Ledgera Yellow Paper version V_0_2

Erwan Mahe, Antonella Del Pozzo, Alvaro Garcia-Perez, Sara Tucci-Piergiovanni
Université Paris Saclay, CEA LIST, Palaiseau, France

2026 06 01

Contents

1 Introduction **2**

2 Architectural overview **2**
2.1 Ledgents & roles . . . 3
2.2 Layers of abstraction . . . 4
2.3 Application-Client API and deployment . . . 4

3 Preliminaries and system model **4**
3.1 Fault tolerance . . . 5
3.2 Public key infrastructure and network assumptions . . . 5

4 Distributed verifiable function instance Execution **6**
4.1 Ledgera application template . . . 6
4.2 Function specifications and function instances . . . 7
4.3 Lifecycle of function instances . . . 8

5 Internal message interface **8**
5.1 Votes and Proofs . . . 8
5.2 Transactions . . . 9
5.3 Requests . . . 9
5.4 Queries . . . 10
5.5 Notifications . . . 11

6 Behavior at the *storage-layer* **11**
6.1 Store data . . . 11
6.2 Query data . . . 12
6.3 Detailed behaviors of Ledgents per involved roles . . . 12

7 Behavior at the *logging-layer* **15**
7.1 Valid history of transactions . . . 15
7.2 Overview of interactions with the secure log . . . 16
7.3 The secure log and its properties . . . 16

8 Behavior at the *coreset-layer* **17**
8.1 Overview . . . 17
8.2 Assignments of unknown inputs . . . 18
8.3 Detailed behaviors of Ledgents per involved roles . . . 20

9 Behavior at the *function-layer* **22**
9.1 Overview . . . 22
9.2 Detailed behaviors of Ledgents per involved roles . . . 23

10 Properties upheld by Ledgera **26**
10.1 Storage availability properties . . . 26
10.2 Function instances safety properties . . . 27
10.3 Function instances liveness properties . . . 28

11 Complexity **30**
11.1 Single-party function instances . . . 30
11.2 Multi-party function instances . . . 30

12 Deployment & Hello World example **30**
12.1 Requirements of the application code . . . 31
12.2 The application code at work . . . 32

13 Discussion **34**
13.1 Synchrony model and BFT consensus . . . 34
13.2 BFT storage . . . 34
13.3 Non-determinism . . . 35
13.4 Membership and reconfiguration . . . 35
13.5 Single fault threshold . . . 35
13.6 Applications . . . 35

1 Introduction

The purpose of this document is the following:

to give a comprehensive specification of Ledgera version `V_0_2`

It should be, as much as possible, a self-contained document.

The document is organized as follows:

- Sec.2 gives an architectural overview of Ledgera.
- Sec.3 describes the system model of Ledgera.
- Sec.4, defines verifiable function and their execution.
- Sec.5 defines internal messages of Ledgera and the concrete types they contain.
- Sec.6, describes the *storage-layer*.
- Sec.7, describes the *logging-layer*.
- Sec.8, describe the *coreset-layer*.
- Sec.9, describes the *function-layer*.
- Sec.10, states and proves important properties upheld by Ledgera.
- Sec.11 discusses communication complexity.
- Sec.12 illustrates how Ledgera can be used on a simple hello world example.
- Sec.13 provides a broader discussion covering related work, current limitations, and directions for future versions.

2 Architectural overview

The aim of Ledgera is to provide a distributed ledger designed for organizations and devices that need to collaborate without relying on a central trusted authority. To this end, Ledgera offers primitive operations for the Byzantine Fault Tolerant distributed execution of verifiable functions (functions whose definition can depend on a single-party or can be multi-party), for the notarization of (the execution of) these functions (via non-revocable anchoring of quorums of signatures, confirming the declaration of the function and, if applicable, proving the integrity of its computed output), as well as for the optional storage of its input and/or output data (via non-revocable anchoring of quorums of signatures guaranteeing the availability of said data). Fig.1 below shows the lifecycle of an instance of a verifiable function in Ledgera.

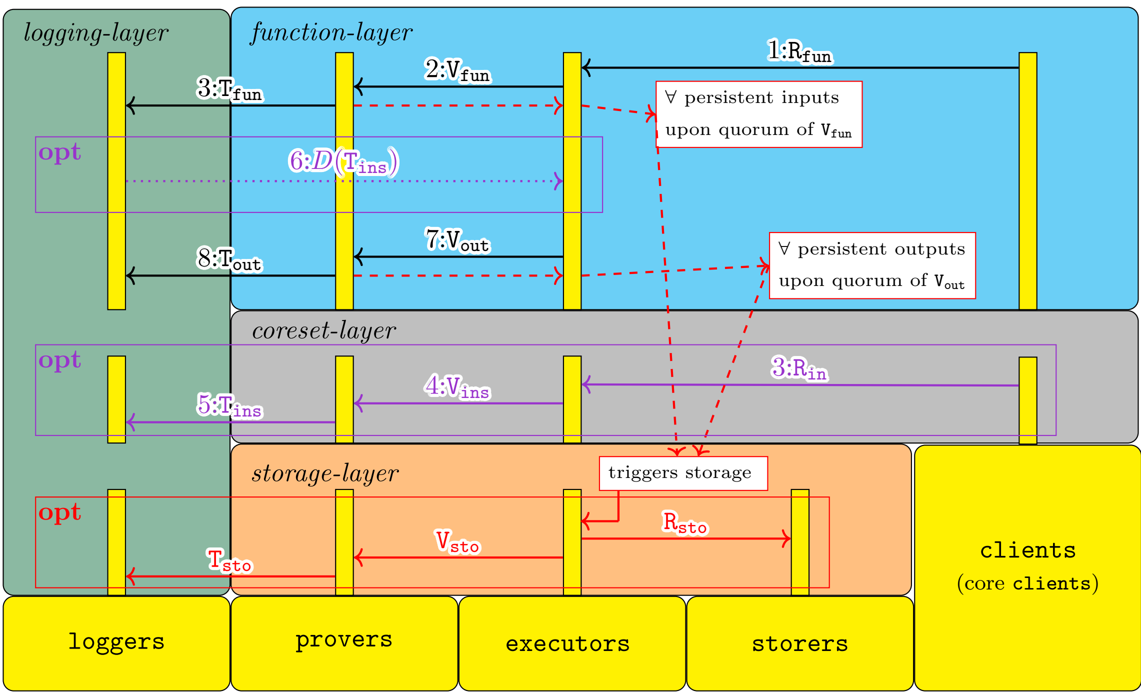


Figure 1: Ledgera roles and how they interact on the different layers of abstraction (feeback messages to the `clients` and queries made to `storers` not represented) during the lifecycle of a function instance

2.1 Ledgents & roles

Nodes in Ledgera are called "Ledgents" (for Ledgera Agents). Ledgents are authenticated unambiguously via a Public/Private key pair.

Roles specify behaviors that can be executed as processes running on Ledgents. A Ledgera system is composed of a set of Ledgents N. Any Ledgent $n \in N$ may or may not play any of the following 5 roles: `client`, `executor`, `prover`, `storer` and/or `logger`. Each role process that runs on a Ledgent inherits the Ledgent's unique identity (Public/Private key pair).

To deploy Ledgera over a set of Ledgents, one needs to assign roles to these Ledgents (a single Ledgent may play several distinct roles and the processes that correspond to the execution of these roles are, as a result, co-located). As illustrated on Fig.1 (c.f. yellow rectangles at the bottom), we distinguish between 5 types of "roles":

- `clients` request the execution of function instances (via $\mathtt{R_{fun}}$ messages) and can propose values (arguments) for the unknowns of multi-party function instances (via $\mathtt{R_{in}}$ messages). It is via the `clients` that users/applications may interact with Ledgera.

- `executors` vote to decide on **(1)** granting execute rights to function instances (via emitting $\mathtt{V_{fun}}$ votes), **(2)** validating a "core-set" of arguments for multi-party function instances (via emitting $\mathtt{V_{ins}}$ votes) and **(3)** after having replicated locally the execution of a function instance, on the result of that function instance (via emitting $\mathtt{V_{out}}$ votes). They also forward storage requests (via $\mathtt{R_{sto}}$ messages) for the inputs and/or outputs of function instances that are flagged as "persistent" and notify they have done so (via emitting $\mathtt{V_{sto}}$ votes). `executors` can also notify `clients` of different events (not represented on Fig.1).

- `provers` collect votes from `executors`, produce quorums of signatures that constitute proofs that various events occurred, package these proofs and send them to the secure log for anchoring:
  - A quorum of $\mathtt{V_{fun}}$ votes constitutes a proof that a function instance has been declared and granted execute right. It is stored as a $\mathtt{T_{fun}}$ transaction on the secure log.
  - A quorum of $\mathtt{V_{ins}}$ votes constitutes a proof that an assignment of values for the unknowns of a multi-party function instance has been validated. It is stored as a $\mathtt{T_{ins}}$ transaction on the secure log.
  - A quorum of $\mathtt{V_{out}}$ votes constitutes a proof of integrity of the output of a function instance. It is stored as a $\mathtt{T_{out}}$ transaction on the secure log.
  - A quorum of $\mathtt{V_{sto}}$ votes constitutes a proof of (shipment to) storage of an input and/or output of a function instance. It is stored as a $\mathtt{T_{sto}}$ transaction on the secure log.

- `storers` implement a distributed (replicated, redundant, Byzantine Fault-Tolerant) storage. They receive storage requests (as $\mathtt{R_{sto}}$ messages) from `executors` and update the distributed storage accordingly. They also answer to queries from `executors` or `clients` that want to retrieve data (not represented on Fig.1).

- finally, `loggers` participate in implementing a "secure log" that totally orders transactions they receive from `executors` into a coherent "valid history" (c.f. Definition 15).
  We consider every Ledgent to be a light-client of the secure log i.e., whenever and iff the secure log delivers a transaction, then, eventually, every Ledgent will have delivered said transaction.
  (Fig.1, represents this abstractly as $D(\mathtt{T_{ins}})$.)

We denote respectively by N_c, N_x, N_p, N_s and N_l the sets of all the `client`, `executor`, `prover`, `storer` and `logger` components.

2.2 Layers of abstraction

As hinted by Fig.1, we may describe the behavior of Ledgera as taking place on 4 different levels of abstractions:

- a low-level *storage-layer*, which implements a (replicated) database in the form of a key-value store that upholds specific properties (such as the availability of data for which a Proof of Storage exists).
- a low-level *logging-layer*, which orders and anchors Proofs that various events took place in the form of transactions. This layer thus provides auditability and traceability of these events. These transactions are totally order, which enables decisions that require events to be ordered.
- a *coreset-layer*, which allows agreement on the inputs of multi-party function instances.
- a *function-layer*, which implements the verifiable execution of function instances.

2.3 Application-Client API and deployment

In order to implement a distributed application on top of Ledgera, one needs to describe how operations of the application get triggered and executed as lower-level Ledgera function instances. To do so, one needs to:

- Implement a Ledgera application template (c.f. Section 4.1) which consists in:
  - providing data types for the values that can be stored, provided as input or produced as output of function instance,
  - providing function types to be used in function instances, and
  - providing predicate types to constraint accepted values for multi-party function instances).
- Write application code that translates higher-level operations into one or several lower-level Ledgera function instances.

Once the Ledgera application template and the application code are written, the application can be deployed as illustrated in the example in Fig.2 below.

Users interact with the application code (outside the scope of Ledgera). This application code must run in a Ledgent that plays the `client` role. The application code communicates with the co-located `client` role via a dedicated "client API". The client API can be used to submit requests to the system (e.g., executing new function instances) or retrieve information from the system (e.g., retrieving values from the distributed storage).

Role processes are hosted on the different Ledgents that play them. For instance, a single node may play, at the same time, the `client`, `executor` and `prover` roles, as does node n_1 in the example of Fig.2.

At operation time, the role processes running on each Ledgent in N communicate with each other via an "internal message interface" (detailed in Section 5).

3 Preliminaries and system model

In what follows, we use the notation $\varnothing$ to denote the "None" case of option types. This $\varnothing$ notation is not to be confused with the $\emptyset$ notation, which denotes an empty set.

We assume the existence of a digest type $\mathfrak{H}$ and of a hash function $\blacktriangledown$ that maps any serializable value into $\mathfrak{H}$ with negligibly probable collisions.

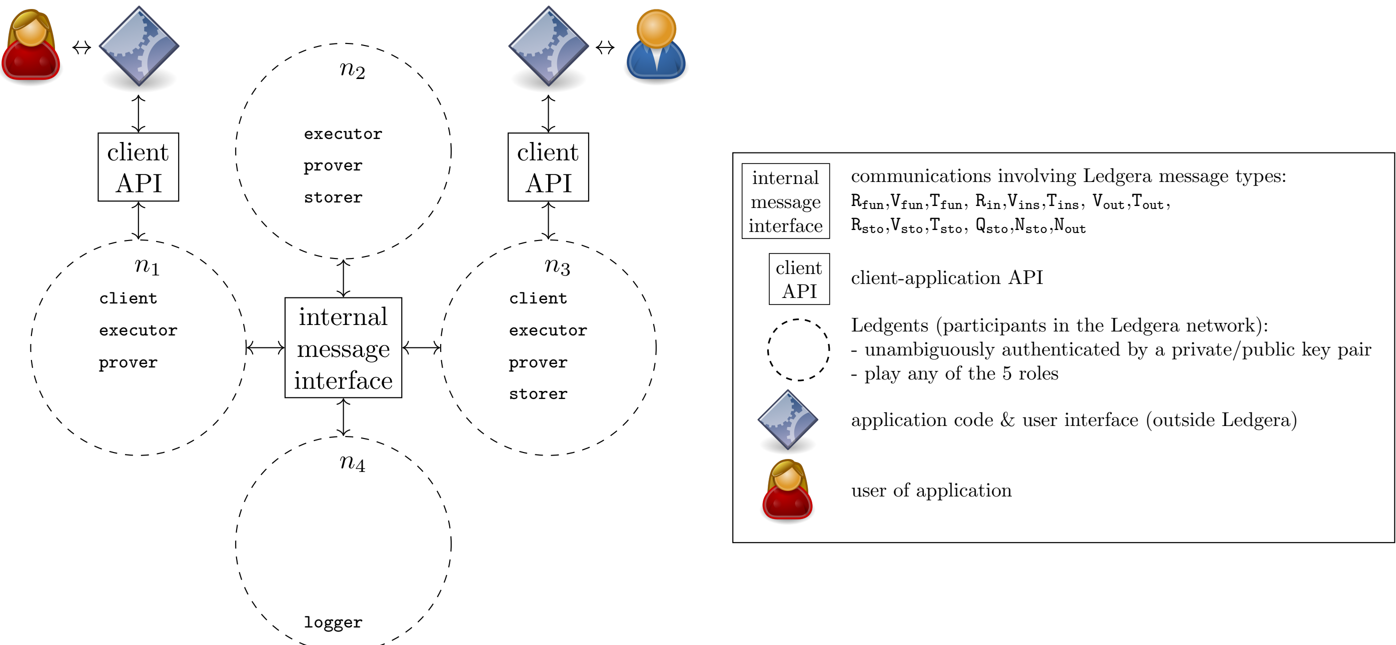


Figure 2: Example of deployment of a Ledgera network

3.1 Fault tolerance

A Ledgera network is Byzantine Fault Tolerant. Given $|N|$ the number of Ledgents, Ledgera tolerates up to f Byzantine Ledgents, where $|N| = 3 * f + 1$. The integer f is the maximum number of Ledgents in N that may deviate arbitrarily from the correct implementation of Ledgera. We may refer to f as the "Byzantine threshold". We may also denote by $F \subset N$ with $|F| \leq f$ the subset of Ledgents that are actually faulty.

In the current version (V_0_2 of the specification) we only consider a single Byzantine threshold f that concern all Ledgents, without any regard to the roles they play.

This has implications on the minimum number of Ledgents that must play specific roles. With the hypothesis that any message of the "internal messages API" that is emitted by a correct Ledgent is eventually received by all the correct Ledgents it targets, we require that there are at least:

- $2f+1$ `executors`. Since in version V_0_2 of Ledgera we consider only deterministic functions, to ascertain that a value v is the correct result, we only need $f+1$ distinct `executors` to produce it. Indeed, if one obtains the same v value from $f+1$ distinct `executors`, we can be sure that at least one of these `executors` is honest and thus v is valid. In the worst case where there are f Byzantine `executors` that abstain from producing a value, we still need to have $f+1$ replicas of v. Thus we need $2f+1$ `executors`.
- $f+1$ `provers` to guarantee that proofs are produced (in the worst case, f `provers` abstain from producing proofs). However, in the paper, we will always consider a specific deployment in which there are $2f+1$ `provers` that are always co-located with the `executors`. This is done so as to simplify the exchange of information so that any honest `executors` immediately learns of the proofs produced by the honest `prover` it is co-located with.
- $f+1$ `storers` for the distributed storage to be functional. This ensures that at any given time, an honest `storer` has a full copy of the data and can make it available by answering queries.
- $3f+1$ `loggers` for the secure log to be functional, as it requires Byzantine-Fault-Tolerant consensus [10] (c.f. Remark 1).

3.2 Public key infrastructure and network assumptions

We assume a permissioned network of N Ledgents with $|N| = 3f+1$, each Ledgent being identified by a unique asymmetric Public/Private key pair.

The set of all cryptographic signatures of arbitrary payloads (serialized messages) produced by any Ledgent in the set N of Ledgents is denoted by $\mathfrak{S}$. For any Ledgent in N and any payload, there exists only one such signature.

In pseudo-code we may write "**broadcast** m **to** X" where m is a message and $X \subseteq N$ a set of Ledgents. We assume m is serialized, that the Ledgent emitting m signs this serialized payload, producing a unique signature $\sigma \in \mathfrak{S}$ and that the

signed serialized message is propagated to all Ledgents in X. We assume that any message broadcast by an honest Ledgent is eventually received by all honest Ledgents in X, as stated in Assumption 1.

Assumption 1. *For any correct Ledgent $n \in N$, if n broadcasts a message to a subset $X \subseteq N$ of Ledgents, then, eventually, every correct Ledgent in X will receive that message.*

With Asmpt.1, a message emitted by a correct Ledgent will eventually be received by all the correct Ledgents it targets. This says nothing about messages emitted by Byzantine Ledgents, which are free to equivocate.

Remark 1. *Byzantine-Fault Tolerant consensus cannot terminate deterministically on a fully asynchronous setting as per the FLP impossibility result [10]. As we need BFT consensus for the secure log (c.f. Sec.7), Ledgera must either rely on randomization under an asynchronous network model, or consider a* partially synchronous *network model [10]. Either way, Assumption 1 guarantees eventual message delivery, which is the key requirement for liveness of BFT consensus.*

In pseudo-code we may write "**upon receiving** m **with signature** $\sigma \in \mathfrak{S}$", signifying that the Ledgent has received a signed serialized message, verified its signature is correct, and deserialized the payload into m.

Definition 1. *For any serializable message m, we define:*

$$\mathcal{Q}(m) = \left\{ q \subset \mathfrak{S} \,\middle|\, \begin{array}{l} |q| = f+1 \\ \forall \sigma \in q,\ \sigma \text{ is the signature of } m \text{ by a Ledgent in } N \end{array} \right\}$$

In Def.1, we introduce the notion of quorum. For any serializable message m, the existence of a quorum $q \in \mathcal{Q}(m)$ guarantees that at least one honest Ledgent has signed m.

4 Distributed verifiable function instance Execution

4.1 Ledgera application template

Higher-level operations from the application are converted and executed as function instances on Ledgera. The execution of these function instances produce proofs of integrity and availability. To encode these application-specific functions, we use a "Ledgera Application Template" LAT, as defined in Definition 2.

Definition 2. *A LAT is a tuple* $(\text{values}, \text{functions}, \text{predicates})$ *where:*

- values *is the union of the types of data that may be stored in the distributed storage and/or used as input and/or output of functions,*
- functions *is a finite set of functions ϕ of finite arity $|\phi| \in \mathbb{N}$ and profile $\phi : \text{values}^{|\phi|} \mapsto \text{values}$, and*
- predicates *is a finite set of predicates ψ of finite arity $|\psi| \in \mathbb{N}$ and defining a relation $\psi \subset \text{values}^{|\psi|}$.*

For any LAT, we may denote by LAT_{val}, LAT_{opn} and LAT_{prd} its respective constituting elements.

Given a LAT, LAT_{val} represents all possible data that may transit through the network and be stored, and may be used as input of functions or produced as the output of the execution of functions.

LAT_{opn} represents all possible atomic functions that can be executed on Ledgera and produce a Proof of Integrity. These functions must be deterministic. For any function $\phi \in LAT_{\text{opn}}$ and any array of inputs $x_1, \cdots, x_{|\phi|} \in LAT_{\text{val}}^{|\phi|}$, we denote by $\phi(x_1, \cdots, x_{|\phi|}) \in LAT_{\text{val}}$ the unique possible result of applying the function on the array of inputs.

LAT_{prd} can be used for multi-party function instances, to constraint the arrays of inputs that are authorized. In this context, given $\psi \in LAT_{\text{prd}}$ and an array of inputs $x_1, \cdots, x_{|\psi|} \in LAT_{\text{val}}^{|\psi|}$, if $\psi(x_1, \cdots, x_{|\phi|})$ holds, then the *coreset-layer* may validate this array of inputs, which will be available to be chosen as the array of inputs of the multi-party function instance.

Serialization. We assume any object in $LAT_{\text{val}} \cup LAT_{\text{opn}} \cup LAT_{\text{prd}}$ is serializable.

Example. Consider the following LAT example:

$$LAT = \left(\mathbb{N},\ \{+\},\ \left\{ \psi_1 \wedge \psi_2 \wedge \psi_3 \,\middle|\, \begin{array}{l} \psi_1 \in \{\top\} \cup \bigcup_{v_1 \in \mathbb{N}} \{x_1 = v_1\}, \\ \psi_2 \in \{\top\} \cup \bigcup_{v_2 \in \mathbb{N}} \{x_2 = v_2\}, \\ \psi_3 \in \{\top,\ x_1 \neq x_2\} \end{array} \right\} \right)$$

In this example, the set of values LAT_{val} is the set of positive integers $\mathbb{N}$. There is a single function, which is the addition on naturals $+ : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$. As for the predicates, they allow constraining the inputs x_1 and x_2 of the addition as follows: fixing the value of x_1, fixing the value of x_2 and/or forcing x_1 and x_2 to be distinct. This LAT enables both single-party and multi-party additions over integers, with the possibility of constraining its inputs as prescribed by LAT_{prd}.

4.2 Function specifications and function instances

The notion of "function" is central to Ledgera. We distinguish between:

- "function specifications", which define a function type and stipulate on which arrays of input a function can be executed, and
- "function instances", which are uniquely identified transient processes run by the Ledgera network and correspond to a given execution of a function specification.

We formalize "function specifications" in Def.3. Our formalization uses the notion of "Proof of Shipment to Storage" ($\texttt{LP}_\texttt{S}$) from Def.6 in Sec.5.1. The $lps \in \texttt{LP}_\texttt{S}$ are used as pointers, referencing a value that is stored on the distributed storage.

Definition 3. *A function specification is a tuple (ϕ, ψ, K) where $\phi \in LAT_{\text{opn}}$ is the function type to execute, $\psi \in LAT_{\text{prd}}$ is a predicate constraining the values that can be used as inputs, and $K : \mathbb{N} \rightharpoonup LAT_{\text{val}} \cup \texttt{LP}_\texttt{S}$ gives the values of the known inputs. We denote by* Spec *the set of all possible function specifications.*

Given a function specification $(\phi, \psi, K) \in \texttt{Spec}$, an input at index $i \in [1, |\phi|]$ is said to be:

- "Known in advance" if $i \in Dom(K)$. In that case, the choice for the concrete value of the argument at index i is fixed in advance by the proposer of the function specification, which is specified by K as follows:
  - If $K(i) \in LAT_{\text{val}}$, the concrete value is directly shipped with the specification.
  - If $K(i) \in \texttt{LP}_\texttt{S}$, then, whoever wants to execute the function specification must, at first, retrieve the concrete value from the distributed storage to which $K(i)$ points.
- "Unknown" if $i \notin Dom(K)$. In that case, the function is multi-party and one must wait for agreement on the unknown inputs before being able to execute it.

A function specification $(\phi, \psi, K) \in \texttt{Spec}$ is valid iff (**1**) the arities of the function and predicate types match i.e., $|\phi| = |\psi|$, (**2**) the known inputs in K correspond to inputs at indices in $[1, |\phi|]$ i.e., $Dom(K) \subset [1, |\phi|]$, (**3**) every $lps \in \texttt{LP}_\texttt{S}$ proposed as known input is valid (c.f. Def.6) and (**4**) if there exists at least one array of input that satisfies ψ i.e.,

$$\exists x_1, \cdots, x_{|\psi|} \in LAT_{\text{val}}^{|\psi|} \text{ s.t. } \forall i \in Dom(K),\ x_i = \texttt{unwrap_val}(K(i))$$

where unwrap_val is defined in Alg.4 of Sec.6.3 and guaranteed to return the value because the $lps \in \texttt{LP}_\texttt{S}$ is valid, and $\psi(x_1, \cdots, x_{|\psi|})$ holds.

Definition 4. *A function specification $(\phi, \psi, K) \in \texttt{Spec}$ is valid iff:*

$$
\begin{array}{ll}
 & (|\phi| = |\psi|) \qquad \textit{/*arities match*/} \\
\wedge & (Dom(K) \subset [1, |\phi|]) \qquad \textit{/*known inputs indices respect arity*/} \\
\wedge & (\ \forall x \in Img(K) \cap \texttt{LP}_\texttt{S},\ x \textit{ is a valid } \texttt{LP}_\texttt{S}\ ) \qquad \textit{/*inputs from storage have valid proofs (as per Def.6)*/} \\
\wedge & \left(\exists x_1, \cdots, x_{|\psi|} \in LAT_{\text{val}}^{|\psi|} \middle| \begin{array}{ll} & \forall i \in Dom(K),\ x_i = \texttt{unwrap_val}(K(i)) \\ \wedge & \psi(x_1, \cdots, x_{|\psi|}) \end{array} \right) \qquad \textit{/*predicate } \psi \textit{ is satisfiable*/}
\end{array}
$$

A "function instance" corresponds to the execution of a "function specification" $(\phi, \psi, K) \in \texttt{Spec}$ with some additional metadata:

- a "persistent output flag", which is a boolean that, if true, will make so that the output result of the function is stored on the distributed storage
- for each "known in advance" argument of index $i \in Dom(K)$ such that $K(i) \in LAT_{\text{val}}$ is a raw value, a "persistent input flag", which is a boolean that, if true, will make so that that input is stored on the distributed storage

4.3 Lifecycle of function instances

The lifecycle of a "function instance" is described on Fig.1 from the overview in Sec.2.

The lifecycle starts with a client broadcasting a $\mathtt{R_{fun}}$ message to the set N_x of all executors. This $\mathtt{R_{fun}}$ message corresponds to a request for the execution of a new function instance. The message contains (among other things) the specification $(\phi, \psi, K) \in \mathtt{Spec}$ of the function.

Upon receiving this specific $\mathtt{R_{fun}}$ message, an executor verifies that the specification it carries is valid (as per Def.4). If this is the case, the executor produces a specific $\mathtt{V_{fun}}$ vote (containing information identifying unambiguously the function instance), signs it, and broadcasts it to the set N_p of all provers.

The provers collect the signatures of the $\mathtt{V_{fun}}$ votes that are broadcast by the executors. Upon collecting a quorum of signatures for such a $\mathtt{V_{fun}}$ vote, a prover constitutes a "Proof of Function Declaration" ($\mathtt{LP_{FD}}$ from Def.6 in Sec.5.1) and submits it to the loggers for anchoring in the secure log, as a $\mathtt{T_{fun}}$ transaction.

The coloured frames below (respectively red and purple) describe the optional segments of the same color in the lifecycle of a function instance in Fig.1.

For each input at index $i \in [1, |\phi|] \cap Dom(K)$ that is known in advance, if the input is shipped as a raw value $K(i) \in LAT_{\text{val}}$ in the initial $\mathtt{R_{fun}}$ message and flagged as a persistent input, the production of an *lpfd* $\in \mathtt{LP_{FD}}$ triggers its storage (red dotted arrows in the figure).

The storage of inputs is carried out by the executors. Each executor broadcasts a $\mathtt{R_{sto}}$ message to the set N_s of all storers, which requests the storage of that input value $v \in LAT_{\text{val}}$, justified by the *lpfd* $\in \mathtt{LP_{FD}}$.

Upon receiving such an $\mathtt{R_{sto}}$ message, a storer stores the input value $v \in LAT_{\text{val}}$ as the pair $(\blacktriangledown(v), v)$ on its local copy of the Key-Value distributed store.

After having sent an $\mathtt{R_{sto}}$ message, an executor also signs and broadcasts the corresponding $\mathtt{V_{sto}}$ vote.

The provers collect the signatures of the $\mathtt{V_{sto}}$ votes that are broadcast by the executors. Upon collecting a quorum of signatures for such a $\mathtt{V_{sto}}$ vote, a prover constitutes a "Proof of Storage" ($\mathtt{LP_S}$ from Def.6 in Sec.5.1) and submits it to the loggers for anchoring in the secure log, as a $\mathtt{T_{sto}}$ transaction.

If the specification of the function instance has unknown inputs (i.e., $[1, |\phi|] \setminus Dom(K) \neq \emptyset$), the system needs to agree on their value them before being able to perform the actual compute.

To do so, clients (including others than the initial requesting client) may propose values for the unknowns via broadcasting $\mathtt{R_{in}}$ messages to the executors. The proposal of inputs by clients different from the requesting one makes the function instance "multi-party".

The executors passively collect input proposals until they can establish a complete assignment of the unknowns inputs of the function that satisfies the constraint ψ. Upon establishing such an assignment, each executor broadcast a unique $\mathtt{V_{ins}}$ vote for this assignment. The executors may also echo the votes of other executors after having verified they constitute valid assignments (i.e., that the assignments found by the other executor also satisfies ψ).

The provers collect the signatures of the $\mathtt{V_{ins}}$ votes broadcast by the executors. Upon collecting a quorum of signatures for such a $\mathtt{V_{ins}}$ vote, a prover constitutes a "Proof of Verification of an Input Assignment " ($\mathtt{LP_{VIA}}$ from Def.6 in Sec.5.1) and submits it to the loggers for anchoring in the secure log, as a $\mathtt{T_{ins}}$ transaction.

The delivery of such a $\mathtt{T_{ins}}$ transaction by the secure log (which must be unique per function instance) ensures agreement on the array of inputs in the multi-party case.

Upon agreement on a unique assignment of values for all the inputs of the function (either directly in the case of single-party functions or after delivering the $\mathtt{T_{ins}}$ transaction in the case of multi-party functions), each executor executes the function locally and signs and broadcasts a $\mathtt{V_{out}}$ vote containing the hash of the output result.

The provers collect the signatures of the $\mathtt{V_{out}}$ votes that are broadcast by the executors. Upon collecting a quorum of signatures for such a $\mathtt{V_{out}}$ vote, a prover constitutes a "Proof of Computation Integrity" ($\mathtt{LP_C}$) from Def.6 in Sec.5.1 and submits it to the loggers for anchoring in the secure log, as a $\mathtt{T_{out}}$ transaction.

If in the initial $\mathtt{R_{fun}}$ declaration the output is flagged as persistent, then the production of an *lpc* $\in \mathtt{LP_C}$ triggers the storage of that output value. This follows the same principle as described above for the case of persistent inputs, except that now both an *lpfd* $\in \mathtt{LP_{FD}}$ and an *lpc* $\in \mathtt{LP_C}$ are required to justify the storage of the output. The executors must include these two proofs in the $\mathtt{R_{sto}}$ they broadcast.

5 Internal message interface

5.1 Votes and Proofs

Def.5 below collects the concrete types of all vote messages that executors broadcast to provers.

Definition 5. *We define the Ledgera vote types as follows:*

$\mathtt{V_{fun}} \begin{pmatrix} fid \in \mathfrak{S}, \\ K' : \mathbb{N} \rightharpoonup \mathfrak{H}, \\ U \subset \mathbb{N}, \\ pi \subset \mathbb{N} \end{pmatrix}$	*vote emitted to accord execute permission for function instance identified by fid (which is the signature of the* $\mathtt{R_{fun}}$ *message broadcast by the client declaring the function) and whose inputs at indices in U are unknowns, and whose known inputs have their digests given by K′ and are flagged as persistent according to pi*
$\mathtt{V_{ins}} \begin{pmatrix} fid \in \mathfrak{S}, \\ \nu : \mathbb{N} \rightharpoonup \mathtt{InputRef} \end{pmatrix}$	*vote emitted to validate an assignment ν of the unknown inputs of a function instance identified by fid, where each unknown index is mapped to a reference in* `InputRef` *(c.f. Def.7)*
$\mathtt{V_{out}} \begin{pmatrix} fid \in \mathfrak{S}, \\ uref \in \{\varnothing\} \cup \mathfrak{H}, \\ r \in \mathfrak{H} \\ po \in \mathbb{B} \end{pmatrix}$	*vote emitted to confirm that the result of the function instance identified by fid, given the agreement on unknown inputs with digests in uref (if multi-party), has a digest of r and ought to be stored iff* $po = \top$
$\mathtt{V_{sto}} \begin{pmatrix} fid \in \mathfrak{S}, \\ h \in \mathfrak{H}, \\ pk \in \{\varnothing\} \cup \mathbb{N} \end{pmatrix}$	*vote emitted to confirm having forwarded a storage request for a value of digest h, involved in the function instance identified by fid if* $pk = \varnothing$*, it corresponds to its output if* $pk \in \mathbb{N}$*, it corresponds to its* pk^{th} *input*

Signatures of these votes are collected by `provers` to constitute various "Ledgera proofs", as defined in Definition 6.

Definition 6. *We denote by:*

$$\begin{array}{lcll} \mathtt{LP_{FD}} & = & \mathtt{V_{fun}} \times \mathcal{P}(\mathfrak{S}) & \textit{the set of Ledgera Proofs of Function Declaration} \\ \mathtt{LP_{VIA}} & = & \mathtt{V_{ins}} \times \mathcal{P}(\mathfrak{S}) & \textit{the set of Ledgera Proofs of Verification of Inputs Assignment} \\ \mathtt{LP_{C}} & = & \mathtt{V_{out}} \times \mathcal{P}(\mathfrak{S}) & \textit{the set of Ledgera Proofs of Computation Integrity} \\ \mathtt{LP_{S}} & = & \mathtt{V_{sto}} \times \mathcal{P}(\mathfrak{S}) & \textit{the set of Ledgera Proofs of Shipment to Storage} \end{array}$$

A Ledgera Proof $(v, q) \in \mathtt{LP_{FD}} \cup \mathtt{LP_{VIA}} \cup \mathtt{LP_{C}} \cup \mathtt{LP_{S}}$ *is valid iff* $q \in \mathcal{Q}(v)$.
(c.f. Def.1, i.e. q is a quorum of signatures of payload v)

The `InputRef` type, appearing in the definition of the $\mathtt{V_{ins}}$ type in Def.5 is defined in Def.7.

Definition 7. *An input reference is a tuple (aid, I, lps) where* $aid \in \mathfrak{S}$ *is a signature,* $I \subset \mathbb{N}$ *corresponds to the indices at which the referred value may be used as input and* $lps \in \mathtt{LP_S}$ *is a Proof of Storage of the value we want to propose as input. We denote by* `InputRef` *the set of all possible input references.*

5.2 Transactions

In Ledgera, "transactions" (introduced in Def.8 below) package various "Ledgera proofs" (c.f. Def.6) for anchoring in the secure log.

Definition 8. *The set of Ledgera transactions is:*

$$\mathtt{T} = \begin{array}{ll} & \bigcup_{lps \in \mathtt{LP_S}} \{\mathtt{T_{sto}}(lps)\} \\ \cup & \bigcup_{lpfd \in \mathtt{LP_{FD}}} \{\mathtt{T_{fun}}(lpfd)\} \\ \cup & \bigcup_{lpvia \in \mathtt{LP_{VIA}}} \{\mathtt{T_{ins}}(lpvia)\} \\ \cup & \bigcup_{lpc \in \mathtt{LP_C}} \{\mathtt{T_{out}}(lpc)\} \end{array}$$

5.3 Requests

Def.9 below introduces the types of all request messages that can be emitted to stimulate Ledgera.

Definition 9. *We define the Ledgera request messages types as follows:*

$\mathtt{R_{fun}}\begin{pmatrix} i \in \mathbb{N}, \\ (\phi,\psi,K) \in \mathtt{Spec}, \\ po \in \mathbb{B}, \\ pi \subset \mathbb{N} \end{pmatrix}$	*requests the execution of function specified by* $(\phi,\psi,K) \in \mathtt{Spec}$ *(c.f. Def.3) and identified by a nonce* i *which uniquely identifies requests from the same client, with the output flagged as persistent by* $po \in \mathbb{B}$*, and inputs flagged as persistent by their indices in* $pi \subset \mathbb{N}$
$\mathtt{R_{in}}\begin{pmatrix} fid \in \mathfrak{S}, \\ I \subset \mathbb{N}, \\ lps \in \mathtt{LP_S} \end{pmatrix}$	*proposes the value stored on the distributed storage and referred to by the Proof of Storage* $lps \in \mathtt{LP_S}$ *(c.f. Def.6), as input at every index in* $I \subset \mathbb{N}$*, of the function instance identified by* $fid \in \mathfrak{S}$
$\mathtt{R_{sto}}\begin{pmatrix} v \in LAT_{\mathtt{val}}, \\ lpfd \in \mathtt{LP_{FD}}, \\ kind \in \mathbb{N} \cup \mathtt{LP_C} \end{pmatrix}$	*requests storage of value* $v \in LAT_{\mathtt{val}}$ *(c.f. Def.2) associated to the function instance identified by the message* $\mathtt{V_{fun}}$ *in the Proof of Function Declaration* $lpfd \in \mathtt{LP_{FD}}$ *(c.f. Def.6, which also proves the instance has been granted execute permission) as either its input at a given index (if* $kind \in \mathbb{N}$*) or as its output (if* $kind \in \mathtt{LP_C}$*), the latter case requiring a Proof Of Integrity (c.f. Def.6)*

The broadcast of a request $\mathtt{R_{fun}}(i,(\phi,\psi,K),po,pi)$ with signature $\sigma \in \mathfrak{S}$, if valid, initiates the execution of a function instance identified by σ, with specification (ϕ,ψ,K), and with persistence metadata specified by po and pi. In Def.10 we define the validity of such requests.

Definition 10. *A request* $\mathtt{R_{fun}}(i,(\phi,\psi,K),po,pi)$ *is valid iff:*

$$\begin{array}{ll} & (\phi,\psi,K) \text{ is valid} \qquad \textit{/*as per Def.4*/} \\ \wedge & pi \subset \{i \in Dom(K) \mid K(i) \in LAT_{\mathtt{val}}\} \qquad \textit{/*input persistence only concerns known inputs shipped as raw values*/} \end{array}$$

A request $\mathtt{R_{in}}(fid, I, lps)$ corresponds to the proposal of a value in storage (referred to by lps) as an input at every index in I for the function instance identified by fid. Such a request can only be validated in the context of the function instance identified by fid. The validity of a request $\mathtt{R_{in}}$ given an existing valid request $r_{fun} \in \mathtt{R_{fun}}$ identified by signature σ (written "$\mathtt{R_{in}}$ is valid w.r.t. r_{fun} of signature σ") is given by Def.11 below.

Definition 11. *Let* $r_{fun} = \mathtt{R_{fun}}(i,(\phi,\psi,K),po,pi)$ *be a valid request (as per Def.10) identified by signature* σ*. We say a request* $\mathtt{R_{in}}(fid, I, lps)$ *is valid w.r.t.* r_{fun} *of signature* σ *iff:*

$$\begin{array}{ll} & fid = \sigma \qquad \textit{/*the proposal concerns the given function instance*/} \\ \wedge & I \neq \emptyset \qquad \textit{/*the proposal is made for some indices*/} \\ \wedge & I \subseteq ([1,|\phi|] \setminus Dom(K)) \qquad \textit{/*the proposal is made for indices of unknowns*/} \\ \wedge & lps \text{ is valid} \qquad \textit{/*as per Def.6*/} \end{array}$$

A request $\mathtt{R_{sto}}(v, lpfd, kind)$ corresponds to the storage of value $v \in LAT_{\mathtt{val}}$. The Proof of Function Declaration $lpfd \in \mathtt{LP_{FD}}$ issued in this request guarantees the existence of a function instance identified by the fid within the $\mathtt{V_{fun}}(fid, K', U, pi)$ contained in $lpfd$. If $kind \in \mathbb{N}$, the value v corresponds to the input of index $kind$ of that function instance. If $kind \in \mathtt{LP_C}$, the value v corresponds to the output of that function instance, which in that case is justified by the Proof of Integrity $kind \in \mathtt{LP_C}$. The validity of a request $\mathtt{R_{sto}}$ is given by Def.12 below.

Definition 12. *Let* $lpfd = (\mathtt{V_{fun}}(fid, K', U, pi), q) \in \mathtt{LP_{FD}}$*. A request* $\mathtt{R_{sto}}(v, lpfd, kind)$ *is valid iff* $lpfd$ *is a valid* $\mathtt{LP_{FD}}$ *(as per Def.6) and we have:*

$$\begin{array}{ll} & \begin{pmatrix} kind \in \mathbb{N} \;\wedge\; kind \in pi \\ \qquad\quad\;\; \wedge\; K'(kind) = \blacktriangledown(v) \end{pmatrix} \qquad \textit{/*input index must match a persistent input*/} \quad \textit{/*value digest for the input must match*/} \\ \vee & \begin{pmatrix} kind = (\mathtt{V_{out}}(fid', uref, r, po), q') \in \mathtt{LP_C} & \wedge\; kind \text{ is a valid } \mathtt{LP_C} \\ & \wedge\; fid' = fid \\ & \wedge\; po = \top \\ & \wedge\; r = \blacktriangledown(v) \end{pmatrix} \qquad \begin{array}{r} \textit{/*as per Def.6*/} \\ \textit{/*identifiers in } \mathtt{LP_{FD}} \textit{ and } \mathtt{LP_C} \textit{ must match*/} \\ \textit{/*output must be flagged persistent*/} \\ \textit{/*output digest must match*/} \end{array} \end{array}$$

5.4 Queries

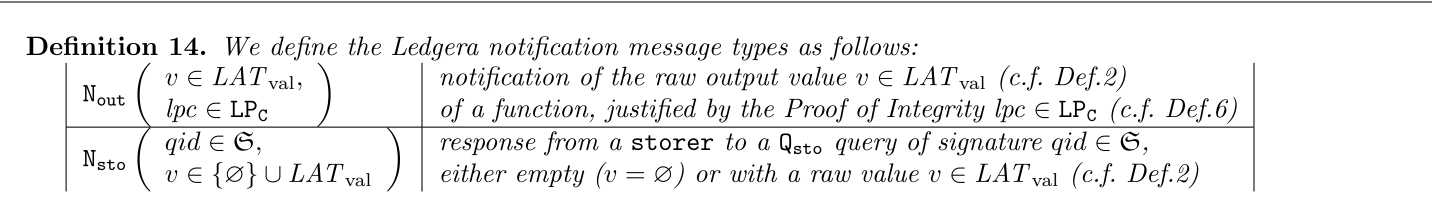
Def.13 below introduces the type of query messages used to retrieve data from the distributed storage.

Definition 13. *We define the Ledgera query message types as follows:*

$\mathtt{Q_{sto}}\left(\begin{array}{l} h \in \mathfrak{H}, \\ lps \in \{\varnothing\} \cup \mathtt{LP_S} \end{array}\right)$	*request to perform a read on the distributed storage at key* $h \in \mathfrak{H}$, *with optional justifying Proof of Storage* $lps \in \mathtt{LP_S}$ *(c.f. Def.6)*

5.5 Notifications

Def.14 below introduces the types of the notification messages that can be sent in reaction to requests (c.f. Def.9) or queries (c.f. Def.13).



Definition 14. *We define the Ledgera notification message types as follows:*

$\mathtt{N_{out}}\left(\begin{array}{l} v \in LAT_{\mathrm{val}}, \\ lpc \in \mathtt{LP_C} \end{array}\right)$	*notification of the raw output value* $v \in LAT_{\mathrm{val}}$ *(c.f. Def.2) of a function, justified by the Proof of Integrity* $lpc \in \mathtt{LP_C}$ *(c.f. Def.6)*
$\mathtt{N_{sto}}\left(\begin{array}{l} qid \in \mathfrak{S}, \\ v \in \{\varnothing\} \cup LAT_{\mathrm{val}} \end{array}\right)$	*response from a* `storer` *to a* $\mathtt{Q_{sto}}$ *query of signature* $qid \in \mathfrak{S}$, *either empty* ($v = \varnothing$) *or with a raw value* $v \in LAT_{\mathrm{val}}$ *(c.f. Def.2)*

6 Behavior at the *storage-layer*

In the following, we present the behavior of Ledgents at the *storage-layer*:

- Sec.6.1 gives an overview of the protocol for "writing" on the distributed storage.
- Sec.6.2 gives an overview of the protocol for "reading" on the distributed storage.
- Sec.6.3 details the behavior of the Ledgents involved in the implementation of the distributed storage using pseudo-code. More precisely, this corresponds to the behavior of individual `executors`, `provers` and `storers`.

For the sake of simplicity, we assume that `executors` and `provers` are present in the same number and always co-located.

6.1 Store data

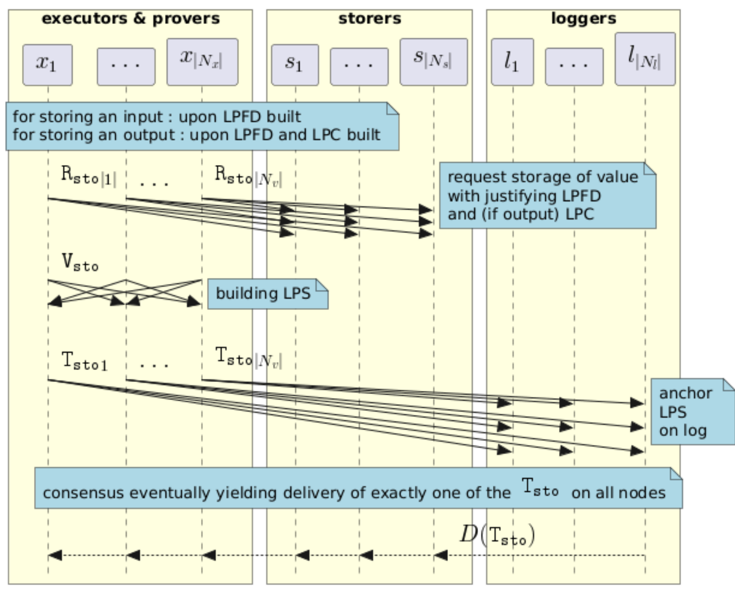


Figure 3: Storing data

The diagram in Fig.3 describes the expected behavior of the Ledgera network whenever a storage request is triggered (for an input, this happens when a "Proof of Function Declaration" ($\mathtt{LP_{FD}}$) is produced, and for an output, when the corresponding "Proof of Integrity" ($\mathtt{LP_C}$) is produced).

To request the storage of some data on the Ledgera network, an `executor` broadcasts an $\mathtt{R_{sto}}$ message, which will eventually be received by all the `storers`.

Upon receiving an $\mathtt{R_{sto}}$ message, a `storer` verifies the "Ledgera proofs" it contains that justify the storage of the data it is shipped with. For storing a persistent input at index i of a function instance, the `storers` verify that the $\mathtt{LP_{FD}}$ is valid and concerns the known input of the function instance at index i, which must be flagged as persistent and have the same digest. For a persistent output of a function instance, `storers` verify that the $\mathtt{LP_{FD}}$ is valid and concerns the output of the function

instance, which must be flagged as persistent, and they verify that the $\texttt{LP}_\texttt{C}$ is valid and the digest of the output corresponds to that of the shipped data. Implementation details are given in Algorithm 3.

Fig.3 shows how each `executor` may send a different $\texttt{R}_\texttt{sto}$ message, which may happen because `executors` do not wait for consensus, and each of them may have different sets of signatures in the quorums that constitute the $\texttt{LP}_\texttt{FD}$ and the $\texttt{LP}_\texttt{C}$.

After emitting a $\texttt{R}_\texttt{sto}$ message, each `executor` also emits a $\texttt{V}_\texttt{sto}$ vote, which justifies that the `executor` has broadcast the storage request (c.f. Algorithm 1).

After having received $f+1$ signed copies of the corresponding $\texttt{V}_\texttt{sto}$ vote from different Ledgents, a `prover` constitutes a quorum of signatures and produces a "Proof of Shipment to Storage" ($\texttt{LP}_\texttt{S}$). The existence of such an $\texttt{LP}_\texttt{S}$ guarantees that the data will eventually be stored and available in the distributed storage. Indeed, it means that at least one correct and honest `executor` has broadcast the corresponding $\texttt{R}_\texttt{sto}$ message. By our hypothesis on the network (c.f. Asmpt.1), this implies that all honest `storers` will eventually receive that specific $\texttt{R}_\texttt{sto}$ message. As a result, they will all store the data on their local copies of the storage, and the data will be available.

Once a `prover` has produced a $\texttt{LP}_\texttt{S}$, the `prover` submits it to the `loggers` in the form of transaction $\texttt{T}_\texttt{sto}$ for anchoring. Implementation details are given in Algorithm 2. Via a consensus mechanism, the secure log guarantees that, eventually, exactly one such $\texttt{T}_\texttt{sto}$ transaction (one per function instance and per persistent input/output) will be delivered.

The content of the secure log being public, non-revocable and non-falsifiable, any of the nodes will eventually have access to it. Thus any node will eventually have the confirmation that one of the $\texttt{T}_\texttt{sto}$ has been included in the log. We denote this fact in Fig.3 by the dotted arrow carrying "$D(\texttt{T}_\texttt{sto})$". Remark that we have used the same dotted arrow for "$D(\texttt{T}_\texttt{ins})$" in Fig.1 to represent the fact that `executors` wait for delivery of a $\texttt{T}_\texttt{ins}$.

6.2 Query data

The diagram on Fig.4 describes the expected behavior of the Ledgera network whenever a node requests data from the distributed storage.

To do so, the node broadcasts a $\texttt{Q}_\texttt{sto}$ query that contains the digest of the data it wants to retrieve, as well as an optional Proof of Shipment to Storage (which, if present, guarantees its availability).

The $\texttt{Q}_\texttt{sto}$ query will eventually be received by all the `storers`.

Upon reception, each individual `storer` answers this query with a $\texttt{R}_\texttt{sto}$ response. If is has the data that corresponds to the provided digest, it returns it with the response. Otherwise, if there is a Proof of Shipment to Storage ($\texttt{LP}_\texttt{S}$) in the $\texttt{Q}_\texttt{sto}$ query, and if this proof is valid and corresponds to the provided digest, then the `storer` is guaranteed to eventually store that value. As a result, it waits to receive the value before answering the query. If, on the contrary, the $\texttt{Q}_\texttt{sto}$ query does not contain a $\texttt{LP}_\texttt{S}$ or contains an invalid $\texttt{LP}_\texttt{S}$, then it immediately answers with a $\texttt{R}_\texttt{sto}$ that does not contain any value. Implementation details are given in Algorithm 3.

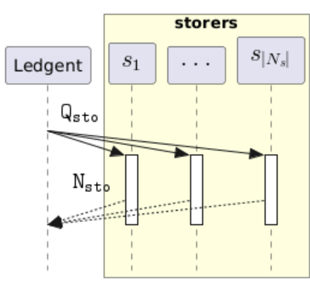


Figure 4: Querying data

6.3 Detailed behaviors of Ledgents per involved roles

Algorithm 1 details the procedures available to the `executors` to trigger the storage of data on the *storage-layer*, as illustrated on Fig.1.

Algorithm 1: Procedures available to `executors` to trigger storing on the *storage-layer*

```
1 Procedure store_input(v ∈ LAT_val, lpfd ∈ LP_FD, i ∈ ℕ):                    /* c.f. Def.2 & Def.6 */
2     dbg_assert!(▼(v) = lpfd.v_fun.K(i)) ;                 /* the value's digest should match that from the lpfd */
3     broadcast R_sto(v, lpfd, i) to N_s ;                                            /* c.f. Def.9 */
4     broadcast V_sto(lpfd.v_fun.fid, lpfd.v_fun.K(i), i) to N_p ;                    /* c.f. Def.5 */
5 Procedure store_output(v ∈ LAT_val, lpfd ∈ LP_FD, lpc ∈ LP_C):                 /* c.f. Def.2 & Def.6 */
6     dbg_assert!(lpfd.v_fun.fid = lpc.v_out.fid) ;    /* both proofs should concerns the same function instance */
7     dbg_assert!(▼(v) = lpc.v_out.r) ;                       /* the value's digest should match that from the lpc */
8     broadcast R_sto(v, lpfd, lpc) to N_s ;                                          /* c.f. Def.9 */
9     broadcast V_sto(lpfd.v_fun.fid, lpc.v_out.r, ∅) to N_p ;                        /* c.f. Def.5 */
```

Algorithm 2 describes the behavior of `provers` w.r.t. the *storage-layer*. On each correct node that hosts a `provers`, there is a dedicated thread that runs the algorithm from Alg.2 and the pertinent messages the node receives are redirected to that thread.

Algorithm 3 describes the behavior of the `storers`. On each correct node that hosts a `storers`, there is a dedicated thread that runs the algorithm from Alg.3 and the pertinent messages the node receives are redirected to that thread.

Algorithm 2: Behavior of the `provers` w.r.t. the *storage-layer*

```
Type : StorageId = 𝔖 × {∅} ∪ ℕ;     /* ∀ fid ∈ 𝔖, we store either output id-ed by ∅ or input at index i ∈ ℕ */
1   proving : Map⟨V_sto ↦ Set⟨𝔖⟩⟩ ← {} ;   /* initialize map to store quorums which are being built
    proved  : Set⟨StorageId⟩     ← {}        and set to ignore pairs for which a quorum has already been built */
2  upon receiving v_sto = V_sto(fid, h, pk) with signature σ ∈ 𝔖 :
3    if (fid, pk) ∉ proved then                          /* if not already produced a PoS for this pair */
4      proving[v_sto] ← proving[v_sto] ∪ {σ} ;            /* add signature to quorum being built */
5      if |proving[v_sto]| ≥ f + 1 then                   /* if there are now enough signatures */
6        q ← proving.pop(v_sto).select(f + 1) ;           /* forming quorum of signatures */
7        for v'_sto ∈ {v'_sto ∈ proving | v'_sto.fid = v_sto.fid ∧ v'_sto.pk = v_sto.pk} do
8          proving.delete(v'_sto);                         /* garbage collecting */
9        proved.append((fid, pk));
10       lps ← (v_sto, q) ;                                /* forming LP_S (c.f. Def.6) */
11       broadcast T_sto(lps) to N_l ;                     /* c.f. Def.8 */
12 upon delivering D(T_sto(lps)) :
13   if (lps.v_sto.fid, lps.v_sto.pk) ∉ proved then
14     for v_sto ∈ {v_sto ∈ proving | v_sto.fid = lps.v_sto.fid ∧ v_sto.pk = lps.v_sto.pk} do
15       proving.delete(v_sto) ;                           /* garbage collecting */
16     proved.append((lps.v_sto.fid, lps.v_sto.pk));
```

Algorithm 3: Behavior of `storers`

```
1   local   : Map⟨𝔥 ↦ LAT_val⟩        ← {} ;   /* local replica of the storage
    pending : Map⟨𝔥 ↦ Set⟨N × 𝔖⟩⟩   ← {}          pending queries */
2  upon receiving r_sto = R_sto(v, lpfd, kind) :                /* c.f. Def.9 */
3    h ← ▼(v) ;
4    if r_sto is valid ∧ h ∉ local then           /* r_sto valid as per Def.12 & not already stored */
5      local[h] ← v;
6      for (n, σ) ∈ pending.pop(h) do                           /* answers pending queries */
7        broadcast N_sto(σ, local[h]) to {n} ;                  /* c.f. Def.14 */
8  upon receiving Q_sto(h, lps) from n ∈ N with signature σ ∈ 𝔖 :   /* c.f. Def.13 */
9    if h ∈ local then                                 /* immediately answers if already in storage */
10     broadcast N_sto(σ, local[h]) to {n} ;                    /* c.f. Def.14 */
11   else if lps ∈ LP_S ∧ lps.v_sto.h = h ∧ lps is valid then  /* delay answer if LP_S validity guarantees eventual availability */
12     pending[h] ← pending[h] ∪ {(n, σ)};
```

Finally, in Algorithm 4, we detail the procedure available to `executors` and `clients` to retrieve data from the distributed storage. After having broadcast the $\mathtt{Q_{sto}}$ query, the node requesting the value waits for a $\mathtt{R_{sto}}$ response carrying a value which digest matches the one in its initial query, waiting for at most $f+1$ responses, to guarantee receiving at least one from an honest `storer`. Duplicate responses from the same node do not count towards this $f+1$ threshold.

Algorithm 4: Procedures available to `executors` and `clients` to retrieve data from the distributed storage

1 **Procedure** `query_storage`($h \in \mathfrak{H},\ lps \in \{\varnothing\} \cup \mathtt{LP_S}$)**:**
2     $resp \leftarrow \emptyset$;
3   **broadcast** $\mathtt{Q_{sto}}(h, lps)$ **with signature** $\sigma \in \mathfrak{S}$ **to** N_s ;
4   **while** $|resp| \leq f + 1$ **do**
5   **upon receiving** $\mathtt{N_{sto}}(\sigma, v)$ ***from*** $n \in N_s$ **:**
6   **if** $v \neq \varnothing \wedge \blacktriangledown(v) == h$ **then**
7   **return** v;
8             $resp \leftarrow resp \cup \{n\}$;
9   **return** $\varnothing$;
10 **Procedure** `unwrap_val`($x \in LAT_{\mathrm{val}} \cup \mathtt{LP_S}$)**:**
11   **if** $x \in LAT_{\mathrm{val}}$ **then**
12   **return** x;
13   **else**
14   **return** `query_storage`($x.v_{sto}.h, x$);

7 Behavior at the *logging-layer*

7.1 Valid history of transactions

The Ledgera secure log orders and stores transactions. It produces a unique list of transactions which form a valid history as defined in Definition 15.

Definition 15. *We define the set of all valid histories of transactions* $\mathtt{HT} \subset \mathtt{T}^*$ *inductively as follows:* $[] \in \mathtt{HT}$ *and for any* $l \in \mathtt{HT}$ *and any* $t \in \mathtt{T}$*,* $l.t \in \mathtt{HT}$ *iff:*

- *if* $t = \mathtt{T_{sto}}(lps)$ *with* $lps \in \mathtt{LP_S}$ *then:*
  - ***storage guaranteed:*** *lps is valid*
  - ***unicity:*** $\nexists lps' \in \mathtt{LP_S}$ *s.t.* $lps'.v_{sto}.fid = lps.v_{sto}.fid \wedge lps'.v_{sto}.pk = lps.v_{sto}.pk \wedge \mathtt{T_{sto}}(lps') \in l$
- *if* $t = \mathtt{T_{fun}}(lpfd)$ *with* $lpfd \in \mathtt{LP_{FD}}$ *then:*
  - ***execute rights given:*** *lpfd is valid*
  - ***unicity:*** $\nexists lpfd' \in \mathtt{LP_{FD}}$ *s.t.* $lpfd'.v_{fun}.fid = lpfd.v_{fun}.fid \wedge \mathtt{T_{fun}}(lpfd') \in l$
- *if* $t = \mathtt{T_{ins}}(lpvia)$ *with* $lpvia \in \mathtt{LP_{VIA}}$ *then:*
  - ***preceded by declaration with unknowns:*** $\exists lpfd \in \mathtt{LP_{FD}}$ *s.t.* $lpfd.v_{fun}.fid = lpvia.v_{ins}.fid \wedge lpfd.v_{fun}.U \neq \emptyset \wedge \mathtt{T_{fun}}(lpfd) \in l$
  - ***argument array verified:*** *lpvia is valid*
  - ***unicity:*** $\nexists lpvia' \in \mathtt{LP_{VIA}}$ *s.t.* $lpvia'.v_{ins}.fid = lpvia.v_{ins}.fid \wedge \mathtt{T_{ins}}(lpvia') \in l$
- *if* $t = \mathtt{T_{out}}(lpc)$ *with* $lpc \in \mathtt{LP_C}$ *then:*
  - ***preceded by declaration:*** $\exists lpfd \in \mathtt{LP_{FD}}$ *s.t.* $lpfd.v_{fun}.fid = lpc.v_{out}.fid \wedge \mathtt{T_{fun}}(lpfd) \in l$
  - ***agreement on unknowns:*** *if* $lpc.v_{out}.uref \neq \varnothing$ *then* $\exists t' = \mathtt{T_{ins}}(lpvia) \in l$ *s.t.* $lpvia.v_{ins}.fid = lpc.v_{out}.fid \wedge \blacktriangledown(t') = lpc.v_{out}.uref$
  - ***integrity verified:*** *lpc is valid*
  - ***unicity:*** $\nexists lpc' \in \mathtt{LP_C}$ *s.t.* $lpc'.v_{out}.fid = lpc.v_{out}.fid \wedge \mathtt{T_{out}}(lpc') \in l$

The `loggers` of the secure log order traces of events occurring on the Ledgera network in the form of transactions.

$\mathtt{T_{sto}}$ transactions correspond to logging a write event that has occurred (or will eventually occur) on the distributed storage, corresponding to storing either an input or an output of a function instance. So that it corresponds to a real write, the $\mathtt{T_{sto}}(lps)$ transaction must carry a valid $lps \in \mathtt{LP_S}$. The same write event should also not be anchored twice on the secure log. As a result, a valid history can only contain a single $\mathtt{T_{sto}}(lps)$ transaction per function id $lps.v_{sto}.fid \in \mathfrak{S}$ and per persistent input-output kind $lps.v_{sto}.pk \in \{\varnothing\} \cup \mathbb{N}$.

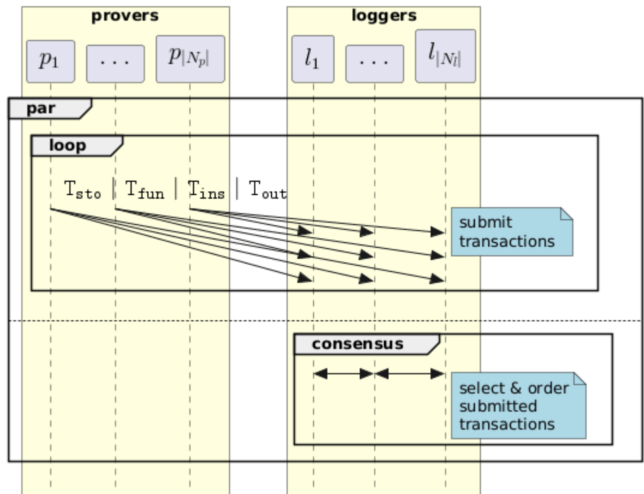
$\mathtt{T_{fun}}$ transactions correspond to logging the declaration of a new function instance. Function instances being uniquely identified by the signature of the initial $\mathtt{R_{fun}}$ message of their declaration, a valid history can only contain a single $\mathtt{T_{fun}}(lpfd)$ per function instance identifier $lpc.v_{fun}.fid \in \mathfrak{S}$. Moreover, $lpfd$ must be a valid $\mathtt{LP_{FD}}$, proving that at least one honest `executor` has granted execute rights for the function instance via signing and broadcasting $lpfd.v_{fun}$.

A $\mathtt{T_{ins}}(lpvia)$ transaction corresponds to logging that an agreement on the concrete values for the unknown arguments of a function instance has been reached. This therefore requires that a previous $\mathtt{T_{fun}}(lpfd)$ has occurred for the same function instance i.e. $lpfd.v_{fun}.fid = lpvia.v_{ins}.fid$. Moreover, a $\mathtt{T_{ins}}$ only appears for function instances that are multi-party i.e., that have unknown arguments. Hence the constraint $lpfd.v_{fun}.U \neq \emptyset$. The embedded $lpvia$ must be valid, which guarantees that at least one honest `executor` has verified that the assignment $lpvia.v_{ins}.\nu$ of the unknowns is correct w.r.t. the function specification corresponding to $lpvia.v_{ins}.fid = lpfd.v_{fun}.fid$. The $\mathtt{T_{ins}}(lpvia)$ must also be unique per function instance id $lpvia.v_{ins}.fid$.

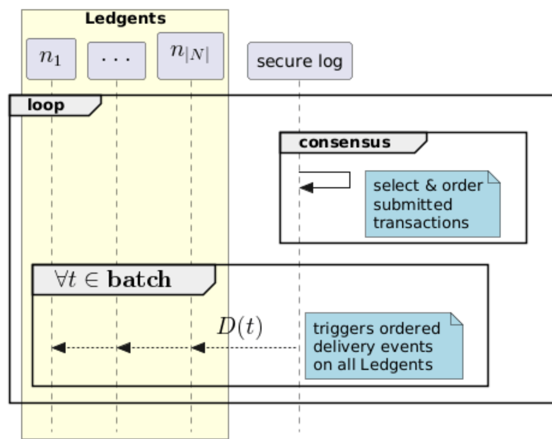
Finally, a $\mathtt{T_{out}}(lpc)$ transaction corresponds to logging that an agreement on the value resulting from the execution of an function instance has been reached. This therefore requires that a previous $\mathtt{T_{fun}}(lpfd)$ has occurred for the same function instance i.e. $lpfd.v_{fun}.fid = lpc.v_{out}.fid$. A valid history can contain a single $\mathtt{T_{out}}(lpc)$ transaction per function instance identifier $lpc.v_{out}.fid \in \mathfrak{S}$ and the associated lpc must be a valid Proof of Integrity that the result of function instance of identifier $lpc.v_{out}.fid$ has a hash value corresponding to $lpc.v_{out}.r \in \mathfrak{H}$. Moroever, if the function instance is multi-party i.e., has unknowns, which is given by $lpc.v_{out}.uref \neq \varnothing$ (and also by $lpfd.v_{fun}.U \neq \emptyset$), the $\mathtt{T_{out}}(lpc)$ must be preceded, in the secure log, by a $\mathtt{T_{ins}}(lpvia)$ referring to the same function instance p $lpvia.v_{ins}.fid = lpc.v_{out}.fid$ and to ensure everyone voted on the result, after having agreed on the same assignment of unknowns (via delivering the $\mathtt{T_{ins}}$), the reference in $lpc.v_{out}.uref$ must correspond to the digest of that $\mathtt{T_{ins}}$ i.e., $lpc.v_{out}.uref = \blacktriangledown(\mathtt{T_{ins}}(lpvia))$.

7.2 Overview of interactions with the secure log

As illustrated on Fig.5a, the `provers` may submit transactions to the secure log, via broadcasting them to the `loggers`. As transactions are being submitted and received, the `loggers` participate in successive instances of a BFT consensus algorithm [10] (c.f. Remark 1) which result in the selection and ordering of these transactions.



(a) Submitting transactions to the secure log



(b) The secure log delivering transactions

Figure 5: Interactions with the secure log

As consensus takes place within the `loggers`, the secure log deliver a subset of the transactions that have been submitted. We assume delivery events occur on every node whenever a transaction is delivered in the secure log, as illustrated on Fig.5b.

7.3 The secure log and its properties

Ledgera does not specify a concrete implementation for the secure log and how the `loggers` implement it. We only assume certain properties any such implementation must uphold.

In the following, we define the Ledgera secure log as a distributed object and a number of properties it may have under various assumptions.

Definition 16. *The LedgerA secure log is an object* 🗎*:*$N \mapsto \mathtt{T}^*$ *such that for any* $n \in N$*:*

- *if* $n \in N \setminus F$, 🗎$(n) \in \mathtt{T}^*$ *correspond to the local order on* n *of "transaction delivery events", denoted as* $D(t)$ *for any* $t \in \mathtt{T}$
- *if* $n \in F$, 🗎(n) *may take any value*

Property 1. *The Ledgera secure log always satisfies:*

- ***initialization****: at the genesis of the system, we have, for all correct node* $n \in N \setminus F$, 🗎$(n) = []$
- ***validity****: at any given time, for any correct node* $n \in N \setminus F$, 🗎$(n) \in \mathtt{HT}$
- ***consistency****: for any two correct nodes* n *and* n' *we either have* 🗎(n) *is a prefix of* 🗎(n') *or* 🗎(n') *is a prefix of* 🗎(n)

Property 2. *Under Asmpt.1, the Ledgera secure log satisfies a form of* ***input liveness****:*
for any correct `prover` $n \in N_p \setminus F$*, if at a given time,* n *broadcasts a transaction* t *to the set* N_l *of all* `loggers` *then, eventually:*

- *if* $t = \mathtt{T_{sto}}(lps)$ *then a certain* $D(\mathtt{T_{sto}}(lps'))$ *event will occur on all correct nodes, for a certain equivalent* $lps' \in \mathtt{LP_S}$ *i.e., s.t.* $lps'.v_{sto}.fid = lps.v_{sto}.fid \wedge lps'.v_{sto}.pk = lps.v_{sto}.pk$
- *if* $t = \mathtt{T_{fun}}(lpfd)$ *then a certain* $D(\mathtt{T_{fun}}(lpfd'))$ *event will occur on all correct nodes, for a certain equivalent*

$lpfd' \in \mathtt{LP_{FD}}$ *i.e., s.t.* $lpfd'.v_{fun}.fid = lpfd.v_{fun}.fid$

- *if* $t = \mathtt{T_{ins}}(lpvia)$ *then a certain* $D(\mathtt{T_{ins}}(lpvia'))$ *event will occur on all correct nodes, for a certain equivalent* $lpvia' \in \mathtt{LP_{VIA}}$ *i.e., s.t.* $lpvia'.v_{ins}.fid = lpvia.v_{ins}.fid$
- *if* $t = \mathtt{T_{out}}(lpc)$ *then a certain* $D(\mathtt{T_{out}}(lpc'))$ *event will occur on all correct nodes, for a certain equivalent* $lpc' \in \mathtt{LP_C}$ *i.e., s.t.* $lpc'.v_{out}.fid = lpc.v_{out}.fid$

Property 3. *Under Asmpt.1, the Ledgera secure log satisfies a form of* ***replication liveness****:*
for any correct node $n \in N \setminus F$*, if a* $D(t)$ *event occurs on* n *for a certain transaction* $t \in \mathtt{T}$ *then, eventually, for any other correct node* $n' \in N \setminus F$*,* $D(t)$ *will have occurred on* n'

The behavior of the secure log can be implemented by having the $3f+1$ `loggers` run a Byzantine Fault Tolerant consensus algorithm [10] (c.f. Remark 1), the validity of transactions being defined w.r.t. Def.15.

8 Behavior at the *coreset-layer*

8.1 Overview

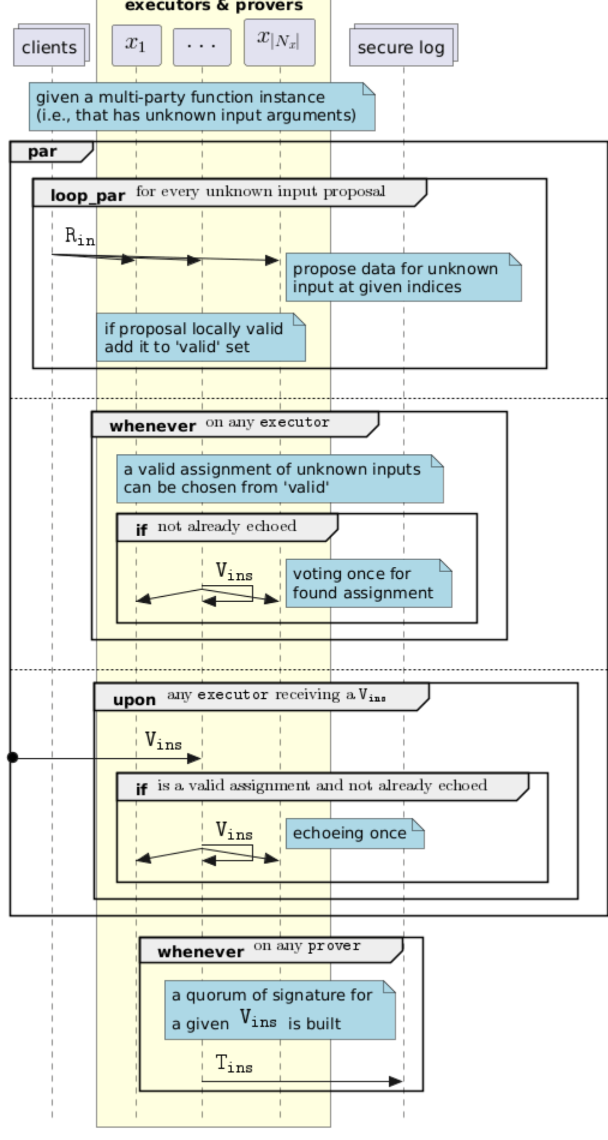


Figure 6: Description of the behavior at the *coreset-layer*

In the case where the function specification contains unknown input arguments, it cannot be immediately executed. One

must first agree on the value of the missing arguments, which is the object of the *coreset-layer* algorithm, which is described on Fig.6.

At any given time (before agreement is reached, as indicated by the occurrence of a deliver event $D(\mathtt{T_{ins}}(lpvia))$), any `client` can propose a value to be chosen as input argument (for a subset of indices) by broadcasting a $\mathtt{R_{in}}(fid, I, lps)$ message to the set N_x of `executors`. In this message, $fid \in \mathfrak{S}$ is the unique identifier of the computation instance, $I \subset \mathbb{N}$ is the subset of the indices of the function at which the value is proposed and $lps \in \mathtt{LP_S}$ is the Proof of (Shipment to) Storage of the value that is proposed.

Upon receiving such a $\mathtt{R_{in}}(fid, I, lps)$ message, a `executor` that has already received and validated the corresponding $\mathtt{R_{fun}}$ declaration (i.e. a $\mathtt{R_{fun}}$ message of signature fid) do as follows. At first it verifies that it has not already received it to eliminate duplicates. If it is not the case, it verifies the set of indices $I \subset \mathbb{N}$ is valid i.e. is not empty and is included in the indices of the unknown arguments of the function instance. It also verifies that the shipped Proof of (Shipment to) Storage lps has a valid quorum of signatures. After validating the received $\mathtt{R_{in}}$ message, it stores it in a set of "locally valid" proposals from which it then may choose in order to build a total assignment of the unknowns of the function instance.

Each addition of a new valid $\mathtt{R_{in}}$ to the set of locally valid proposals may trigger the proposition of an assignment. This assignment must satisfy the constraint ψ associated with the function instance. When such an assignment is found for the first time, the `executor` builds a $\mathtt{V_{ins}}$ message and broadcasts it to $N_x \cup N_p$, provided it has not already echoed the same vote.

`executors` may echo $\mathtt{V_{ins}}$ votes received from other `executors` to guarantee finding an assignment of unknowns in cases where Byzantines `clients` propose different subsets of valid proposals to different subsets of `executors`. Upon receiving a $\mathtt{V_{ins}}$, a `executors` verifies it contains a valid assignment of unknowns (given ψ) and that each assigned unknown refers to a locally valid signed $\mathtt{R_{in}}$. If these conditions are met, and if the `executor` has not already echoed the same vote, it echoes it i.e., broadcasts it to N_p.

As `executors` produce and echo $\mathtt{V_{ins}}$ votes, `provers` collect them and eventually produce a Proof of Unknowns Assignment Verification ($\mathtt{LP_{VIA}}$) that they submit to the secure log in the form of a $\mathtt{T_{ins}}$ transaction. Note that if ψ admits multiple satisfying assignments, different subsets of honest `executors` may independently find different valid assignments, causing different `provers` to submit competing $\mathtt{T_{ins}}$ candidates to the `loggers`. Agreement on which assignment becomes canonical is delegated to the secure log's consensus: the **unicity** condition of Def. 15 guarantees that at most one $\mathtt{T_{ins}}$ per function-instance identifier is ever committed. This separation is by design: the *coreset-layer* is responsible for validating candidate assignments while the *logging-layer* is responsible for selecting one among them.

Then, eventually, a $D(\mathtt{T_{ins}}(lpvia))$ delivery event occurs on every node, closing, on the `executors`, the procedure corresponding to collecting the unknown arguments for the $lpvia.v_{ins}.fid$ function instance.

8.2 Assignments of unknown inputs

The goal of the *coreset-layer*is to find valid assignments of the unknown inputs of a multi-party function instance. In the following we define what is this "validity of assignments" in Definition 17 and how to verify it in Algorithm 5.

Definition 17. *Let* $r_{fun} = \mathtt{R_{fun}}(i, (\phi, \psi, K), po, pi)$ *be a valid request (as per Def.10) identified by signature* σ. *We say an assignment of unknowns* $\nu \in \mathtt{InputRef}^{\mathbb{N}}$ *is valid w.r.t.* r_{fun} *of signature* σ *iff:*

$$
\begin{aligned}
& Dom(\nu) = [1, |\phi|] \setminus Dom(K) && \text{/*fills exactly the unknowns*/} \\
\wedge \quad & \forall i \in Dom(\nu), r_{in} = \mathtt{R_{in}}(\sigma, \nu(i).I, \nu(i).lps) \text{ s.t. } \begin{pmatrix} r_{in} \text{ valid w.r.t. } r_{fun} \text{ of signature } \sigma \\ \wedge\ r_{in} \text{ accepts } \nu(i).aid \text{ as signature} \end{pmatrix} && \text{/* can reconstitute valid proposal per Def.11 */} \\
\wedge \quad & \forall i \in Dom(\nu),\ i \in \nu(i).I && \text{/*match desired index*/} \\
\wedge \quad & \begin{pmatrix} \text{given } \forall i \in [1, |\phi|],\ v_i = \begin{cases} \textbf{if } i \in Dom(\nu) & \blacktriangledown^{-1}(\nu(i).lps.v_{sto}.h) \\ \textbf{elif } i \in Dom(K) \cap LAT_{\text{val}} & K(i) \\ \textbf{else} & \blacktriangledown^{-1}(K(i).v_{sto}.h) \end{cases} \\ \text{we have that } \psi(v_1, \cdots, v_{|\phi|}) \text{ holds} \end{pmatrix} && \text{/*satisfies predicate*/}
\end{aligned}
$$

In Definition 17:

- the first condition signifies that the assignment gives values exactly to the unknown inputs of the function instance corresponding to r_{fun} of signature fid
- the second condition signifies that every argument reference in the assignment corresponds to a valid and signed (non-forged) $\mathtt{R_{in}}$ input proposal
- the third that the index at which the input proposal is assigned is compatible for the indices for which the proposal was made
- the fourth that, after retrieving the raw values of all known inputs and all unknown inputs assigned by the assignment, the resulting inputs array satisfies the predicate of the function instance

Algorithm 5: Procedures to verify and find assignments in the *coreset-layer*

```
1  Procedure check_assignment(fid ∈ 𝔖, r_fun = R_fun(_, (_, ψ, K), _, _), knowns : ℕ ⇀ LAT_val, ν : ℕ ⇀ InputRef):
2      dbg_assert(Dom(knowns) = Dom(K));
3      dbg_assert(∀i ∈ Dom(K), knowns(i) = K(i) if K(i) ∈ LAT_val else ▼^{-1}(K(i).v_sto.h));
4      if Dom(ν) ≠ [1, |ψ|] \ Dom(K) then
5          return ⊥ ;            /* invalid due to not assigning proposals exactly to the unknowns */
6      inputs ← knowns ;         /* prepares array of inputs to test predicate */
7      for i ∈ Dom(ν) do
8          r_in ← R_in(fid, ν(i).I, ν(i).lps) ;   /* reconstitutes the referred-to input proposal */
9          if (   i ∉ ν(i).I
                ∨ r_in is not valid w.r.t. r_fun of signature fid
                ∨ ν(i).aid not signature of r_in ) then      /* as per Def.11 */
10             return ⊥ ;        /* invalid due to invalidity of a specific argument reference */
11         inputs[i] ← query_storage(ν(i).lps.v_sto.h, ν(i).lps) ;   /* completes inputs array (query guaranteed as ν(i).lps valid) */
12     return ψ(inputs(1), ··· , inputs(|ψ|)) ;   /* assignment valid if ψ satisfied */
13 Procedure find_assignment(valid ∈ Set⟨R_in × 𝔖⟩, fid ∈ 𝔖, r_fun = R_fun(_, (_, ψ, K), _, _), knowns : ℕ ⇀ LAT_val):
14     U ← [1, |ϕ|] \ Dom(K) ;   /* indices of all unknown inputs */
15     potential ← {ν ∈ InputRef^U | ∀i ∈ U, ∃(r_in, aid) ∈ valid, ν(i) = (aid, r_in.I, r_in.lps)} ;
16     if ∃ν ∈ potential s.t. check_assignment(fid, r_fun, knowns, ν) = ⊤ then   /* c.f. Remark 2 */
17         return ν ;            /* returns valid assignment found via mapping locally valid proposals */
18     return ∅ ;                /* could not find any valid assignment */
```

Remark 2. *The* `find_assignment` *procedure leaves the concrete search strategy abstract. In practice, an implementation must search potential for a ν satisfying* `check_assignment`. *Exhaustively iterating over all combinations of valid proposals across the $|U|$ unknown indices yields worst-case complexity $O(|valid|^{|U|})$, exponential in the number of unknowns. An implementation can reduce this by using a constraint-satisfaction solver or backtracking search.*

In Property 4 we show that the procedure `check_assignment` from Algorithm 5 determines the validity of an assignment w.r.t. a function instance, as defined in Definition 17.

Property 4. *For all valid $r_{fun} = \mathtt{R_{fun}}(_, (_, \psi, K), _, _)$ of signature $fid \in \mathfrak{S}$, given $knowns : \mathbb{N} \rightharpoonup LAT_{\mathrm{val}}$ such that $Dom(knowns) = Dom(K)$ and $\forall i \in Dom(K)$, $knowns(i) = K(i)$ **if** $K(i) \in LAT_{\mathrm{val}}$ **else** $\blacktriangledown^{-1}(K(i).v_{sto}.h)$ we have:*

$$\forall\ \nu \in \mathtt{InputRef}^{\mathbb{N}},\ \ \left(\ \mathtt{check_assignment}(fid, r_{fun}, knowns, \nu) = \top\ \Leftrightarrow\ \nu \text{ is valid w.r.t. } r_{fun} \text{ of signature } \sigma\ \right)$$

Proof. In Definition 17 (which defines the validity of an assignment of unknowns w.r.t. a signed $\mathtt{R_{fun}}$ request) we have that:

- the first condition is guaranteed by lines 4-5 of Alg.5
- the second and third by lines 7-10
- the fourth by the construction of "*inputs*" on lines 6 (with the hypothesis on "*knowns*") and 11 as well as the verification of ψ on line 12

□

In Property 5 we show that the procedure `find_assignment` from Algorithm 5 returns a valid assignment is one such assignment can be constituted using the received signed $\mathtt{R_{in}}$ requests stored in the *valid* variable.

Property 5. *For all valid $r_{fun} = \mathtt{R_{fun}}(_, (_, \psi, K), _, _)$ of signature $fid \in \mathfrak{S}$, given $knowns : \mathbb{N} \rightharpoonup LAT_{\mathrm{val}}$ such that $Dom(knowns) = Dom(K)$ and $\forall i \in Dom(K)$, $knowns(i) = K(i)$ **if** $K(i) \in LAT_{\mathrm{val}}$ **else** $\blacktriangledown^{-1}(K(i).v_{sto}.h)$ we have:*

for any valid $\in Set\langle \mathtt{R_{in}} \times \mathfrak{S}\rangle$ *we have:*

$$\exists \nu \in \left\{ \nu \in \mathtt{InputRef}^U \;\middle|\; \begin{array}{l} \forall i \in U,\ \exists (r_{in}, aid) \in valid,\ \nu(i) = (aid, r_{in}.I, r_{in}.lps) \\ \wedge\ \nu \textit{ is valid w.r.t. } r_{fun} \textit{ of signature } fid \end{array} \right\}$$

$$\Leftrightarrow$$

$$\left(\ \mathtt{find_assignment}(valid, fid, r_{fun}, knowns) \in \mathtt{InputRef}^{\mathbb{N}} \textit{ is valid w.r.t. } r_{fun} \textit{ of signature } fid\ \right)$$

Proof. The existential condition in `find_assignment` tests whether some $\nu \in potential$ satisfies $\mathtt{check_assignment}(fid, r_{fun}, known$ $\top$. By Prop.4, this is equivalent to ν being valid w.r.t. r_{fun} of signature fid. Hence `find_assignment` returns a valid ν if and only if such a ν exists in *potential*. □

8.3 Detailed behaviors of Ledgents per involved roles

On the *coreset-layer*, the `provers` simply collect signatures of $\mathtt{V_{ins}}$ votes and produce quorums of such signatures as Proofs of Unknowns Assignment Verification ($\mathtt{LP_{VIA}}$). This is explicited in Algorithm 6. Notice this is quite similar to their behavior on the *storage-layer* (which we have seen in Algorithm 2).

Algorithm 6: Behavior of the `provers` w.r.t. the *coreset-layer*

1. $\begin{array}{lcll} proving & : & Map\langle \mathtt{V_{ins}} \mapsto Set\langle \mathfrak{S}\rangle\rangle & \leftarrow \{\} \\ proved & : & Set\langle \mathfrak{S}\rangle & \leftarrow \{\} \end{array}$; `/* initialize map to store quorums being built and set to ignore` $fids$ `for which a` $\mathtt{LP_{VIA}}$ `has already been built */`
2. **upon receiving** $v_{ins} = \mathtt{V_{ins}}(fid, \nu)$ **with signature** $\sigma \in \mathfrak{S}$ **:**
3.   **if** $fid \notin proved$ **then** `/* if not already produced a` $\mathtt{LP_{VIA}}$ `for this` fid `*/`
4. $proving[v_{ins}] \leftarrow proving[v_{ins}] \cup \{\sigma\}$; `/* add signature to quorum being built */`
5.   **if** $|proving[v_{ins}]| \geq f + 1$ **then** `/* if there are now enough signatures */`
6. $q \leftarrow proving.pop(v_{ins}).select(f+1)$; `/* forming quorum of signatures */`
7.   **for** $v'_{ins} \in \{v'_{ins} \in proving \mid v'_{ins}.fid = v_{ins}.fid\}$ **do**
8. $proving.delete(v'_{ins})$; `/* garbage collecting */`
9. $proved.append(fid)$;
10. $lpvia \leftarrow (v_{ins}, q)$; `/* forming` $\mathtt{LP_{VIA}}$ `(c.f. Def.6) */`
11.   **broadcast** $\mathtt{T_{ins}}(lpvia)$ **to** N_l ; `/* c.f. Def.8 */`
12. **upon delivering** $D(\mathtt{T_{ins}}(lpvia))$ **:**
13.   **if** $lpvia.v_{ins}.fid \notin proved$ **then**
14.   **for** $v_{ins} \in \{v_{ins} \in proving \mid v_{ins}.fid = lpvia.v_{ins}.fid\}$ **do**
15. $proving.delete(v_{ins})$; `/* garbage collecting */`
16. $proved.append(lpvia.v_{ins}.fid)$;

In Algorithm 7, we detail the `collect_unknowns` procedure, available to `executors`, and which allow them to **(1)** collect input argument proposals for the unknowns of a function instance, **(2)** build assignments for these unknowns and **(3)** vote on the validity of such assignments via emitting $\mathtt{V_{ins}}$ votes. Notice that this procedure is called for a pair $fid \in \mathfrak{S}$ and $r_{fun} \in \mathtt{R_{fun}}$ characterizing a specific function instance and for a map "*knowns*" that gives the raw values of the known inputs.

Algorithm 7: Behavior of the `executors` w.r.t. the *coreset-layer*

```
1  Procedure collect_unknowns(fid ∈ 𝔖, r_fun = R_fun(_, (φ, ψ, K), _, _), knowns : ℕ ⇀ LAT_val):
2      dbg_assert!(fid is signature of r_fun);
3      U ← [1, |φ|] \ Dom(K) ;                                  /* indices of all unknown inputs */
4      dbg_assert!(U ≠ ∅) ;                        /* there must be at least one unknown to agree on */
5      dbg_assert!(Dom(knowns) ⊎ U = [1, |φ|]) ;   /* U gives indices of unknowns and knowns the values of known inputs */
6      valid : Set⟨R_in × 𝔖⟩ ← {} ;                  /* memorizes locally valid input argument proposals */
7      echoed : Set⟨V_ins⟩ ← {} ;                            /* memorizes already emitted V_ins votes */
8      has_proposed_assignment : 𝔹 ← ⊥ ;       /* wether or not the executor has proposed an assignment itself */
9      upon receiving r_in = R_in(fid, I, lps) with signature σ ∈ 𝔖 :   /* here the R_in (Def.9) involves the same fid */
10         if r_in is valid w.r.t. r_fun of signature fid then                          /* as per Def.11 */
11             valid ← valid ∪ {(r_in, σ)} ;                  /* add signed r_in to locally valid proposals */
12             if ¬has_proposed_assignment then
13                 if ν = find_assignment(valid, fid, r_fun, knowns) ≠ ∅ then             /* c.f. Alg.5 */
14                     v_ins ← V_ins(fid, ν) ;                                            /* c.f. Def.5 */
15                     if v_ins ∉ echoed then
16                         broadcast v_ins to N_p ∪ N_x;
17                         echoed ← echoed ∪ {v_ins};
18                     has_proposed_assignment ← ⊤;

19     upon receiving v_ins = V_ins(fid, ν) :                                             /* c.f. Def.5 */
20         if v_ins ∉ echoed then
21             if check_assignment(fid, r_fun, knowns, ν) = ⊤ then
22                 broadcast v_ins to N_p ∪ N_x;
23                 echoed ← echoed ∪ {v_ins};

24     upon delivering D(t = T_ins(lpvia)) with lpvia.v_ins.fid = fid :   /* procedure returns upon consensus */
25         return (▼(t), lpvia) ;                                         /* used at line 15 of Alg.10 */
```

9 Behavior at the *function-layer*

9.1 Overview

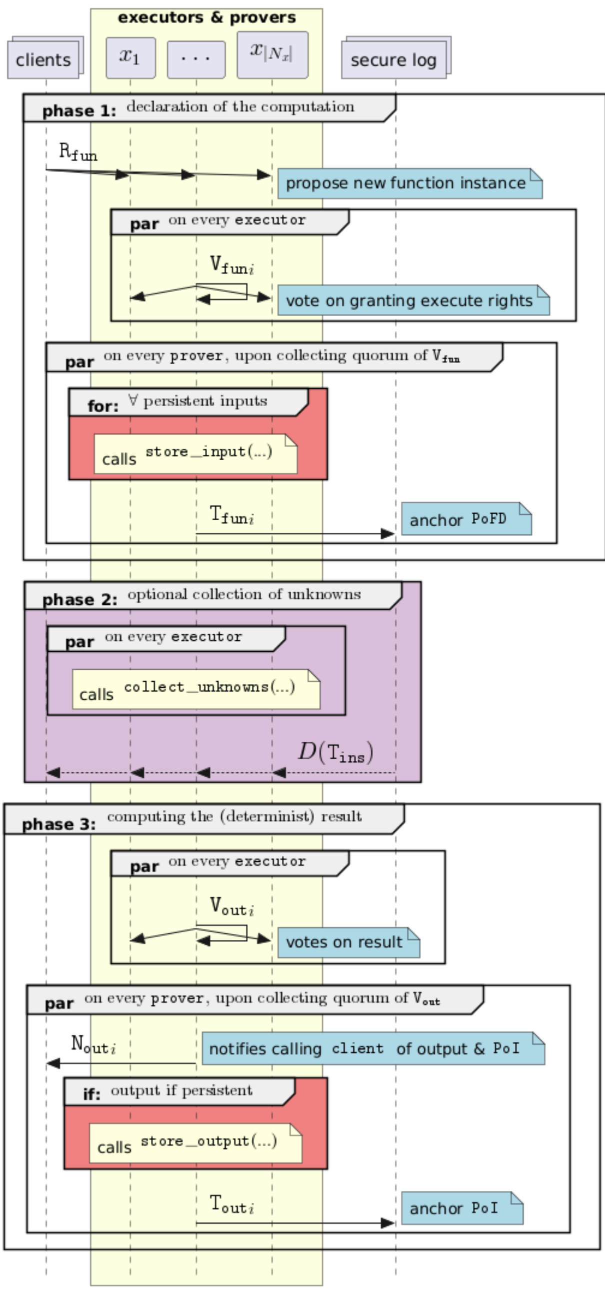


Figure 7: Description of the behavior at the *function-layer*

On Fig.7, we describe in more details (and focusing on the behavior of `executors` and `provers`) the lifecycle of a function instance, which we had already described on Fig.1 and Section 4.3. We can now provide some additional details, as we have introduced all data types as well as the procedures `store_input` and `store_output` in Algorithm 1 from Section 6.3 and the procedure `collect_unknowns` in Algorithm 7 from Section 8.3.

To request the execution of a new function instance, a `client` broadcasts a $\mathtt{R_{fun}}$ message (which contains its specification (ϕ, ψ)) to the `executors`.

Upon receiving the $\mathtt{R_{fun}}$, the `executors` decide on wether or not to grant execute rights to the function via broadcasting $\mathtt{V_{fun}}$ votes to the `provers`.

After having collected $f+1$ signatures of the same $\mathtt{V_{fun}}$ vote, whenever a `prover` produces a Proof of Function Declaration ($\mathtt{LP_{FD}}$) for the first time, it submits a $\mathtt{T_{fun}}$ transaction to the secure log to anchor the declaration (but delivery of the transaction is not required to progress further in the lifecycle and thus it lies outside the critical path of the lifecycle).

In the meanwhile, whenever a `executor` (co-located with a `prover`) learn about a $\mathtt{LP_{FD}}$ that was produced, if the function instance has persistent inputs to store, it calls the `store_input` procedure (Alg.1) accordingly.

Then, if (and only if) the function instance is multi-party (i.e., has unknown input arguments), the `executor` calls the `collect_unknowns` procedure (Alg.7), which only returns when agreement has been reached on the unknowns, upon delivery of a $D(\mathtt{T_{ins}})$ event directed by the secure log. When this procedure returns, the `executor` has the required information to retrieve the concrete values of all the unknown inputs.

Eventually, every honest `executor` acquire the information to retrieve reliably the concrete values of all known input arguments and all unknown arguments (if there are any). In the first case it is either via receiving the $\mathtt{R_{fun}}$ or any valid $\mathtt{LP_{FD}}$ or the delivered $D(\mathtt{T_{fun}})$. In the second, it can only be via the delivered $D(\mathtt{T_{ins}})$ (making `collect_unknowns` return).

As a result, an honest `executor` can compute the result of the function locally. Once done, it emits a $\mathtt{V_{out}}$ containing the digest of the obtained result.

After having collected $f + 1$ signatures of the same $\mathtt{V_{out}}$ vote, whenever a `prover` produces a Proof of Integrity ($\mathtt{LP_C}$) for the first time, it submits a $\mathtt{T_{out}}$ transaction to the secure log to anchor the result of the function. The `prover` can also notify the `client` that requested the execution of the function instance via sending to that `client` its local version of the $\mathtt{LP_C}$ (which might differ in its selected subset of signatures from the $\mathtt{LP_C}$ that will eventually be anchored on the secure log).

In the meanwhile, whenever a `executor` (co-located with a `prover`) learn about a $\mathtt{LP_C}$ that was produced, if the output of the function instance is flagged as persistent, it calls the `store_output` procedure (Alg.1).

9.2 Detailed behaviors of Ledgents per involved roles

Algorithm 8: Behavior of the `provers` w.r.t. the declaration phase of the *function-layer*

```
1   proving : Map⟨V_fun ↦ Set⟨𝔖⟩⟩ ← {}  ;   /* initialize map to store quorums being built
    proved  : Set⟨𝔖⟩              ← {}       and set to ignore fids for which a LP_FD has already been built */
2   upon receiving v_fun = V_fun(fid, K, U, pi) with signature σ ∈ 𝔖 :
3     if fid ∉ proved then                                  /* if not already produced a LP_FD for this fid */
4       proving[v_fun] ← proving[v_fun] ∪ {σ} ;             /* add signature to quorum being built */
5       if |proving[v_fun]| ≥ f + 1 then
6         q ← proving.pop(v_fun).select(f + 1);
7         for v'_fun ∈ {v'_fun ∈ proving | v'_fun.fid = fid} do
8           proving.delete(v'_fun);
9         proved.append(fid);
10        lpfd ← (v_fun, q);
11        broadcast T_fun(lpfd) to N_l;
12  upon delivering D(T_fun(lpfd)) :
13    if lpfd.v_fun.fid ∉ proved then
14      for v_fun ∈ {v_fun ∈ proving | v_fun.fid = lpfd.v_fun.fid} do
15        proving.delete(v_fun);
16      proved.append(lpfd.v_fun.fid);
```

In the *function-layer*, the behavior of `provers` is, as always, the same, as illustrated on Alg.8 and Alg.9 but with different votes, proofs and transactions.

As for the behavior of `executors`, it is described in Algorithm 10.

Algorithm 9: Behavior of the `provers` w.r.t. the computation phase of the *function-layer*

```
1  proving : Map⟨V_out ↦ Set⟨𝔖⟩⟩ ← {}
   proved  : Set⟨𝔖⟩             ← {} ;      /* initialize map to store quorums being built
                                                and set to ignore fids for which a LP_C has already been built */
2  upon receiving v_out = V_out(fid, uref, r, po) with signature σ ∈ 𝔖 :
3      if fid ∉ proved then                  /* if not already produced a LP_C for this fid */
4          proving[v_out] ← proving[v_out] ∪ {σ} ;      /* add signature to quorum being built */
5          if |proving[v_out]| ≥ f + 1 then
6              q ← proving.pop(v_out).select(f + 1);
7              for v'_out ∈ {v'_out ∈ proving | v'_out.fid = fid} do
8                  proving.delete(v'_out);
9              proved.append(fid);
10             lpc ← (v_out, q);
11             broadcast T_out(lpc) to N_l;
12 upon delivering D(T_fun(lpc)) :
13     if lpc.v_out.fid ∉ proved then
14         for v_out ∈ {v_out ∈ proving | v_out.fid = lpc.v_out.fid} do
15             proving.delete(v_out);
16         proved.append(lpc.v_out.fid);
```

Algorithm 10: Behavior of the **executors** w.r.t. the *function-layer*

```
1  specs      : Map⟨𝔖 ↦ (Spec × 𝔹 × Set⟨ℕ⟩)⟩ ← {}
   senders    : Map⟨𝔖 ↦ N⟩                  ← {}
   lpfds      : Map⟨𝔖 ↦ LP_FD⟩              ← {} ;
   outputs    : Map⟨𝔖 ↦ LAT_val⟩            ← {}
   terminated : Set⟨𝔖⟩                      ← {}
      /* pending function instances' specifications
         pending function instances' senders
         pending function instances' LP_FDs
         pending function instances' outputs
         terminated function instances */
2  upon receiving r_fun = R_fun(_, (φ, ψ, K), po, pi) with signature σ ∈ 𝔖 from n ∈ N for the first time :
3    if σ ∉ terminated ∧ r_fun is valid per Def.10 then
4      specs[σ] ← ((φ, ψ, K), po, pi) ;                         /* keeps track of specification */
5      senders[σ] ← n ;                                         /* keeps track of sender */
6      K' ← [i ↦ ▼(K(i)) if K(i) ∈ LAT_val else K(i).v_sto.h | i ∈ Dom(K)];   /* digests of all known inputs */
7      U ← [1, |φ|] \ Dom(K) ;                                  /* indices of all unknown inputs */
8      v_fun ← V_fun(σ, K', U, pi) ;                            /* c.f. Def.5 */
9      broadcast v_fun to N_p;
10     inputs : Map⟨ℕ ↦ LAT_val⟩ ← ∅ ;                          /* initialize inputs array */
11     for i ∈ Dom(K) do
12       inputs[i] ← unwrap_val(K(i)) ;                         /* fill-in raw value of known input, c.f. Alg.1 */
13     uref ← ∅;
14     if U ≠ ∅ then
15       (uref, lpvia) ← collect_unknowns(σ, r_fun, inputs) ;   /* c.f. Alg.7, awaits consensus at coreset-layer */
16       for i ∈ lpvia.v_ins.ν do                               /* c.f. Def.5 and Def.6 */
17         (_, _, lps) ← lpvia.v_ins.ν[i] ;                      /* c.f. Def.7 */
18         inputs[i] ← unwrap_val(lps) ;                        /* fill-in raw value of unknown input */
19     output ← φ(inputs[1], ··· , inputs[|φ|]) ;               /* computes output */
20     outputs[σ] ← output ;                                    /* keeps track of output */
21     v_out ← V_out(σ, uref, ▼(output), po) ;                  /* c.f. Def.5 */
22     broadcast v_out to N_p;
23 upon building lpfd ∈ LP_FD with lpfd.v_fun.fid ∈ 𝔖 for the first time :   /* e.g. on co-localized prover running Alg.8 */
24   once lpfd.v_fun.fid ∈ specs :
25     ((φ, ψ, K), po, pi) ← specs[lpfd.v_fun.fid];
26     for i ∈ pi do                                            /* stores inputs with the persistent input flag */
27       store_input(K(i), lpfd, i) ;                           /* c.f. Alg.1 */
28     lpfds[lpfd.v_fun.fid] ← lpfd;
29 upon building lpc ∈ LP_C with lpc.v_out.fid ∈ 𝔖 for the first time :      /* e.g. on co-localized prover running Alg.9 */
30   once lpc.v_out.fid ∈ lpfds ∧ lpc.v_out.fid ∈ outputs :
31     ((φ, ψ, K), po, pi) ← specs.remove(lpc.v_out.fid) ;      /* retrieve & garbage collects specification */
32     n ← senders.remove(lpc.v_out.fid) ;                      /* retrieve & garbage collects sender */
33     lpfd ← lpfds.remove(lpc.v_out.fid) ;                     /* retrieve & garbage collects Proof of Function Declaration */
34     output ← outputs.remove(lpc.v_out.fid) ;                 /* retrieve & garbage collects output */
35     n_out ← N_out(output, lpc) ;                             /* c.f. Def.14 */
36     broadcast n_out to {n};
37     if po then                                               /* stores output if it has persistent output flag */
38       store_output(output, lpfd, lpc) ;                      /* c.f. Alg.1 */
39     terminated ← terminated ∪ {lpc.v_out.fid} ;              /* tracks terminated functions' identifiers */
```

10 Properties upheld by Ledgera

10.1 Storage availability properties

In the following, we reformulate the behavior of `storers` from Section 6 as a "distributed storage" distributed object and then prove a number of properties it has under certain assumptions.

Definition 18. *The Ledgera distributed storage is a simple key-value store* $\faDatabase : (N_s \times \mathfrak{H}) \mapsto (LAT_{\text{val}} \cup \{\varnothing\})$ *such that for any* $n \in N_s$ *and any* $h \in \mathfrak{H}$*:*

- *if* $n \in N_s \setminus F$ *then* $\faDatabase[n, h]$ *corresponds to the value of* $local(h)$ *in the local variable "local" in* n *as per Alg.3*
- *if* $n \in F$ *then* $\faDatabase[n, h]$ *may take any value in* $LAT_{\text{val}} \cup \{\varnothing\}$

The fact that we do not have deletion and modification (because the key must be the hash of the value, it is not possible to overwrite a value) makes so that we have no risk of conflict on a key.

Property 6. *The Ledgera distributed storage always satisfies:*

- ***initialization****: at the genesis of the system, we have for any correct* `storer` $n \in N_s \setminus F$*, for any key* $h \in \mathfrak{H}$*,* $\faDatabase[n, h] = \varnothing$
- ***validity****: at any time, for any correct* `storer` $n \in N_s \setminus F$*, we have, for any key* $h \in \mathfrak{H}$*,* $\faDatabase[n, h] \in \{\faFilter^{-1}(h), \varnothing\}$
- ***immutability****: for any correct* `storer` $n \in N_s \setminus F$*, for any key* $h \in \mathfrak{H}$*, if, at a certain time, we have* $\faDatabase[n, h] = \faFilter^{-1}(h)$ *then, at any later time, we also have* $\faDatabase[n, h] = \faFilter^{-1}(h)$

Proof. For each point:

- **initialization** see Def.18 and line 1 of Alg.3
- **validity** see lines 3-8 of Alg.3
- **immutability** can only write on a key in line 6 or 8 of Alg.3 with same value, no deletion possible

□

Property 7. *Under Asmpt.1, the Ledgera distributed storage satisfies* ***storage-eventual-availability****: for any* $h \in \mathfrak{H}$*, if there exists a valid* $lps \in \texttt{LP}_\texttt{S}$ *with* $lps.v_{sto}.h = h$ *then, eventually, for all correct replicas* $n \in N_s \setminus F$ *we will have* $\faDatabase[n, h] = \faFilter^{-1}(h)$

Proof. If lps is valid, then $f + 1$ Ledgents have signed the associated $lps.v_{sto} = \texttt{V}_{\texttt{sto}}(fid, h, pk)$ and at least one of them is an honest and non-faulty `executor`.

An honest `executor` only emits $\texttt{V}_{\texttt{sto}}$ votes via either the `store_input` or the `store_output` procedure of Alg.1 (line 4 or 9 in Alg.1). As a result, it must have previously broadcast a corresponding $\texttt{R}_{\texttt{sto}}(\faFilter^{-1}(h), lpfd, kind)$ (at either line 3 or line 8 in Alg.1) with valid content to all the `storers` in N_s.

As per Asmpt.1 all the correct `storers` $n \in N_s \setminus F$ will eventually receive that $\texttt{R}_{\texttt{sto}}$, triggering lines 2-11 of Alg.3.

If the honest `storer` $n \in N_s \setminus F$ that receives the $\texttt{R}_{\texttt{sto}}$ message already has $h \in local$, then, by Prop.7 it can only be $\faFilter^{-1}(h)$ and thus $\faDatabase[n, h] = \faFilter^{-1}(h)$.

If on the other hand, it has $h \notin local$, then, because the $\texttt{R}_{\texttt{sto}}$ message has valid content (being sent by an honest `executor`), the conditions on line 4 and either 5 or 7 of Alg.3 are satisfied and the `storer` will insert $\faFilter^{-1}(h)$ in $local[h]$ at either line 6 or 8 of Alg.3, fulfilling $\faDatabase[n, h] = \faFilter^{-1}(h)$. □

Property 8. *Under Asmpt.1, the Ledgera distributed storage satisfies* ***storage-query-liveness****: for any* $h \in \mathfrak{H}$ *and any valid* $lps \in \texttt{LP}_\texttt{S}$ *with* $lps.v_{sto}.h = h$*, if any correct Ledgent* $n \in N \setminus F$ *calls* `query_storage`(h, lps) *of Alg.4 then, eventually, this call will return a value* $x = \faFilter^{-1}(h)$

Proof. As per Alg.4, the Ledgent n broadcast a $\texttt{Q}_{\texttt{sto}}(h, lps)$ (having signed it with its private key, yielding signature σ) to all the `storers` in N_s. Then, as per Asmpt.1, all correct `storers` will eventually receive that query. Given that there are at least $f + 1$ `storers`, one can consider a correct `storer` $n_s \in N_s \setminus F$.

Upon receiving the query, if the local copy of the storage on n_s has a value corresponding to h, as per ***validity*** of Prop.6, it can only be $\blacktriangledown^{-1}(h)$. As a result, the `storer` will send respond to the query (on lines 13-14 in Alg.3) with a $\mathtt{N_{sto}}(\sigma, \blacktriangledown^{-1}(h))$ notification.

Otherwise, the *lps* being valid and having $lps.v_{sto}.h = h$, as per lines 15-16 in Alg.3, n will not answer immediately, and keep track of the query in its set '*pending*' of pending queries.

On the other hand, the *lps* being valid, as per ***storage-eventual-availability*** (Prop.7), we also have that, eventually, n_s will include it in its local copy of the storage so that $\text{🛢}[n_s, h] = \blacktriangledown^{-1}(h)$. This can only be done via either line 6 or line 8 of Alg.3 being run upon receiving a certain $\mathtt{R_{sto}}(\blacktriangledown^{-1}(h), _, _, _)$. As a result, lines 9-11 will also be eventually executed and, at that point, $\text{🛢}[n_s, h] = local[h] = \blacktriangledown^{-1}(h)$ therefore the query will be answered on line 11 with a $\mathtt{N_{sto}}(\sigma, \blacktriangledown^{-1}(h))$ notification.

Going back to the `query_storage` procedure of Alg.4, we remark that our initial node n waits for $f+1$ $\mathtt{N_{sto}}(\sigma, v)$ responses. In the worst case, there are f Byzantine Ledgents that respond first with $v \neq \blacktriangledown^{-1}(h)$. And in any case, we are guaranteed to receive at least one response (among the $f+1$) that comes from an honest `storer`, which we have proven above will contain $v = \blacktriangledown^{-1}(h)$. As a result, line 7 of Alg.4 runs and the procedure returns the value $\blacktriangledown^{-1}(h)$. ☐

10.2 Function instances safety properties

In the following, we prove properties regarding the safety/correctness of the execution of function instances. With:

- the ***function-valid-declaration*** of Prop.9, we prove that the existence of a valid $\mathtt{LP_{FD}}$ implies the validity of the corresponding $\mathtt{R_{fun}}$ function instance request
- the ***function-valid-inputs-selection*** of Prop.10, we prove that the existence of a valid $\mathtt{LP_{VIA}}$ implies the validity of the assignment of unknowns it contains w.r.t. the corresponding $\mathtt{R_{fun}}$ function instance request
- the ***function-valid-output*** of Prop.11, we prove that the existence of a valid $\mathtt{LP_C}$ implies the validity of the output which digest it contains w.r.t. the corresponding $\mathtt{R_{fun}}$ function instance request

Property 9. *The execution of function instances in Ledgera satisfies* ***function-valid-declaration****: for any $r_{fun} \in \mathtt{R_{fun}}$ broadcast by a Ledgent with signature $\sigma \in \mathfrak{S}$, if there exists a valid lpfd $\in \mathtt{LP_{FD}}$ with $lpfd.v_{fun}.fid = \sigma$ then r_{fun} is valid (as per Def.10).*

Proof. The validity of *lpfd* implies $f+1$ Ledgents have signed $lpfd.v_{fun}$, among them at least one honest `executor`. Signing such a vote can only be done at line 9 of Alg.10. This can only be done in the context of receiving, at line 2 of Alg.10, a $\mathtt{R_{fun}}$ message with the same signature σ (which is found as an attribute of the $\mathtt{V_{fun}}$ vote). Having the same signature, this $\mathtt{R_{fun}}$ message can only be r_{fun}.

Reaching line 9 then requires having passed the validity check at line 3, which implies r_{fun} is valid as per Def.10. ☐

Property 10. *The execution of function instances in Ledgera satisfies* ***function-valid-inputs-selection****: for any $r_{fun} \in \mathtt{R_{fun}}$ broadcast by a Ledgent with signature $\sigma \in \mathfrak{S}$, if there exists a valid lpvia $\in \mathtt{LP_{VIA}}$ with $lpvia.v_{ins}.fid = \sigma$ then we have that $lpvia.v_{ins}.\nu$ is valid w.r.t. r_{fun} of signature σ (as per Def.17).*

Proof. Given *lpvia* is valid, at least one honest `executor` signed the vote $lpvia.v_{ins}$. This can only be done at either line 16 or line 22 of Alg.7 in the context of `executor` having called the procedure `colect_unknowns` on the same function identifier $fid = \sigma$, which requires it having knowledge of r_{fun} (as it can only be called with a r_{fun} that accepts σ as signature).

In that context, if the $lpvia.v_{ins}$ is emitted at line 16 of Alg.7, this means that it is the honest `executor` itself that has found the assignment on line 13 of Alg.7. This in turn implies, by lines 17-18 of Alg.5 and by Prop.4 that $lpvia.v_{ins}.\nu$ is valid w.r.t. r_{fun} of signature σ.

If the $lpvia.v_{ins}$ is emitted at line 22 of Alg.7, this means the `executor` has had knowledge of an assignment found by another Ledgent, has verified it and has echoed it on, respectively, lines 19, 21 and 22 of Alg.7. This verification implies, by Prop.4 that $lpvia.v_{ins}.\nu$ is valid w.r.t. r_{fun} of signature σ.

☐

In Definition 19 we formalize the validity of an output value w.r.t. a function instance as the fact that it must be the result of applying the function on the raw values of the known inputs and, if there are unknowns, on raw values assigned to them via a valid assignment.

Definition 19. *Let $r_{fun} = \mathtt{R_{fun}}(i, (\phi, \psi, K), po, pi)$ be a valid request (as per Def.10) identified by signature σ. We say*

a value $v \in LAT_{\text{val}}$ *is a valid output of* r_{fun} *of signature* σ *iff* $\exists\ \nu\ \in \texttt{InputRef}^{\mathbb{N}}$ *s.t.:*

$$\nu \text{ is valid w.r.t. } r_{fun} \text{ of signature } \sigma \qquad \text{/* as per Def.17 */}$$

$$\wedge \left(\begin{array}{l} \text{given } \forall i \in [1, |\phi|],\ v_i = \begin{cases} \textbf{if } i \in Dom(\nu) & \blacktriangledown^{-1}(\nu(i).lps.v_{sto}.h) \\ \textbf{elif } i \in Dom(K) \cap LAT_{\text{val}} & K(i) \\ \textbf{else} & \blacktriangledown^{-1}(K(i).v_{sto}.h) \end{cases} \\ \text{we have that } v = \phi(v_1, \cdots, v_{|\phi|}) \end{array} \right) \qquad \text{/* value is result of function applied on inputs */}$$

Property 11. *The execution of function instances in Ledgera satisfies* ***function-valid-output****: for any* $r_{fun} \in \texttt{R}_{\texttt{fun}}$ *broadcast by a Ledgent with signature* $\sigma \in \mathfrak{S}$*, if there exists a valid* $lpc \in \texttt{LP}_\texttt{C}$ *with* $lpc.v_{out}.fid = \sigma$ *then:*

(1) *we have* $\blacktriangledown^{-1}(lpc.v_{out}.r)$ *is a valid output of* r_{fun} *of signature* σ

(2) *and there cannot exist any* $lpc' \in \texttt{LP}_\texttt{C}$ *with* $lpc'.v_{out}.fid = \sigma$ *and* $lpc'.v_{out}.r \neq lpc.v_{out}.r$

Proof.

■ Proof of **(1)**:
The *lpc* being valid, at least one honest `executor` has signed the $lpc.v_{out}$ vote, which can only be done at line 22 of Alg.10. This can only be done in a context triggered by having received on line 2 a $\texttt{R}_{\texttt{fun}}$ message with the same signature σ (which is found as an attribute of the $\texttt{V}_{\texttt{out}}$ vote). Having the same signature, this $\texttt{R}_{\texttt{fun}}$ message can only be r_{fun}.

The digest $lpc.v_{out}.r$ must therefore correspond to the digest of the "*output*" variable computed at line 19 i.e., we have $\blacktriangledown^{-1}(lpc.v_{out}.r) = output$. And this "*output*" is a valid output of the function instance as per Def.19. Indeed:

- If there are no unknowns (i.e. $Dom(K) = [1, |\phi|]$), the *output* results from applying ϕ to the known inputs which raw values we retrieve at lines 11-12. This corresponds in Def.19 to the case of the empty assignment ν with $Dom(\nu) = \emptyset$.
- If there are unknowns, then the raw values for the unkowns are retrieved from the $\texttt{LP}_{\texttt{VIA}}$ resulting from the call to `collect_unknowns` at line 15. This procedure can only return on line 25 of Alg.7, following the occurrence of a delivery event $D(\texttt{T}_{\texttt{ins}}(lpvia))$ for a *lpvia* s.t. $lpvia.v_{ins}.fid = \sigma$. As per the properties of the secure log (Prop.1 and the notion of valid history of anchored transaction in Def.15) the *lpvia* it contains must be valid. Then, as per Prop.10, we must have $lpvia.v_{ins}.\nu$ is valid w.r.t. r_{fun} of signature σ and this is this ν that we can use as witness in Def.19.

■ Proof of **(2)**:
Following the reasoning above, it is impossible for two distinct honest `executors` to compute different values for the "*output*" at line 19 of Alg.10. Indeed, they must have the exact same context, including the same definition of ϕ, K (from line 2) and, if applicable, the same value for *lpvia* resulting from calling `collect_unknowns` at line 15. Indeed, as per the properties of the secure log (Prop.1, Def.15) there can only be one such event and it is the same accross all honest Ledgents. As a result, any two honest `executor` builds the exact same "*inputs*" variable on lines 10, 11-12 and 15-18 of Alg.10 and because ϕ is deterministic, they obtain the same "*output*" at line 19 and therefore they cannot produce votes v_{out} and v'_{out} with $v'_{out}.fid = v_{out}.fid$ and $v'_{out}.r \neq v_{out}.r$. □

10.3 Function instances liveness properties

In the following, we prove properties regarding the liveness of the execution of function instances. More specifically, these property signify that collaboration between honest `clients` to execute a function is guaranteed to eventually succeed. With:

- the ***function-declaration-liveness*** of Prop.12, we prove that, if a valid $\texttt{R}_{\texttt{fun}}$ request is broadcast by an honest `client`, then a corresponding valid $\texttt{LP}_{\texttt{FD}}$ will eventually be produced.
- the ***function-inputs-agreement-liveness*** of Prop.13, we prove that, if a valid $r_{fun} \in \texttt{R}_{\texttt{fun}}$ request which function specification has unknown inputs, as well as enough $\texttt{R}_{\texttt{in}}$ requests to produce a valid assignment for that r_{fun} are broadcast by honest `clients`, then a valid $\texttt{LP}_{\texttt{FD}}$ will eventually be produced for that function instance (which may carry any valid assignment).
- the ***function-execution-liveness*** of Prop.14, we prove that, if a valid $\texttt{R}_{\texttt{fun}}$ request is broadcast by an honest `client`, if it has no unknwon inputs, or if we are in the conditions above, then, a corresponding valid $\texttt{LP}_\texttt{C}$ will eventually be produced.

Property 12. *Under Asmpt.1, the execution of function instances in Ledgera satisfies* ***function-declaration-liveness:*** *If an honest* `client` *broadcasts a valid* $r_{fun} \in \texttt{R}_{\texttt{fun}}$ *request with signature* $\sigma \in \mathfrak{S}$*, then, eventually, a valid* $lpfd \in \texttt{LP}_{\texttt{FD}}$

with $lpfd.v_{fun}.fid = \sigma$ is produced at an honest `prover`*.*

Proof. The `client` broadcasting r_{fun} of signature σ being honest, as per Asmpt.1, at least $f+1$ honest `executors` will eventually receive it. Upon reception, this triggers the logic at lines 2-22 of Alg.10. The r_{fun} request being valid, the check at line 3 is passed and the same $v_{fun} = \mathtt{V_{fun}}(\sigma, _, _, _)$ is broadcast by the $f+1$ honest `executors` (with different signatures) at line 9.

As per Asmpt.1, at least one honest `prover` is guaranteed to eventually receive all of these votes and their signatures. Each reception of a vote triggers the logic at lines 2-11 of Alg.8. The check at line 5 is eventually passed and a $lpfd \in \mathtt{LP_{FD}}$ s.t. $lpfd.v_{fun}.fid = \sigma$ is produced at line 10. □

Property 13. *Under Asmpt.1, the execution of function instances in Ledgera satisfies* ***function-inputs-agreement-liveness****:*
Let us suppose an honest `client` *broadcasts a valid $r_{fun} = \mathtt{R_{fun}}(_, (\phi, _, K), _, _)$ request such that $U = [1, |\phi|] \setminus Dom(K) \neq \emptyset$ with signature $\sigma \in \mathfrak{S}$.*
Let us also suppose that honest `clients` *broadcast $|U|$ valid $\mathtt{R_{in}}$ requests, that we denote by $\forall i \in U,\ r^i_{in}$ of signatures σ_i.*
Let us then consider the assignment $\nu \in \mathtt{InputRef}^U$ s.t. $\forall i \in U,\ \nu(i) = (\sigma_i, r^i_{in}.I, , r^i_{in}.lps)$.
If ν is valid w.r.t. r_{fun} of signature σ, then, eventually, a valid $lpvia \in \mathtt{LP_{VIA}}$ with $lpvia.v_{ins}.fid = \sigma$ is produced at an honest `prover`*.*

Proof. The `client` broadcasting r_{fun} of signature σ being honest, as per Asmpt.1, at least $f+1$ honest `executors` will eventually receive it. Upon reception, this triggers the logic at lines 2-22 of Alg.10. The r_{fun} request being valid, the check at line 3 is passed and as $U \neq \emptyset$, line 15 is reached and the `collect_unknowns` procedure is called. This triggers the logic in Alg.7.

For all $i \in U$, the r^i_{in} of signature σ_i being emitted by an honest `client`, as per Asmpt.1, it is guaranteed to be eventually received by at least $f+1$ honest `executors`.

Upon reception on an honest `executor`, as long as lines 24-25 are not executed, the reception of the r^i_{in} request of signature σ_i triggers the logic at lines 9-18 of Alg.7. Because the r^i_{in} request is valid, (r^i_{in}, σ_i) is added to "*valid*" at line 11.

As a result, as long as lines 24-25 are not yet executed (and because the "*has_proposed_assignment*" is initialized to $\perp$ at line 8), we are guaranteed that, eventually, "*valid*" will contain enough $\mathtt{R_{in}}$ message so that the call of the `find_assignment` at line 13 returns a valid assignment as per Prop.5.

As a consequence, the honest `executor` will eventually broadcast a $v_{ins} = \mathtt{V_{ins}}(\sigma, \nu)$ with an assignment ν that is valid w.r.t. r_{fun} of signature σ at line 16 (if it has not already received that same vote from another `executor` and echoed it at line 22).

In any case, it is guaranteed that an honest `executor` will eventually broadcast a valid $v_{ins} = \mathtt{V_{ins}}(\sigma, \nu)$ vote to $N_x \cup N_p$. Because it is honest, as per Asmpt.1, all f other honest `executors` will eventually receive that v_{ins} vote, which will trigger the logic at lines 19-23 of Alg.7. On any such `executor`, if it has not itself already broadcast that same v_{ins}, it will do so at line 22.

Therefore, we are guaranteed that, for a given valid assignment ν, at least $f+1$ honest `executors` will eventually broadcast $\mathtt{V_{ins}}(\sigma, \nu)$ to $N_x \cup N_p$ (on either line 16 or line 22 of Alg.7).

As per Asmpt.1, an honest `prover` will eventually receive these $f+1$ votes and use them to produce a valid $lpvia \in \mathtt{LP_{VIA}}$ at line 10 of Alg.6.

□

Property 14. *Under Asmpt.1, the execution of function instances in Ledgera satisfies* ***function-execution-liveness:***
If an honest `client` *broadcasts a valid $r_{fun} = \mathtt{R_{fun}}(_, (\phi, _, K), _, _)$ request, given $U = [1, |\phi|] \setminus Dom(K)$:*

- *if $U = \emptyset$, then, eventually, a valid $lpc \in \mathtt{LP_C}$ with $lpc.v_{out}.fid = \sigma$ is produced at an honest* `prover`*.*
- *if $U \neq \emptyset$ then, if honest* `clients` *broadcast $|U|$ valid $\mathtt{R_{in}}$ requests - denoted by $\forall i \in U,\ r^i_{in}$ of signatures σ_i - such that the assignment $\nu \in \mathtt{InputRef}^U$ s.t. $\forall i \in U,\ \nu(i) = (\sigma_i, r^i_{in}.I, , r^i_{in}.lps)$ is valid w.r.t. r_{fun} of signature σ, then, eventually, a valid $lpc \in \mathtt{LP_C}$ with $lpc.v_{out}.fid = \sigma$ is produced at an honest* `prover`*.*

Proof. The `client` broadcasting r_{fun} of signature σ being honest, as per Asmpt.1, at least $f+1$ honest `executors` will eventually receive it. Upon reception, this triggers the logic at lines 2-22 of Alg.10. The r_{fun} request being valid, the check at line 3 is passed.

For all known inputs, their raw value is inserted at the correct argument index of the "*inputs*" variable at lines 10-12. This yields the exact same "*inputs*" variable on all honest `executors` because **(1)** using σ as an identifier for r_{fun} makes so that all the honest `executors` agree on the definition of K, **(2)** for all known inputs that are references to storage, they take the form of valid $\mathtt{LP_S}$ (r_{fun} being valid) and the `unwrap_val` (c.f. Alg.4) calls `query_storage` which, as per Prop.7 is guaranteed to eventually return the same raw value.

- if $U = \emptyset$, the code at lines 14-18 is ignored and an honest `executor` computes the output of the function instance at line 19. It then broadcasts a $v_{out} = \mathtt{V_{out}}(\sigma, \varnothing, \blacktriangledown(output), _)$ at line 22. The function being deterministic, and applied on the same *inputs*, all $f+1$ honest `executors` compute the same value and emit the same vote.

- if $U \neq \emptyset$ and under the additional conditions of honest `clients` broadcasting the required $\mathtt{R_{in}}$ requests, we can follow the same reasoning as in the proof of Prop.10. This results in the `collect_unknowns` procedure returning, on all honest `executors` with the same $(uref, lpvia)$ result at line 15 of Alg.10. This results in the "*inputs*" variable being filled in the same way (at lines 16-18) and then the same $v_{out} = \mathtt{V_{out}}(\sigma, uref, \blacktriangledown(output), _)$ being broadcast at line 22.

In any case, as per Asmpt.1, an honest `prover` eventually gather $f+1$ signatures of the v_{out} vote and produce a valid $pi \in \mathtt{LP_C}$ at line 10 of Alg.9. □

11 Complexity

We analyze the communication complexity of a single function-instance lifecycle, distinguishing the *critical path* (the sequence of steps a `client` must wait through before receiving its $\mathtt{N_{out}}$ notification) from steps that proceed off the critical path.

11.1 Single-party function instances

For a single-party function instance ($U = \emptyset$, no unknown inputs), the critical path consists of three sequential communication rounds:

1. **Declaration.** The `client` broadcasts a $\mathtt{R_{fun}}$ message to the $2f+1$ `executors`.

2. **Voting.** Each `executor`, upon receiving the $\mathtt{R_{fun}}$ message, immediately computes the output and concurrently broadcasts a $\mathtt{V_{fun}}$ vote and a $\mathtt{V_{out}}$ vote to the `provers` (c.f. Alg. 10).

3. **Notification.** A `prover` assembles $f+1$ signatures of $\mathtt{V_{fun}}$ messages into a $\mathtt{LP_{FD}}$ and $f+1$ signatures of $\mathtt{V_{out}}$ messages into a $\mathtt{LP_C}$; once both are available, it sends $\mathtt{N_{out}}$ to the `client` (c.f. Alg. 10, 8, 9).

Each round involves $O(|N|)$ messages. Critically, **BFT consensus is not in the critical path**: the $\mathtt{T_{fun}}$ and $\mathtt{T_{out}}$ transactions are submitted to the `loggers` and anchored via BFT consensus asynchronously, independently of the $\mathtt{N_{out}}$ notification already being delivered to the `client` (c.f. Sec. 7 and the explicit remark in Sec. 9).

11.2 Multi-party function instances

For a multi-party function instance ($U \neq \emptyset$), the `executor` calls `collect_unknowns` (Alg. 7), which blocks until the secure log delivers a $D(\mathtt{T_{ins}})$ event (c.f. Sec. 8). This places **one full instance of BFT consensus in the critical path**, in addition to the rounds above.

The critical path is extended between steps 1 and 2 as follows:

(1') **Proposal collection.** `clients` broadcast $\mathtt{R_{in}}$ proposals for the unknown inputs to `executors` (this may overlap with step 1).

(2') **Assignment voting.** Once an `executor` has accumulated enough locally valid $\mathtt{R_{in}}$ proposals to satisfy ψ, it broadcasts a $\mathtt{V_{ins}}$ vote to `provers` and `executors`.

(3') **Consensus on assignment.** A `prover` assembles $f+1$ signatures of $\mathtt{V_{ins}}$ messages into a $\mathtt{LP_{VIA}}$ and submits a $\mathtt{T_{ins}}$ transaction to the `loggers`. The `loggers` run BFT consensus [10] (c.f. Remark 1) and deliver $D(\mathtt{T_{ins}})$ to all correct nodes, at which point `collect_unknowns` returns and execution proceeds.

As in the single-party case, the anchoring of $\mathtt{T_{fun}}$ and $\mathtt{T_{out}}$ remains off the critical path.

12 Deployment & Hello World example

In Fig.8, we recall the deployment scheme illustrated in Fig.1 but with some more details and focusing on the application layer built on top of Ledgera.

On Fig.8, Ledgera is composed of the 5 Ledgents represented by circles. Among these Ledgents, 3 of them play the `client` role, and each one is co-located with (i.e., runs on the same machine that) the process that runs application-specific code (represented by diamonds).

Users interact with the application code to make higher-level application-specific requests (represented as green dashed arrows on Fig.8). The application code may apply these requests by **(1)** translating them into a number of lower-level Ledgera requests ($\mathtt{R_{fun}}$, $\mathtt{R_{in}}$, $\mathtt{Q_{sto}}$) and submitting them to Ledgera, and awaiting responses (if request call is synchronous)

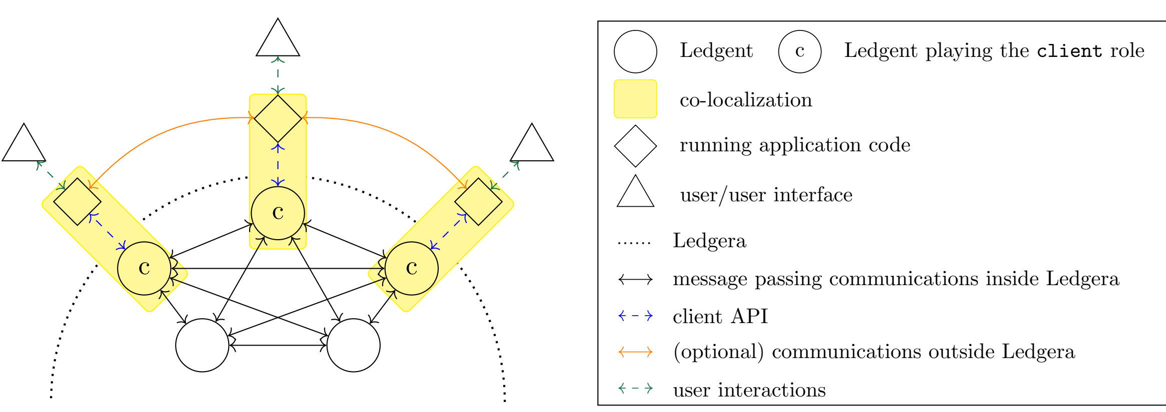


Figure 8: Ledgera user-facing applications

or not (if request call is asynchronous) (represented as blue dashed arrows on Fig.8), and/or **(2)** partaking in additional communications outside Ledgera (represented as orange arrows on Fig.8).

In the following, we present a basic example of a distributed application `app_string_concat` built on top of Ledgera that allows one to create and concatenate strings that may be securely stored.

12.1 Requirements of the application code

Ledgera Application Template. For our string concatenation application, the LAT is such that:

- $LAT_{\text{val}} = \texttt{string}$
- $LAT_{\text{opn}} = \{\texttt{concat}\}$ where $\texttt{concat} : \texttt{string} \times \texttt{string} \mapsto \texttt{string}$ is the concatenation of two strings
- and $LAT_{\text{prd}} = \Big(\{\top\} \cup \bigcup_{length \in \mathbb{N}} \{|x_1| > length\}\Big) \wedge \Big(\{\top\} \cup \bigcup_{length \in \mathbb{N}} \{|x_2| > length\}\Big) \wedge (\{\top,\ x_1 \neq x_2\})$

This template allows us to define function specifications such as, for instance:

- the concatenation of the string "*apple*" with whatever string is stored in storage at the digest given in the valid $pos \in \texttt{LP}_\texttt{S}$, corresponding to $(\texttt{concat}, \top, [1 \mapsto \text{"}apple\text{"}, 2 \mapsto pos])$ (which is single-party)
- the concatenation of the string "*apple*" with whatever string of length greater than 2, corresponding to $(\texttt{concat}, |x_2| > length, [1 \mapsto \text{"}apple\text{"}])$ (which is multi-party)
- the concatenation of the string "*apple*" with whatever string of length greater than 2 distinct from "*apple*", corresponding to $(\texttt{concat}, (|x_2| > length) \wedge (x_1 \neq x_2), [1 \mapsto \text{"}apple\text{"}])$ (which is multi-party)

Monikers. Whenever one wants to refer to a function instance (e.g., in order to propose an argument to it via broadcasting a $\texttt{R}_\texttt{in}$ message), one must use its 64 bytes identifier $fid \in \mathfrak{S}$. To avoid human users having to remember and write 128 character hexadecimal strings, we propose that the application code maintains a simple dictionary of shorthand names (called "monikers") $\texttt{dict}_\texttt{fun} : \texttt{string} \mapsto \mathfrak{S}$. Whenever the underlying `client` gains knowledge of a new function instance (via a valid $\texttt{LP}_\texttt{FD}$), it creates a new moniker (e.g., $c1, c2, \cdots$) and completes its $\texttt{dict}_\texttt{fun}$ accordingly.

Similarly, a user might want to refer to a value that is stored on the distributed storage at digest $h \in \mathfrak{H}$ for which it knowns at least one valid $pos \in \texttt{LP}_\texttt{S}$ such that $pos.v_{sto}.h = h$. Instead of using the 64 characters long hexadecimal representation of h, we allow the user to rather use a moniker provided by $\texttt{dict}_\texttt{val} : \texttt{string} \mapsto \mathfrak{H}$. Whenever the underlying `client` gains knowledge of a new stored value (via a valid $\texttt{LP}_\texttt{S}$), it creates a new moniker (e.g., $d1, d2, \cdots$) and completes its $\texttt{dict}_\texttt{val}$ accordingly.

Representation of `client` knowledge. We propose to represent the Ledgera knowledge that the `client` has (and so has the application code co-located with it) as follows:

- for each value stored on the distributed storage, we may hold the following information:
  - The value's digest.
  - The various valid proofs $\texttt{LP}_\texttt{S}$ that guarantee storage of the value as either an input or an output of a function instance.
  - Its raw value.
- for each function instance that is declared, we may hold the following information:

- The function's identifier and a valid $\texttt{LP}_{\texttt{FD}}$.
- The function's specification.
- The $\texttt{LP}_{\texttt{S}}s$ for the function's persistent inputs.
- If the function is multi-party, a $\texttt{LP}_{\texttt{VIA}}$ delivered on the secure log that attests agreement on its unknown inputs assignment.
- A $\texttt{LP}_{\texttt{C}}$ that attests agreement on the function's output value.
- The raw value of the function's output.
- If the output is persistent, a corresponding $\texttt{LP}_{\texttt{S}}$.

Application Code. For the `app_string_concat` application, we implement the following:

- The application maintains local monikers for function instances (via a $\texttt{dict}_{\texttt{fun}}$) and stored values (via a $\texttt{dict}_{\texttt{val}}$). These monikers may differ across distinct `clients` due to different orders with which they may learn about the existence of $\texttt{LP}_{\texttt{FD}}s$ and $\texttt{LP}_{\texttt{C}}s$.
- The application provides a Text User Interface from which the user might easily:
  - browse the knowledge the underlying `client` has of Ledgera, and
  - submit $\texttt{R}_{\texttt{fun}}$ and $\texttt{R}_{\texttt{in}}$ requests, built from parsing user-friendly commands that use the monikers.

12.2 The application code at work

Let us consider the example deployment represented in Fig.9. The users interact with local instances of the Text User Interface.

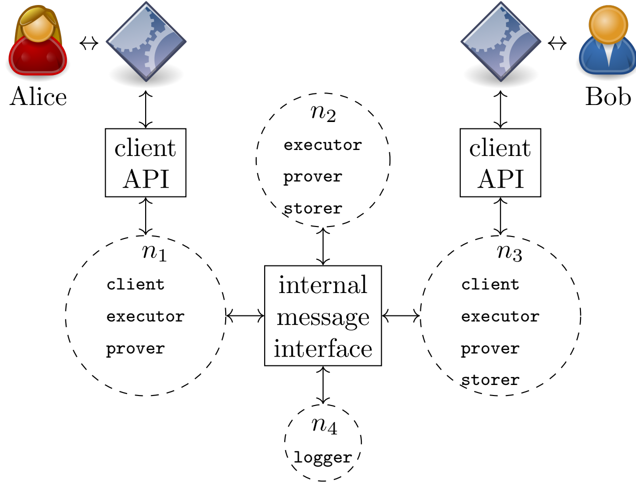


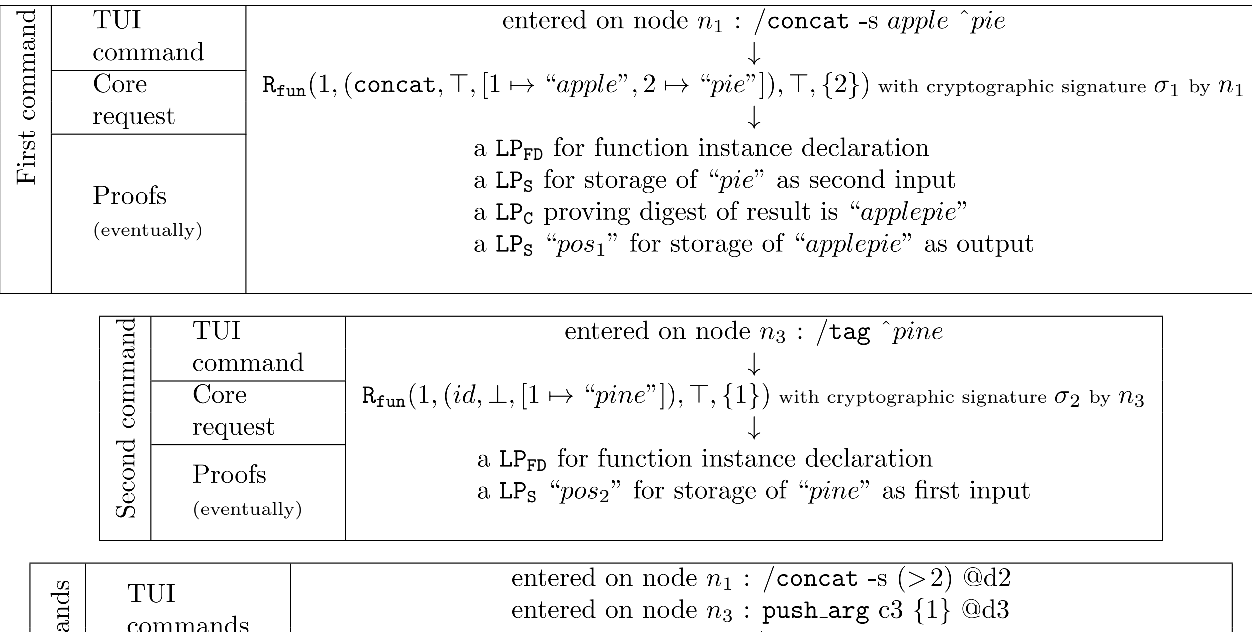
Figure 9: Deployment example for the string concatenation application

The scenario of utilization represented on Fig.10 describes the three commands that we detail below.

First command. Let's suppose that Alice (the red user on the left on Fig.9) enters the command "`/concat` -s *apple ˆpie*" (first command on the top of Fig.10) on the TUI co-localized with Ledgent n_1. This command is translated into a valid $\texttt{R}_{\texttt{fun}}$ request, cryptographically signed by node n_1 (yielding signature σ_1) and broadcast within the network of Ledgents.

Ledgera assigns nonce 1 to this first $\texttt{R}_{\texttt{fun}}$ sent by n_1, (first argument of the $\texttt{R}_{\texttt{fun}}$ constructor). The function to execute is "`concat`" (see second argument of the $\texttt{R}_{\texttt{fun}}$ constructor, which is the function specification within parenthesis). The two inputs "*apple ˆpie*" of the TUI command are translated into two known concrete raw string values : "*apple*" and "*pie*" in the function specification $[1 \mapsto \text{"}apple\text{"}, 2 \mapsto \text{"}pie\text{"}]$. As we did not specify any constraints on our arguments ("*apple*" and "*pie*", which are known in advance), the predicate inside the function specification is $\top$ (constraint is always satisfied). The "-s" flag in the TUI command determines output persistence, which corresponds to the $\top$ third argument of the $\texttt{R}_{\texttt{fun}}$ constructor. The second input "*pie*" is flagged as persistent (via the "ˆ" symbol) in the TUI command, which corresponds to the fourth argument $\{2\}$ of the $\texttt{R}_{\texttt{fun}}$ constructor. Here "*apple*" will not be stored in the secure storage while "*pie*" will be.

Since the $\texttt{R}_{\texttt{fun}}$ request is valid, once the request is signed by n_1 (yielding signature σ_1) and broadcast to the Ledgents, we eventually will have a number of Ledgera Proofs anchored as transactions in the Secure Log (by the liveness properties of Sec.10.3). These proofs are given on Fig.10 in the order with which they are expected to appear on the secure log (c.f. notion of valid history of Def.15). We enumerate these proofs below:



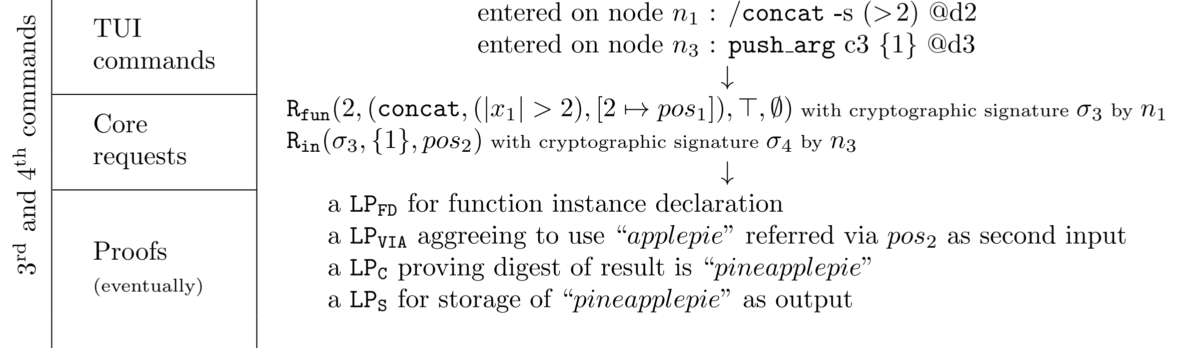


Figure 10: Commands details

(1) The $\mathtt{LP_{FD}}$ proves that at least one honest Ledgent has associated signature σ_1 to the function instance specification described in the corresponding $\mathtt{R_{fun}}$ request.

(2) The $\mathtt{LP_S}$ proves that the value “*pie*” is reliably stored in the Secure Storage (and, we keep track, in the Secure Log, that it is the second input of the function instance identified by σ_1).

(3) The third will be a $\mathtt{LP_C}$ proving that at least one honest Ledgent has locally performed the computation concatenating “*apple*” to “*pie*” and has obtained the expected “*applepie*” as the result of the function instance identified by σ_1.

(4) The $\mathtt{LP_S}$ proves that the value “*applepie*” is reliably stored in the Secure Storage (and, we keep track, in the Secure Log, that it is the output of the function instance identified by σ_1).

Second command. Now suppose Bob (the blue user on the right on Fig.9) enters the command “/tag ˆ*pine*” (second command in the middle of Fig.10) on the TUI co-localized with Ledgent n_3. Here the /tag operation corresponds to an identity function (which does not compute anything). We use it here simply to store the “*pine*” value, which is marked as a persistent input. The commands proceeds similarly to the first command above, the only difference being that the function specification is for the identity function, and that the output is not marked as persistent.

Third and Fourth commands. Suppose that now Alice declares another concatenation function instance. This function takes as second argument the result “*applepie*” of the previous concatenation function instance, via a reference to the storage provided by the $\mathtt{LP_S}$ proof pos_1 previously produced. The command specifies that the first input is an unknown with the only requirement that is must correspond to a string that is longer than 2 characters. This makes this function instance multi-party, and thus the function will not immediately return, but instead it will wait for the missing input to be filled-in.

Now Bob proposes the value “*pine*” that was previously stored (via the tag operation) as the first argument. As illustrated on Fig.10, this proposal corresponds to the $\mathtt{R_{in}}(\sigma_3, \{1\}, pos_2)$ broadcast by n_1. Because this is a valid proposal, the function instance will eventually be executed and produce a number of Ledgera Proofs (as implied by the liveness properties of Sec.10.3). Among these proofs, an $\mathtt{LP_{VIA}}$ is produced that guarantees agreement on using “*pine*” as a first argument. Also, an $\mathtt{LP_C}$ is produced that guarantees the integrity of the “*pineapplepie*” output returned by the function.

Anchored objects. Ledgera anchors objects in both its Secure Storage and Secure Log. Fig.11 below provides a graphical representation of the various objects that have been anchored in our scenario.

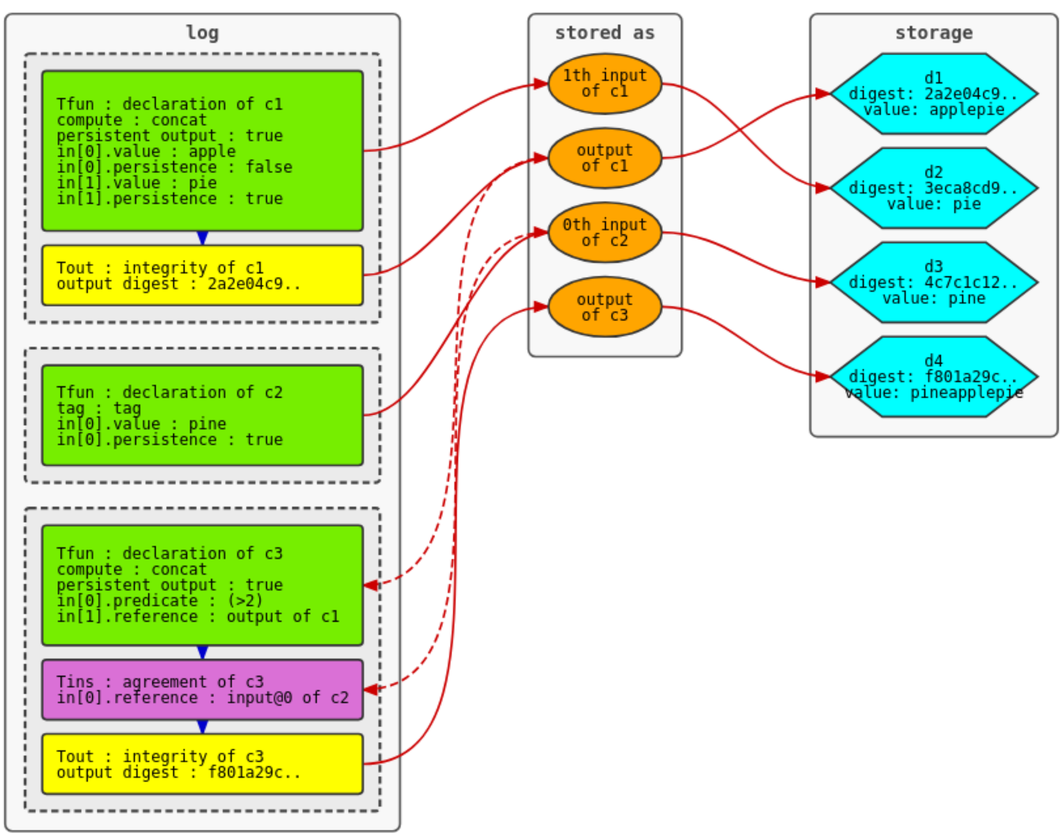


Figure 11: Dependencies between objects anchored (both in Secure Storage and Secure Log) in the scenario on Fig.10

13 Discussion

13.1 Synchrony model and BFT consensus

As noted in Remark 1, Ledgera's liveness relies on a partially synchronous network model [10]: eventual message delivery is the key assumption that would make BFT consensus terminate in a potential implementation of a BFT tolerant secure log that we could plug into Ledgera Core. Ledgera deliberately leaves the concrete BFT protocol unspecified, treating the secure log as a black box satisfying Props. 1–3. Similarly to Hyperledger Fabric's ordering service [2], which abstracts over the underlying consensus protocol and allows it to be swapped out depending on the deployment's trust and performance requirements, Ledgera does not prescribe a specific consensus algorithm for the secure log.

13.2 BFT storage

BFT storage has been studied extensively, with a recurring theme of separating the data plane from the control plane to reduce replication cost. MDStore [6] reduces data replicas from $3t + 1$ to $2t + 1$ by delegating write authorization to a separate metadata service, while remaining fully asynchronous. Similarly, Farsite [1] separates file data (replicated without BFT consensus) from directory metadata (maintained by BFT state-machine replication), using certificates to authorize data writes. PoWerStore [9] introduces *Proofs of Writing* (PoW), which are certificates that attest that a write has been completed, enabling efficient strongly-consistent storage; this concept is directly analogous to Ledgera's $\mathtt{LP}_\mathtt{S}$ (Proof of Shipment to Storage, c.f. Prop. 7). The works by Padilha and Pedone [14] address throughput scalability by partitioning the storage and running independent BFT instances per partition, which are all of them coordinated by an atomic commit protocol.

Ledgera's storage layer adopts the data/control separation principle but with a distinct authorization model: write access is guarded by computation proofs ($\mathtt{LP}_\mathtt{FD}$ for inputs, $\mathtt{LP}_\mathtt{FD} + \mathtt{LP}_\mathtt{C}$ for outputs), tying a write directly to the computation lifecycle of a function instance, rather than to a dedicated metadata service (c.f. Alg. 1). Addressing-by-content—i.e. each stored value has its hash as (a suffix of) the key—makes writes idempotent and reduces the required `storer` pool to $f + 1$ replicas. Storage replication is thereby kept off the critical path of the function-instance lifecycle (c.f. Sec. 11). Confidentiality of stored

data, as studied in Belisarius [13] via Shamir secret sharing across $3f+1$ servers, is not addressed in Ledgera `V_0_2` and is left to future versions of this document.

13.3 Non-determinism

Ledgera `V_0_2` restricts verifiable function instances to deterministic functions. This is the standard assumption underlying threshold-signature certification: all honest `executors` must independently produce the same result from the same inputs for the $f+1$ quorum to be meaningful.

Cachin, Schubert and Vukolić [7] provide a general treatment of non-determinism in BFT replication, distinguishing three models: *modular* (the application is a black box), *master-slave* (a designated leader resolves non-deterministic choices), and *cryptographically secure* (for operations requiring strong randomness, handled via verifiable random functions). In their protocol, called *Sieve*, replicas execute an operation speculatively and independently, and they exclude outlying results to limit divergence.

Extending Ledgera to support non-deterministic verifiable functions is a natural direction for future versions; the appropriate mechanism would depend on the source of non-determinism (occasional divergence, randomized algorithms, or cryptographic operations).

13.4 Membership and reconfiguration

Ledgera `V_0_2` assumes a static set N of Ledgents. Handling permanent addition or removal of Ledgents, and the associated view-change protocol for the underlying BFT consensus, is a notoriously complex problem and is left to future versions.

13.5 Single fault threshold

As noted in Sec. 3, Ledgera `V_0_2` uses a single Byzantine threshold f across all roles. In practice, different roles may have different trust profiles: `storers` and `loggers` may belong to distinct administrative domains with different failure rates. Generalising the fault model to role-specific thresholds is an acknowledged simplification left to future versions.

13.6 Applications

The design of Ledgera was originally motivated by the tool Fantastyc for decentralized federated learning [5].

Blockchain-based federated learning systems have a key scalability bottleneck: storing large model data and running aggregation directly on-chain is prohibitively expensive. Fantastyc addresses this bottleneck by separating concerns: a BFT set of servers performs robust aggregation and produces validity proofs; model updates and aggregated models are stored in an off-chain fault-tolerant key-value store; and the blockchain anchors only cryptographic hashes and proofs, keeping blockchain overhead independent of model size. Ledgera generalises this architecture from federated learning aggregation to arbitrary verifiable computations, formalising the three-layer structure (storage, logging, function execution) that Fantastyc instantiates for FL.

Federated learning. Fantastyc's servers perform Byzantine-robust aggregation to filter out poisoned client updates [11]. In Ledgera, each FL training round maps directly to a multi-party verifiable function instance: clients submit their model updates as unknown inputs via $\texttt{R}_{\texttt{in}}$ proposals, the coreset-layer certifies agreement on which updates are included, and `executors` run the aggregation function. The resulting proof and the updated model are anchored and stored exactly as in Ledgera's standard lifecycle.

Decentralised collaborative fine-tuning. In [8], we use Ledgera for a decentralised collaborative fine-tuning use case, motivated by the need for cross-institutional AI governance without a central authority. The application combines two components built over Ledgera. First, participants collaboratively construct a training knowledge base using Conflict-free Replicated Data Types (CRDTs); each document and each update is anchored on Ledgera as a sequence of $\texttt{LP}_{\texttt{FD}}$, $\texttt{LP}_{\texttt{C}}$, and $\texttt{LP}_{\texttt{S}}$ transactions, creating an immutable, independently verifiable edit history. Second, participants with sufficient computational resources execute fine-tuning jobs locally; the initial model, hyperparameters, training parameters, and final model checkpoint are each anchored on Ledgera, producing a complete end-to-end provenance trail covering data, computation, and results.

Confidential MPC. In Ledgera `V_0_2`, all raw input and output values are visible to the `executors` that perform the computation. The coreset-layer provides agreement on the input set for multi-party function instances (c.f. Sec. 8), but this is a coordination guarantee, it does not hide inputs from `executors`. In other words, multi-party function instances in Ledgera are not to be understood as confidential MPC.

For confidential multi-party computation [4], where parties wish to compute a joint function without revealing their private inputs to one another, Ledgera can provide coordination. Indeed, the coreset-layer can be used to agree on the input shares of each input that should be used to compute output shares and then to agree on the subset of output shares used

to recompose the output. Using Ledgera for this coordination makes the execution auditable, while the actual confidential computation (e.g., based on secret sharing) runs separately.

Payment transactions. A central motivation for Ledgera's architecture is that total order is not always required for application correctness. This is well established for payment transactions: it has been known that single-owner asset transfer can be solved without consensus using Byzantine quorum systems ($2f + 1$ validations per transaction in an asynchronous system). Even non-conflicting payments from the same balance must be processed by a common correct validator. Bazzi and Tucci-Piergiovanni [3] break this bottleneck by introducing *fractional spending transactions* (transactions that spend only a small fraction of a balance) validated using a novel class of probabilistic (k_1, k_2)-quorum systems with both lower and upper intersection bounds. This enables up to k_1 non-conflicting payments to be validated concurrently with fewer than $f + 1$ validations each in a fully asynchronous system under an adaptive adversary, while a *settlement transaction* (using a larger quorum) reclaims remaining funds and prevents double-spending of the full balance.

Access control. Frey, Gestin and Raynal [12] study the synchronization power of AllowList and DenyList objects under Herlihy's consensus hierarchy, in a crash-prone asynchronous setting. They show that the consensus number of AllowList is 1i.e., an AllowList (opt-in, set-membership proofs) can be implemented without any consensus. By contrast, the consensus number of DenyList corresponds to the number of processes authorized to prove set-non-membership: only those verifiers need to synchronize, not all processes. These results are applied to concrete systems including anonymous asset transfer, e-voting, and decentralised identity management, providing guidelines for right-sizing the coordination each application actually requires. This directly motivates Ledgera's modular architecture: stateless verifiable computation (akin to AllowList-only applications) needs no consensus in the critical path, while conflict-detecting applications such as asset transfer (akin to DenyList-based systems) require a limited form of agreement among the relevant validators.

References

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2003.

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Rida Bazzi and Sara Tucci-Piergiovanni. The fractional spending problem: Executing payment transactions in parallel with less than f+1 validations. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, PODC '24, page 295–305, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 52–61, New York, NY, USA, 1993. Association for Computing Machinery.

[5] William Boitier, Antonella Del Pozzo, Álvaro García-Pérez, Stephane Gazut, Pierre Jobic, Alexis Lemaire, Erwan Mahe, Aurelien Mayoue, Maxence Perion, Tuanir Franca Rezende, Deepika Singh, and Sara Tucci-Piergiovanni. Fantastyc: Blockchain-based federated learning made secure and practical. In *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*, pages 260–270, Sep. 2024.

[6] Christian Cachin, Dan Dobre, and Marko Vukolić. Separating data and control: Asynchronous bft storage with $2t+1$ data replicas. In Pascal Felber and Vijay Garg, editors, *Stabilization, Safety, and Security of Distributed Systems*, 2014.

[7] Christian Cachin, Simon Schubert, and Marko Vukolic. Non-Determinism in Byzantine Fault-Tolerant Replication. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[8] Antonella Del Pozzo, Erwan Mahe, Sami Souihi, and Sara Tucci-Piergiovanni. Ledgera and its application to decentralised collaborative fine-tuning. In *IEEE ICDCS 2026 Posters/Demonstrations*, 2026.

[9] Dan Dobre, Ghassan O Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolic. Powerstore: proofs of writing for robust and efficient storage. In ACM, editor, *CCS 2013, 20th ACM Conference on Computer and Communications Security, 4-7 November 2013, Berlin, Germany*, Berlin, 2013. ACM, 2013. Author's version. The definitive version was published in CCS 2013, Berlin, Germany. `http://dx.doi.org/10.1145/2508859.2516750`.

[10] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. 1988.

[11] El-Mahdi El-Mhamdi, Sadegh Farhadkhani, Rachid Guerraoui, Arsany Guirguis, Lê-Nguyên Hoang, and Sébastien Rouault. Collaborative learning in the jungle (decentralized, byzantine, heterogeneous, asynchronous and nonconvex learning). In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc.

[12] Davide Frey, Mathieu Gestin, and Michel Raynal. The Synchronization Power (Consensus Number) of Access-Control Objects: the Case of AllowList and DenyList. In Rotem Oshman, editor, *37th International Symposium on Distributed Computing (DISC 2023)*, volume 281 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:23, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[13] Ricardo Padilha and Fernando Pedone. Belisarius: Bft storage with confidentiality. In *2011 IEEE 10th International Symposium on Network Computing and Applications*, pages 9–16, 2011.

[14] Ricardo Padilha and Fernando Pedone. Scalable byzantine fault-tolerant storage. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 171–175, 2011.