HERMES: Fault-Tolerant Middleware for Blockchain Interoperability

Rafael Belchior [1], André Vasconcelos [2], Miguel Correia [2], and Thomas Hardjono [2]

[1]INESC-ID
[2]Affiliation not available

October 30, 2023

Abstract

The emergence of blockchain interoperability is reducing the risk of investing in blockchain by avoiding vendor lock-in, leveraging interoperation, and providing migration capabilities. However, to fully unlock the internet of blockchains, it is necessary to provide enterprise interoperability mechanisms that allow service providers to comply with different regulations, e.g., data privacy regulations. Each blockchain can be reached via a gateway, allowing to interconnect value, to provide different services, and to enable self-sovereignty. To realize this vision, we propose Hermes, a fault-tolerant middleware that connects blockchain networks and is based on the Open Digital Asset Protocol (ODAP). Hermes is crash fault-tolerant by allying a new protocol, ODAP-2PC, with a log storage API that can leverage blockchain to secure logs, providing them transparency, auditability, availability, and non-repudiation. We introduce a use case benefiting from Hermes, digital cross-jurisdiction promissory notes. We show that cross-chain transactions can be achieved securely with Hermes, given that gateways are complying with legal frameworks.

HERMES: Fault-Tolerant Middleware for Blockchain Interoperability

Rafael Belchior, André Vasconcelos, Miguel Correia, Thomas Hardjono

***Abstract*—Blockchain interoperability is reducing the risk of investing in blockchain by avoiding vendor lock-in, leveraging interoperation, and providing migration capabilities. However, unlocking the *internet of blockchains* requires enterprise interoperability mechanisms that allow service providers to comply with different regulations, e.g., data privacy regulations. Each blockchain can be reached via a *gateway*, allowing to interconnect value, to provide different services, and to enable self-sovereignty. To realize this vision, we propose HERMES, a fault-tolerant middleware that connects blockchain networks and is based on the Open Digital Asset Protocol (ODAP). Hermes is crash fault-tolerant by allying a new protocol, ODAP-2PC, with a log storage API that can leverage blockchain to secure logs, providing them transparency, auditability, availability, and non-repudiation. We introduce a use case benefiting from HERMES, digital cross-jurisdiction promissory notes. We show that cross-chain transactions can be achieved securely with HERMES, given that gateways are complying with legal frameworks.**

I. INTRODUCTION

The interoperability of *blockchains* is key to the area [9]. Several works point out that a blockchain, or more generically a Distribute Ledger Technology (DLT) system – we use the two terms interchangeably –, can deliver more value when connected to multiple business ecosystems and data providers [31], creating an interconnected network of value, similarly to the rise of the Internet [20], [38].

Recent efforts on the blockchain space are already enabling the connection of heterogeneous (i.e., different) blockchains (e.g., [4], [39], [27]), enabling the *internet of blockchains*, and thus a multiple-blockchain approach for developing applications (such as [25], [33], as surveyed by [9]). For instance, Hyperledger Cactus allows defining business logic plugins that implement use cases connecting both public and private blockchains [27], capable of complying with legal frameworks and regulations. Although several dozens of solutions and architecture proposals exist for interoperating blockchains, practical solutions are scarce and limited [9]. In particular, current solutions are often lacking standardization and provide interoperation capabilities for limited use cases (e.g., asset transfers [13], [35], [10]).

The notion of *blockchain gateway* (*gateway* for short) envisions each blockchain as an *autonomous system*, providing the emerging internet of blockchains survivability, cost-effectiveness, and capabilities for accomodating different services and networks [20]. Autonomous systems are able to comply with different jurisdictions, and also *hierarchically aggregate information*, providing routing capabilities for *cross-blockchain transactions* (CB-Tx) with an arbitrary payload. Examples of cross-blockchain transactions are asset transfers and smart contract calls [25]. Gateways are autonomous systems that deliver messages to other gateways, following the end-to-end principle, where the application layer delegates message delivery reliability, data encryption, and key management to a gateway (transport layer) [19]. The client application handles the delivered item (value layer), allowing for great flexibility. In other words, gateways provide the infrastructure for applications issuing CB-Tx, by connecting to DLT systems (e.g., blockchains). Within the Internet Engineering Task Force (IETF) there is currently ongoing work on an asset transfer protocol that operates between two gateway devices, the *Open Digital Asset Protocol* (ODAP) [22], [7]. ODAP is the first cross-chain communication protocol handling multiple digital asset cross-border transactions by leveraging asset profiles (the schema of an asset) and the notion of gateways. The process of transferring an asset among blockchains is equivalent to an atomic swap, that locks an asset in a blockchain and creates its representation on another.

We present HERMES, a middleware for blockchain interoperability that focuses on reliability. The main component of Hermes is an extension of the ODAP protocol that we also introduce in this paper. We denominate this protocol *ODAP-2PC* as it is inspired in the two-phase commit protocol (2PC) [11]. Hermes also leverages a log storage API that comes with different storage options: local, cloud, or blockchain-based. By modeling and developing this system, we expect to address three *research questions*: *RQ1* What is the reliability of a cross-chain transaction issued by a gateway, i.e., how can one be sure that a gateway can effectively deliver transactions? *RQ2* What is the trade-off between resiliency and performance of gateways? *RQ3* How decentralized is HERMES, and how can it be accountable for the transactions it manages? and *RQ4* What to expect in terms of security and privacy of gateway-based interoperability solutions?

Our work focuses on the *reliability* of gateways, and thus on ODAP-2PC, as crashes can pose severe threats to the correct operation of gateways: what happens if a gateway initiates a CB-Tx and then crashes? Systems that perform sensitive operations such as virtual asset transfers among ledgers have to possess a degree of resiliency and fault tolerance in the face of possible crashes [30]. Since gateways handle information transfer between parties, it is desirable that either all operations are executed successfully or none is executed. Atomic commitment protocols such as 2PC attempt at providing transaction *atomicity*, where multiple parties consistently carry out a single logical action, typically distributed. Other properties desirable for the reliability of distributed transactions are *consistency*, *isolation* and *durability*. These ACID properties are the basis for transactional systems for more than twenty years [26], and are discussed throughout this paper. A key component of ODAP-

2PC's crash recovery guarantees that the source and target DLTs are modified consistently, i.e., that assets taken from the source DLT are persisted into the recipient DLT, and no double spend can occur. To address this challenge, ODAP-2PC maintains logs that enable either the gateways to resume partially completed transfers, allowing for a gateway to recover in case of a crash [11].

The contributions of this paper are three-fold: first, the blockchain gateway concept is explained, from a theoretical and practical perspective. Second, we present the HERMES fault-tolerant middleware, instantiated with the ODAP protocol and ODAP-2PC. Lastly, we provide a comprehensive discussion on Hermes as a solution for blockchain interoperability, focusing on consistency, performance, and decentralization. We also briefly explore a use case for cross-jurisdiction asset transfers, illustrating how one can leverage Hermes to achieve blockchain interoperability compliant with legal and regulatory frameworks.

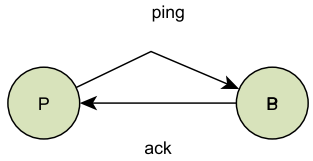
The rest of this paper is structured as follows: Section II presents the background. After that, we introduce the gateway concept and Hermes, in Section III. After, in Section IV, we present ODAP, including the message and logging procedure, the log storage API, and the distributed recovery protocol (Sections IV-B, IV-C, and IV-D, respectively). Section V, presents a use case that benefits from Hermes. Section VI presents our discussion on gateways, ODAP, and ODAP-2PC in the light of the presented research questions. The related work follows, in Section VII. Finally, we conclude the paper in Section VIII.

II. BACKGROUND

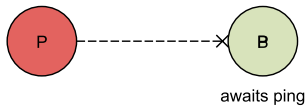
This section presents the background on fault tolerance models and nomenclature both on logging and blockchain interoperability.

Fault Tolerance: A *fault* is an event that alters the expected behavior of a system. Faults can imply the transition from a correct state of the system to an incorrect state, called *errors*. Errors can provoke *failures* if the system deviates from its specification, possibly causing loss of information or compromising business logic. Nodes can experience failures where for various reasons (e.g., power outage, network partitions, faulty components). We consider four failure types: message loss, communication link failure, site failure, and network partition. Albeit common, failures can be detected by different mechanisms, such as timeouts, defined as the upper bound δ_t that a message is expected [11].

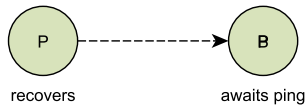
Fault Recovery: Typically, crash fault-tolerant (CFT) services can tolerate $\frac{n}{2}$ nodes crashing, with n being the number of nodes. As long as there is a majority of nodes with the latest state, failures can be tolerated. The *primary-backup model* defines a set of n hosts (or nodes) that, as a group, assures service resiliency, thus improving availability. In this model, an application client sends messages to a primary node $\mathcal{P}$. The primary nodes redirects the message updates to a set of replicas (backups) $\mathcal{B} = \{\mathcal{B}_1, ..., \mathcal{B}_n\}$, when it receives a message. The backup server k propagates the new incoming message to the backup server *k+1*, $k \leq n, k \in \mathbb{R}$. Node $\mathcal{P}$ is then notified an update when n-host resiliency is met, i.e., the message was at least replicated in n nodes. Should such acknowledgment fail to be retrieved by $\mathcal{P}$, a message update request is re-sent. If $\mathcal{P}$ crashes, then a new leader $\mathcal{P}_{new} \in \mathcal{B}$ is elected. If a backup node receives a request from the application client, it redirects it to $\mathcal{P}$, only accepting it when the latter sends the update request. When an update is received, $\mathcal{P}$ sends the message update to its right-hand neighbor, sending back an acknowledgment.



(a) Both nodes operational. P sends ping to B and gets ACK response.



(b) Node P crashes, leaving B awaiting for a ping.



(c) Node P recovers, re-establishing the connection to B.

Fig. 1: Self-healing mode with two nodes

Another recovery mechanism is *self-healing* [11]. In self-healing, when nodes crash, they are assumed to recover eventually. While this mode is cheaper than primary-backup, as there are fewer nodes, less exchanged messages and lower storage requirements, it comes at the expense of availability. In particular, the protocol may block until nodes recover. Figure 1 depicts a simplified self healing protocol for two nodes. Node P sends a ping to node B, that responds with an ACK. In case of a crash, node B awaits for a ping.

Atomic Commit Protocols: An atomic commit protocol (ACP) is a protocol that guarantees a set of operations being applied as a single operation. An atomic transaction is indivisible and irreducible: either all operations occur, or none does. ACPs consider two roles: a *Coordinator* that manages the execution of the protocol, and *Participants* that manage the resources that must be kept consistent. ACPs assume stable storage with a write-ahead log (a history of operations are persisted before actions are executed). Example of ACPs are the two-phase commit protocol, 2PC, the three-phase commit protocol, 3PC, and non-blocking atomic commit protocols [11].

2PC achieves atomicity even in case of temporary system failure, accounting for a wide adoption both in the academia and in the industry. It has two phases: the voting phase and the commit phase. In the voting phase, the coordinator prepares all participants to take place in a distributed transaction by inspecting each participant's local status. Each participant executes eventual local transactions required to complete the distributed transaction. If those are successful, participants send a *YES* response to the coordinator, and the protocol continues. Else, if the *NO* response is sent, it means that the participant chose to abort; this happens when there are problems at the local partition. Next, in the commit phase, when the coordinator obtains *YES* from all participants, a *COMMIT* message is sent to the participants that voted *YES*. This message triggers the

execution of local transactions that implement the distributed transaction. Otherwise, the coordinator sends an *ABORT* message, triggering a rollback on each local partition.

Logging: A log $\mathcal{L}$ is a list of log entries $\{l_1, l_2, ..., l_n\}$ such that entries have a total order, given by the time of its creation. A log is considered *shared* when a set of nodes can read and write from the log. On the other hand, a log is *private* (or *local*) when only one node can read and write it. Logs are associated to a process p running *operations* on a certain node. We denote the n^{th} step of process p as (n, p). We denote the i^{th} log entry, as l_i, and the log entry referring to process p and step k as $l^{p,k}$. Both i and k are monotonically increasing positive integers. To manipulate the log, we define a set of *log primitives*, that translate *log entry requests* from a process p into log entries. The log primitives are *writeLogEntry* (writes a log entry), *getLogLength* (obtains the number of log entries), and *getLogEntry(i)* (retrieves a log entry l_i). A log entry request typically comes from a single event in a given protocol.

A *log storage API* provides access to the primitives. Log entry requests have the format <phase, step, operation, nodes>, where the field *operation* corresponds to an arbitrary command, and the field *nodes* to the parties involved in the protocol p. We define four operations types to provide context to the protocol being executed. Operation type *init-* states the intention of a node to execute a particular operation, and operation *exec-* expresses that the node is executing the operation. The operation type *done-* states when a node successfully executed a step of the protocol, while *ack-* refers to when a node acknowledges a message received from another. Conversely, we use the type *fail-* to refer to when an agent fails to execute a specific step. The field *nodes* contains a tuple with a node A issuing a command, or a node A commanding a node B the execution of a command c, if the form is A or $A \rightarrow B$ (c may be ommitted), respectively.

Figure 2 illustrates the logging procedure of a process A that has three steps and includes two nodes using a log storage API. While typically each log has its own log (and log storage API), we only represent one for simplicity. Note that nodes can also have a common log. Log entry $l_1 = l^{init,1}$ corresponds to the node's first message to the log storage API, which on its turn persists it on a log, using the `writeLogEntry` primitive. The log storage API writes the message that is received. For instance, in step 2 the log storage API executes writeLogEntry<Process A, 1, init-node, Node>. Log entry l_1 is created in step (2), coming from the command issued at step 1. Conversely, writing $l_2 = l^{init,2}$ (steps 4 and)) corresponds to the command that the node issues towards node 2 (step 6), *initAllNodes*, which causes node 2 to issue an init operation. Log entry $l^{init,3}$ corresponds to the execution of init by Node 2 (step 9). At step 12, `getLogLength` returns 3.

Blockchain Interoperability: A recent survey classifies blockchain interoperability studies in three categories: Public Connectors, Blockchain of Blockchains, and Hybrid Connectors [9]. The survey uses the terms *Cross-Chain Transaction (CC-Tx)* and *Cross-Blockchain Transaction (CB-Tx)* respectively for the cases where the blockchains are homogeneous (same technology) and heterogeneous (the opposite). Here we do not distinguish the use and adopt the acronym CC-Tx to designate such transactions. These transactions are implemented using *Cross-Chain Communication Protocols* (CCCPs).

A recent survey classifies blockchain interoperability studies in three categories: Cryptocurrency-directed interoperability approaches, Blockchain Engines, and Blockchain Connectors [9]. Cryptocurrency-directed approaches are directed to enabling the transfer of digital assets (e.g., cryptocurrencies) across homogeneous and heterogeneous blockchains. The cryptocurrency-directed approaches typically rely on protocols leveraging public blockchains, as they assume that gateways are not trusted. As a result, these approaches are difficult to integrate with permissioned blockchains, that support with arbitrary assets and smart contracts. The second category are the blockchain engines, which enable creating application-specific blockchain that can communicate with its other instances. These solutions can benefit from implementing gateways, providing each application-specific blockchain (e.g., applications running on a parachain) self-sovereignty, regarding communications with outter blockchains. The third category, blockchain connectors include trusted relays, blockchain agnostic protocols, blockchain of blockchains solutions, and blockchain migrators. Trusted relays are software components, typically centralized, where escrows route cross-blockchain transactions.

The same survey points out several attempts to design an architecture for blockchain interoperability and bring about the common vocabulary used in this context. A pair of *homogeneous blockchains* are blockchains powered by the same environment (e.g., an EVM-based [42] blockchain pair). On the other hand, *heterogeneous blockchains* are power by different environments (e.g., Ethereum and Hyperledger Fabric [6]). *Cross-Chain Transaction (CC-Tx)* and *Cross-Blockchain Transaction (CB-Tx)* are transactions across blockchains, where the blockchains are homogeneous and heterogeneous, respectively. We do not distinguish the terms and assume that CC-Tx can be across heterogeneous systems for simplicity.

CC-Txs enable the *Internet of Blockchains (IoB)*, a system where blockchains can communicate, enabling the transfer of value and data. By enabling an IoB, we can enable a *Blockchain of Blockchains (BoB)*, a set of systems that realize a business use case by leveraging multiple blockchains. For this, *Cross-Chain Communication Protocols* (CCCPs) are needed. CCCPs are protocols allowing the communication of multiple blockchains. For simplicity, we do not distinguish between CCCPs and cross-blockchain communication protocols. While proper advances have been made enabling interoperation between public blockchains, interoperability amongst private blockchains is still an open problem.

III. The Architecture of Hermes

In this section, we introduce the gateway concept, as well as Hermes. A *gateway* is a DLT system node based on an underlying DLT-based system and functionally capable of performing CC-Tx, including asset transfers [20]. A *primary gateway* is the DLT system node acting as a gateway in a CC-Tx. Primary gateways may be supported by *backup gateways* for fault tolerance. Primary gateways can be a *source gateway* $\mathcal{G}_S$ or a *recipient gateway* $\mathcal{G}_R$, depending on the role they play in a CC-Tx. Source gateways initiate the gateway protocol, e.g., an asset transfer, data pushing/pulling. Gateways use machine-resolvable addresses (e.g. URIs/URLs) in order to communicate

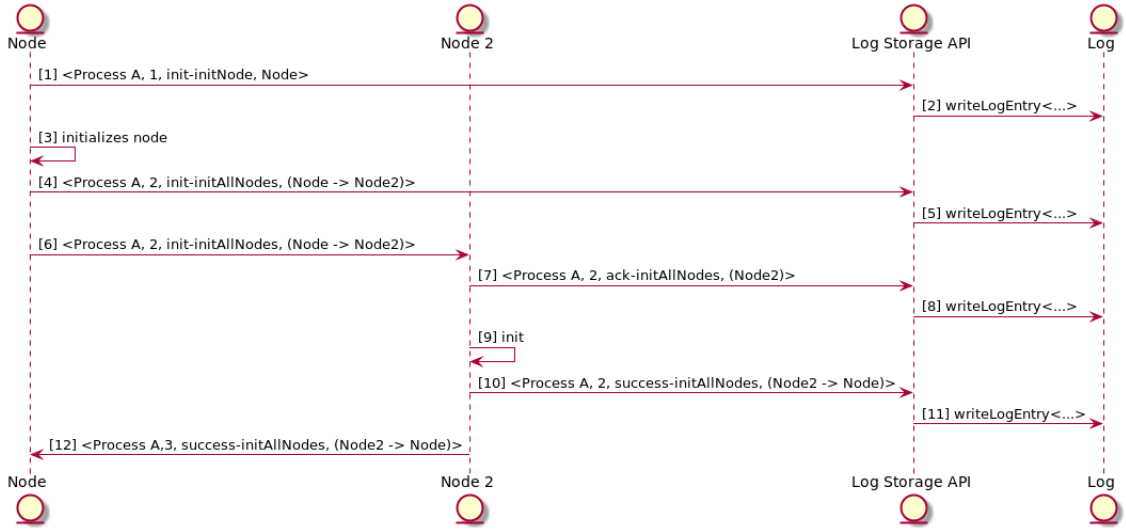


Fig. 2: Message exchange flow example. The log storage API writes the incoming message. Just one log shown for simplicity.

with other gateways, obtaining information such as public-key certificates and protocol-specific messages.

For gateways to be crash fault-tolerant, they keep track of each operation they do in a log (of operations). The log is a sequence of log entries, each entry representing a step of the gateway protocol. Each message has a schema, defining the parameters and the payload employed in each message flow. The log data comprises the log information retained by a gateway within a protocol using gateways. A gateway protocol specifies the set of messages and procedures between two gateways for their correct functioning. The gateway protocol considered in this paper is ODAP [22], [7].

A. Blockchain Interoperability with Hermes

HERMES is a gateway system that enables DLT interoperability based on gateways. This system has four layers, allowing for end-to-end communication. The gateway protocol layer implements any standards that a specific gateway implementation needs to comply with (e.g., travel rule [19]). ODAP, a gateway-based CCCP that realizes asset transfers, allows realizing technical interoperability for asset transfers. It is built on top of a distributed recovery protocol, providing reliability in the presence of crashes. On top of the gateway protocol stands a concrete implementation of a gateway. Jointly with the gateway protocol, it provides support for semantic interoperability [9], unlocking the value level. More specifically, in the value level, the business logic is defined for clients using gateways, allowing them to attribute value to the assets exchanged with ODAP. The whole stack provides atomicity, consistency, isolation, and durability of CC-Tx. Figure 3 represents HERMES' architecture.

Our architecture is flexible and modular, as its components are pluggable. Modularity allows building a system that can be adapted to specific needs. In this paper, we instantiate HERMES with the ODAP-Gateway, the ODAP CCCP, and its crash fault-tolerant distributed recovery protocol, ODAP-2PC. The whole stack allows a business case, gateway-to-gateway asset transfers, providing the basis for unidirectional asset transfers, expressed in detail in Section V. The Hermes Client allows to implement the business logic, realizing semantic interoperability.

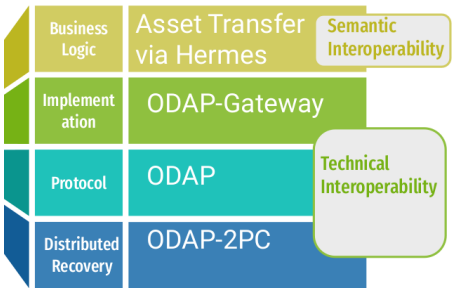


Fig. 3: The architecture of Hermes

B. System Model

We consider an asynchronous distributed system composed of two types of participants: clients and gateways. Clients are in charge of starting transactions and are connected to gateways that are connected to blockchains. More specifically, every gateway is a node from a DLT system authorized to act on it (manage assets, identify participants) via, for example, smart contracts. Gateways can communicate with other gateways and can crash (i.e., becoming unresponsive). We assume that there are no Byzantine or arbitrary faults. We assume blockchains are *secure*, so they fail only by crashing. We consider a blockchain secure if the data stored is immutable, transparent to all their participants, traceable, and, generally, the consensus mechanism cannot be subverted by malicious parties.

To recover from these crashes, gateways store in persistent storage data about the step of their protocol. Gateways are *honest-but-curious*, i.e., follow the protocol, but will attempt to learn all possible information from legitimately. received messages. HERMES provides the following properties:

- *P1 Atomicity:* Transactions either commit on all underlying ledgers or entirely fail.
- *P2 Consistency:* All gateways that reach a decision on a CC-Tx reach the same, either commit or abort. The state of the underlying ledgers reflect that decision.
- *P3 Durability:* Once a transaction has been committed, it must remain so regardless of any component crashes.
- *P4 Isolation:* When a transaction is issued, all the underlying assets are locked.

- *P5 Auditability:* Any CC-Tx executed can be inspected by the involved parties.
- *P6 Termination:* If a gateway proposes a transaction, it is eventually committed or aborted.

To satisfy these properties, HERMES leverages ODAP and ODAP-2PC.

C. Threat Model

ODAP assumes a trusted, secure communication channel between gateways (i.e., messages cannot be spoofed or altered by an adversary) using TLS 1.3 or higher, i.e., the receiver of the communication will be able to ascertain the authenticity and validity of the communication. Each gateway has a public and private key pair. New TLS sessions [34] are created when a gateway crashes and then recovers. Clients connect to gateways using a credential scheme such as OAuth2.0 [15].

The distributed recovery protocol has assumptions regarding log management. Log entries need integrity, availability, and confidentiality guarantees, as they are an attractive attack point [32]. Every log entry contains a hash of its payload for guaranteeing integrity. If extra guarantees are needed (e.g., non-repudiation), a log entry might be signed by the gateway creating it (e.g., with ECDSA [23]). Availability is guaranteed using the log storage API, which connects a gateway to dependable storage (local, external, or DLT-based). Each underlying storage provides different guarantees. Access control can be enforced via the *access control profile* that each log can have associated with, i.e., the profile can be resolved, indicating which client can access the log in which condition. Access control profiles can be implemented with access control lists for simple authorization. The authentication of the entities accessing the logs is done at the log storage API level (e.g., username and password authentication in local storage vs. blockchain-based access control in a DLT). We assume the log is not tampered with or lost.

While we consider both gateways to be trusted, we consider a probabilistic, polynomial-time adversary who can corrupt any gateway to prevent the protocol from achieving liveness. The adversary can do this by causing a gateway crash, interrupting an asset transfer. However, we assume that gateways do not deviate from the protocol.

IV. OPEN DIGITAL ASSET PROTOCOL

In this section, we present the building blocks of Hermes: ODAP, along with its messaging and logging flow, and the distributed recovery mechanism, ODAP-2PC.

A. ODAP and Properties

The ODAP protocol is a gateway-to-gateway unidirectional asset transfer protocol that uses gateways as the systems conducting the transfer [22]. An asset transfer is represented in the form $T : G_1 \overset{a,x}{\rightarrow} G_2$, where a source gateway G_1 transfers x asset units from type a from a source ledger $\mathcal{B}_S$ to a recipient ledger $\mathcal{B}_R$, via a gateway G_2.

The source gateway issues a transfer such that x asset units will be unavailable at the source DLT and become available at the target DLT. A recipient gateway is the target of an asset transfer, i.e., follows instructions from the source gateway. Hermes provides as strong durability guarantees as to the underlying durability guarantees of the chosen data store. If the datastore is a blockchain, Hermes can be considered to achieve transaction durability, if transactions are immutable and permanently stored in a secure decentralized ledger.

Durability

Hermes provides the durability guarantees that the infrastructure gateways are connected to.

The transfer process is started by a client (application) that interacts with the source gateway. The source gateway then deals with the complexity of translating an asset transfer request to transactions targeting both the source and target DLT systems. The gateway also knows other gateways, either directly or via a decentralized gateway registry. ODAP has several operating modes, but here we solely consider the relay mode. The relay mode realizes client-initiated gateway to gateway asset transfers.

In ODAP, a client application interacts with its local gateway (source gateway GS) over a Type-1 API. The existence of this API allows the client to provide instructions to GS (corresponding to the source gateway) concerning the assets stored in the source DLT and the target DLT (via the recipient gateway, GR). It is possible that the client has complex business logic code that triggers behavior on the gateways. Hence, ODAP allows three flows: the *transfer initiation flow*, where the process is bootstrap, and several identification procedures take place; the *lock-evidence flow*, where gateways exchange proofs regarding the status of the asset to be transferred; and the *commitment establishment flow*, where the gateways commit on the asset transfer. The schema of the messages exchanged by the ODAP protocol is depicted in Appendix A.

Figure 4 represents ODAP. When an end-user wants to perform an asset transfer, gateways conduct such process. In the transfer initiation flow (Phase 1), both gateways resolve identities, asset information (via the asset profiles) and establish a secure channel. This verification includes verifying the asset profile validity, the travel rule status, and the pair originator-beneficiary of the transaction [22]. In the lock-evidence verification flow (Phase 2), claims on the status of assets are exchanged, and their correspondent proofs are persisted. The persistence of asset status proof allows for non-repudiation and accountability, proving useful in case of a dispute.

Theorem 1 (Isolation): Let there be an instance of ODAP, with a source gateway $\mathcal{G}_S$ and a recipient gateway $\mathcal{G}_R$, operating on an asynchronous environment. Given a lock primitive `LOCK` that prevents assets from being used, if there is a timeout δ_t (applied to steps 2.3 and 3.3 of ODAP), then ODAP provides transaction isolation.

Proof: In this context, transaction isolation implies that a certain asset is locked. At various points of the protocol, both $\mathcal{G}_S$ and $\mathcal{G}_R$ are waiting for messages before proceeding. In particular, in steps 2.3 and 2.4, where the logging procedure depends on the asset lock's success. A trigger δ_t, defining an interval before an asset is used to assure that an asset is securely locked, even in probabilistic-based consensus blockchains. After δ_t counterparty $\mathcal{G}_R$ can produce a log entry with the asset locking proof. When `LOCK` is called, assets are locked, rendering any attempt of writing fruitless. A similar process occurs in step 3.3. Thus,

as assets cannot be changed up to step 3.7, ODAP guarantees transaction isolation. □

Isolation

ODAP-2PC provides transaction isolation by pre-locking assets before the commitment of an asset transfer.

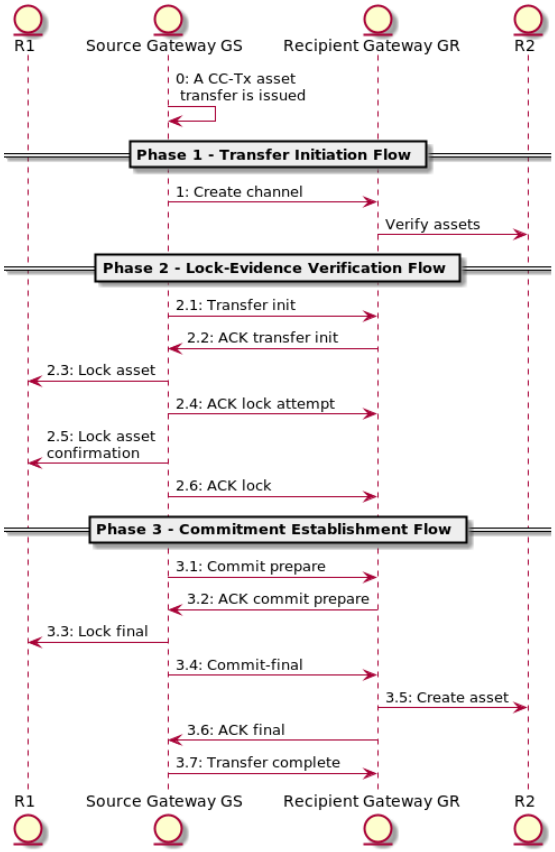


Fig. 4: Simplified sequence diagram depicting ODAP. A transfer is issued by an end-user to the gateway (G1), which then manages on-chain resources (L1), and communicates with a counterparty gateway (G2). The asset transfer corresponds to the creation of L2.

Finally, at the Commitment Establishment Flow (Phase 3), assets are escrowed. In practice, assets are locked on the source ledger and represent those created on the target ledger. The lock of assets prevents double-spend attacks. ODAP aims at providing termination, a non-trivial problem when considering distributed transactions [5]. Thus, we consider three processes on ODAP, p_1 = *transfer initiation flow*, p_2 =*lock-evidence flow*, and p_3 = *commitment establishment flow*. Process p_1 has 2 steps, p_2 has 6 steps, and $p3$ has 6 steps. Thus, a normal end-to-end ODAP flow would have 14 steps.

B. Message and Logging Flow

We consider the set of logging nodes $\mathcal{N} = \{\mathcal{G}_S, \mathcal{G}_R\}$, with log entry requests with the format <phase, step, type-operation operation, nodes>. Within processes, two types of operations are considered: *private operations* and *public operations*. Private operations involve only one gateway, requiring two log entries, the intention of executing a command and the confirmation of the execution. This serves to handle crashes in systems with only one node. Public operations are operations which state is known by more than one node. Intuitively, a private operation only is only known by the node executing it, whereas public operations involve several nodes, and thus are perceived by more nodes than the one executing it.

The message flow generates a variable number of log entries, depending on the situation: i) a private operation completes successfully, generating three log entries (init-X, exec-X, done-X); ii) a private operation fails, generating three log entries (init-X, exec-X, fail-X); iii) a public operation completes successfully, generating at least four log entries (init-X, exec-X, done-X, ack-X), and iv) a public operation fails, generating four log entries (init-X, exec-X, fail-X, ack-X). Given that a normal ODAP flow has 14 steps, one would expect at least 42 log entries.

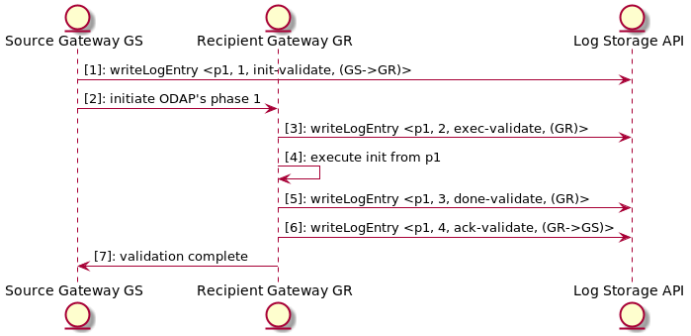


Fig. 5: Message flow regarding the validation operation, of ODAP's phase 1.

Let us consider an example where there is an asset transfer $\mathcal{G}_S \overset{a,1}{\rightarrow} \mathcal{G}_R$. We depict a message exchange with content m from $\mathcal{G}_S$ to $\mathcal{G}_R$ by $\mathcal{G}_S \overset{m}{\rightarrow} \mathcal{G}_R$. The reply from $\mathcal{G}_R$ to $\mathcal{G}_S$ is represented by $\mathcal{G}_R \overset{\alpha(m)}{\rightarrow} \mathcal{G}_S$, where α is a function that given an operation, step, and input from a counterparty gateway, returns the response to it. Figure 5 illustrates part of the message flow involving the public operation $p1$, the ODAP's first phase. Note that one operation has been performed before, corresponding to three log entries (init, exec, done), and to a $\mathcal{G}_S$ client issuing an asset transfer. Thus, the first log entry from p_1 has index 4. In the transfer initiation flow, where $\mathcal{G}_S$ initiates a transfer of one asset a to $\mathcal{G}_R$, the first step is to resolve identities. To fulfill step 1, $\mathcal{G}_S$ takes two actions: 1) it expresses that $\mathcal{G}_R$ will be informed to initiate an asset transfer; and 2) it sends that message to $\mathcal{G}_R$. These messages are sent to the Log Storage API, that generates the appropriate log entries $l_4 = l^{p_1,1} =< p_1$, 1, init-validate, $(\mathcal{G}_S \rightarrow \mathcal{G}_R)$¿, $l_5 = l^{p_1,2} =< p_1$,2, init, $(\mathcal{G}_R) >$, $l_6 = l^{p_1,3} =< p_1$, 3, done-init, $(\mathcal{G}_R) >$, and $l_7 = l^{p,4} =< p_1$, 4, ack-validate, $\mathcal{G}_R >$. Table I summarizes the exchanged messages and the log entries they generate. Note that these log entries are simplified, for illustration purposes. ODAP logs have a well-defined schema, and extra parameters, illustrated later in Section IV-C.

We consider a log storage API that allows developers to be abstracted from the storage details (e.g., relational vs. non-relational, local vs. cloud vs. DLT-based) and handles access control if needed. In the next section, we detail the functioning of the log storage API.

TABLE I: Logging flow regarding the validation operation, of ODAP's phase 1

Event	From	To	Log ID	Log Content	Operation	Type
$\mathcal{G}_S$ triggers the validation operation	$\mathcal{G}_S$	$\mathcal{G}_R$	$l_4 = l^{p1,1}$	<p1, 1, init-validate, (GS->GR)>	validate	init
$\mathcal{G}_R$ executes the validation operation	$\times$	$\mathcal{G}_R$	$l_5 = l^{p1,2}$	<p1, 2, exec-validate, (GR)>	validate	exec
$\mathcal{G}_R$ completes the validation operation	$\times$	$\mathcal{G}_R$	$l_6 = l^{p1,3}$	<p1, 3, done-validate, (GR)>	validate	done
$\mathcal{G}_R$ informs $\mathcal{G}_S$	$\mathcal{G}_R$	$\mathcal{G}_S$	$l_7 = l^{p1,4}$	<p1, 4, ack-validate, (GR->GS)>	validate	ack

C. Log Storage API

The log storage API allows developers to abstract operations on the log, focusing on the development of gateway protocols. Our API uses the following primitives:

- `initializeLog(`γ`)`: returns a reference to an empty log $\mathcal{L}$, stored on the support γ. The support can be local γ_{local}, cloud γ_{cloud}, or a blockchain γ_{bc}.
- `getLogSupport()`: returns the support γ.
- `writeLogEntry(`$l, \mathcal{L}$`)`: writes a log entry l in the log $\mathcal{L}$, stored on the support γ.
- `getLogEntry(`i`)`: returns the log entry l_i.
- `getLogLength`: returns the length of the log, i.e., $|\mathcal{L}|$.
- `getLatestLogEntry`: returns the log entry l_j such that $\nexists l_i : i > j$
- `getLog`: returns $\mathcal{L}$.

This API can be exposed as a REST API, allowing for the log storage API to be hosted in an execution environment different from the one running the gateway implementation. We consider the log file to be a stack of log entries. Each time a log entry is added, it goes to the top of the stack (has the highest index). Logs can be saved either locally (e.g., γ_{local} = computer's disk) and may also be saved in an external service (e.g., γ_{cloud} = cloud storage service) or even in a DLT (e.g., γ_{bc} = Ethereum).

Depending on the support, logs will have different privacy levels. On support γ_{local}, logs are isolated, each gateway keeping its entries private. In case of a crash, the crashed gateway will retrieve the most updated version of the log: if it is local, it needs to require it from other gateways (and thus being susceptible to misbehavior from other gateways). This mode thus requires substantial trust on other gateways. The DLT-based repository, γ_{bc}, offers strong reliability concerning log-saving, due to its immutability, transparency, and traceability [32], [8]. In particular, this method offers accountability because persisted log entries are non-repudiable, traceable, and cannot be changed; it offers high-availability because they are replicated across all nodes participating in the network. The cloud support γ_{cloud} offers a tradeoff between γ_{local} and γ_{bc}, both in terms of cost and integrity guarantees. As this support is mediated by a cloud provider, trust is put on the provider instead of uniquely on the counterparty gateway. However, it is likely to be more costly than the local support.

Auditability

Given a secure γ = γ_{bc}, Hermes provides auditability by keeping an immutable, traceable, distributed chain of log entries. Extra guarantees are needed if γ = $\gamma_{local}, \gamma_{cloud}$.

Format of log entries: The log entries' format should account for three phases, in case the gateway protocol is ODAP. In Section IV-B we introduced a simplified version of a log entry for illustration purposes. The mandatory fields for a log entry for ODAP-2PC are:

ODAP-2PC Log Schema – Mandatory Fields

1) **Session ID**: unique identifier (UUIDv2) representing an ODAP interaction (corresponding to a particular flow)
2) **Sequence Number**: represents the ordering of steps recorded on the log for a particular session
3) **ODAP Phase ID**: flow to which the logging refers to. Can be Transfer Initiation flow, Lock-Evidence flow, and Commitment Establishment flow.
4) **Source Gateway ID**: the public key of the gateway initiating a transfer Source DLT ID: the ID of the gateway initiating a transfer
5) **Recipient Gateway ID**: the public key of the gateway involved in a transfer Recipient DLT ID: the ID of the gateway involved in a transfer
6) **Timestamp**: timestamp referring to when the log entry was generated (UNIX format)
7) **Payload**: Message payload: contains subfields *Votes* (optional), *Msg*, *Message type*. The field Votes refers to the votes parties need to commit in the 2PC. Msg is the content of the log entry. Message type refers to the different logging actions (e.g., command, backup).
8) **Payload Hash**: hash of the current message payload

Apart from mandatory log fields, the log schema for ODAP-2PC contains optional fields. The *logging profile* field contains the profile regarding the logging procedure. If not present, $\gamma = \gamma_{local}$ is assumed. The *Source Gateway UID* is the unique identifier (UID) of the gateway initiating a transfer. The *Recipient Gateway UID* is the UID of the gateway involved in a transfer. The *Message Digest* is a gateway signature over the log entry. The *Last Log Entry* is the hash of the previous log entry. Finally, the *Access Control Profile* is the field specifying a profile regarding the confidentiality of the log entries being stored; in particular, this field can be used to parse access control policies to the supports managing logs. Next, we introduce the ODAP-2PC, a distributed recovery mechanism for gateways.

D. Distributed Recovery Procedure

One of the key deployment requirements of gateways for asset transfers is a high degree of gateways availability. A distributed recovery procedure then increases the resiliency of a HERMES gateway by tolerating faults. Next, we present an overview of ODAP-2PC.

ODAP-2PC Overview: The protocol is crash fault-tolerant, so the gateways are trusted to operate the ODAP protocol as specified unless they stop. We envisage ODAP-2PC to support two strategies to increase the availability of gateways [7]: (1) *self-healing mode:* after a crash, a gateway eventually recovers, informs other parties of its recovery, and continues executing the protocol; (2) *primary-backup mode:* after a crash, a gateway

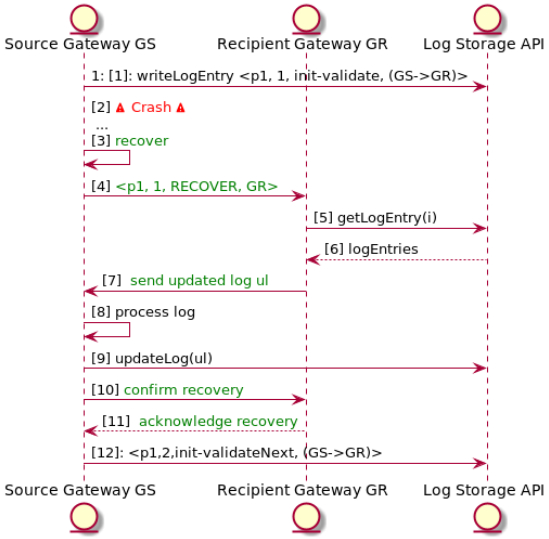


Fig. 6: $\mathcal{G}_S$ crashing before issuing init-validation to $\mathcal{G}_R$

Fig. 7: $\mathcal{G}_S$ crashing after issuing the init command to $\mathcal{G}_R$

may never recover, but that timeout can detect this failure [5]; if a node is crashed indefinitely, a backup is spun off, using the log storage API to retrieve the log's most recent version.

In self-healing mode – the mode we detail in this paper – when a gateway restarts after a crash, it reads the state from the log, and continues executing the protocol from that point on. We assume that the gateway does not lose its long-term keys (public-private key pair) and can reestablish all TLS connections. In Primary-backup mode, we assume that after a period δ_t of the failure of the primary gateway, a backup gateway detects that failure unequivocally and takes the role of the primary. The failure is detected using heartbeat messages and a conservative value for δ_t. For that purpose, the backup gateway does essentially the same as the gateway in self-healing mode: reads the log and continues the process. In this mode, the log must be shared between the primary and the backup gateways. If there is more than one backup, a leader-election protocol must be executed to decide which backup will take the primary role.

In both modes, logs are written before operations (write-ahead) to provide atomicity and consistency to the protocol used for asset exchange. The log-data is considered as resources that may be internal to the DLT system, accessible to the backup gateway and possible other gateway nodes.

There are several situations when a crash may occur. Figure 6 represents the crash of $\mathcal{G}_S$ before it issues a validation operation to $\mathcal{G}_R$ (steps 1 and 2). Both gateways keep their log storage APIs, with γ_{local}. For simplicity, we only represent one log storage API. In the self-healing mode, the gateway eventually recovers (step 3), building a recovered message in the form `<phase, step, RECOVER, nodes>` (step 4). The non-crashed gateway queries the log entries that the crashed gateway needs (steps 5, 6). In particular, $\mathcal{G}_S$ obtains the necessary log entries at step 7 and compares them to its current log. After that, $\mathcal{G}_S$ attempts to reconcile the changes with its current state (step 8). Upon processing, if both log versions match, then the log is updated, and the process can continue. If the logs differ, then $\mathcal{G}_S$ calls the primitive `updateLog`, updating its log (step 9) and thus allowing the crashed gateway to reconstruct the current state. In this particular example, step 9 would not occur because operations exec-validate, done-validate, and ack-validate were not executed by $\mathcal{G}_R$. If the log storage API is on the shared mode, no extra steps for synchronizations are needed. After that, it confirms a successful recovery (steps 10, 11). Finally, the protocol proceeds (step 12).

Figure 7 represents a recovery scenario requiring further synchronization. At the retrieval of the latest log entry, $\mathcal{G}_S$ notices its log is outdated. It updates it, upon necessary validation, and then communicates its recovery to $\mathcal{G}_R$. The process then continues as normal. (for instance, corresponding to exec-validate, done-validate, and ack-validate)

1) The ODAP-2PC Protocol: In this section, we present the ODAP-2PC protocol. In particular, this protocol is used at ODAP's Phase 3, crucial for the atomicity and the consistency of asset transfers. We consider two parties: the coordinator $\mathcal{G}_S$, and the participant $\mathcal{G}_R$. The coordinator manages the protocol execution while the participant follows the coordinator's instructions.

ODAP-2PC is a 2PC protocol able to detect and recover from crashes, delivering the effort to execute an asset transfer starting at ODAP's phase 3: the commitment establishment flow. Crashes at other phases of the ODAP are handled by the self-healing mechanism, supported by the messaging and logging mechanism. In phase 3, sensitive messages that include the lock and unlocking of assets may not arrive due to failures (e.g., communication failures, gateway crash due to power outage). To detect crashes, we use a timeout δ_C. However, processes may wait for the crashed gateway to recover for an unbounded timespan, wasting resources (e.g., locked assets). To avoid this, we introduce an additional timeout $\delta_{rollback}$. When a gateway does not recover before this timeout, a *timeout action* is triggered, corresponding to the *rollback protocol*. A possible rollback protocol cancels the current transactions by issuing transactions with the contrary effect, guaranteeing the consistency of the DLT whose gateway is not crashed. Upon recovery, the crashed gateway is informed of the rollback, performing a rollback too. This process guarantees the consistency of both underlying

DLTs.

Algorithm 1: ODAP-2PC Protocol

Input: Coordinator $\mathcal{G}_S$, Participant $\mathcal{G}_R$, Asset a, Gateway primitives PRE_LOCK, LOCK, COMMIT,CREATE_ASSET, COMPLETE, ROLLBACK
Result: Asset a transferred from $\mathcal{G}_S$ to $\mathcal{G}_R$

1 $PO_{\mathcal{G}_S} = \perp$ ▷ rollback list for $\mathcal{G}_S$
2 $PO_{\mathcal{G}_R} = \perp$ ▷ rollback list for $\mathcal{G}_R$
3 ▷ Pre-Voting Phase
4 preLock $= \mathcal{G}_S$.PRE_LOCK(a) ▷ step 2.3
5 $PO_{\mathcal{G}_S}$.append(preLock)
6 ▷ Voting Phase
7 $\mathcal{G}_S \overset{vote-req}{\longrightarrow} \mathcal{G}_R$ ▷ step 3.1
8 **wait until** $\mathcal{G}_R \overset{\alpha(vote-req)}{\longrightarrow} \mathcal{G}_S$ ▷ step 3.2
9 ▷ Decision Phase
10 **if** $\mathcal{G}_R \overset{\alpha(vote-req)}{\longrightarrow} \mathcal{G}_S = NO$ **then**
11 $\quad \mathcal{G}_S \overset{abort()}{\longrightarrow} \mathcal{G}_R$ ▷ otherwise, $\mathcal{G}_R \overset{\alpha(vote-req)}{\longrightarrow} \mathcal{G}_S$ = YES
12 $\quad \mathcal{G}_S$.ROLLBACK($PO_{\mathcal{G}_S}$) ▷ undo $\mathcal{G}_S.preLock(a)$
13 **end if**
14 $lock = \mathcal{G}_S$.LOCK(a) ▷ step 3.3
15 $PO_{\mathcal{G}_S}$.append(lock)
16 $commit = \mathcal{G}_S$.COMMIT() ▷ step 3.4
17 **if** $commit = \perp$ **then**
18 $\quad \mathcal{G}_S \overset{abort()}{\longrightarrow} \mathcal{G}_R$
19 $\quad \mathcal{G}_S.rollback(PO_{\mathcal{G}_S})$ ▷ undo $\mathcal{G}_S$.LOCK(a)
20 **end if**
21 $\mathcal{G}_S \overset{commit}{\longrightarrow} \mathcal{G}_R$
22 $a' = \mathcal{G}_R$.CREATE_ASSET() ▷ step 3.5
23 $PO_{\mathcal{G}_R}$.append(a')
24 **wait until** $\mathcal{G}_R \overset{\alpha(commit)}{\longrightarrow} \mathcal{G}_S$ ▷ step 3.6
25 **if** $\mathcal{G}_R \overset{\alpha(commit)}{\longrightarrow} \mathcal{G}_S = COMMIT$ **then**
26 $\quad \mathcal{G}_S$.COMPLETE() ▷ step 3.8
27 **end if**
28 **else**
29 $\quad \mathcal{G}_S \overset{abort()}{\longrightarrow} \mathcal{G}_R$ ▷ otherwise, $\mathcal{G}_R$ failed the commit
30 $\quad \mathcal{G}_S$.ROLLBACK($PO_{\mathcal{G}_S}$) ▷ undo $\mathcal{G}_S$ locks
31 $\quad \mathcal{G}_R$.ROLLBACK($PO_{\mathcal{G}_R}$) ▷ undo $\mathcal{G}_R$.CREATE_ASSET()
32 **end if**
33 return ▷ asset transferred

Algorithm 1 depicts the ODAP-2PC. A coordinator $\mathcal{G}_S$ and a participant $\mathcal{G}_R$ perform a CC-Tx T, that typically is an asset transfer of x number of a assets, i.e., $T : \mathcal{G}_S \overset{a,x}{\rightarrow} \mathcal{G}_R$. Any time a party ABORTS, the protocol stops, and that transaction is considered invalid (and thus the run of the protocol fails). We define a set of gateway primitives Σ = {PRE_LOCK, UNLOCK, LOCK, COMMIT, CREATE_ASSET, COMPLETE, ROLLBACK}, such that they realize pre-locking an asset, locking an asset, unlocking an asset, committing to a CC-Tx, creating an asset, asserting for the end of the protocol, and performing a rollback, respectively. The gateway primitives are divided into two types: off-chain primitives, and on-chain primitives, represented by $\sigma^{offchain}$ and $\sigma^{onchain}$, respectively. Some off-chain primitives call their respective on-chain primitive. The protocol receives a set of gateway primitives that realize the commit, locking, rollback and other operations. Lists $PO_{\mathcal{G}_S}$ and $PO_{\mathcal{G}_R}$ track the operations to be rolledback in case of failure for $\mathcal{G}_S$or $\mathcal{G}_R$, respectively.

First, in the session opening, the asset to be transferred is agreed on. At the pre-voting phase, the source gateway initiates the process, pre-locking an asset (executing the transaction right to the point before its commitment, at step 2.3, line 4). The recipient gateway confirms this pre-locking, issuing a VOTE-REQ to its counterparty (line 7). The recipient gateway replies either YES or ABORT (line 8), starting the decision phase. Note that the eventual ABORT, at line 8, does not require a rollback, because so far no on-chain operations took place. At the beginning of the decision phase, if $\mathcal{G}_R$ replies NO, then the pre-lock is rolledback, and the transaction aborted (lines 11 and 12). Otherwise, $\mathcal{G}_S$ tries to lock the asset to be transferred (line 14) and commit that action (line 16). The recipient gateway completes the pending transactions (line 22) and sends an acknowledgment message back to the source gateway (line 24). Upon the second commit, the source gateway completes the process, closing the session (line 26). However, if $\mathcal{G}_S$ cannot commit (line 25 is not COMMIT), the transaction is aborted, and the respective rollbacks are triggered.

If the participant $\mathcal{G}_R$ does not reply on the blocking operations (within $t < \delta_R$, $\mathcal{G}_S$ considers $\mathcal{G}_R$ crashed, and starts the *recovery protocol*). The recovery protocol may be trivial: in ODAP-2PC, firstly, the gateway awaits for the counterparty gateway to recover (by assumption, it does). Upon recovery, the process depicted by steps 4-11 from Figure 6 take place. Conversely, if $\mathcal{G}_S$ does not respond within $t < \delta_S$, the same process occurs. It is worth noting that the coordinator may issue the rollback at any point $t > \delta_{rollback}$, where $\delta_{rollback} > \delta_R$, i.e., it does not need to wait indefinitely for the participant to recover. For both cases, if the recovering awaiting period is greater than the rollback timeout protocol, i.e., $t > \delta_{rollback}$, the *rollback protocol* is triggered.

Consistency

ODAP-2PC provides transaction consistency by employing a self-healing strategy based on a write-ahead log: all parties either COMMIT or ABORT the CC-Tx (i.e., the asset transfer). The rollback protocol assures the consistency of the underlying DLTs.

Theorem 2 (Termination): Let there be an instantiation of ODAP-2PC in the self-healing mode, with a coordinator $\mathcal{G}_S$ and a participant $\mathcal{G}_R$, operating on an asynchronous environment. Given a coordinator timeout δ_S and a participant timeout δ_R, ODAP-2PC assures that ODAP terminates.

Proof: At various points of the protocol, both $\mathcal{G}_S$ and $\mathcal{G}_R$ are waiting for messages before proceeding, in particular at lines 3, 4, and 17. In line 3, $\mathcal{G}_R$ waits for a VOTE-REQ message. Since this gateway can decide to abort before it votes YES, if it has the timeout action triggered, it can abort and stop the process. In line 4, $\mathcal{G}_S$ is waiting for a YES or NO message. At this stage, there is still no decision on how to proceed ($\mathcal{G}_R$ still did not decide to COMMIT). Thus, the coordinator can decide abort in case of timeout by sending ABORT to the other gateway and stopping the process. In line 17, in case it voted YES, gateway $\mathcal{G}_R$ is waiting for a COMMIT or ABORT message. In case of a crash, $\mathcal{G}_R$ gateway remains blocked until $\mathcal{G}_S$ recovers. By assumption, there is an upper bound in which gateways recover from crashes. Thus, gateways will be able to communicate and thereby reach a decision. □

Termination

ODAP-2PC does not block indefinitely, providing liveness regarding its termination.

2) Rollback Protocol: The process of rolling back blockchain-based transactions is not trivial. As most blockchains are immutable, rolling back means issuing a transaction with the opposite effect of the first. We call this a *cancelling* a transaction. For example, cancelling a PRE_LOCK(a) and LOCK would imply issuing a transaction unlocking a, whereas CREATE_ASSET would imply the destruction of a created asset. The rollback protocol includes two parties: the cancelling gateway, and the counterparty gateway. The cancelling gateway realizes the need of cancelling one or more transactions, initiating the rollback protocol, and propagating eventual corrective measure commands to the counterparty gateway. There is a need of involving a counterparty gateway, to ensure the consistency of the protocol.

The rollback process occurs as follows: 1) the cancelling gateway undoes the transactions to be rolledback, by issuing transactions with the contrary effect; 2) the same gateway sends an acknowledgment back to the conterparty gateway; and 3) conterparty gateway undoes all its pending transactions, which can lead back to step one, where the counterparty gateway serves as the cancelling gateway. This recursive protocol may generates a cascade effect where several transactions from both blockchains need to be cancelled. Our rollback protocol is triggered at step 3.3 or 3.5. At step 2.3, if a lock is unsuccessful, there is still no transaction to undo (an ABORT is sent). Steps 2.4 and 3.7 are assumed to be successful, i.e., issuing a transaction that creates a log entry succeeds. In particular, if the log entry cannot be persisted in the blockchain support, an alternative support is used by the Log Storage API, and the respective party is warned.

Atomicity

The ODAP-2PC protocol provides transaction atomicity,

It is worth noting that the ODAP-2PC, and its rollback protocol depend on the implementation of a set of gateway primitives, as well as a specific asset schema. In the next section, we briefly present a use case leveraging gateway primitives.

V. Use Case: Gateway-Supported Cross-Jurisdiction Promissory Notes

In this section, we present a use case implementing digital asset transfers, benefiting from the gateway paradigm. The digital assets to be exchanged are defined as an *asset profile*, which is ongoing work at the IETF [37]. An asset profile is "the prospectus of a regulated asset that includes information and resources describing the virtual asset". A virtual asset, on its turn, is "a digital representation of value that can be digitally traded" [37]. Asset profiles can be emitted by authorized parties, having the capability to legally represent real-world assets (e.g., real estate).

A. Asset Profile

The *Asset Profile Definitions for DLT Interoperability* draft presents an unambiguous manner of representing a digital asset, independently of its concrete implementation [37]. This is notably for tokenization, as a physical asset might be represented in a multitude of ways. Thus, it is important to find a sufficiently generic schema that allows representing an arbitrary digital asset, and thus enable asset transfers. Perhaps most importantly, its definition assures that heterogenous DLTs refer to the same asset within a transfer. An asset profile contains the following fields (from [37]): issuer, asset code, asset code type, issuance and expiration dates, verification endpoint, digital signature, ledger requirements, among others.

Asset Profile Schema

1) **Issuer**: The registered name or legal identifier of the entity issuing this asset profile document.
2) **Asset Code**: The unique asset code under an authoritative namespace assigned to the virtual asset.
3) **Asset Code Type**: The code-type to which the asset code belongs under an authoritative namespace.
4) **Issuance date**: The issuance date of the Asset Profile JSON document.
5) **Expiration date**: The expiration of the Asset Profile JSON document in terms of months or years.
6) **Verification Endpoint**: The URL endpoint where anyone can check the current validity status of the Asset Profile JSON file.
7) **Digital signature**: The signature of the Issuer of the Asset Profile.
8) **Prospectus Link**: The link to any officially published prospectus, or non-applicable.
9) **Key Information Link**: The link to any Key Information Document (KID), or non-applicable.
10) **Keywords**: The list of keywords to make the Asset Profiles easily searchable. Can be blank or non-applicable.
11) **Transfer Restriction**: Information about transfer restrictions (e.g. prohibited jurisdictions etc.), or non-applicable.
12) **Ledger Requirements**: Information about the specific ledger mechanical requirement, or non-applicable..

We refer to this asset profile as $\mathcal{A}_p$. For generic protocols manipulating assets (e.g., transfer, creating), this asset profile can provide the necessary attributes for trust establishment. For instance, gateways should be able to verify its counter party identity in case of an asset transfer. Moreover, the asset profile and asset code should be identifiable and retrievable, allowing different attributes to be parsed as inputs to the asset gateway primitives.

B. Asset Gateway Primitives

Based on the proposed digital asset schema, we present pseudo-code for the gateway primitives used in ODAP-2PC. We recall the gateway primitives: off-chain primitives (COMMIT, ROLLBACK, and COMPLETE) and on-chain primitives (PRE-LOCK, LOCK, UNLOCK, and CREATE_ASSET). The sequencing of off-chain operations, performed by gateways, and on-chain operations allow the asset transfer. For instance, based on a specific asset profile $\mathcal{A}_p$, gateways validate eventual restrictions (e.g., jurisdiction restrictions) on a certain asset, at the validation phase, prior to PRE_LOCK an asset (in case the protocol comprises transferring an asset).

To implement the primitives, we define an additional field on $\mathcal{A}_p$ to represent a digital asset: *state*. Four possible states exist: an asset is unlocked (can be used without constraints on that ledger), pre-locked (asset will be transfered, and thus cannot be used), locked (asset was transfered, and cannot be used), and burnt (asset was destroyed or permanently locked).

Algorithm 2: On-chain set state

Input: Asset a, Ledger connector c, lock level l
Result: Asset a locked at l

```
assetRepresentation = c.getStateById(a.assetCode)   ▷ DLT-specific
assetRepresentation.state = l                       ▷ pre-lock, lock, unlocked, burnt
c.setState(assetRepresentation)                     ▷ DLT-specific
```

Algorithm 2 depicts the procedure to implement a PRE-LOCK, LOCK, and UNLOCK, if the level is pre-lock, lock, or unlock, respectively. If the level is burnt, then and additional, DLT-specific operation needs to happen to eliminate (burn) the asset. The PRE_LOCK primitive issues a LOCK, temporarily locking an asset on $\mathcal{G}_S$, setting the state of the asset to *pre-locked*. After that, the gateway awaits for a confirmation from the counterparty gateway of such operation. In case the protocol fails before COMMIT, a ROLLBACK is issued by $\mathcal{G}_S$, triggering an UNLOCK transaction. The UNLOCK simply sets the state of the pre-locked asset to *unlocked*, reverting the effect of the PRE-LOCK.

If a COMMIT is successful, then two operations happen: 1) in $\mathcal{G}_S$ a LOCK is issued, setting the state of the asset to *locked*, meaning it cannot be used; 2) $\mathcal{G}_R$ issues a CREATE_ASSET, creating a representation of the original asset on the recipient ledger. If the whole process is successful, according to ODAP-2PC, $\mathcal{G}_S$ issues a COMPLETE. All operations are logged via the log storage API; an additional on-chain primitive LOG is considered in case the logging takes place on-chain.

C. Using Hermes to Exchange Promissory Notes

Promissory notes are freely transferable financial instruments where issuers denote a promise to pay another party (payee) [41]. Notes are globally standardized by several legal frameworks, providing a low-risk instrument to reclaim liquidity from debt. Notes contain information regarding the debt, such as the amount, interest rate, maturity date, and issuance place. Notes are useful because they allow parties to liquidate the debts and conduct financial transactions faster, overcoming market inefficiencies. In practice, promissory notes can be both payment and credit instruments. A promissory note typically contains all the terms about the indebtedness, such as the principal amount, credit rating, interest rate, expiry date, date of issuance, and issuer's signature. Despite their benefits, paper promissory notes are hard to track, require hand signatures and not-forgery proofs, accounting for cumbersome management. To address these challenges, recent advances in promissory notes' digitalization include FQX's eNote [17]. Blockchain-supported digital promissory notes (eNotes) worth about half a million dollars were used by a "Swiss commodity trader to finance a transatlantic metal shipment" [2]. eNotes are stored in a trusted ledger covered by the legal framework, belonging to a specific jurisdiction. Consider the following supply chain scenario: a producer (P) produces a certain amount of goods that sells to a wholesaler (W). W accepted the goods, and now P issues an invoice of value V. The wholesaler could pay in, for example, 90 days. Because P does not want to wait up to 90 days for its payment, it requests a promissory note from W, stating that V will be paid in 90 days. This way, P can sell that same promissory note to a third party. The promissory note is abstract from any physical good being exchanged. Depending on the issuer, collateral might not be needed, as the accountability for liquidating the debt is tracked by the blockchain where it is stored.

Promissory Note Example

1) **Issuer**: FQX AG
2) **Asset Code**: CH0008742519
3) **Asset Code Type**: ISIN
4) **Keywords**: Electronic Promissory Note; eNote; Debt
5) **Prospectus Link**: N/A
6) **Key Information Link**: N/A
7) **Transfer Restriction**: shall not be transferred to the U.S., Canada, Japan, United Kingdom, South Africa. Shall not be transferred to non-qualified investors anywhere.
8) **Ledger Requirements**: Hyperledger Fabric v2.x.
9) **Original Asset Location**: N/A
10) **Previous Asset Location**: N/A
11) **Issuance date**: 04.09.2020
12) **Verification Endpoint**: https://fqx.ch/profile-validate
13) **Signature Value**: (signature blob)

Blockchain-based promissory notes belonging to a particular jurisdiction are stored in a certified blockchain that exposes a gateway. When a promissory note needs to change jurisdictions (e.g., a promissory note issued in the USA that needs to be redeemed in Europe), the gateways belonging to the source and target blockchains perform an asset transfer, where the asset is a digital promissory note. Alternatively, the gateway extends to several jurisdictions. Below is an example of an asset profile of a digital promissory note. Such digital promissory notes can be trivially exchanged between blockchains using Hermes and the ODAP-2PC protocol, where gateways belonging to different jurisdictions (e.g., representing different blockchains regulated by different entities) perform asset transfers.

VI. Discussion

Hermes can include different gateway implementations, different gateway protocols, and different distributed recovery mechanisms. Modularity and pluggability allow Hermes to be flexible regarding different legal frameworks, supporting different privacy and performance requirements. In particular, Hermes can be instantiated in blockchains supporting smart contracts that implement functionality for locking and unlocking assets. The gateway paradigm allows integrating DLT-based systems to centralized legacy systems by leveraging existing legal frameworks. For extra robustness, data integrity and counterparty performance can be attested, using trusted hardware [21], [14]. Remote attestations are particularly important, since provably exposing internal state to external parties is a crucial requirement for CC-Txs [4].

Gateways can also be leveraged for tasks other than asset transfers; they can perform the function of oracles, either centralized or decentralized [9], allowing to integrate blockchains with external systems and data providers. An oracle's general goal is to retrieve data, validate and deliver it to a blockchain, or pull information from a blockchain [28]. An oracle may provide extra functions, such as showing proof of original data, incentivizing oracle services (e.g., rewarding nodes providing information to the oracle), and even privacy (encrypting data). As a gateway, Hermes can implement asset transfers through the ODAP protocol or serve as an oracle.

A. RQ1: Reliability

On primary-backup mode, n-host resiliency is provided by sequencing backups and using acknowledgment messages. These messages assure that the update has progressed at least to the following backup beyond itself. However, primary backup introduces a latency overhead, as the client application only retrieves the output from the message update request after n replicas have been updated. On the other hand, the self-healing mechanism, allied to a resilient log storage API, provides means for developers to save the ODAP state, even in the presence of crashes.

ODAP and ODAP-2PC assume a trade-off between reliability and efficiency, according to the end to end principle [36]. The more reliable a gateway is (in terms of accountability, termination, and ACID properties), the higher the overhead is in terms of performance. The storage capability of gateways, abstracted by the log storage API, determine gateways' robustness, as logs are used to dispute resolution and accountability. Shared, non-repudiable, and immutable log entries provide better guarantees than locally stored logs [32], [8]. Thus, the log storage API serves two purposes: 1) it provides a reliable mean to store logs created by all gateways involved in an asset transfer, and thus ensures consistency, atomicity, and isolation; and 2) promotes accountability across parties.

B. RQ2: Performance

As mentioned, a tradeoff between reliability and performance exists. Storing logs in local storage typically has lower latency but deliver weaker integrity and availability guarantees than store them on the cloud or in a ledger. Generally, the more resilient the support γ is, the higher the latency ($\gamma_{bc} > \gamma_{cloud} > \gamma_{local}$). For critical scenarios where strong accountability and traceability are needed (e.g., financial institution gateways), blockchain-based logging storage may be appropriate. Conversely, for gateways that implement interoperability between blockchains belonging to the same organization (i.e., a legal framework protects the legal entities involved), local storage might suffice.

ODAP-2PC exchanges messages to assure atomicity, leading to blocking operations, where operations depend on the state of the other gateway. In particular, γ_{bc} implies issuing a blockchain transaction, several orders of magnitude slower than writing on disk or even writing on a cloud-based storage [12], especially if one awaits for a confirmation, depending on the blockchain, it may require up to dozens of minutes. The self-healing mode is compatible with the three types of logs, but the primary-backup mode could require the log storage API on support external to the gateway.

C. RQ3: Decentralization

Gateway-to-gateway business transactions depend on the social and technological trust that stakeholders build. In particular, as every operation is saved on a log, this log can be used for disputes, in case of misbehavior by any stakeholder. In particular, in case of dispute, the involved parties can inspect the logs and recur to the legal frameworks [8] from the jurisdiction in which the asset transfer occurs. Thus, for the legislated spaces, and for a proper log storage support, Hermes might sufficiently decentralized. While this is acceptable for enterprise scenarios, as accountability is guaranteed, there may be cases in which gateways are not trusted. Considering non-trusting gateways, Hermes might not be sufficiently decentralized. Besides picking the appropriate log storage support, one could choose from several techniques to decentralize gateways or to enhance the accountability level.

A first option is to implement a gateway as a smart contract: this does not allow a gateway to deviate from its configured behavior but has shortcomings, such as inflexibility, lack of scalability, and operation costs. In particular, smart contracts often lack the possibility of being integrated with external resources and systems; oracles may provide some extra flexibility [9]. Smart contract-based gateways could also need to pay transaction fees in public blockchains, such as gas on Ethereum [42], raising additional costs. Additional costs imply that adding gateways on the same blockchain is not scalable.

Second, to decentralize Hermes, one could implement a Byzantine fault-tolerant version of a gateway, similarly to what is planned on Cactus [27]. In this case, it is not a single gateway conducting the message delivery process, but a quorum of gateways that belong to different stakeholders. In a permissioned scenario, stakeholders could represent different departments, with the caveat should periodically publish proofs of state in an external repository [32]. If gateways are sufficiently decentralized, gateways do not need to be implemented as smart contracts. This allows better scalability than the smart contract and flexibility in integrating legacy systems and infrastructure with the gateways.

A third option is to secure computation leveraging trusted hardware to enable remote attestation [21], [14]. Remote attestation is a method allowing a device to authenticate its hardware and software to a centralized service, proving its integrity, and thus its trustworthiness. Working as an additional security layer, device-level attestations would enable gateways to provide truthful evidence of their internal state. Evidence would then promote trust across gateways, diminishing the risk of collusion and misbehavior. This solution would be essential for financial institution gateways, involving digital asset transfers with monetary value.

D. RQ4: Security and Privacy

Gateways should assure the integrity and non-repudiation of log entries and ensure that the protocol terminates. If an adversary performs a denial-of-service on either gateway, the asset transfer is denied but ODAP-2PC assures eventual consistency of the underlying DLTs. Accountability promoted by robust storage can diminish the impact of these attacks. The connection between gateways should always provide an authentication and authorization scheme, e.g., based on OAuth and OIDC [1], and use secure channels based on TLS/HTTPS [34].

Gateways should be flexible enough to accommodate not only different legal frameworks but also different notions of privacy. Reasoning about different privacy levels, one key question is: what should be the privacy granularity level regarding an issuer and beneficiary transaction of a digital asset? Some regulations imply that both parties are identified, and such records are maintained for several years; However, for cryptocurrency exchange

across public blockchains, privacy might be of a more significant concern. A second question follows: what are the privacy guarantees of the gateway performing such transfers, mainly if logging functions are jointly performed, on blockchain-based support? This question can be answered with privacy-policies, and cherry-picking the information written in publicly-available logs. Future research on the security and privacy of gateways is needed before they are ready for production use.

Other privacy-related aspect is the encapsulation of internal asset representation. Although gateways are working with a specific asset schema, each gateway needs to be aware of the asset representation by the underlying DLT (or at least DLT client), i.e., it needs to be able to convert ODAP messages to blockchain-specificc transactions. Thus, the gateway has the responsability of converting a standard representation on a DLT-specific one. If desirable, gateways can hide representation details, providing privacy regarding asset management.

VII. Related Work

Hardjono et al. proposed a gateway-based architecture inspired by the architecture of the Internet [20], further expanded by recent work [19]. Vo et al. propose decentralized blockchain registries that can identify and address blockchain oracles [38]. Such registries can be extended to support gateways. Hyperledger Cactus [27] is a trusted relay connecting DLTs, whereby a consortium of Cactus Nodes endorses transactions. Cactus uses two families of software components that, on its sum, constitute a gateway: validators and connectors. Validators are components that retrieve state from blockchains, while connectors are active components that issue transactions. The consortium can run arbitrary business logic, including logic for asset transfers, making Cactus a suitable infrastructure to implement gateways. Like Cactus, HERMES is a trusted relay directed to enterprise use cases. Our system can be decentralized using one of the approaches detailed in Section VI. Other trusted relays can realize the concept of gateway (e.g., [3], [4], [24], [16]). For the sake of space, we refer readers interested in interoperability to [9].

Generally, 2PC is not used for blockchain consensus [29], but rather for communication across blockchains. Fynn et al. presented a Move operation that can migrate accounts and arbitrary computation across Ethereum virtual machine based chains [18]. An atomic Move operation can be implemented with 2PC. Wang et al. [40] presented a 2PC protocol for conducting CB-Tx. In this scheme, a blockchain is elected as the coordinator, managing the process between an arbitrary number of blockchains. This protocol includes a heartbeat monitoring mechanism to guarantee liveness. However, it is not clear how are ACID properties assured, e.g., atomicity, as the authors do not provide a rollback protocol. Our work provides ACID properties via ODAP-2PC and the rollback protocol.

VIII. Conclusion

This paper introduced HERMES, a middleware that enables blockchain interoperability across DLT-systems that can operate under different legal frameworks. HERMES is instantiated with ODAP, an asset transfer protocol between two gateway devices. Hermes supports ACID properties and can assure accountability by keeping an off-chain or on-chain shared log of operations. We propose and discuss ODAP-2PC, a distributed recovery mechanism, guaranteeing asset transfers between blockchains to be atomic and secure. By studying Hermes' reliability, performance, decentralization, security, and privacy, we explore the potential of gateways to respond to the current interoperability challenge. By presenting the digital promissory note use case, we show that Hermes is an appropriate trust anchor for enterprise use cases requiring cross-blockchain asset transfers. Future work will enable several gateways to be involved in an asset transfer (ODAP-3PC), paving the way for efficient multiparty atomic swaps.

References

[1] Final: OpenID Connect Core 1.0 incorporating errata set 1.

[2] Transatlantic Shipment of Metals Financed via FQX eNote — Treasury Management International.

[3] E. Abebe, D. Behl, C. Govindarajan, Y. Hu, D. Karunamoorthy, P. Novotny, V. Pandit, V. Ramakrishna, and C. Vecchiola. Enabling Enterprise Blockchain Interoperability with Trusted Data Transfer. In *Proceedings of the 20th International Middleware Conference Industrial Track*, pages 29–35. Association for Computing Machinery, 2019.

[4] E. Abebe, D. Karunamoorthy, J. Yu, Y. Hu, V. Pandit, A. Irvin, and V. Ramakrishna. Verifiable Observation of Permissioned Ledgers. *arXiv 2012.07339v2*, 2021.

[5] P. ALsberg and J. Day. A Principle for Resilient Sharing of Distributed Resources. *Journal of Chemical Information and Modeling*, 1976.

[6] E. Androulaki, A. Barger, V. Bortnikov, S. Muralidharan, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Murthy, C. Ferris, G. Laventman, Y. Manevich, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th EuroSys Conference, EuroSys 2018*, pages 1–15, New York, New York, USA, apr 2018. Association for Computing Machinery, Inc.

[7] R. Belchior, M. Correia, and T. Hardjono. DLT Gateway Crash Recovery Mechanism (draft-belchior-gateway-recovery-00). Technical report, 2021.

[8] R. Belchior, A. Vasconcelos, and M. Correia. Towards Secure, Decentralized, and Automatic Audits with Blockchain. In *European Conference on Information Systems*, 2020.

[9] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia. A Survey on Blockchain Interoperability: Past, Present, and Future Trends. *arXiv 2005.14282*, 2020.

[10] M. Belotti, S. Moretti, M. Potop-Butucaru, and S. Secci. Game Theoretical Analysis of Atomic Cross-Chain Swaps. *Hal Archives-Ouverte hal-02414356*, 2020.

[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[12] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *ACM Transactions on Storage*, volume 9, pages 1–33, nov 2013.

[13] M. Borkowski, M. Sigwart, P. Frauenthaler, T. Hukkinen, and S. Schulte. DeXTT: Deterministic Cross-Blockchain Token Transfers. *IEEE Access*, 7:111030–111042, aug 2019.

[14] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10(2):63–81, jun 2011.

[15] E. D. Hardt. RFC 6749 - The OAuth 2.0 Authorization Framework, 2012.

[16] G. Falazi, U. Breitenbücher, F. Daniel, A. Lamparelli, F. Leymann, and V. Yussupov. Smart Contract Invocation Protocol (SCIP): A Protocol for the Uniform Integration of Heterogeneous Blockchain Smart Contracts. In *International Conference on Advanced Information Systems Engineering*, volume 12127 LNCS, pages 134–149, 2020.

[17] FQX. eNI™ Infrastructure - fqx.ch - Electronic Negotiable Instruments - FQX, 2020.

[18] E. Fynn, F. Pedone, and B. Alysson. Smart Contracts on the Move. In *Dependable Systems and Networks*, 2020.

[19] T. Hardjono. Blockchain Gateways, Bridges and Delegated Hash-Locks. *arXiv 2102.03933*, 2021.

[20] T. Hardjono, A. Lipton, and A. Pentland. Towards an Interoperability Architecture Blockchain Autonomous Systems. *IEEE Transactions on Engineering Management*, 67(4):1298–1309, June 2019.

[21] T. Hardjono and N. Smith. Towards an Attestation Architecture for Blockchain Networks (to appear). *World Wide Web Journal – Special Issue on Emerging Blockchain Applications and Technology*, 2021.
[22] M. Hargreaves and T. Hardjono. Open Digital Asset Protocol (draft-hargreaves-odap-01), 2020.
[23] D. Johnson, A. Menezes, and S. Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, aug 2001.
[24] L. Kan, Y. Wei, A. Hafiz Muhammad, W. Siyuan, G. Linchao, and H. Kai. A Multiple Blockchains Architecture on Inter-Blockchain Communication. *Proceedings - 2018 IEEE 18th International Conference on Software Quality, Reliability, and Security Companion, QRS-C 2018*, pages 139–145, 2018.
[25] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, and Y. C. Hu. Hyperservice: Interoperability and programmability across heterogeneous blockchains. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 549–566, 2019.
[26] N. Lynch. Multilevel Atomicity—A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4):484–502, 1983.
[27] H. Montgomery, H. Borne-Pons, J. Hamilton, M. Bowman, P. Somogyvari, S. Fujimoto, T. Takeuchi, T. Kuhrt, and R. Belchior. Hyperledger Cactus Whitepaper, 2020.
[28] R. Mühlberger, S. Bachhofner, E. Castelló Ferrer, C. Di Ciccio, I. Weber, M. Wöhrer, and U. Zdun. Foundational Oracle Patterns: Connecting Blockchain to the Off-Chain World. In *Lecture Notes in Business Information Processing*, volume 393 LNBIP, pages 35–51. Springer Science and Business Media Deutschland GmbH, sep 2020.
[29] J. Nijsse and A. Litchfield. A Taxonomy of Blockchain Consensus Methods. *Cryptography*, 4(4):32, 2020.
[30] R. J. Patton. Fault-Tolerant Control: The 1997 Situation. *IFAC Proceedings Volumes*, 30(18):1029–1051, 1997.
[31] L. Pawczuk, M. Gogh, and N. Hewett. Inclusive Deployment of Blockchain for Supply Chains: A Framework for Blockchain Interoperability. Technical report, World Economic Forum, 2020.
[32] B. Putz, F. Menges, and G. Pernul. A secure and auditable logging infrastructure based on a permissioned blockchain. *Computers & Security*, 87:101602, 2019.
[33] Quant Foundation. Overledger Network Whitepaper v0.3. Technical report, Quant, 2019.
[34] E. Rescorla. RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3, 2014.
[35] K. Sai and D. Tipper. Disincentivizing Double Spend Attacks Across Interoperable Blockchains. In *First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications*, 2019.
[36] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, nov 1984.
[37] A. Sardon, T. Hardjono, and Benedikt Schuppli. Asset Profile Definitions for DLT Interoperability (draft-sardon-blockchain-interop-asset-profile-00). Technical report, 2021.
[38] H. Tam Vo, Z. Wang, D. Karunamoorthy, J. Wagner, E. Abebe, and M. Mohania. Internet of Blockchains: Techniques and Challenges Ahead. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1574–1581, 2018.
[39] S. Thomas and E. Schwartz. A Protocol for Interledger Payments, 2015.
[40] X. Wang, O. T. Tawose, F. Yan, and D. Zhao. Distributed Nonblocking Commit Protocols for Many-Party Cross-Blockchain Transactions, jan 2020.
[41] J. S. Waterman. The Promissory Note as a Substitute for Money. *Minnesota Law Review*, 14:313, 1929.
[42] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. Byzantium version 7e819ec. Technical report, 2019.

APPENDIX A - ODAP MESSAGE FORMAT

ODAP messages are exchanged between client applications and gateway servers (DLT nodes). They consist of functional messages allowing protocol negotiation [22]. Messages are encoded in JSON format, allowing for serialization, with protocol specific mandatory fields. Support for authentication and authorization is provided, allowing for plaintext or encrypted payloads. This servers enterprise needs.

Mandatory Log Fields for ODAP

1) **Version**: ODAP protocol Version (major, minor)
2) **Resource URL**: Location of Resource to be accessed.
3) **Developer URN**: Assertion of developer / application identity.
4) **Action/Response**: GET/POST and arguments (or Response Code)
5) **Credential Profile**: Specify type of auth (e.g. SAML, OAuth, X.509)
6) **Credential Block**: Credential token, certificate, string
7) **Payload Profile**: Asset Profile provenance and capabilities
8) **Application Profile**: Vendor or Application specific profile
9) **Payload**: Payload for POST, responses, and native DLT txns
10) **Sequence Number**: Sequence Number.