{RA}}.$$

The bounds use the budgets of the corresponding composed contexts. Replacing another declared interface requires its own realization and error term. No identifier error is added to this realization bound because no hash procedure is replaced. For the full composed context $\mathcal{C}$, a client assertion that a hash binds complete transaction bytes uses the unconditional bound $\epsilon_{\text{id}, \mathcal{C}}$ in (1). If a joint bound already charges the same cutoff tail, it adds only the off-tail collision term $\beta_{\text{id}, \mathcal{C}}$. Progress still requires the separately stated progress assumptions and time allowances.

The trusted and local implementations have $\epsilon_{\text{RA}} = 0$, so their total error is ϵ_{auth} . The configured hash program and public index remain unchanged in these comparisons, so they add no identifier error. The separate claim that a hash binds encountered complete transaction bytes has the probabilistic bound in the transaction-identifiers paragraph of Section 2.11. Other channel, hash, setup, or ledger substitutions require separate proofs and error bounds.

5 Usage of AsyncLS

We illustrate use of $\mathcal{F}_{\text{AsyncLS}}$ under the realization and composition assumptions with an ORDI payment and an application interpreting ordered block proposals.

The examples use the constructions in Section 3.3 and the application requirements in Section 2.14. The old session’s outstanding requests retain their status when a fresh session starts.

5.1 An ORDI interactive payment

This construction pays the amount already reserved in one valid unused ORDI transfer inscription. Our construction follows the original BRC-20 rules [16] and the latest Open Protocol Indexer [8].

Source ledger and immutable bootstrap In the selected baseline, write $b(k)$ and $a(k)$ for an account’s overall and available balances. A valid transfer inscription e for amount u reserves available balance at source s while leaving its overall balance unchanged. Its first normal movement consumes that reservation and transfers u to the actual destination. Self-movement and movement of the inscription into transaction fees restore availability rather than making the payment considered here. Thus inscription creation is not a positive receipt. The source’s `p/op/tick/amt` JSON does not itself bind our invoice, nonce or checkpoint: the transaction wrapper below supplies that binding. The decoder must keep the baseline’s `deploy/mint`, numerical and event-order rules needed to establish the reservation.

Fix the four-letter ticker `ordi` under the pinned baseline rules, a permanent source prefix at ledger height b_{res} , and its valid unused transfer inscription e of amount $u > 0$, owned by source script s . The source decoder D_{ORDI} recovers the selected BRC-20 balances, inscription records and ordered inscription events from complete ledger history. Its requirements include Ordinals ownership and satoshi assignment, the creation/movement distinction, inscription identities, ledger transaction/occurrence links, numerical checks, used-inscription handling, and complete event order. Correctness of source extraction and ledger validation is an explicit assumption. The selected interpretation must also satisfy the causal input condition below.

Selected causal ORDI interpretation. Use a source state containing the prior Ordinals ownership/position data, relevant transaction-output data, token definitions, balances and inscription-use records. At a block boundary, retain its authentic height and source-format context. For each next complete record ρ at (ℓ, j) , the selected source-format extractor must determine its inscription events and occurrence links from this preceding state and ρ alone, with a fixed order for events in that record. In particular, restricting a complete permanent source history cannot change an earlier event, destination, event ordinal or record link. This is the explicit causal input obligation on the assumed Ordinals extraction, in addition to its correctness.

For the **AsyncLS** interpretation, the decoder applies the selected baseline’s deploy, mint, inscription and movement handlers to those current events in order, retaining all source updates whether or not the transaction is an **AsyncLS** candidate. Each handler tests the preceding token/balance/inscription records and the current event’s fields and updates that source state. Emit $(d_{\text{before}}, d_{\text{after}}, \text{events}, \text{links})$ from this one record’s transition and save it unchanged while **AsyncLS** checks the request. Block boundaries are emitted even when there are no events or transactions. At the pinned OPI revision, the BRC-20 block pass selects the current block’s **ord.transfers** in ascending event order and executes these handlers sequentially [8]. That block pass supports the selected handler order.

This defines an instance of the transition F in Section 2.11. To prove prefix consistency, compare two authentic prefixes at their first common item. Their initial source states and authentic item agree. At a boundary the source book-keeping therefore agrees. At a record the causal extractor yields identical events and links, and induction over those events gives identical intermediate and final handler states. Thus the emitted contexts agree and the next source states agree. Induction over all common items proves prefix consistency, including skipped **AsyncLS** records and empty blocks. Saving the resulting source state and pending item gives the same next transition as an uninterrupted scan, proving the resume law. The selected mathematical interpretation is therefore causal under its stated source-input assumption.

The application configuration fixes e, s, u, b_{res} , outpoint U , its value and inscription offset, the transaction format/size limit, owner p and owner maps. Bootstrap follows Section 2.8’s registration, synchronization and authentic read procedure. Before saving the returned P_0 of height $b_0 \geq b_{\text{res}}$, it runs D_{ORDI} over that complete prefix and checks the reservation at b_{res} , nonuse through b_0 , that U remains unspent and contains e at its stated position, and ownership of s, U by this endpoint. If a check fails, bootstrap aborts. Successful bootstrap initializes version zero, $\text{Reserved}(e, s, u)$ and empty **AsyncLS** nonce/outcome maps. The source-decoder state at b_0 is retained or recomputed from authentic P_0 . **AsyncLS** reconstruction starts after this anchor. Replicas import this same authenticated setup. Subsequent relative height i denotes ledger height $b_0 + i$.

For the illustrative transaction format, U is a single-key pay-to-witness-public-key-hash (P2WPKH) input. It contains no other inscription whose movement would change a represented token balance. The transaction-signing owner uses only the prescribed wrapper-signing route for spends of U , including before any adversarially delayed candidate could have been produced: in particular, there is no earlier bare or conflicting spend authorization for this input. Other transactions and source owners remain unrestricted by this instance. Let $\epsilon_{\text{tx}}(\lambda)$ bound violation of the declared transaction-spend authenticity and signed-field binding requirement, counting every permitted use of the transaction-signing key. This assumption includes the relevant script/key and digest binding. It is separate from the **AsyncLS** signature theorem. Ledger double-spend exclusion and the complete permanent history remain selected-ledger assumptions. Distinct transaction-signing and **AsyncLS** keys avoid silently identifying their signing APIs.

For an ordinary account k , let $\mathcal{T}_k$ be its valid unused transfer inscriptions and $u_{e'}$ their reserved amounts.

Lemma 3 (Reservation accounting). *For a correctly decoded source prefix Q , write $b_Q(k), a_Q(k)$ for the overall and available balances of an ordinary account k , and $\mathcal{T}_k(Q)$ for its valid unused transfer inscriptions, with stored amounts $u_{e'}(Q)$. Under the selected baseline rules,*

$$\forall Q \forall k: \quad b_Q(k) - a_Q(k) = \sum_{e' \in \mathcal{T}_k(Q)} u_{e'}(Q).$$

For every permitted decoded extension $Q' \succeq Q$,

$$e' \in \mathcal{T}_k(Q) \wedge e' \text{ remains unused in } Q' \implies e' \in \mathcal{T}_k(Q') \wedge u_{e'}(Q') = u_{e'}(Q).$$

Proof. Induct from the empty source history over the selected decoder's ordered events. Set $R_k = \sum_{e' \in \mathcal{T}_k} u_{e'}$ and $J_k = b(k) - a(k) - R_k$. Initially $J_k = 0$. For one event, let $\Delta b, \Delta a, \Delta R$ be the changes at the indicated account. The complete cases are

event at the account	Δb	Δa	ΔR
deployment or invalid/used event	0	0	0
valid mint of v	v	v	0
reservation of $u_{e'}$	0	$-u_{e'}$	$u_{e'}$
first normal movement, source	$-u_{e'}$	0	$-u_{e'}$
first normal movement, destination	$u_{e'}$	$u_{e'}$	0
self-movement or movement into fees	0	$u_{e'}$	$-u_{e'}$

Every row has $\Delta J_k = \Delta b - \Delta a - \Delta R = 0$. A reservation inserts exactly that unused inscription. Each consuming movement removes it, and self-movement or movement into fees restores availability without changing the source's overall balance. Accounts not touched by the event have zero changes.

These cases are the selected deploy, mint, inscription, normal-transfer and fee-transfer rules [16,8]. Their numerical and availability checks exclude creation of an invalid reservation. They exhaust the stated ordinary-account interpretation, so the invariant holds at each event boundary and hence every complete decoded prefix. An event that leaves a particular inscription unused does not change its stored reserved amount. Other reservations and available funds may change, but its summand persists until its own first consuming event. The invariant is conditional on correct source decoding.

Application state and transaction template. The application state S contains the reserved version field and a delivery record, initially $\text{Reserved}(e, s, u)$ and later terminal $\text{Delivered}(e, s, t, u)$. The version follows the common block-update rule from the shared anchor P_0 . Its typed request is $\text{Transfer}(s, t, u)$, for a declared ordinary recipient $t \neq s$. With delivery record $\text{Reserved}(e, s, u)$, its transition δ records delivery and returns $\text{Applied}(\text{Transfer}(s, t, u, e))$. A later evaluation would

reject without changing S . The source correspondence below justifies that delivery effect. The synchronization query **Total** returns the fixed reserved amount u . A status query describes this reservation/delivery record. The queries concern only this earmarked amount and its delivery record.

Invoice, approval and complete ledger transaction Fix a payment deployment psid binding this service sid , currency ordi , source/recipient owner maps and invoice-authentication convention. The invoice amount must be the bootstrapped u . Its specified encoding is

$$I = \text{Encode}(\text{PAY/invoice}, \text{psid}, m, p, \text{iid}, t, u), \quad \text{Owner}(t) = m, \quad t \neq s.$$

Here m is the merchant, $p = \text{Owner}(s)$ the intended payer, and iid a merchant-local issuance identifier. Reusing an identifier for a different tuple is an operation error. Distinct intents may refer to the same invoice.

For a payer-local intent η , which identifies one intended payment, the nonce binding that intent is

$$\nu = \text{Encode}(\text{PAY/attempt}, \text{psid}, p, \eta, 0, I, \chi_x).$$

The attempt index is fixed to zero: this infinite-lifetime instance permits one approved correctly encoded request per intent and only submission retries preserving that request. The complete x authenticates $\text{Transfer}(s, t, u)$, the invoice and transaction context. The configured format admits a bounded full encoding. Once an intent has an approved x , the payment rules permit no different approved request for that intent, even under uncertainty, rejection or missing merchant acceptance.

By issuing an actual **IssueInvoice** input, the merchant approves the full I and its publication. Its local owner module signs

$$m_{\text{inv}}(m, I) = \text{Encode}(\text{PAY/invoice-auth}, \text{psid}, m, I).$$

This domain differs injectively from the **AsyncLS** request domain of x . The merchant uses the same $\mathcal{G}_{\text{auth}}$ through $\text{Sign}(m, m_{\text{inv}}(m, I))$ on its actual authorized owner connection. Readiness uses **Register**(m) on that connection. The payer calls $\text{Verify}(m, m_{\text{inv}}(m, I), w_I)$ with the received certificate w_I . Delivered **NoBinding** ends that invoice-validation invocation with **Abort** without rejecting the invoice permanently. A later validation invocation may issue a new **Verify**. After the verifier has obtained a positive binding, a verification result of 0 rejects the invoice, whereas a result of 1 permits the payer/service/recipient/amount checks. Every permitted use of the shared owner key, including invoice and **AsyncLS** domains, counts in the common authentication bound. Invoice certificates remain public on their declared network route, which may delay, replay, reorder, or drop them.

A **PaymentNotice**($\text{psid}, \text{iid}, x$) supplies an untrusted candidate x for **Check**. The merchant establishes payment through the acceptance checks below.

Prepare provisionally checks the unsigned proposal’s encoding and transaction context, including size constraints decidable without the **AsyncLS** certificate. It obtains an actual checkpoint c_h with $h \geq 0$ and uses the invoice/intent/transaction-context nonce above. A transaction template in the required encoding χ_x specifies the sole input U , its certified value and script context, all output values and scripts, version, sequence, locktime, signing mode, and the source decoder’s required position of the inscription at recipient t . The template includes a specified zero-value metadata output with a payload slot. It also has a separate input-witness slot. The payload slot is filled with

$\text{Encode}(\text{ALS/ORDI-carrier}, x, w).$

The output script is **OP_RETURN** followed by the uniquely encoded data push of the displayed bytes: a one-byte length up to 75 bytes, otherwise the smallest fitting **PUSHDATA1**, **PUSHDATA2** or **PUSHDATA4** length field with its specified little-endian serialization. The format fixes a maximum payload size B under its configured admission and relay policies. The chosen bound must admit at least one complete proposal for this instance. The full request and certificate must fit that limit. After the **AsyncLS** certificate is produced, the exact payload is checked against this limit before transaction signing. Admission of every such full wrapper under a default Bitcoin relay policy is *not* assumed.

For the sole input U , let v_U be its certified ledger value and v_j the output values in the approved template. The network transaction fee is $v_U - \sum_j v_j$. Construction checks value conservation, funding and its fixed fee budget/selection policy, including the pinned size/rate rules where required, using the already specified authenticated input data. An insufficient budget or failed policy check aborts. Final signed size is checked before returning a usable transaction. A wallet/relay fee-rate preference is not necessarily a consensus minimum, and later ledger validation may still reject the transaction. A positive network fee is compatible with paying the token recipient: the prohibited event above is movement of inscription e itself into fees. Saved retries preserve all inputs, outputs and fees.

The metadata and transaction-witness slots avoid recursive copies of x, w . After the payer approves the complete x , **TransactionCreation**(x) obtains w from $\mathcal{G}_{\text{auth}}.\text{Sign}(p, x)$, inserts the exact payload into the approved template and checks its size/context before invoking the separate transaction-signing route. It checks final signed size and fee policy before returning the complete transaction, including transaction signatures η_{tx} . Only request signing invokes $\mathcal{G}_{\text{auth}}.\text{Sign}$. Its **AuthOutput** and the transaction-signing response remain separate. Only w appears in the metadata slot.

A failed pre-signing payload check returns **Abort** before transaction signing. The completed **AsyncLS** certificate, approved intent, asset reservation, disclosed messages and every issued authentication or transaction-signing request remain saved under the saved-invocation procedure in Section 2.13. The abort neither proves expiry nor authorizes a replacement request for the same intent. Only the payer’s approved operation may authorize a spend of U . Section 2.11’s common

public hash configuration defines $tx.id = id_{hk}(ser_{full}(tx))$ on the complete signed serialization, including every witness. This derived accessor is computed from finished bytes and is not itself a serialized or signed field.

The parser recomputes that same identifier from the complete bytes and recovers x, w, η_{tx} from the metadata and witness slots, all exposed by the selected ledger record. Section 2.11’s collision-resistance assumption supplies the probabilistic complete-byte binding guarantee for the entire payment context, including both replicas, all participating owners, direct ledger submissions and independently used shared interfaces. This hash assumption follows from neither authentication nor transaction-signature unforgeability. The identifier need not equal Bitcoin’s conventional transaction identifier, which need not commit to witnesses. Section 2.13’s shared preparation serializes both signing stages, retains the sole completed transaction before verification/insertion, and governs later joins/reuse. Signing responses retain their original operations, and sends recheck terminal status.

The illustrative transaction signature uses SIGHASH_ALL without ANY-ONECANPAY. BIP143 specifies commitments to version, all input outpoints and sequences, the signed input’s outpoint, script code and amount, all output amounts/scripts, locktime, and signing mode [28]. Thus the chosen metadata output is among the signed outputs. The actual `MatchesRequeste` check also validates the full transaction template, correctly encoded payload, actual input/output context and decoded destination. It excludes self-movement, movement of the inscription into transaction fees, and burn outputs. Other senders’ signatures and identifiers may vary only within the context-equivalent copies of Section 2.11. Both configured lifetimes are $W = W_L = \infty$. A mempool timeout cannot certify that the request will have no application effect.

Concrete event, request and receipt mapping. Reconstruction keeps the `AsyncLS` certificate w distinct from the transaction signatures recovered from the same complete record. On each entire prefix returned by the unchanged ledger route, the local driver runs the selected causal `DORDI` fold in its fixed order, including every ledger record, to maintain the staged source state. For `AsyncLS` processing, the driver type-checks and decodes the full wrapper, checks its payment-deployment binding, and applies Figure 2’s preliminary encoding, checkpoint, height and existing-outcome tests. Before calling the authentication interface, it checks the actual input U , inscription e , and every χ_x field through `MatchesRequeste`, together with the decoder’s first normal movement of e to the request’s t in that same record. These locally decidable context and event checks invoke no authentication operation. A failing record is skipped for `AsyncLS` processing without discarding the decoder’s source-state update. Only a candidate surviving these checks calls the common `Verify` on p, x, w . Consuming a delivered `NoBinding` aborts that recovery with `Abort` and discards all its staging, leaving the committed decoder state and `AsyncLS` committed checkpoint C unchanged. Under the recovery-ownership procedure in Section 2.13, the aborted recovery fails its unresolved recovery demands. Their caller invocations apply the stated terminal priority, and previously closed return connections stay closed. A later public call

starts from that checkpoint and decoder state and decodes the same movement again before a fresh **Verify**. A verification result of 0 skips the record. A result of 1 enters the common first-nonce/outcome rule and transition. Creation/reservation, an unrelated movement, or a record without the required event produces no positive receipt.

This maps the event to the unique complete x in its own transaction. The receipt keeps the ordinary format

$$\text{receipt} = (\text{gid}, x, \text{txid}, (i, j), \text{Applied}(\text{Transfer}(s, t, u, e))),$$

with the application result specified above and $\text{txid} = tx.\text{id}$ recomputed from the complete included transaction bytes. The authentic position (i, j) selects its particular occurrence in the ledger. Ledger-assigned occurrence metadata and the decoder's source movement-record identifier **using_tx_id** are separate values. The pinned decoder's occurrence relation must map the full transaction at (i, j) to its source record and event ordinal k , by the selected source-format parsing rule rather than equality with the receipt identifier. That event must be the first normal movement of e in this very transaction. An ambiguous or unverifiable relation gives no positive receipt. The signed transaction binds the movement to its exact authenticated request and invoice.

For this bootstrapped instance and a prefix P_H processed by the specified reconstruction procedure, $\text{TransferApplied}(x, \text{receipt}, s, t, a)$ holds exactly when all of the following conditions hold:

1. $a = u > 0$ and $s \neq t$. The correctly encoded request x belongs to this deployment, names $\text{Transfer}(s, t, u)$, and has application caller $p = \text{Owner}(s)$, with the invoice/transaction-context encoding specified above.
2. The first-outcome entry computed from that prefix $O_H[x]$ equals **receipt** in the ordinary format above.
3. The authentic ledger record at receipt position (i, j) yields the complete transaction tx , satisfies $tx.\text{id} = \text{txid}$ under the declared parsing/binding rule, contains the exact wrapper x , and passes the selected authentication and MatchesRequest_e checks.
4. On the authenticated history, the decoder's occurrence relation identifies event k , in that same record, as the first normal movement of the reserved e from s to t for exactly u .

Set $\zeta_e = (\infty, \infty, d_{\min}, d_{\max})$ with release delays in Section 2.3's domains. The direct local implementation and ideal use this same session and specified configuration, including the separate transaction-signing route and bootstrap's actual ledger read.

Write $\mathcal{A}_{\text{ORDI}}$ for the complete application tuple of Definition 3 with the domains, reserved-inscription core and transaction procedures just specified. It includes the ORDI parsing and MatchesRequest_e procedures in addition to the four-field core $\mathcal{M}$. The real and ideal instantiations use the same hash program, public index and serialization of Section 2.11. Both satisfy the same probabilistic binding requirement. Their operational comparison uses their actual matching

hash outputs, including on executions in which different complete transactions collide.

The ideal ORDI payment interface

Fig. 5. The ideal ORDI payment interface with separately connected payer and merchant AsyncLS replicas.

Description of the ideal ORDI payment interface:

Participating roles: The payer p and merchant m on their actual connections.
Corruption model: The honest payer and merchant of Theorem 4. Other owners retain the signer cutover of Appendix A.1.
Protocol parameters: The fixed ORDI deployment and $\zeta_e = (\infty, \infty, d_{\min}, d_{\max})$.

Operation-structure adaptation of the payment interface:

Implemented role(s): The payer recipient handles **Prepare**, **Authorize** and **Progress_{pay}**. The merchant recipient handles **IssueInvoice** and **CheckPayment**, on their actual connections.
Subroutines: The separately connected payer and merchant $\mathcal{F}_{\text{AsyncLS}}^{\zeta_e}$ replicas and the shared interfaces in the dispatch block below.
Initialization: The one completed authenticated ORDI bootstrap supplies the shared (sid, P_0) .
EntityInitialization: Each assigned replica imports that same authenticated setup and keeps its own state and actual ledger-party/response connections, as specified below. These labels describe the existing setup, not new first-message hooks.
Interface, internal state and initialization:
¹ Define $\mathcal{F}_{\text{ORDI-pay}}[(\mathcal{F}_{\text{AsyncLS}}^{\zeta_e})_p, (\mathcal{F}_{\text{AsyncLS}}^{\zeta_e})_m]$ with merchant inputs **IssueInvoice**, **CheckPayment** and payer inputs **Prepare**, **Authorize**, **Progress_{pay}**(x) on their actual connections. It stores invoices, proposals, approved intents, candidates and accepted receipts.
(a) The subscripts name the payer's and merchant's separately connected monolithic AsyncLS replicas and leave the logical sid unchanged. Both import the same authenticated (sid, P_0) from the one completed setup and share the ORDI interpretation, its initial application and decoder states, ζ_e and the transaction-identifier configuration. Their common shared- $\mathcal{G}_{\text{auth}}$ procedure supplies consistent completed historical-authentication decisions on their distinct verifier connections.
(b) Each replica owns its committed checkpoint, application state, nonce and outcome tables, committed and staged source-decoder states, recovery owner and staging, preparation records, E , operation queue and saved requests and responses. Their committed prefixes and unfinished work may differ.
Subroutines and actual dispatch:
² The replicas share one retained copy each of $\mathcal{G}_{\text{LEDGER}}$, $\mathcal{G}_{\text{clock}}$, $\mathcal{G}_{\text{auth}}$, the registration board and the separate transaction-signing interface. Each replica uses its actual configured ledger-party, clock-requester, verifier, caller and response connections,

preserving their permissions and public or adversarial roles. The two ledger-party identities are distinct.

- (a) The payer's **Prepare** dispatches its **Register** and any required **Read** to $(\mathcal{F}_{\text{AsyncLS}}^e)_p$, **Authorize** dispatches **Submit** there, and **Progress_{pay}** dispatches its **Check** and any permitted fresh **Submit** there.
- (b) Merchant **CheckPayment** dispatches **Check** to $(\mathcal{F}_{\text{AsyncLS}}^e)_m$, and acceptance uses that query's receipt or its previously accepted tuple.
- (c) Invoice registration/signing and payer invoice verification use their declared $\mathcal{G}_{\text{auth}}$ connections directly. The saved-invocation procedure in Section 2.13 governs every saved operation. The payment rules are:

Main:

- 3 **Issue and deliver an invoice.** On an actual merchant **IssueInvoice** input, the merchant endpoint checks the required encoding of the owner, recipient and amount, fixes (m, id) to I , and starts its invoice $\mathcal{G}_{\text{auth}}$. **Sign** invocation on the merchant's owner connection. Publish only the resulting signed message on the declared network route.
 - (a) An exact repeat preserves the operation. A changed tuple under the same handle is rejected.
 - (b) The payer validates a received invoice through the actual common owner/session **Verify** checks, including their delays. For an honest merchant, an accepted invoice must match its issued tuple. Malformed, wrong-payer, cross-deployment or inconsistent invoices fail. Replaying a valid invoice creates no payment.
- 4 **Prepare.** On the actual payer connection, obtain registration and, when needed, a **Read** result on the assigned service connection. Return the complete unsigned proposal, using its saved returned checkpoint, validated invoice, fresh intent nonce and checked transaction template. Keep the proposal and pending route without refreshing a delayed checkpoint. Failure of these provisional unsigned-proposal checks aborts without starting either signing route. Exact full-payload admission is checked after the **AsyncLS** certificate exists.
- 5 **Authorize.** Receive the user's actual approval of the complete x . Check the validated invoice, source owner, recipient, amount, context and exclusive intent namespace. Record that intent's sole approved request before dispatching its **Submit** invocation on p 's private assigned service connection.
 - (a) At this payment interface, an exact repeated approval keeps the intent's existing live operation. A different request for the intent is rejected.
 - (b) Publish the untrusted notice without inferring any **Submit** completion. Only this approval on the payer's connection enables the two-stage signing route. The ensuing construction follows the two-stage procedure and its failure-retention rules above.
 - (c) An operation-level **Abort** neither clears the approved intent/reservation nor cancels issued authentication or transaction-signing requests.
- 6 **Progress_{pay}(x).** From the actual payer p , require x to equal an already stored approved request of that payer. Otherwise return **Abort**. Save x , the caller and this invocation's return connection.
 - (a) If its status-query route is occupied by another invocation, retain this **Progress_{pay}(x)** invocation unchanged until the route is free and a later permitted activation can start its own fresh **Check(x)**. A query or result for

- another request cannot answer this $\text{Progress}_{\text{pay}}(x)$ invocation, and no occupied query is switched or cancelled.
- (b) **Pending** closes the owned query and its originating $\text{Progress}_{\text{pay}}(x)$ invocation. A later $\text{Progress}_{\text{pay}}(x)$ polls the same approved request afresh. A pending result permits one policy-authorized fresh **Submit** for this saved x when the direct send route is free. An occupied send route keeps its original **Submit** invocation.
 - (c) That **Submit** obtains its own owner **Register** result and uses the shared-preparation rule to reuse, join or start construction, retaining the same completed transaction. Its own verification and direct send checks remain required. This adds no signing authority or fee replacement.
 - (d) A **Confirmed(receipt)** result is terminal whether its application result is **Applied** or **Rejected**, and permits no successor. A query **Abort** completes the originating $\text{Progress}_{\text{pay}}$ invocation with that error and schedules no **Submit**. An endpoint **Submit** error closes that submission without clearing approval or completed transactions.
 - (e) A later $\text{Progress}_{\text{pay}}$ may retry the same request through a new query and, after **Pending**, a new policy-authorized **Submit**. **Abort** ends only its operation. Since $W = W_L = \infty$, finite expiry never retires this request.
- ⁷ **Candidate requests and merchant status queries.** A notice supplies only a correctly encoded invoice-bound candidate x . A new $\text{CheckPayment}(I, x)$ invokes a fresh **Check** on the merchant's own instance. **Pending** or **Abort** closes both invocations with that result, so further observation requires another actual poll. Recovery that has not aborted may continue after **Pending**. A later recovery abort cannot reopen either closed return connection. A payer's earlier read cannot advance the merchant's separate cache.
- ⁸ **Acceptance.** Require an actual matching **Confirmed(receipt)** with the complete request, ledger identity, occurrence and $\text{TransferApplied}(x, \text{receipt}, s, t, u)$ relation defined above. Its invoice and transaction context must agree with the merchant's issued I . For an honest payer the request must be its exact approved intent. Record $\text{Accepted}(m, I, x) = \text{receipt}$ only on the first acceptance. Repeats may return that cached tuple.

The ledger effect and its **AsyncLS** receipt may exist before notice or merchant acceptance. The interaction completes at the first merchant result that passes the acceptance checks, without an additional acknowledgment.

Protocol and complete proof

Description of the real ORDI payment protocol:

Participating roles: The same payer p and merchant m , with the same dispatch and connection map.
Corruption model: The honest endpoints and independent-owner corruption behavior stated below.
Protocol parameters: The same fixed ORDI deployment and ζ_e .

Operation-structure adaptation of the payer and merchant implementation:

Implemented role(s): The actual payer and merchant recipients implement the same respective payment operations and dispatch map as the ideal interface.
Subroutines: The direct local replicas $(\Pi_{\text{AsyncLS}}^{\zeta_e})_p$ and $(\Pi_{\text{AsyncLS}}^{\zeta_e})_m$, direct invoice $\mathcal{G}_{\text{auth}}$ connections, and the retained shared interfaces and channels below.
Initialization: The same single completed ORDI bootstrap fixes the authenticated (sid, P_0) .
EntityInitialization: Each local replica uses its existing setup connection to import that same completed setup, retaining separate state and its distinct configured ledger identity. This does not generate a fresh anchor for a role or caller.
Main, retained state and subroutines:

- ¹ The real merchant and payer execute the preceding interface rules using their own saved endpoint records: issued or validated invoices, exact intents/proposals, transactions and outstanding connections.
- (a) Replace the two ideal replicas by direct local instances $(\Pi_{\text{AsyncLS}}^{\zeta_e})_p$ and $(\Pi_{\text{AsyncLS}}^{\zeta_e})_m$ under exactly the same dispatch and connection map. They retain separate service states and the shared authenticated setup and global interfaces specified above. They use the declared invoice/notice channels.
- (b) Invoice signing follows the merchant's **IssueInvoice** input. The payer's exact approval enables request and transaction signing. The saved-invocation procedure in Section 2.13 governs response matching.
- (c) The payer and merchant endpoints are assumed honest, so the signer-corruption rule of Appendix A.1 does not apply to them. Independent nonhonest owners retain that corruption rule and all public roles. Compromise of an **AsyncLS** request key does not compromise the separate transaction-spend key.

Fig. 6. The ORDI payment protocol using direct local **AsyncLS** instances.

Theorem 4 (ORDI payment and ledger-effect correspondence). *Fix the ORDI deployment, ζ_e and the two replicas' dispatch and connection map above. Assume the stated bootstrap, causal baseline interpretation, ledger validation, meta-data admission, ownership, protected signing, exclusive-intent, parsing/identifier and request-arrival clock conditions. Write*

$$R_{\text{pair}} = ((\Pi_{\text{AsyncLS}}^{\zeta_e})_p, (\Pi_{\text{AsyncLS}}^{\zeta_e})_m), \quad I_{\text{pair}} = ((\mathcal{F}_{\text{AsyncLS}}^{\zeta_e})_p, (\mathcal{F}_{\text{AsyncLS}}^{\zeta_e})_m).$$

In the common ideal-authentication hybrid, a fixed PPT simulator $\mathcal{S}_{\text{pair}}$ couples these systems with

$$\forall \mathcal{E} \in \text{Env} \ \forall \lambda \ \forall \text{hk}: \quad \Pr[\text{View}_{\mathcal{E}, R_{\text{pair}}} \neq \text{View}_{\mathcal{E}, \mathcal{S}_{\text{pair}}, I_{\text{pair}}}] = 0.$$

The displayed view notation omits 1^λ and the same fixed valid public hash index hk . The views retain one copy of each shared global interface and every actual connection. Thus the joint service-computation error is zero even on hash-collision executions and remains zero after averaging over the shared index generation.

Let Π_{pay} be the real payment protocol above and $F_{\text{pay}} = \mathcal{F}_{\text{ORDI-pay}}[I_{\text{pair}}]$ its specified ideal interface. For honest payer and merchant endpoints on the stated private connections, there is one PPT simulator $\mathcal{S}_{\text{pay}}$ such that

$$\forall \mathcal{E} \in \text{Env} \ \forall \lambda: \quad \text{Adv}_{\mathcal{E}}(\Pi_{\text{pay}}, F_{\text{pay}}, \mathcal{S}_{\text{pay}}; \lambda) \leq \epsilon_{\text{auth}}(\lambda) + \epsilon_{\text{tx}}(\lambda).$$

Both simulators are chosen before the environment. The transaction-authenticity bound is the one above, and the authentication bound includes invoice owners and every permitted signing domain. Put $\text{Bad}_{\text{pay}} = \text{Bad}_{\text{auth}} \cup \text{Bad}_{\text{tx}}$ for these failure events. In every execution outside Bad_{pay} ,

$$\begin{aligned} \text{Accepted}(m, I, x) = \text{receipt} \neq \perp &\implies \text{TransferApplied}(x, \text{receipt}, s, t, u), \\ \forall (p, \eta) \text{ approved}: \quad |\text{Effects}(p, \eta)| &\leq 1. \end{aligned}$$

Here t is the recipient in I , and $\text{Effects}(p, \eta)$ is the set of occurrences of nonzero represented payment movements authorized by that intent. The first implication identifies the authorized first normal movement of the selected e , for exactly u from s to t .

In the complete payment context $\mathcal{C}_{\text{pay}}$, each receipt's hash identifier additionally binds its complete transaction among the transaction bytes processed in that context outside $\text{Bad}_{\text{id}, \mathcal{C}_{\text{pay}}}$, whose probability is at most $\epsilon_{\text{id}, \mathcal{C}_{\text{pay}}}$. Choose one common polynomial cutoff for the authentication and identifier bounds, with its tail already included once in ϵ_{auth} . Then the joint failure bound is

$$\Pr[\text{Bad}_{\text{pay}} \cup \text{Bad}_{\text{id}, \mathcal{C}_{\text{pay}}}] \leq \epsilon_{\text{auth}} + \epsilon_{\text{tx}} + \beta_{\text{id}, \mathcal{C}_{\text{pay}}}.$$

Here $\beta_{\text{id}, \mathcal{C}_{\text{pay}}}$ is the off-tail collision bound in (1). This probability averages over honest hash-index generation. The payment-effect implications use the full request and authentic occurrence relation and do not require hash injectivity.

Proof. Causal interpretation and bootstrap. Fix the deployment, direct local code and ζ_e before choosing the environment. Bootstrap’s actual synchronized read, source decoding and ownership checks establish the common authenticated P_0 , unused reservation and unspent U . Section 5.1 proves that the selected decoder satisfies Section 2.11’s prefix-consistency and resumption laws. Complete Ordinals extraction and source-ledger correctness remain the explicit source assumptions. We first verify the five conditions required by Definition 3 for $\mathcal{A}_{\text{ORDI}}$ under these stated requirements.

Condition 1. The bounded typed transfer and reserved/delivered record determine the total transition: the reserved case records the stated delivery, while the delivered case rejects with exactly the same S . The fixed-amount and status queries are deterministic, bounded and read-only. This is an application-state check, not a general BRC-20 balance transition. Source balances remain in the causal decoder.

Condition 2. Successful bootstrap gives version zero at P_0 . The reserved-to-delivered update and rejection preserve that field. Only the common boundary step changes it to the relative ledger height, including at empty blocks.

Condition 3. The specified slots and unique payload encoding give deterministic parsing, and each usable constructor result contains its exact x, w and the same recomputed hash identifier on both sides. Section 2.11’s collision-resistance assumption gives the binding bound $\epsilon_{\text{id}, C_{\text{pay}}}$ for the complete payment context, including all shared replicas and interfaces. The actual input, outputs, template and linked first normal movement determine MatchesRequest_e from the immutable complete record context. These deterministic checks and the operational comparison below remain defined even on hash-collision executions.

Condition 4. After authentication and the nonce check, the common ordered update evaluates δ at the immediately preceding reserved/delivered state. A nonce conflict records its rejection without evaluating δ . Ownership, input and destination correspondence are checked by MatchesRequest_e against the causal source context. Lemma 3 supplies the still-reserved amount. Thus the source debit and credit used below follow from the selected normal-movement rule, not from an additional balance test inside δ . Application rejection preserves S , while the common L, O updates and the source decoder retain their specified effects.

Condition 5. The canonical checkpoint encoding binds this session and relative height, and the common guard requires the movement’s height to exceed the request checkpoint. The source’s unique first consuming movement and the record link provide the first-outcome effect correspondence. Here $W = W_L = \infty$, so there is no finite upper-height or strict-expiry branch and no finite resolution conclusion. These checks establish $\text{AdmissibleApp}(\mathcal{A}_{\text{ORDI}}, \zeta_e, \text{sid}) = 1$ for the common hash-based configuration. No collision-free conditioning is needed for the service comparison.

Service comparison. For each endpoint $q \in \{p, m\}$, let $\mathcal{R}_q$ be the service-record part of Lemma 2’s relation between its real and ideal replica, extended with the ORDI source-decoder state, immutable current event context, complete unsigned template and saved transaction-signing operations and identities. This

pairs that replica's committed and staged states, recovery owner and demands, preparation records, E , queue positions, send permissions and every occupied response connection. Use the product $\mathcal{R}_p \wedge \mathcal{R}_m$, coupling each shared global interface once. It equates the corresponding real and ideal states separately for each endpoint and permits different payer and merchant states.

For complete paired configurations U, V , write the product invariant as

$$\mathcal{R}_{\text{pair}}(U, V) = \mathcal{R}_p(U_p, V_p) \wedge \mathcal{R}_m(U_m, V_m) \wedge g_U = g_V,$$

where g_U, g_V contain the one shared state of each retained global interface. A local step at endpoint q preserves $\mathcal{R}_q$ by the operational lemma and leaves the other replica's local state unchanged. An interface request or delivery uses the same actual identity, original connection and order in both executions. Coupled shared-interface states and tapes therefore produce the same results, observations and scheduling opportunities even when requests from the replicas interleave with independent users' requests. Induction gives $\mathcal{R}_{\text{pair}}(U_n, V_n)$ and equal visible prefixes for every n , without duplicating an interface interaction or imposing a common recovery schedule.

The retained interfaces, common fixed **hk** and coupled constructor coins give the same two-stage construction at the payer replica: obtain or reuse the request certificate, fill and check its payload, then invoke the separate transaction-signing route. A payload/fee/size failure returns the same operation error while preserving approvals, certificates and issued requests. In particular, a failed pre-signing payload check produces no transaction-signing request. Constructor ownership, waiters and atomic transaction retention apply across both stages. A later submission reuses that transaction on its own registration, verification and send connections. If a different transaction has a hash already saved in that replica's E , both systems take the same existing save-conflict branch. The unused-identifier guard keeps each replica's saved identifiers distinct independently of global collisions.

Within each replica pair, equal records and saved decoder states give equal pre-authentication wrapper, checkpoint, height, duplicate and **MatchesRequest_e** checks. Source updates survive skipped records. A surviving candidate uses the shared- $\mathcal{G}_{\text{auth}}$ procedure, whose immutable decisions discharge the historical-authentication requirement in Section 2.11. A zero verification result causes both replicas in the pair to skip that record. Consumed **NoBinding** discards that replica pair's staging and ends its demands with the same **Abort**, preserving its committed checkpoint and decoder state. New public retries repeat the same causal fold and verification interfaces. These recovery rescans neither publish discarded staged results nor create another source movement. Thus the product relation gives zero service-computation error in the common ideal-authentication hybrid. This comparison retains the actual setup, ledger, clock and transaction-signing interfaces.

In particular, after the payer has obtained **Confirmed**, a lagging merchant replica can issue its own ledger read and return **Pending** before recovery completes in both executions. If one endpoint's replica instead consumes **NoBinding**, only

that endpoint's matched real and ideal recoveries and their unresolved demands abort, with issued responses retained on their original connections. The other endpoint's replica keeps any terminal receipt it has already committed. The shared anchor, ledger, ζ_e , ORDI reconstruction programs and initial states, hash configuration and immutable shared- $\mathcal{G}_{\text{auth}}$ decisions satisfy Theorem 1's common-data premises. That theorem equates the payer's and merchant's terminal outcomes when both have obtained them, retaining their distinct identities and connections. Their intermediate statuses and read schedules may differ.

From an applied receipt to a movement. Exclude the authentication and transaction-authorization bad events. By the discharge of Conditions 3 and 4, an eligible payment record passes MatchesRequest_e and contains the first normal movement of e to the request's ordinary recipient t . The full transaction, complete request and authentic occurrence link identify this record's event even if its hash collides with another transaction's hash. Lemma 3 gives its still-reserved amount u . The normal-movement rule debits $b(s)$, credits $b(t)$ and $a(t)$ and consumes the reservation, precisely the delivery represented by the applied transition at the immediate application state. Creation, self-movement, inscription movement into fees, burns and unrelated transfers fail the payment predicate and give no positive receipt.

Any spend of U must, outside the transaction-authenticity event, use authorization produced by the permitted wrapper-signing route. The assumption excluding earlier bare or conflicting authorization is needed here. Let $\prec_\rho$ denote strict event order in the execution. That route enforces

$$\begin{aligned} & \text{Approve}(p, x) \prec_\rho \text{Produce}_{\mathcal{G}_{\text{auth}}}(p, x, w), \\ & \text{Produce}_{\mathcal{G}_{\text{auth}}}(p, x, w) \prec_\rho \text{ProduceTxSignature}(tx) \prec_\rho \text{Consume}(e, tx). \end{aligned}$$

These symbols name the existing approval, protected request-signature production, transaction-signature production and consuming ledger event. The wrapper embeds the certificate before transaction signing. Signed-field binding ties the spending authorization to the protected invoice, recipient, amount and template. This argument uses the separate transaction-spend assumption and the wrapper's prescribed order. Request certificates, transaction signatures, full-byte transaction identifiers and ledger/source occurrences keep their distinct roles.

Uniqueness and completed observation. Ledger double-spend exclusion and the used-inscription rule permit at most one consuming movement of this e . Every structurally eligible payment request in this instance requires that same first normal movement in its own record. Consequently no earlier eligible nonce-conflicting request can already have claimed this instance's nonce without consuming the same unique reservation. Starting from the bootstrapped reserved state, the first authenticated eligible occurrence therefore applies the specified delivery and records its first receipt. The checkpoint was recovered before approval/signing, so its height precedes the later movement, and $W = W_L = \infty$ imposes no finite upper-height guard on that movement. A completed scan covering it obtains the stated TransferApplied relation. A NoBinding abort or a lagging view can delay that observation without establishing a contradictory completed result. Eventual binding and later public retries are required to complete an aborted recovery.

Immutable first outcomes and the causal source interpretation preserve the receipt under rescans and extensions. Repeating the same transaction creates no second source movement. Unrelated source activity may change balances or other reservations but cannot alter this unused inscription's amount or revise its already recorded historical delivery. Formally, if $\text{Moves}(e)$ is the set of consuming movements of this inscription, double-spend exclusion and the used-inscription rule give

$$\text{Effects}(p, \eta) \subseteq \text{Moves}(e), \quad |\text{Moves}(e)| \leq 1 \implies |\text{Effects}(p, \eta)| \leq 1.$$

The exclusive-intent rule fixes the sole approved request to which any such effect is attributed.

Payment-interface correspondence. Conjoin $\mathcal{R}_p \wedge \mathcal{R}_m$ with the relation pairing payer and merchant records to the ideal's invoices, proposals, approved intents, candidates and accepted receipts. Preserve the dispatch map and the exact caller, service endpoint and saved response connection of every network, service and signing operation. For an honest merchant, the absence of the monitored authentication bad event and the injective invoice domain imply that an invoice passing authentication was signed on its permitted route. That route signs only the tuple fixed by `IssueInvoice`, establishing the ideal's requirement that the invoice match an issued tuple. Both systems then apply the same deployment, payer, recipient and amount checks to that tuple, so their final invoice-validation decisions agree. Replayed valid invoices retain their tuple and create no payment by themselves.

An `IssueInvoice` input fixes and signs the same tuple or rejects an identifier conflict. `Prepare` uses the same saved registration/read results and checkpoint to return the same unsigned proposal without starting signing. `Authorize` checks the same invoice/context/intent records, records the same sole approved x before `Submit`, preserves exact repeats and rejects a changed request. The notice is only a candidate pointer. The service comparison accounts for both signing stages and all retained work after an operation abort.

For `Progresspay`, an unapproved or wrong-payer request aborts before query or send. Otherwise both executions retain an owned unfinished query, wait on a route occupied by another invocation, or start the same fresh query at a permitted activation. Only that query's result may answer the payment-progress invocation that owns it. Merchant `CheckPayment` likewise keeps its own request/query binding. Each `Pending` reply completes its matched query and payment invocation in both executions, while recovery that has not aborted may continue. A query `Abort` closes those returns with the same error and grants no submission. A later actual poll starts a new query, and a policy-authorized submission uses the same request and transaction with its own registration, verification and send permission. A terminal result prohibits a successor, while an operation abort clears neither approval nor issued work. Matching confirmed receipts, invoices and the proved movement relation then give the same one-time acceptance, including observation after the movement has already occurred.

Simulator and error. A fixed forwarder preserves declared network and adversarial inputs and observations once under their original roles and replica

connections. Actual public inputs reach the payment ideal’s own rules, and the simulator obtains no approval, owner-signing or honest ledger-send authority. The product and endpoint relations match every finite visible prefix, including independently used interfaces and unfinished requests. These are compatible complete environmentally bounded systems, so iUC composition applies [13, Corollary 1]. Substitute authentication once over the complete context containing both replicas, all participating owners and permitted domains, including invoice signing. Choose a common polynomial cutoff for the authentication and identifier bounds in Appendix A.2 and Section 2.11. Appendix A.2’s authentication bound includes the common negligible tail once, with its forgery budget evaluated at this cutoff. Each $\mathcal{R}_q$ pairs unset timestamps, original clock requests and their actual returned values under Section 4.3. Earlier simulators’ clock-affecting actions reach the same retained clock once and are matched without assuming an intermediate $t_0 = \tau_{\text{arr}}$. That equality remains the sampling condition on the advertised execution class, with τ_{arr} used only in the trace. The remaining transaction-authenticity failure has probability at most ϵ_{tx} . A union bound gives the payment-realization error without independence. Hash collisions do not change this comparison because both systems run the same total hash-based programs. The simulator is fixed before the admitted environment. Infinite lifetimes supply no finite confirmation or resolution guarantee. For the resulting coupling, these conclusions give the explicit bound

$$\begin{aligned} \text{Adv}_{\mathcal{E}}(\Pi_{\text{pay}}, F_{\text{pay}}, S_{\text{pay}}; \lambda) &\leq \Pr[\text{View}_{\mathcal{E}, \Pi_{\text{pay}}} \neq \text{View}_{\mathcal{E}, S_{\text{pay}}, F_{\text{pay}}}] \\ &\leq \Pr[\text{Bad}_{\text{pay}}] \\ &\leq \epsilon_{\text{auth}} + \epsilon_{\text{tx}}. \end{aligned}$$

Receipt-identifier binding. By Section 2.11’s collision-resistance assumption, the probability that distinct transaction bytes processed in $\mathcal{C}_{\text{pay}}$ share a receipt identifier is at most $\epsilon_{\text{id}, \mathcal{C}_{\text{pay}}}$. For the joint claim, split all failures at the common cutoff: the tail and off-tail authentication forgery terms are already bounded by ϵ_{auth} , transaction-authorization failure contributes at most ϵ_{tx} , and off-tail hash collision contributes at most $\beta_{\text{id}, \mathcal{C}_{\text{pay}}}$. If the authentication bound is clipped at one, the joint inequality is immediate. Otherwise this partition gives the stated sum, charging neither a second tail nor one hash term per replica. No independence between these events is required.

5.2 Consensus ordering and blockchain execution

Section 5 separates ordered proposals from deterministic execution. The ordering ledger supplies Appendix B.1’s interface. Application balances, contract execution and execution-state roots are checked by the state machine below.

The ordering interface. Specialize the unchanged parameterized $\mathcal{G}_{\text{LEDGER}}$ [3, Secs. 4.1–4.3] by choosing admission rules `Validate`, complete record format `Blockify` and extension policy `ExtendPolicy`. The chosen parameters must satisfy their admissibility conditions and the requirements of Appendix B.1

and Section 2.11, even with permissive application admission. These requirements include block boundaries, ledger identifiers, recorded submitters, clock/registration/synchronization behavior, independent public roles and delayed reads.

Execution state machine. Fix bounded domains with specified encodings for proposers, complete block bytes and validation context, together with a proposer predicate $\text{MayPropose}(p)$ and a pinned total deterministic program V . Its input is the complete execution data A , validation context κ and candidate B . It returns either $\text{Invalid}(f)$ or $\text{Valid}(A', \kappa', \rho_B)$. V must check target-chain authorization, parent/context, complete block validity and the computed commitments under the selected rules. ρ_B contains every inner transaction outcome, including specified effects of failed inner execution. Bounds on block size, execution-data size and execution work make V total within the declared work budget.

The initial application state is $Z_0 = (0, A_0, \kappa_0, \beta_0, 0, [])$, with fields ($\text{ver}, A, \kappa, \beta, n, \log$) for AsyncLS version, execution data, validation context, accepted tip, accepted-block count after the initial tip, and accepted block records. Initial tip β_0 and execution data (A_0, κ_0) must correspond at the immutable setup anchor. Thus $n = 0$ at β_0 , independent of that tip's absolute target-chain height. The request is $r = \text{Propose}(\beta_{\text{parent}}, B)$, with complete B in the required encoding. Define $\text{bid}(B) = \text{Encode}(\text{ALS/block}, B)$ for this construction. Using a target-chain digest instead requires its own binding requirement. The unsigned request includes a fresh nonce and recovered checkpoint, and only the actual proposer invokes Submit . The data-only $\text{TransactionCreation}(x)$ constructs $tx = \text{Pack}(x, w)$ with $W_L = \infty$. Its derived accessor $tx.\text{id}$ hashes the finished envelope bytes under Section 2.11's common public index and is not serialized as an additional field. During reconstruction, eligible historical requests use the common shared- $\mathcal{G}_{\text{auth}}$ verification procedure of Section 3.3. A configured finite W limits only the represented AsyncLS execution effect. Taking $W = \infty$ gives the unbounded instance. The carrier's represented effect is the AsyncLS execution transition.

At block i of the ordering ledger, AsyncLS sets only $Z.\text{ver} = i$. For the first eligible request, δ_{chain} rejects a request if its fields are malformed, its proposer fails MayPropose or $\beta_{\text{parent}} \neq \beta$, returning typed reasons and unchanged application state Z . Otherwise it runs $V(A, \kappa, B)$. An Invalid result leaves Z unchanged. A Valid result sets $A \leftarrow A'$, $\kappa \leftarrow \kappa'$, $\beta \leftarrow \text{bid}(B)$, $n \leftarrow n + 1$ and appends (n, β, B, ρ_B) to $\log$, preserving ver . Denote this updated application state by Z' . Its complete application output is $o_B = (n, \beta, \rho_B)$. Thus

$$\delta_{\text{chain}}(p, Z, r) = \begin{cases} (Z', \text{Applied}(\text{BlockProcessed}(o_B))) & \text{if accepted,} \\ (Z, \text{Rejected}(f)) & \text{otherwise.} \end{cases}$$

Queries Tip , State and $\text{Block}(k)$ return respectively (n, β) , (A, κ) and $\log[k]$ for $1 \leq k \leq n$ (or a typed absent value otherwise). The list indexes blocks accepted after β_0 starting at 1. These three queries are total read-only functions on the accepted application state Z . The outer receipt remains $(\text{gid}, x, \text{txid}, (i, j), z)$. It

is distinct from the inner receipt vector ρ_B . All outer branches for nonce conflicts, duplicates, records beyond the last eligible height, and authentication follow the common AsyncLS algorithm exactly.

Application admissibility. Denote this complete application tuple by $\mathcal{A}_{\text{chain}}$. It comprises the stated domains, the core $\mathcal{M}$ with initial state Z_0 and transition δ_{chain} , and the canonical data-only transaction procedures. We verify the five conditions required by Definition 3 from this construction as follows.

Condition 1 follows from the bounded encodings, total deterministic V and explicit update: every rejecting branch returns exactly Z , and the three queries are bounded read-only functions. For an accepted block, the application update retains all inner outcomes and their specified execution effects in A' and ρ_B , including failed inner executions. Condition 2 follows from $Z_0.\text{ver} = 0$, preservation of that field by δ_{chain} , and the service's boundary assignment, which changes none of $(A, \kappa, \beta, n, \log)$.

For Condition 3, the canonical data-only envelope parses deterministically to its exact request and certificate and recomputes the hash identifier from the complete bytes. For the full execution-layer context $\mathcal{C}_{\text{chain}}$, Section 2.11's collision-resistance assumption gives the complete-byte binding bound $\epsilon_{\text{id}, \mathcal{C}_{\text{chain}}}$ under the same shared hash configuration. The data-only decoder preserves every complete record and boundary with singleton source state and empty event/link lists, so its emitted contexts satisfy Section 2.11's laws. The canonical context predicate checks the complete envelope against its request. The separately defined $\text{bid}(B)$ injectively encodes the complete block bytes. Substituting a target-chain digest would still require its own binding argument. For Condition 4, the common nonce test precedes δ_{chain} , whose proposer, parent and validation checks use the request and immediate (A, κ, β) under the pinned rules. Nonce conflict bypasses δ_{chain} .

For Condition 5, the canonical checkpoint binds the session and relative ordering height, and the common guards enforce $h < i \leq D(x)$ and $O[x] = \perp$. The carrier's only represented effect is this AsyncLS application transition, so the same guards enforce its first-outcome and upper-height correspondence. With finite W and $W_L = \infty$, complete processing beyond $D(x)$ without an outcome excludes that represented effect. With $W = \infty$, there is no strict-expiry conclusion. Thus $\text{AdmissibleApp}(\mathcal{A}_{\text{chain}}, \zeta, \text{sid}) = 1$ under the stated configuration. No realization of a target chain has been assumed or proved by this check.

Lemma 4 (Execution-layer invariant). *Fix the stated data-only interpretation, authenticated initial (A_0, κ_0, β_0) and total deterministic validator V . For every completely processed authentic permanent ordering prefix P_H , let $(u_1, \dots, u_n)$ be the ledger-ordered subsequence of exactly the first eligible proposals passing the outer nonce/outcome, proposer, immediate-parent and complete-validation checks at their respective preceding states. Write $B_k, x_k, \text{txid}_k, (i_k, j_k)$ for the block, request, transaction identifier and occurrence of u_k . Starting at $(A^{(0)}, \kappa^{(0)}, \beta^{(0)}) = (A_0, \kappa_0, \beta_0)$, the accepted sequence satisfies, for $1 \leq k \leq n$,*

$$\begin{aligned}\beta_{\text{parent},k} &= \beta^{(k-1)}, \\ V(A^{(k-1)}, \kappa^{(k-1)}, B_k) &= \text{Valid}(A^{(k)}, \kappa^{(k)}, \rho_k), \\ \beta^{(k)} &= \text{bid}(B_k).\end{aligned}$$

Let $o_k = (k, \beta^{(k)}, \rho_k)$ be the complete application output of accepted proposal k . The reconstructed application state and applied outer receipts are

$$\begin{aligned}Z_H &= (H, A^{(n)}, \kappa^{(n)}, \beta^{(n)}, n, [(k, \beta^{(k)}, B_k, \rho_k)]_{k=1}^n), \\ o_H[x_k] &= (\text{gid}, x_k, \text{txid}_k, (i_k, j_k), \text{Applied}(\text{BlockProcessed}(o_k))).\end{aligned}$$

Every applied outer receipt arises from exactly one member of this subsequence and contains all its inner outcomes. Other records leave the execution data, accepted tip, count and log unchanged. Boundaries change only the reserved version, giving $Z_H.\text{ver} = H$.

Proof. Use the data-only decoder specified above when verifying Condition 3: singleton source state, unchanged boundaries and complete records, and empty event/link lists. Its prefix-consistency and resumption laws, together with Section 3.3's direct authentication procedure, allow application of Lemma 1.

Induct over the ordering prefix's boundaries and records, starting with $n = 0$, tip β_0 and an empty log. By Condition 2, a boundary sets only the reserved version to its relative height. It changes neither execution data nor the accepted count or tip, including at an empty block. A malformed, context-invalid, duplicate or late outer record is skipped. An authentication rejection is skipped, and an authenticated nonce conflict records a rejection without invoking δ_{chain} . Each completed case preserves the execution fields and accepted log, although the nonce-conflict branch records its outer outcome.

By Condition 4, every remaining first eligible proposal invokes δ_{chain} once at its immediate state. Let $Z = (i, A, \kappa, \beta, k, \log)$ be that state. Malformed fields, an unauthorized proposer, a wrong parent or V returning *Invalid* give $\delta_{\text{chain}}(p, Z, r) = (Z, \text{Rejected}(f))$, as required by Condition 1. This leaves Z unchanged but retains the common nonce and outer-outcome updates. For a proposal passing these checks with $V(A, \kappa, B) = \text{Valid}(A', \kappa', \rho_B)$, set $\beta' = \text{bid}(B)$ and $o'_B = (k+1, \beta', \rho_B)$. The update is exactly

$$\begin{aligned}Z' &= (i, A', \kappa', \beta', k+1, \log \parallel [(k+1, \beta', B, \rho_B)]), \\ z &= \text{Applied}(\text{BlockProcessed}(o'_B)).\end{aligned}$$

Thus the induction appends one accepted record, applies the stated recurrence and copies the complete output into the first outer receipt. Any fee or nonce effects of a valid block's failed inner execution remain in A' and ρ_B .

These branches establish both directions of the subsequence claim: every appended record passed all named tests, and every first eligible proposal passing them appends one record in its ledger order. Existing-outcome checks prevent reevaluation of the same request. After all items finish, atomic commitment gives version H and the stated query values for the accepted log and execution state. More precisely, replicas sharing the logical session and common reconstruction data of the session-configuration paragraph in Section 2.11, and the same complete authenticated prefix P_H reconstruct this same Z_H by Lemma 1. Condition 1 then gives identical **Tip**, **State** and **Block(k)** answers for identical query arguments, and Condition 2 gives their common outer version H . Equal heights alone do not establish equality of execution data. Validation accepts or rejects each complete target block. The invariant is relative to the configured V . It uses complete candidate blocks and the exact $\text{bid}(B)$ encoding for parent/tip identity. Outer transaction hashes are recomputed receipt fields, not keys for selecting or deduplicating candidate blocks. The invariant and these query equalities therefore hold even if distinct outer transactions have the same hash.

The envelope authenticates the **AsyncLS** caller, while V checks target-chain authorization. The invariant identifies the transition's complete execution effects. The verification of Condition 5 explains why strict expiry excludes that represented transition. Carrier fees/funding remain distinct from V 's execution rules. For this interpreter and data-only format, Proposition 1 or 2 supplies the component realization under Corollary 1's configuration, identifier, honesty and request-arrival-clock assumptions. Finite resolution requires Section 2.16's progress bounds.

The validator must retain every effect of an accepted target-chain execution, including fee and nonce effects of inner Ethereum Virtual Machine (EVM) reverts [7,12,19], and reject invalid commitments under pinned fork rules. Pending validation is not rejection [18]. For Bitcoin, full script, spending, value, coinbase, header and contextual validity is required [10]. Implementing a target chain requires establishing these properties of V .

Checkpoints and target-chain tips. The checkpoint c_H identifies the complete ordering-ledger prefix and its reconstructed Z_H . For an authenticated initial tip of target-chain height h_{init} , the accepted target-chain height is $h_{\text{init}} + n$.

A target-chain embedding must relate the exported permanent history to execution-valid accepted blocks and their parents, including validity feedback to consensus where required [18,10]. It must state its finalized or stable-prefix interpretation and any residual common-prefix error [3]. The exported history must satisfy **AsyncLS**'s permanence assumption.

Conditional replacement of the ordering ledger. Replacement requires a separate complete iUC realization of the declared ordering interface, including

actual callers, public/adversarial roles, submission metadata, authentic reads, synchronization, clock effects and unresolved requests. For compatible complete environmentally bounded compositions, iUC composition [13, Corollary 1] carries its error and assumptions through AsyncLS and its client, retaining consensus and authentication errors.

6 Conclusion and future directions

AsyncLS exposes immutable request outcomes above a configured ledger. Its reference protocol realizes the monolithic ideal under the stated ledger, clock, authentication, setup, authorization, and effect assumptions. Our trusted and local **Relay** and **Adapter** realizations compose with authentication. Finite terminal resolution requires separate progress bounds.

Future work can relate the authorization, checkpoints, and effects of AsyncLS instances across ledgers. The present theorem assumes permanent history. Supporting rollback requires a corresponding specification and proof.

References

1. Andrews, J., Ciampi, M., Zikas, V.: Etherless Ethereum tokens: Simulating native tokens in Ethereum. *Journal of Computer and System Sciences* **135**, 55–72 (2023). <https://doi.org/10.1016/j.jcss.2023.02.001>, <https://eprint.iacr.org/2021/766>, section and theorem references follow the linked full version, Cryptology ePrint Archive 2021/766
2. Arapinis, M., Gkaniatsou, A., Karakostas, D., Kiayias, A.: A formal treatment of hardware wallets. In: *Financial Cryptography and Data Security. Lecture Notes in Computer Science*, vol. 11598, pp. 426–445. Springer (2019). https://doi.org/10.1007/978-3-030-32101-7_26, <https://eprint.iacr.org/2019/034>, section references follow the linked ePrint 2019/034 version, accessed 8 September 2026
3. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. *Journal of Cryptology* **37**(2) (2024). <https://doi.org/10.1007/s00145-024-09493-7>, <https://link.springer.com/article/10.1007/s00145-024-09493-7>
4. Baum, C., Chiang, J.H.y., David, B., Frederiksen, T.K.: Eagle: efficient privacy preserving smart contracts. In: Baldimtsi, F., Cachin, C. (eds.) *Financial Cryptography and Data Security*. p. 270–288. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-47754-6_16, <https://eprint.iacr.org/2022/1435>, technical references follow ePrint 2022/1435, accessed 9 September 2026
5. Baum, C., David, B., Dowsley, R.: Insured MPC: efficient secure computation with financial penalties. In: Bonneau, J., Heninger, N. (eds.) *Financial Cryptography and Data Security*. p. 404–420. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51280-4_22, <https://eprint.iacr.org/2018/942>, technical references follow ePrint 2018/942, accessed 9 September 2026
6. Baum, C., David, B., Frederiksen, T.K.: P2dex: privacy-preserving decentralized cryptocurrency exchange. In: Sako, K., Tappenhauer, N.O. (eds.) *Applied Cryptography and Network Security*. p. 163–194. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-78372-3_7, <https://eprint.iacr.org/2021/283>, technical references follow ePrint 2021/283, accessed 9 September 2026
7. Beregszaszi, A., Mushegian, N.: EIP-140: REVERT instruction (2017), <https://eips.ethereum.org/EIPS/eip-140>
8. Best in Slot: Open protocol indexer: Historical BRC-20 indexer and api (2024), <https://github.com/bestinslot-xyz/OPI/tree/a398daa6f04c000c626584c089435d83a94c5142>, revision a398daa6f04c000c626584c089435d83a94c5142, dated 17 October 2024, modules/brc20_index/brc20_index.py and modules/brc20_api/api.js. Accessed 8 September 2026
9. Bitcoin Core Project: Mempool retention and transaction finality rules (2026), <https://github.com/bitcoin/bitcoin/tree/33a363ea250839ca31ea043b7500789d2e5d844b>, revision 33a363ea250839ca31ea043b7500789d2e5d844b, src/kernel/mempool_options.h and src/consensus/tx_verify.cpp. Accessed 8 September 2026
10. Bitcoin Developer Documentation Contributors: Developer guide: Block chain, https://developer.bitcoin.org/devguide/block_chain.html, block validation, transactions, and chain selection. Accessed 9 September 2026

11. Brown, R.G., Carlyle, J., Grigg, I., Hearn, M.: Corda: An introduction. Tech. rep., R3 (Aug 2016), <https://docs.r3.com/en/pdf/corda-introductory-whitepaper.pdf>
12. Buterin, V., Conner, E., Dudley, R., Slipper, M., Norden, I., Bakhta, A.: EIP-1559: Fee market change for ETH 1.0 chain (2019), <https://eips.ethereum.org/EIPS/eip-1559>
13. Camenisch, J., Krenn, S., Küsters, R., Rausch, D.: iUC: Flexible universal composability made simple. In: Advances in Cryptology – ASIACRYPT 2019, Part III. Lecture Notes in Computer Science, vol. 11923, pp. 191–221. Springer, Kobe, Japan (2019). https://doi.org/10.1007/978-3-030-34618-8_7, <https://eprint.iacr.org/2019/1073>
14. Canetti, R., Shahaf, D., Vald, M.: Universally composable authentication and key-exchange with global PKI. Cryptology ePrint Archive, Report 2014/432 (2014), <https://eprint.iacr.org/2014/432>, full version dated 9 October 2014
15. Canetti, R., Shahaf, D., Vald, M.: Universally composable authentication and key-exchange with global pki. In: Public-Key Cryptography – PKC 2016, Part II. Lecture Notes in Computer Science, vol. 9615, pp. 265–296. Springer (2016). https://doi.org/10.1007/978-3-662-49387-8_11
16. domo: BRC-20 experiment: Original rules (2023), <https://domo-2.gitbook.io/brc-20-experiment/brc-20.md>, original-author Markdown snapshot archived with this manuscript, SHA-256 661c67df3852b411e5c4768c8123b2713f7b78b75a20a7e1f05f7c3c2eb371a6. Retrieved 8 September 2026
17. Dziembowski, S., Faust, S., Hostáková, K.: General state channel networks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 949–966. ACM (2018). <https://doi.org/10.1145/3243734.3243856>, <https://eprint.iacr.org/2018/320>, technical references follow ePrint 2018/320, accessed 9 September 2026
18. Ethereum Execution API Contributors: Engine API: Paris, <https://github.com/ethereum/execution-apis/blob/main/src/engine/paris.md>, payload validation and fork-choice interfaces. Accessed 9 September 2026
19. Ethereum Execution Specifications Contributors: Cancun: Transaction processing and EVM execution, <https://github.com/ethereum/execution-specs/tree/0cc100eb190b64b23baba72dac0165652eaec252/src/ethereum/forks/cancun>, pinned execution specification, commit 0cc100e: process.transaction and process.message. Accessed 9 September 2026
20. Ethereum.org Contributors: Nodes and clients, <https://ethereum.org/developers/docs/nodes-and-clients>, local node validation and wallet access. Accessed 21 September 2026
21. Ethereum.org Contributors: Nodes as a service, <https://ethereum.org/developers/docs/nodes-and-clients/nodes-as-a-service>, provider-operated node access and client-held signing keys. Accessed 21 September 2026
22. Garay, J., Kiayias, A., Leonardos, N.: The Bitcoin backbone protocol: Analysis and applications. Journal of the ACM **71**(4), 1–49 (2024). <https://doi.org/10.1145/3653445>
23. Geth Authors: Introduction to Clef, <https://geth.ethereum.org/docs/tools/clef/introduction>, separate transaction signing and owner approvals. Accessed 21 September 2026
24. Graf, M., Rausch, D., Ronge, V., Egger, C., Küsters, R., Schröder, D.: A security framework for distributed ledgers. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. ACM (2021). <https://doi.org/10.1145/3460120.3485362>, <https://eprint.iacr.org/2021/145>

25. Hansen, A.B., Nielsen, J.B., Simkin, M.: OCash: Fully anonymous payments between blockchain light clients. In: Jager, T., Pan, J. (eds.) *Public-Key Cryptography – PKC 2025*. Lecture Notes in Computer Science, vol. 15678, pp. 169–202. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-91832-2_6, <https://eprint.iacr.org/2024/246>, technical references follow ePrint 2024/246, accessed 9 September 2026
26. Karakostas, D., Kiayias, A., Larangeira, M.: Account management in proof of stake ledgers. In: *Security and Cryptography Networks*. pp. 3–23. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-57990-6_1, <https://eprint.iacr.org/2020/525>, section and theorem references follow the linked full version dated 5 May 2020
27. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: *2016 IEEE Symposium on Security and Privacy*. pp. 839–858. IEEE (2016). <https://doi.org/10.1109/SP.2016.55>, <https://eprint.iacr.org/2015/675>, technical references follow ePrint 2015/675, accessed 9 September 2026
28. Lau, J., Wuille, P.: BIP 143: Transaction signature verification for version 0 witness program (2016), <https://github.com/bitcoin/bips/blob/24e96e870fffaa257b465ce1f0370c14aac588e8/bip-0143.mediawiki>, specification and signing-field definitions at the cited revision. Byte-identical to the source retrieved 8 September 2026, verified 9 September 2026
29. Lee, C., Park, S.J., Kejriwal, A., Matsushita, S., Ousterhout, J.: Implementing linearizability at large scale and low latency. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. pp. 71–86. ACM (2015). <https://doi.org/10.1145/2815400.2815416>
30. Overko, R.: A study on shared objects in Sui smart contracts. In: *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. pp. 1–7. IEEE (2024). <https://doi.org/10.1109/ICBC59979.2024.10634357>
31. Rauchs, M., Glidden, A., Gordon, B., Pieters, G.C., Recanatini, M., Rostand, F., Vagueur, K., Zhang, B.Z.: Distributed ledger technology systems: A conceptual framework. *SSRN Electronic Journal* (2018). <https://doi.org/10.2139/ssrn.3230013>
32. Ravi, D., Ramachandran, S., Vignesh, R., Falmari, V.R., Brindha, M.: Privacy preserving transparent supply chain management through Hyperledger Fabric. *Blockchain: Research and Applications* **3**(2), 100072 (2022). <https://doi.org/10.1016/j.bcra.2022.100072>
33. Tikhomirov, S.: Ethereum: State of knowledge and research perspectives. In: *Foundations and Practice of Security*. pp. 206–221. Springer (2018). https://doi.org/10.1007/978-3-319-75650-9_14

A Shared authentication

A.1 One shared authentication functionality

The owner-bound $\mathcal{G}_{\text{auth}}$ provides Register, Sign, and Verify. It adapts owner-bound certification [14, Sec. 4.1, Fig. 4], with explicit readiness, a missing-registration result and the iUC signer-corruption convention below. Appendix A.2 proves realization by the signature implementation.

The configuration fixes bounded encodings for party identifiers, messages m and certificates, and binds each party p to its actual owner identity ω_p and authorized response connections. Applications may use distinct encoded message domains through this shared functionality. All allowed signing domains count in the security bound for that owner’s key.

Authentication records and actual callers The global authentication functionality keeps owner initialization/publication phases, corruption flags, the set Signed_ω of messages for which protected signing has produced a certificate, and previously set decisions $\text{Cert}_\omega[m, w] \in \{0, 1\}$. The functionality records the proposed initialization value separately from the immutable value returned by the board. Readiness requires them to match. Positive binding caches and pending retrievals are indexed by the actual verifier’s response connections, so each verifier uses its own privately received registration result.

The setup uses the authenticated global first-write board of [14, Fig. 3]. A publisher registers only under its actual identity. Every permitted machine instance keeps its public retrieval role. Publication notifies the adversary and stores only the first value after permission is granted under that board construction’s rules. A retrieval saves the board value at its actual read step, and delayed delivery preserves that value. A bottom value means that no key was registered at that read step. Thus an old bottom response may follow successful registration. A positive immutable binding, once returned by the board, is not erased by such a response. The board’s independently visible interactions and its permitted corruption operations remain part of the complete protocol system.

An owner-only Register or Sign call must arrive on the connection authorized for ω_p . The primitive ignores malformed inputs or invalid owner identities without a successful result. The public Verify interface starts neither owner registration nor signing.

Signer corruption and existing connections. The recipient of each configured owner connection implements its iUC signer role. The existing public Verify recipients implement verifier roles. These roles use the existing configured entities and retain distinct response connections even when they share a machine [13, Secs. 3.2–3.3 and App. B, Figs. 7–9]. We use dynamic corruption and the entity-specific hooks of [13, App. A]. The adversarial $\text{Corrupt}(p)$ denotes the framework corruption request targeting that owner’s signer. The corruption permission excludes verifier roles and owners assumed honest by the claim. Signer corruption compromises this signer’s local request-signing state, not the board, other owners

or a separately declared transaction-signing interface. Initial corruption is decided before owner initialization or service processing, with no generated key. The framework’s corruption-status exchanges are retained on both sides.

After accepted corruption, the framework handles that signer’s I/O before the service code: an incoming message is forwarded on its existing network connection as `CorrMsgForward` with its actual sender [13, App. I, Figs. 22–23]. The framework’s `AllowAdvMessage` uses the existing configured authorization relation. Public verification and board retrieval keep their declared permissions, while publication and private returns keep their actual-owner checks. Corruption grants no other owner’s route or honest ledger-send connection. Thus, later owner-route inputs and old board responses remain observable. The adversary may compute certificates offline and use its legitimate board and public verification roles. Such offline computation leaves the authentication records unchanged. `Verify` still executes at its configured verifier role, including for a corrupted signing owner.

Corruption preserves received observations, counters, saved inputs, occupied response connections and produced results. A pending owner continuation stops making protected productions. Its issued board operations are not cancelled or rebound: their responses reach the original, now-corrupted entity and follow the framework forwarding rule. A result produced before corruption retains its recorded value, certificate decision and one-time `AuthOutput`, even if delivery is later. The adversary controls delivery and subsequent I/O on the corrupted signer’s existing connections. These sends remain adversarial traffic. Late ideal initialization/signing control responses cannot resume the corrupted owner’s service code. Verifier continuations and responses remain active under their own roles.

At initial corruption, the ideal’s `LeakedData` returns the initial input and actual sender, as in the framework default. At a later corruption it exposes only the targeted owner’s saved operations, readiness/binding, route counters and owner-specific message history. It exposes neither another owner’s data nor verifier decision tables and contains no secret key or signature-scheme coins. The framework includes this value in its existing network confirmation (`CorruptionStatus`, `true`, ℓ). Appendix A.2 specifies concrete leakage and how the simulator supplies its existing key in this same confirmation. An absent key is never inferred from a board value. Corruption is irreversible and preserves previously fixed certificate decisions.

Saved authentication operations. Authentication invocations follow the common saved-operation and response-connection rules of Section 2.13.

Register, Sign, and Verify

Description of the shared functionality $\mathcal{G}_{\text{auth}}$:

Participating roles: Signer roles at the configured owner-connection recipients and verifier roles at the public verification recipients. Sharing a machine does not merge them.

Corruption model: Dynamic signer corruption under Appendix A.1.

Protocol parameters: The fixed encodings and actual owner/connection map above.

Operation-structure adaptation of the signer and verifier service:

Implemented role(s): Each configured owner recipient implements its signer role. Each public verification recipient implements its verifier role, on the original response connections.

Subroutines: The authenticated first-write registration board, with its existing publisher and retrieval permissions.

Internal state: Owner phases, corruption flags, Signed_ω , Cert_ω , binding caches, route counters and saved operations as specified above.

Corruption behavior: Appendix A.1 fixes forwarding, leakage and the transfer of pending owner operations.

Initialization: The authenticated registration board, encodings and owner/connection map fixed above.

EntityInitialization: Owner setup at the first permitted owner initialization or lazy signing initialization below. These labels retain the existing operation triggers.

Adversarial observations:

- 1 On each Register, Sign or Verify input admitted to the uncorrupted recipient's service code, save its full input, actual caller and return connection. Allocate $cid = (\text{caller-route}, n_{\text{call}})$ and emit $\text{AuthInput}(cid, op, args)$, then yield. The route is its declared connection identity, and the counter is local to that route in this $\mathcal{G}_{\text{auth}}$ instance. Resumption preserves cid . Each new service invocation increments the counter. Corruption forwarding does not.
- 2 When the operation produces a result, atomically save that result and the updated authentication records and emit $\text{AuthOutput}(cid, result)$ once, then yield before delivery on the saved return connection. Delivery consumes the saved result without rerunning the operation.

on Register (p) from the actual owner ω_p :

- 3 Reuse completed readiness. Otherwise, keep its pending initialization. On first uncorrupted initialization, send $(\text{Initialize}, p)$ to the simulator/adversary and wait for matching $(\text{Initialized}, p, v)$. Save (ω_p, v) once and publish through that owner's actual setup connection. A repeat cannot replace the value.
- 4 Obtain the matching positive setup response on the owner's connection and return $(\text{Registered}, p)$. A captured bottom response leaves readiness pending and permits a later retrieval. A different immutable value returns $(\text{BindingError}, p)$. Appendix A.1 governs corruption before completion.

on Sign (p, m) from the actual owner ω_p :

- 5 Enforce the owner check and setup-based readiness above while the signer is uncorrupted. Lazy initialization is allowed for an authorized application. Corruption transfers the pending owner operation under Appendix A.1 instead of executing a corrupted-signing branch.

```

6 Send  $(\text{Sign}, p, m)$  to the simulator/adversary and wait for matching
   $(\text{Signature}, p, m, w)$ . Reject certificate bytes that do not use the specified encoding.
  If  $\text{Cert}_{\omega_p}[m, w] = 0$ , return  $(\text{SigningError}, p, m)$  without changing the decision.
7 Otherwise add  $m$  to  $\text{Signed}_{\omega_p}$ , set  $\text{Cert}_{\omega_p}[m, w] = 1$ , and return
   $(\text{Signature}, p, m, w)$ .
on Verify  $(p, m, w)$  from a permitted verifier:
8 Obtain the positive binding returned to that verifier by setup, or use its existing
  immutable binding. A delivered bottom returns  $(\text{Verified}, p, m, \text{NoBinding})$ 
  without fixing an entry in the table of certificate decisions. Malformed API inputs
  are rejected, while every byte string in the required encoding remains a
  verification candidate.
9 After positive binding, send  $(\text{Verify}, p, m, w)$  and wait for matching
   $(\text{Verified}, p, m, b)$  with  $b \in \{0, 1\}$  for every invocation. Preserve an existing entry.
  Otherwise, if the signer is uncorrupted and  $m \notin \text{Signed}_{\omega_p}$ , record 0. In every
  remaining case record  $b$ .
10 Return  $(\text{Verified}, p, m, \text{Cert}_{\omega_p}[m, w])$ . No branch obtains an owner-only connection.
    
```

Fig. 7. The shared authentication functionality $\mathcal{G}_{\text{auth}}$.

These service phases apply to independent applications, cache hits and authentication errors. Ignored inputs produce no result or **AuthOutput**. A delivered **NoBinding** completes this **Verify** without a Boolean decision. Its later retry is a new **Verify** invocation with its own counter and response connection. **AsyncLS** invokes that retry only under a later actual public call after aborting the affected invocation or recovery (the saved-invocation procedure in Section 2.13). Internal readiness during **Sign** is not a **Register** invocation. Observation messages are correlated with service invocations through declared routes and their counters. The receipt, production and delivery boundaries grant no completion, cancellation or signing authority. Initialization, signing and verification control exchanges with the simulator determine results under the rules above and remain separate from observations.

For an honest owner, a positive **Verify** result returned by $\mathcal{G}_{\text{auth}}$ implies an earlier actual **Sign** production on the same message. Ordinary unforgeability does not require that the earlier protected signing step produced these exact certificate bytes.

A.2 Realizing the shared authentication functionality

Signing executes in the declared owner’s protected boundary, while verification uses public bindings. The implementation retains Appendix A.1’s authenticated registration setup and public role. That setup establishes the identity binding.

The source construction uses explicit keys and a local signing module [14, Sec. 4.2, Fig. 7 and Claim 4.3]. The proceedings version is [15]. Our **Register** operation and **NoBinding** result separate readiness from verification. The service emits observations when it receives inputs and produces results. Appendix A.1 uses iUC’s signer-role corruption, not the source’s postponed simulation of corrupted signing [14, Claim 4.1]. The lemma below establishes realization of this shared iUC interface.

Fix $\text{Sig} = (\text{KeyGen}, \text{Sign}, \text{Ver})$ with specified bounded encodings, polynomial-time algorithms, perfect correctness, deterministic total verification, and existential unforgeability under chosen-message attack (EUF-CMA). Let λ be the security parameter, p an application party, m an encoded message and w a certificate in the specified encoding domain. The configuration assigns p its actual signing owner ω_p . Write (sk_p, v_p) for that owner's secret and verification keys when generated by $\text{KeyGen}(1^\lambda)$. The local algorithm $\text{Sign}(sk, m)$ signs m with secret key sk , whereas $\mathcal{G}_{\text{auth}}.\text{Sign}(p, m)$ is the owner-bound interface call. The algorithm $\text{Ver}(m, w, v)$ verifies w on m under verification key v . Every signature produced under an honestly generated key verifies and lies in the configured bounded certificate domain. The bounds defined below may depend on the fixed complete context, as in Section 4.

Concrete operations and global setup Write $\Pi_{\mathcal{G}_{\text{auth}}}$ for the common implementation. Its authentication operations are **Register**, **Sign**, and **Verify** on the same actual caller and response connections as $\mathcal{G}_{\text{auth}}$, with the same signer/verifier role map and framework corruption dispatch. The setup board keeps its public publisher/retriever role. Ordinary operations expose no secret key or raw KeyGen interface. After signer corruption the adversary may run $\text{Sign}(sk, m)$ offline if it knows sk , without an $\mathcal{G}_{\text{auth}}.\text{Sign}$ production. Public mathematical verification and permitted setup/verification connections remain available.

Uncorrupted service code follows Appendix A.1's phases on its own adversarial interface. The iUC corruption dispatcher precedes that code. Observations require no acknowledgment or permission change.

Description of the implementation $\Pi_{\mathcal{G}_{\text{auth}}}[\text{Sig}]$:

Participating roles: The same configured signer and verifier roles as $\mathcal{G}_{\text{auth}}$.
Corruption model: The signer dispatch and saved-operation transfer of Appendix A.1, with concrete leakage specified below.
Protocol parameters: The signature scheme, bounded encodings and owner/connection map fixed above.

Operation-structure adaptation of the concrete signer and verifier service:

Implemented role(s): The configured owner recipients implement signer roles and the public verification recipients implement verifier roles, as in $\mathcal{G}_{\text{auth}}$.
Subroutines: The unchanged authenticated registration board on the existing owner and verifier connections.
Internal state: Generated owner keys, readiness and binding records, saved operations and owner-specific histories as specified here.
Initialization: The retained registration board and fixed owner/connection map above.
EntityInitialization: The existing first permitted owner initialization generates one key pair and completes board readiness, as specified in **Register** below. Lazy **Sign** initialization uses that same owner instance and trigger.
Framework dispatch and observations: The iUC corruption dispatcher precedes uncorrupted service code. Input and output observations follow Appendix A.1.

Main:

- 1 **Register.** At the uncorrupted signer's first permitted owner initialization, generate and keep (sk_p, v_p) once.
 - (a) Publish v_p through that owner's actual first-write setup connection. The owner becomes ready only after its saved retrieval returns that same value. Cache readiness after that response.
 - (b) Preserve pending initialization on repeats. A captured bottom allows only a later bounded retry, and a conflicting first value gives the binding error without regeneration. Prior initialization in another admitted session reuses the same owner instance.
- 2 **Sign.** While the signer is uncorrupted, apply the owner/session and readiness rules, initializing lazily if needed. Compute $w \leftarrow \text{Sign}(sk_p, m)$ and atomically produce $(\text{Signature}, p, m, w)$ with its **AuthOutput** at that computation step. Only the authorized owner route causes this protected production. Later delivery does not recompute it.
- 3 **Verify.** Use this endpoint's saved positive-binding cache or issue its actual setup retrieval. A delivered bottom returns **NoBinding** without a permanent negative decision.
 - (a) With the immutable binding v_p returned by setup, return $\text{Ver}(m, w, v_p)$. Invalid signature bytes within the specified encoding domain give zero.
 - (b) These local computations create no public raw-signature packet or additional signature query. All registration-board publications and retrievals and independently invoked authentication operations remain observable through their declared interfaces.

Fig. 8. The concrete authentication implementation $\Pi_{\mathcal{G}_{\text{auth}}}[\text{Sig}]$.

Appendix A.1 gives the saved-operation and signer-corruption rules, including the transfer of saved operations at corruption. Before corruption, the protected signer alone publishes its reserved entry, or the entry is its previously established matching binding in the modeled execution. After corruption the same owner retains its legitimate first-write publication role. It may choose an arbitrary still-unregistered value, even if its generated module key is different.

For the targeted signer, concrete **LeakedData** contains its retained owner state and owner-specific message/coin history, including (sk_p, v_p) only if that key was actually generated. It includes that owner's pending and produced operation records. Initial corruption of a fresh signer precedes key generation and exposes only its initial input/sender. This uses the source's entity-specific leakage hook without assuming secure erasure [13, Sec. 3.2 and App. A]. An existing key can be exposed before readiness or publication completes. This is the implementation's request-key exposure event. No key is generated by corruption, and no secret for an arbitrary board value is inferred. The separate transaction-spend key is outside this signer role.

One simulator and authentication realization The fixed simulator $\mathcal{S}_{\mathcal{G}_{\text{auth}}}$ retains each simulated concrete owner's key, coin history and operation records. At the matched uncorrupted initialization step, it runs **KeyGen** and supplies only v_p to the ideal's initialization control exchange before the next external yield. The

ideal performs the actual publication and saved retrieval under the original owner identity. At a matched protected signing production the simulator computes the same signature. At every registered verification control request it supplies the mathematical bit under that verifier's actual board binding. These bounded control exchanges are internal to the simulation. Actual service **AuthInput** and **AuthOutput** are forwarded once with their unchanged routes and counters.

Write ℓ_{ideal} for the targeted owner's ideal leakage from Appendix A.1, and ℓ_{real} for its concrete **LeakedData** payload specified above. On an allowed corruption request the simulator corrupts exactly that same signer entity. It receives the ideal framework confirmation (**CorruptionStatus**, true, ℓ_{ideal}) on the existing network route. It combines the disclosed owner-operation state with its saved key, coins and concrete history, omitting ideal-only control exchanges, and sends the environment (**CorruptionStatus**, true, ℓ_{real}) with the same entity header and timing. An undisclosed, undelivered board response is not reconstructed or guessed. The ideal never receives or stores sk_p . This simulation of the existing corruption payload is distinct from forwarding actual service observations. Initially corrupted owners have no simulated generated key. If publication is pending or a different board value has won, the simulator exposes only its actual module key, never a key for that board value.

After cutover, the simulator applies Appendix A.1's existing-connection forwarding and emulates adversarial local computation without ideal **Sign**. Late board responses arrive through that framework dispatch, not a private notification. Actual verifier operations remain unchanged.

Key-generation slots and verification events. In the probability bounds and reductions below, $j = 1, 2, \dots$ indexes successive owner-key generation events performed by the uncorrupted signer service. Slot j records generation order, with owners chosen adaptively. Reusing that owner's key in another session retains the same slot. A slot not reached in a run has no bad event in that run. The API parameter p continues to name the application party on its existing connection.

A registered verification invocation has obtained, on its own setup connection, a positive immutable binding to the generated key of the slot under consideration. It accepts when its service code produces and saves result 1 on (m, w) . Define Bad_j to be the event that such acceptance occurs before the slot's signer is corrupted and without an earlier protected **Sign** production on m by that owner. All permitted signing domains and sessions at that key count. After corruption, offline signatures and subsequent verification do not trigger Bad_j . The union $\bigcup_{j \geq 1} \text{Bad}_j$ ranges over all slot positions.

More explicitly, index the execution's transitions by $n \in \mathbb{N}$. Let $M_j^{<n}$ contain the messages with protected signing production at slot j before transition n , and let $n_{\text{cor},j} \in \mathbb{N} \cup \{\infty\}$ be that signer's first corruption transition. Write $\text{Accept}_j(n, m, w)$ for production of a registered verification result 1 at transition

n under that slot's immutable key binding. Then

$$\begin{aligned} \text{Bad}_j &= \{\exists n, m, w: n < n_{\text{cor},j} \wedge \text{Accept}_j(n, m, w) \wedge m \notin M_j^{<n}\}, \\ \text{Bad}_{\text{auth}} &= \bigcup_{j \geq 1} \text{Bad}_j. \end{aligned}$$

An unreached slot has no acceptance event. These transition indices describe event order, not ledger-clock time.

One complete-context runtime tail. For each fixed admitted environment $\mathcal{E}$, its complete context includes connected protocols, independent interface users and any previously composed simulators. Event probabilities refer to this complete concrete execution at security parameter λ . Choose a polynomial cutoff $B_{\text{ctx}}(\lambda)$ for its total relevant computation, including connected-protocol and registered-verification work. Define Tail_{ctx} as the event that this computation exceeds $B_{\text{ctx}}(\lambda)$. Environmental boundedness supplies a negligible function $\rho_{\text{ctx}}(\lambda)$ with $\Pr[\text{Tail}_{\text{ctx}}] \leq \rho_{\text{ctx}}(\lambda)$. If the context has a hard polynomial bound, choose that bound and take $\rho_{\text{ctx}} = 0$. The reductions alone stop at this joint cutoff, chosen before selecting a key-slot reduction.

Let $N_{\text{ctx}} = N_{\text{ctx}}(\lambda)$ be a polynomial upper bound on the number of such generation slots below this cutoff. The finite sum in (2) ranges over the fixed indices $1 \leq j \leq N_{\text{ctx}}$. On $\neg \text{Tail}_{\text{ctx}}$ no later slot occurs. Unreached slots contribute no bad event.

Reduction budgets and error bounds. For $1 \leq j \leq N_{\text{ctx}}$, let $t_j = t_j(\lambda)$ and $Q_j = Q_j(\lambda)$ bound the running time and signing-oracle queries of the reduction targeting slot j up to the joint cutoff or that signer's corruption. The time bound includes the surrounding computation and verification checks. The query bound counts every protected signing production at that owner across all permitted domains and sessions before corruption. Let $\epsilon_{\text{EUF},j}(\lambda, t, Q)$ bound the success probability of its single-key EUF-CMA game with at most t computation steps and Q signing queries. Define $\epsilon_{\text{auth}}(\lambda)$ as the sum of the one tail allowance and these slot bounds, clipped at one, as displayed in (2).

For the uniform envelope used at the end of the proof, let $t_{\text{ctx}}, Q_{\text{ctx}}$ be polynomial bounds dominating all t_j, Q_j for $j \leq N_{\text{ctx}}$, and let $\epsilon_{\text{EUF}}(\lambda, t, Q)$ be a common EUF-CMA bound for the fixed scheme covering every slot. Dependence of these budgets on λ is suppressed. These context-dependent bounds are analysis quantities. The simulator $\mathcal{S}_{\text{auth}}$ defined above is chosen independently of them and before $\mathcal{E}$. The comparison preserves actual service observations, corruption and existing-connection forwarding, and setup roles.

Lemma 5 (Shared authentication realization). *Fix the perfectly correct EUF-CMA signature scheme with deterministic verification, owner setup, shared registration and signer-corruption convention above. Let $\text{Env}_{\mathcal{G}_{\text{auth}}}$ be the class of admitted universally bounded, compatible, responsive complete contexts with environmentally bounded connected protocols. There exists one PPT simulator $\mathcal{S}_{\mathcal{G}_{\text{auth}}}$ such that, for every $\mathcal{E} \in \text{Env}_{\mathcal{G}_{\text{auth}}}$ and every λ , a coupling satisfies*

$$\Pr[\text{View}_{\mathcal{E}, \Pi_{\mathcal{G}_{\text{auth}}}}(1^\lambda) \neq \text{View}_{\mathcal{E}, \mathcal{S}_{\mathcal{G}_{\text{auth}}}, \mathcal{G}_{\text{auth}}}(1^\lambda) \wedge \neg \text{Bad}_{\text{auth}}] = 0.$$

For the complete-context cutoff and reduction budgets defined above,

$$\Pr\left[\bigcup_{j \geq 1} \text{Bad}_j\right] \leq \epsilon_{\text{auth}}(\lambda) = \min \left\{ 1, \rho_{\text{ctx}}(\lambda) + \sum_{j=1}^{N_{\text{ctx}}} \epsilon_{\text{EUF},j}(\lambda, t_j, Q_j) \right\}. \quad (2)$$

Consequently $\Pi_{\mathcal{G}_{\text{auth}}} \leq_{\text{iUC}}^{\epsilon_{\text{auth}}} \mathcal{G}_{\text{auth}}$. The order is $\exists \mathcal{S}_{\mathcal{G}_{\text{auth}}} \forall \mathcal{E} \in \text{Env}_{\mathcal{G}_{\text{auth}}}$, while $B_{\text{ctx}}, N_{\text{ctx}}, t_j, Q_j$ and ρ_{ctx} may depend on that complete context. The same registration board, roles, observations, corruption dispatch and occupied response connections remain in both systems.

Proof. Endpoint coupling. Fix the preceding simulator before the environment and couple its generation/signing coins with the concrete endpoint. Retain the same first-write board and configured signer/verifier entities. Pair framework corruption status, actual routes, service counters, inputs, stages and saved responses. Before signer corruption, admitted receipt produces the same **AuthInput** and yield. At the corresponding local initialization step the simulator generates one key and supplies its public value before the next external yield. Actual publication and captured retrieval then use the same owner identity. Matching values establish readiness, conflicting first values give the same error, and repetitions preserve the initialization. An old bottom stays with its original retrieval and erases no positive cache or decision. Issued board requests and their response connections remain paired. Initial corruption occurs before these owner steps on both sides, so neither side has a generated key or an invented readiness result.

Uncorrupted signing and decisions. Successful readiness binds the key generated by the uncorrupted signer to its immutable board entry. A protected signing computation and its service production form one step, and perfect correctness makes the certificate's verifier bit one. For a key in slot j , registered verification of an unsigned message before that signer's corruption returns zero outside **Bad_j**, matching the ideal's forced zero. For a signed message the simulator supplies the mathematical bit. Deterministic verification and immutable binding preserve every earlier decision. For a message and certificate with saved decision zero, perfect correctness prevents honest signing from later producing that certificate on that message. Thus the ideal signing-error test creates no unmatched protected result off the bad event. Malformed inputs, binding errors, cache hits and retries have the same service transitions.

Corruption confirmation and pending work. A permitted corruption targets the same signer entity in both executions. Its framework status changes before

any further service step. The ideal’s owner-scoped leakage supplies the actual pending/produced records, including any response already delivered to that owner. The simulator’s coupled coins and keys complete the concrete payload on the same corruption-confirmation route. If generation has not run, this payload contains no key. If generation precedes corruption but publication/readiness is unfinished, it contains that generated key and the same pending board operation. A later conflicting first-write value changes neither key. No secret for the conflicting value is required. The simulator does not expose global verifier tables or another owner’s state.

A received **Sign** with no produced result retains its old **AuthInput** and counter, but the dispatcher prevents a later protected production. If production already occurred, both executions retain the same certificate, decision and **AuthOutput** before signer corruption. The saved result and return connection remain without recomputation or a replacement error. Since the dispatcher skips the corrupted signer’s service code, later sends follow the same adversarial choice in both worlds. They do not overwrite the old produced record. Late board responses reach the original entity and are forwarded with their actual captured values. After corruption, ideal-only control exchanges cannot reactivate signer service code. These cases cover corruption before initialization, during registration, before signing production and between production and delivery.

Post-corruption computation and verification. Couple each permitted adversarial action through the corrupted entity’s existing I/O and the same identity checks. Forwarded inputs, sends and local computations create no service counter increment, **AuthOutput** or certificate-table update. The adversary may compute $\text{Sign}(sk, m)$ offline if it knows a suitable key, or publish an arbitrary still-available board value without its secret. A conflicting value can win after corruption while the generated key’s publication is unfinished. No protected signature was then produced under successful matching readiness, and an already established board binding cannot change. Verifier roles use their own actual board responses. For a new pair naming a corrupted signer, the ideal accepts the simulator’s mathematical bit without an ideal signing event. Saved certificate decisions still equal that deterministic bit outside the pre-corruption bad event. This covers arbitrary board values, absent or mismatched module keys and independent nonhonest owners.

One-key reduction. Fix an index $1 \leq j \leq N_{\text{ctx}}$ and target the j th uncorrupted owner-key generation. A forger receives the challenge key privately, installs it at that slot’s actual generation step, and generates other keys normally. It simulates the complete concrete context up to the cutoff. It stops with failure if that cutoff is exceeded or if its challenge signer is corrupted before a bad event. It therefore never has to reveal the unknown challenge secret or its generation coins. Before that cut it forwards each protected signing production at the owner assigned to slot j to its oracle across all domains and sessions. Receipt of a pending **Sign** causes no oracle query. At each registered verification under the challenge key, it outputs (m, w) if verification accepts and m is absent from its signing-oracle query set.

Call this forger $\mathcal{B}_j$. Write v_j for the verification key assigned to generation slot j , namely this forger's challenge key. As above, j indexes generation order, whereas p names an API party. At every transition before its stopping condition, its signing-oracle message set is exactly $M_j^{<n}$. A Bad_j event outside Tail_{ctx} therefore supplies a pair satisfying

$$\text{Ver}(m, w, v_j) = 1, \quad m \notin M_j^{<n}, \quad n < n_{\text{cor},j}.$$

In the coupled execution this is a winning EUF-CMA output for $\mathcal{B}_j$, before either the runtime cutoff or the challenge signer's corruption. Its budgets include the surrounding computation, verification checks and all Q_j pre-cutover signing productions. Consequently

$$\Pr[\text{Bad}_j \cap \neg \text{Tail}_{\text{ctx}}] \leq \Pr[\mathcal{B}_j \text{ wins its EUF-CMA game}] \leq \epsilon_{\text{EUF},j}(\lambda, t_j, Q_j).$$

Complete views and error. Outside the monitored bad event, protected service results and the corresponding $\text{AuthInput}/\text{AuthOutput}$ observations agree, as do the corruption status, simulated concrete leakage and framework forwarding. The records saved before corruption and the original response connections survive corruption. Public verifier and retained-board operations remain paired, including independent users and their pending results. Induction over these transitions and every permitted adversarial send preserves the complete view. Only ideal control exchanges are internal. The simulator fabricates no service observation or retained-interface response and gains no honest-owner authority. Thus the views coincide outside $\bigcup_{j \geq 1} \text{Bad}_j$. Decomposing by the one joint runtime event gives

$$\begin{aligned} \Pr\left[\bigcup_{j \geq 1} \text{Bad}_j\right] &\leq \Pr[\text{Tail}_{\text{ctx}}] + \sum_{j=1}^{N_{\text{ctx}}} \Pr[\text{Bad}_j \cap \neg \text{Tail}_{\text{ctx}}] \\ &\leq \rho_{\text{ctx}} + \sum_{j=1}^{N_{\text{ctx}}} \epsilon_{\text{EUF},j}(\lambda, t_j, Q_j). \end{aligned}$$

Clipping at one proves (2). An owner initialized only after the cutoff is covered by the same tail event. The union bound charges the joint tail and each slot's forgery bound once.

Using the previously defined common EUF-CMA bound and budget envelope gives

$$\epsilon_{\text{auth}} \leq \min\{1, \rho_{\text{ctx}} + N_{\text{ctx}} \epsilon_{\text{EUF}}(\lambda, t_{\text{ctx}}, Q_{\text{ctx}})\}.$$

The scheme's uniform EUF-CMA bound and the one contextual tail are negligible, so this envelope is negligible for each admitted complete context. The simulator itself does not depend on these contextual bounds or the environment. Its bounded work per operation and preserved continuations establish the stated iUC comparison with quantifier order $\exists \mathcal{S}_{\text{auth}} \forall \mathcal{E}$.

B Ledger and clock

B.1 The transaction ledger

We use the configured $\mathcal{G}_{\text{LEDGER}}$ of Badertscher et al. [3, Secs. 4.1–4.3] and its underlying global components. Its parameters are `windowSize`, `Delay`, `Validate`, `ExtendPolicy`, `Blockify` and `predict-time`. The cited functionality defines its ledger history, transaction buffer, party records, preprocessing and all public/adversarial operations. The ledger and $\mathcal{G}_{\text{clock}}$ retain these interfaces and properties.

Every ledger activation first reads its clock and then executes the source’s applicable branch. Registration and deregistration have their own party-management branches. Block extension, buffer revalidation and pointer reset occur only in the honest-party ledger-operation branch. The source ledger checks, `clock-read` and `clock-update` operations, and outputs remain unchanged. In particular, `(read, gid)` from a synchronized honest party P returns `(read, gid, stateP)` with that party’s current authentic permanent prefix. Other parties may receive different prefixes. The source’s pointer and `windowSize` rules govern these views. Reads and submissions alone do not promise a clock increment. Adversarial ledger-history reads, block proposals, view controls and corruption retain the exact interfaces summarized in Appendix B.2.

Late honest registrants may be in the ledger’s desynchronized set. The source’s strict synchronization condition is $\tau_P^{\text{reg}} < \tau^L - \text{Delay}$, where τ_P^{reg} is registration time and τ^L the current ledger-clock sample.

Complete transactions and ledger occurrences. The selected transaction format supplies the complete encoded bytes tx . The `AsyncLS` accessor $tx.\text{id}$ is derived from those bytes by the configured hash procedure in Section 2.11. The transaction fields and validity rules belong to the ledger configuration. Ledger-assigned occurrence metadata and the submitting party are separate from those bytes.

For `(submit, gid, tx)` from an actual registered party P , the ledger allocates occurrence metadata rid_L and forms $\text{BTX} = (tx, \text{rid}_L, \tau^L, P)$. It buffers the entry exactly when $\text{Validate}(\text{BTX}, \text{state}, \text{buffer}) = 1$ and outputs `(submit, BTX)` to its adversary regardless of admission. This metadata identifies a ledger occurrence. $tx.\text{id}$ is the supplied transaction’s `AsyncLS` hash, with the probabilistic binding guarantee above. The caller receives no admission acknowledgment.

The chosen `Validate` uses the ledger history `state` and transaction buffer supplied to it. Together with the extension and block-selection rules, it must enforce the configured transaction validity, including authorization, funding, nonce and fee requirements where applicable. Candidate blocks must be checked against the ledger history they extend, and only valid blocks may enter [3, Secs. 4.3.1 and 6]. This requirement also covers adversarial submissions and proposals. A constructor’s stricter fee-selection or relay preference is not automatically a ledger-validity rule.

The selected `Blockify` formats each included buffered entry so that the complete transaction, ledger-assigned occurrence metadata, and actual submitting-party

field are recoverable in a complete ordered block. Empty-block boundaries and the order within each block are kept. These are declared parameters of the selected instance, permitted by the parameterized output-format treatment [3, Sec. 6]. The ledger-assigned field identifies the submitting party.

B.2 Clock and ledger adversary interfaces

We use BMTZ’s clock [3, Fig. 2 and Sec. 3.3], with its actual session/identity interpretation. In the source’s notation, $(\text{register}, \text{sid}_C)$ adds the permitted actual party P , sets its update bit $d_P = 0$, initializes a new managed session’s time $\tau_{\text{sid}} = 0$, and returns $(\text{register}, \text{sid}, P)$ to the caller. $(\text{de-register}, \text{sid}_C)$ removes a registered party and returns $(\text{de-register}, \text{sid}, P)$. The adversary can invoke these on behalf of corrupted parties. The source’s reply uses sid , whereas its operation input uses sid_C . No private correlation field is added.

A registered party’s $(\text{clock-update}, \text{sid}_C)$ is ignored unless its providing dummy party encodes the correct party identifier. Otherwise the clock sets $d_P = 1$, performs **Round-Update**, and forwards $(\text{clock-update}, \text{sid}_C, P)$ to the adversary. $(\text{clock-read}, \text{sid}_C)$ from a machine instance with the source’s identity shape for a managed session performs **Round-Update** and returns $(\text{clock-read}, \text{sid}_C, \tau_{\text{sid}})$ to that requester. A fresh ITI with the permitted identity shape in that managed session may read subject to ξ_{sync} , including before registration. Unmanaged sessions are ignored. **Round-Update** increments each managed session’s time if every uncorrupted party in that session has update bit 1, then clears that session’s bits. Thus a read executes the actual clock code, including any enabled round update.

The clock’s adversary input $(\text{get-registered}, \text{sid}_C)$ returns $(\text{get-registered}, \text{sid}, \mathcal{P}_C)$, with the registered set $\mathcal{P}_C$. Its $(\text{corrupt}, \text{sid}_C, P_i)$ for a registered party marks it corrupted, executes **Round-Update**, and returns the same corruption tuple. The identity-bound context restriction ξ_{sync} prevents the environment from claiming an existing ITI’s clock identity or subsequently creating an ITI under an identity it already claimed externally. It also excludes direct access to the clock’s session-corruption information. Figure 2 supplies no separate get-corruption-set operation. All other permitted clock interactions remain observable and are matched.

The ledger’s public party operations are **register**, **de-register**, **read**, **submit**, and **(maintain-ledger, gid, minerID)**. Maintenance issues **clock-update** precisely when the activating party is honest and $\text{predict-time}(I_T^H) > \tau^L$. Otherwise it forwards that input to the adversary.

Table 2: The BMTZ ledger’s unchanged adversarial interface. Each row retains the source’s clock read and applicable branch and identity checks.

Ledger input	Ledger update and output
(read, gid)	Return the permanent ledger history, transaction buffer, and timed honest-input log ($\text{state}, \text{buffer}, I_T^H$).
next-block with an ordered list of ledger-assigned occurrence identifiers rid_L	Keep the list entries that identify buffered transactions, append that ordered candidate to NxtBC, and output (next-block, ok). The selected ExtendPolicy decides extension on an honest ledger activation.
set-slack with party/pointer pairs	Apply the source’s honest-party/view/pointer checks [3, Sec. 4.3], update accepted pointers and return (set-slack, ok).
desync-state with party/ledger-view pairs	For the source-admitted desynchronized parties, replace their readable ledger views and return (desync-state, ok). Such views supply no permanence guarantee.
(get-registered, gid)	Return (get-registered, gid, $\mathcal{P}_L$) for the actual set of registered ledger parties.
(corrupt, gid, P_i)	Apply the source’s honest-log/set changes. If $\text{predict-time}(I_T^H) > \tau^L$, issue the actual clock-update. Otherwise return the input to the adversary. The theorem excludes only corruption forbidden by its explicit honesty assumptions.

Messages on the ledger’s adversarial interface are delivered directly to the ledger’s existing adversarial role. Registered parties retain their separate submission identities.