Future Generation Computer Systems 129 (2022) 236–251

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

HERMES: Fault-tolerant middleware for blockchain interoperability

Rafael Belchior [c,b,*], André Vasconcelos [c,b], Miguel Correia [c,b], Thomas Hardjono [a]

[a] MIT Connection Science & Engineering, Massachusetts Institute of Technology, Cambridge, USA
[b] Department of Computer Science and Engineering, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
[c] INESC-ID, Lisbon, Portugal

ARTICLE INFO

Article history:
Received 30 August 2021
Received in revised form 3 November 2021
Accepted 8 November 2021
Available online 8 December 2021

Keywords:
Distributed ledger technology
Blockchain
Interoperability
Digital asset

ABSTRACT

Blockchain interoperability reduces the risk of investing in blockchain systems by avoiding vendor lock-in, enabling a new digital economy, and providing migration capabilities. However, seamless interoperability among enterprises requires service providers to comply with different regulations, e.g., data privacy regulations and others that apply to financial services. For supporting interoperability, organizations can connect to each blockchain via a *gateway*. However, these gateways should be resilient to crashes to maintain a consistent state across ledgers. 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 transparency, auditability, availability, and non-repudiation. We briefly explore a use case for cross-jurisdiction asset transfers, illustrating how one can leverage HERMES to support cross-chain transactions compliant with legal and regulatory frameworks.

1. Introduction

There is an increasing interest in digital currencies and virtual assets as the foundation of the next generation digital economy. Blockchain technology has been shown to be a dependable enabler due to its properties, such as immutability, transparency, and auditability [1–3]. Both private organizations and governments are actively investigating and investing in blockchain-based digital assets by, for example, promoting new platforms for digital transactions [4]. A key challenge to enabling this digital economy is to safely connect different networks, enabling network effects among them [5–7].

Interoperability of *blockchains* is, therefore, key to the area [2, 8–10]. Hash lock time contracts, sidechains, oracles and relays, and exchanges are already allowing users to take advantage of this new digital economy in a permissionless environment (see [11] for a detailed survey on the topic).

Although significant progress on interoperability has been made, public blockchains, private blockchains, and legacy systems cannot communicate seamlessly yet [11]. Moreover, current solutions are not standardized and do not offer the possibility to seamlessly transfer data and value across legal jurisdictions, hampering enterprise adoption of blockchain. There is a need for building solutions capable of complying with legal frameworks and regulations.

We believe that similar to Internet routing gateways, which enabled interoperability around private networks, and fostered the rise of the Internet, the global network of decentralized ledgers (DLTs) will require blockchain gateways [2,6]. Gateways permit digital currencies and virtual assets to be transferred seamlessly between these systems. 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) [2]. ODAP is a cross-chain communication protocol handling multiple digital asset cross-border transactions by leveraging blockchain gateways. Gateways agree on the assets to be exchanged via an asset profile, i.e., a structured, regulation-compliant data model for representing assets (e.g., digital, physical). Transferring an asset among blockchains via gateways is equivalent to an atomic swap that locks an asset in a blockchain and creates its representation on another. However, how can one guarantee a fair exchange of assets (either all parties receive the assets they requested, or none do) across gateways?

To assure the properties that enable a fair exchange of assets, blockchain gateways must operate reliably and be able to withstand a variety of attacks. Thus, a crash-recovery strategy must be a core design factor of blockchain gateways, where specific recovery protocols can be designed as part of the digital asset transaction protocol between gateways. A recovery protocol, allied to a crash recovery strategy, guarantees that the source and

* Corresponding author at: Department of Computer Science and Engineering, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal.
E-mail address: rafael.belchior@tecnico.ulisboa.pt (R. Belchior).

https://doi.org/10.1016/j.future.2021.11.004
0167-739X/

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.

We present Hermes,[1] a middleware for blockchain interoperability that focuses on providing crash recovery capabilities to gateways. 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) [13,14]. Hermes also leverages a log storage API that persists evidence on asset transfer processes across gateways for posterior accountability and dispute-resolution. Hermes's architecture is layered, allowing for cross-chain logic (implemented by business logic plugins, or *BLPs* [15]) to be executed among gateways.

By modeling and developing this system, we expect to address four *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?
- *RQ4* What to expect in terms of security and privacy of gateway-based interoperability solutions?

The contributions of this paper are three-fold: first, the blockchain (or DLT) *gateway concept* is explained from a theoretical and practical perspective. Second, we present the Hermes fault-tolerant middleware and its main component, *ODAP-2PC*, a new protocol that provides ACID properties for cross-blockchain transactions. ODAP is also presented. A preliminary implementation of ODAP[2] is available at Hyperledger Cactus, an Hyperledger Foundation project dedicated to DLT interoperability.[3] We provide a comprehensive discussion on Hermes as a solution for blockchain interoperability, focusing on the four research questions we address. Third, and lastly, we present a use case on the exchange of promissory notes across jurisdictions. This use case illustrates how one can leverage Hermes to achieve blockchain interoperability compliant with legal and regulatory frameworks. To be clear, Hermes and ODAP-2PC are contributions of this paper, whereas ODAP is a protocol being designed in the context of the IETF, although by some of the authors of the present paper.

The rest of this paper is structured as follows: Section 2 presents the background. We introduce the gateway concept and Hermes, in Section 3. After, in Section 4, we present ODAP, including the message and logging procedure, the log storage API, and the distributed recovery protocol (Sections 4.2, 4.3, and 4.4, respectively). Section 5, presents a use case that benefits from Hermes. Section 6 presents our discussion on gateways, ODAP, and ODAP-2PC in the light of the presented research questions. The related work follows, in Section 7. Finally, we conclude the paper in Section 8.

2. Preliminaries

This section presents the background on fault tolerance, atomic commit, and nomenclature both on logging and blockchain interoperability.

[1] A preliminary, short, version of this paper appears in IEEE SCC 2021 [12].

[2] We plan to study the performance of our implementation as future work.

[3] https://github.com/hyperledger/cactus.

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 [13].

Fault recovery

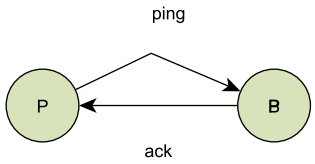
Typically, crash fault-tolerant (CFT) services can tolerate $\frac{n}{2}$ nodes crashing, with n being the number of nodes. As long as there is a majority of nodes with the latest state, failures can be tolerated. The *primary-backup model* defines a set of n hosts (or nodes) that, as a group, assures 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) $\mathcal{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$, 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 of 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.

Another recovery mechanism is *self-healing* [13]. In self-healing, when nodes crash, they are assumed to recover eventually. While this mode is cheaper than primary backup, fewer nodes, fewer exchanged messages, and lower storage requirements, it comes at the expense of availability. In particular, the protocol may block until nodes recover. Fig. 1 depicts a simplified self healing protocol for two nodes. Node P sends a ping to node B, which responds with an ACK. In case of a crash, node B awaits a ping.

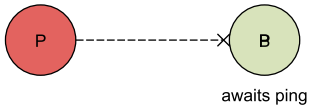
Atomic commit protocols

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 is persisted before executed). Examples of ACPs are the two-phase commit protocol, 2PC, the three-phase commit protocol, 3PC, and non-blocking atomic commit protocols [13].

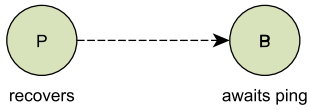
2PC achieves atomicity even in case of temporary system failure, accounting for a wide adoption in academia and in the industry. It has two phases: the voting phase and the commit phase. In the voting phase, the Coordinator prepares all participants to take place in a distributed transaction by inspecting 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



(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.

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 $\{l_1, l_2, \ldots, l_n\}$ 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 nth step of process p as (n, p). We denote the ith 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 process p. We define five operations types to provide context to the protocol being executed:

- Operation type *init*- states the intention of a node to execute a particular operation.
- Operation type *exec*- expresses that the node is executing the operation.
- Operation type *done*- states when a node successfully executed a step of the protocol.
- Operation type *ack*- refers to when a node acknowledges a message received from another.
- Operation type *fail*- indicates 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 omitted), respectively.

Fig. 2 illustrates the logging procedure of some process (or protocol) A, executed by two nodes: Node and Node 2. Process A has three steps. While typically each gateway has its 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: Cryptocurrency-directed interoperability approaches, Blockchain Engines, and Blockchain Connectors [11]. Cryptocurrency-directed approaches enable 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 challenging to integrate with permissioned blockchains that support arbitrary assets and smart contracts.

The second category is the blockchain engines, enabling an 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 other blockchains.

The third category, blockchain connectors, includes 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.

3. Hermes

In this section, we introduce out interoperability middleware, Hermes.

3.1. The concept of gateway

A *gateway* is a hardware device running software capable of interacting with blockchains (e.g., issuing transactions, reading state), and performing computation based on such interactions. Depending on the distributed ledger gateways are connected, they might need to be full nodes, i.e., they may need to implement the whole functionality of a node of that blockchain (e.g., Ethereum). By being present in different DLTs, gateways can perform *cross-chain transactions* (CC-Tx), i.e., transactions including both blockchains, including asset transfers [6]. 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-to-gateway protocol, e.g., an asset transfer, data pushing/pulling. Gateways use

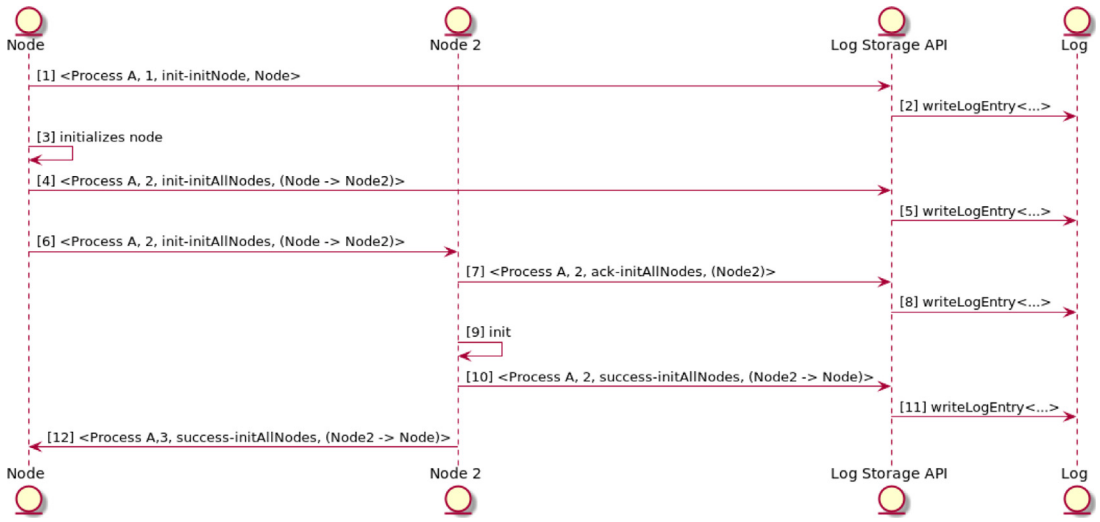


Fig. 2. Example of logging in the context of a process/protocol *A* executed by two nodes. The log storage API writes the incoming message. A single log represents the logs of nodes Node and Node2, but the intent is clear.

machine-resolvable addresses (e.g., URIs/URLs) to communicate 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-to-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-to-gateway protocol specifies the set of messages and procedures between two gateways for their correct functioning. The gateway-to-gateway protocol considered in this paper is ODAP [2].

3.2. The architecture of hermes

Hermes is a middleware that enables DLT interoperability by implementing part of the software component of gateways. ODAP defines the set of messages (the protocol) for asset transfers at the base layer, realizing technical interoperability. On top of it, ODAP-2PC, a fault-tolerant gateway-to-gateway protocol, provides reliability in the presence of crashes. In case a cross-chain transaction is aborted, ODAP-2PC attempts to issue a rollback on the affected DLTs. ODAP-2PC also provides support for disputes and accountability by providing logging capabilities via the log storage API. Finally, business logic plugins can be implemented (i.e., asset transfer), providing the core rules for a gateway to operate (when to initiate or refuse a transfer). This layer implements semantic interoperability. Clients can use Hermes to support standards that a specific gateway implementation needs to comply with (e.g., Travel Rule [16]). Fig. 3 represents the layers of Hermes.

Our architecture is flexible and modular, as its components are pluggable. By decoupling the protocol from the crash recovery component, and the latter from the business logic plugins, our system can be adapted to specific needs. For instance, in a trustless environment, where gateways do not fully trust each other, a gateway might have a more robust fault recovery mechanism; conversely, if gateways operate in a permissioned environment and completely trust each other, logging capabilities might be reduced to a local log. In this paper, we instantiate Hermes with ODAP and its crash fault-tolerant distributed recovery protocol, ODAP-2PC. The chosen business logic plugin allows promissory note exchanges, presented in detail in Section 5. We presented

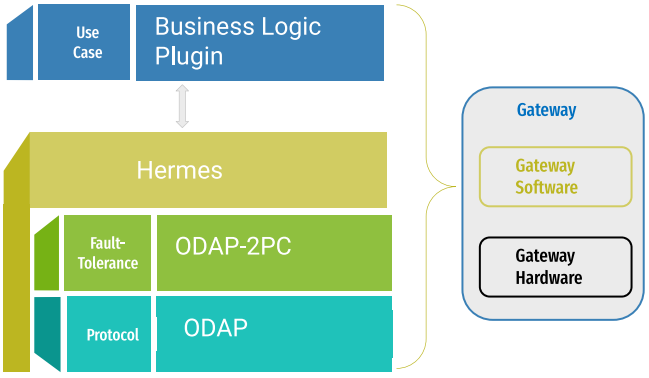


Fig. 3. Hermes's layers.

the architecture of a single Hermes-enabled gateway. In Fig. 4, we present a network compressed of two organizations (A and B), each one with its gateway (Gateway A and Gateway B, respectively). Gateway A is connected to DLT 1, while Gateway B is connected to DLT 2, and a centralized system (e.g., invoice system). This network connects data and assets from DLT 1 to DLT 2 (via Gateway A), DLT 2 to DLT 1 (via Gateway B).

Gateway A establishes a connection to Gateway B via ODAP, exchanging protocol messages. Each gateway has an instance of ODAP-2PC, that guarantees the current state to be preserved. State is written to and read from the distributed log storage, i.e., DLT-based or cloud-based. This storage is accessible by both gateways. Each gateway has its local log, where private information on the gateways operations might be saved (e.g., for data analytics). Hermes redirects ODAP messages to business logic plugins that, in its turn, issue transactions against distributed ledgers (or centralized systems).

3.3. System model

We consider a partially synchronous distributed system (there are unknown bounds on transmission delay and processing time) composed of two types of participants: clients and gateways. Participants have access to a globally synchronized clock (although minor deviations are tolerated).

Clients are in charge of starting transactions and are connected to gateways that are connected to blockchains. More specifically,

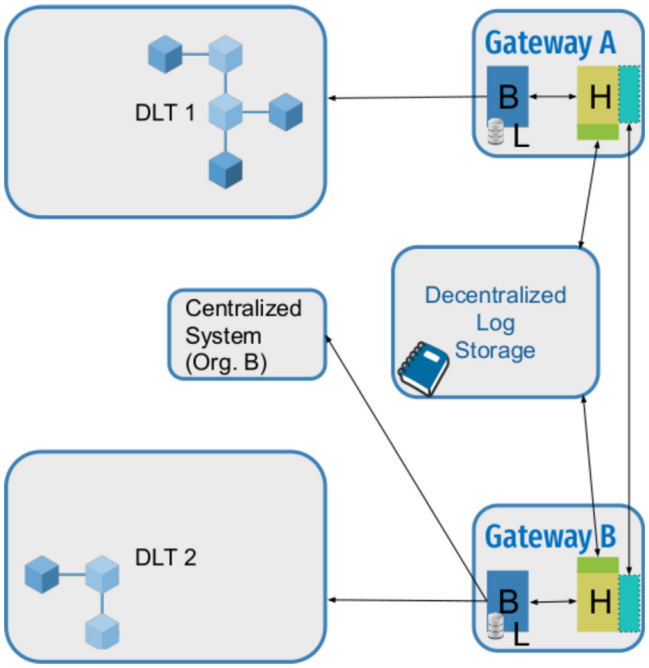


Fig. 4. A Hermes-powered gateway network compressed of two gateways. Within each gateway, the dark blue box represents a business logic plugin (B); the local log is represented by L; the yellow box represents Hermes (H); the light blue box represents ODAP (dotted border); the green box represents ODAP-2PC.

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.

Gateways store log data about the step of their protocol. This information allows a gateway to construct a state, and recover, in case of a crashe. 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 decide on a CC-Tx reach the same, either commit or abort. The state of the underlying ledgers reflects 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.

3.4. 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 ascertain the authenticity validity of the communication. Each gateway has a public and private key pair. New TLS sessions [17] are created when a gateway crashes and then recovers. Clients connect to gateways using a credential scheme such as OAuth2.0 [18].

The distributed recovery protocol has assumptions regarding log management. Log entries need integrity, durability, availability, and confidentiality guarantees, as they are an attractive attack point [19]. 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 [20]). 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. We assume the underlying ledgers where gateways operate are safe (i.e., consensus cannot be subverted by an adversary).

4. ODAP-2PC

In this section, we present Hermes' main building block: ODAP-2PC. We start by presenting ODAP, in which ODAP-2PC is based.

4.1. ODAP

The ODAP protocol is a gateway-to-gateway unidirectional asset transfer protocol that uses gateways as the systems conducting the transfer [2]. An asset transfer is represented in the form $T : G_1 \xrightarrow{a,x} G_2$, 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 leveraged ODAP to provide as strong durability guarantees as to the underlying durability guarantees of the chosen data store. If the datastore is a blockchain, Hermes can 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 the 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). The client may have a 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 the "Simplified ODAP Message Format" figure.

Fig. 5 represents ODAP. When an end-user wants to perform an asset transfer, gateways conduct such a 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 [2]. 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 proofs in resolving 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 (*informal*)**.** 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, the logging procedure depends on the success of the asset lock. 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.

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 [21]. 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.

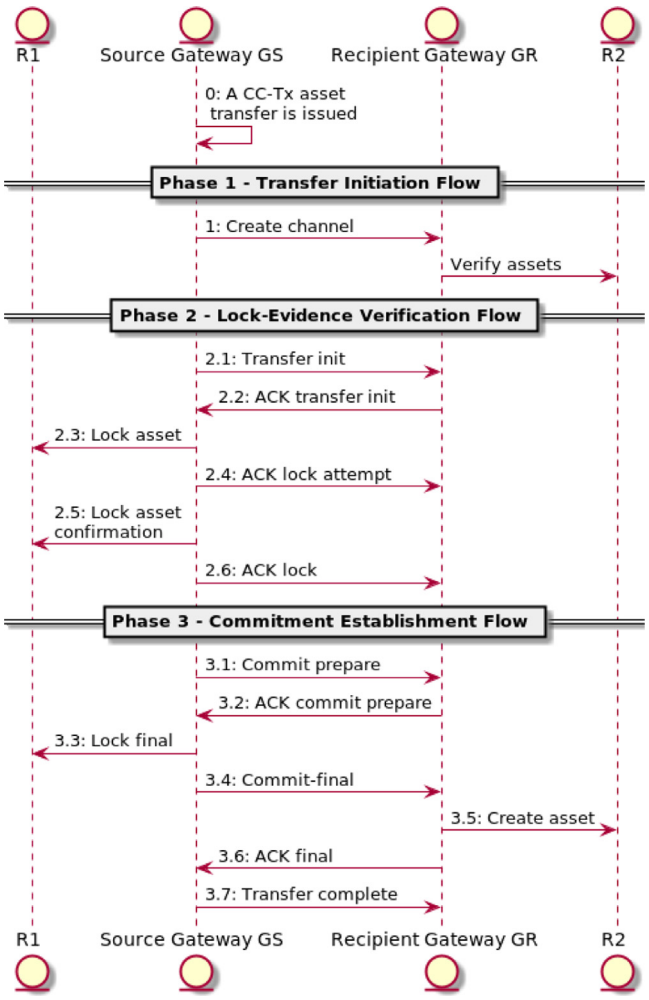


Fig. 5. Simplified sequence diagram depicting ODAP. A transfer is issued by an end-user to the gateway (G1), which then manages on-chain resources (L1), and communicates with a counterparty gateway (G2). The asset transfer corresponds to the creation of L2.

4.2. ODAP-2PC: Message and logging flow

ODAP-2PC aims to solve an important practical limitation of ODAP. ODAP does not handle gateway crashes. If they crash, the protocol may leave the DLTs in some inconsistent state. ODAP-2PC allows the ODAP to continue operating when the faulty gateway recovers, e.g., when the server where it runs reboots.

ODAP messages are exchanged between client applications and gateway servers (DLT nodes). They consist of functional messages allowing protocol negotiation [2]. 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. ODAP-2PC stores these messages in logs to allow recovery.

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 execution's confirmation. This serves to handle crashes in systems with only one node. Public operations are operations in which a state is known by more than one node. Intuitively, a private

Table 1
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	$\mathcal{G}_S$	$\mathcal{G}_R$	$l_4 = l^{p_1,1}$	`<p1, 1, init-validate, (GS->GR)>`	validate	init
$\mathcal{G}_R$ executes the validation operation	✗	$\mathcal{G}_R$	$l_5 = l^{p_1,2}$	`<p1, 2, exec-validate, (GR)>`	validate	exec
$\mathcal{G}_R$ completes the validation operation	✗	$\mathcal{G}_R$	$l_6 = l^{p_1,3}$	`<p1, 3, done-validate, (GR)>`	validate	done
$\mathcal{G}_R$ informs $\mathcal{G}_S$	$\mathcal{G}_R$	$\mathcal{G}_S$	$l_7 = l^{p_1,4}$	`<p1, 4, ack-validate, (GR->GS)>`	validate	ack

operation is only known by the node executing it, whereas public operations involve several nodes and are thus perceived by more nodes than those executing it.

Simplified ODAP Message Format

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.

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.

Let us consider an example where there is an asset transfer $\mathcal{G}_S \xrightarrow{a,1} \mathcal{G}_R$. We depict a message exchange with content m from $\mathcal{G}_S$ to $\mathcal{G}_R$ by $\mathcal{G}_S \xrightarrow{m} \mathcal{G}_R$. The reply from $\mathcal{G}_R$ to $\mathcal{G}_S$ is represented by $\mathcal{G}_R \xrightarrow{\alpha(m)} \mathcal{G}_S$, where α is a function that given an operation, step, and input from a counterparty gateway, returns the response to it. Fig. 6 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 1 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 .

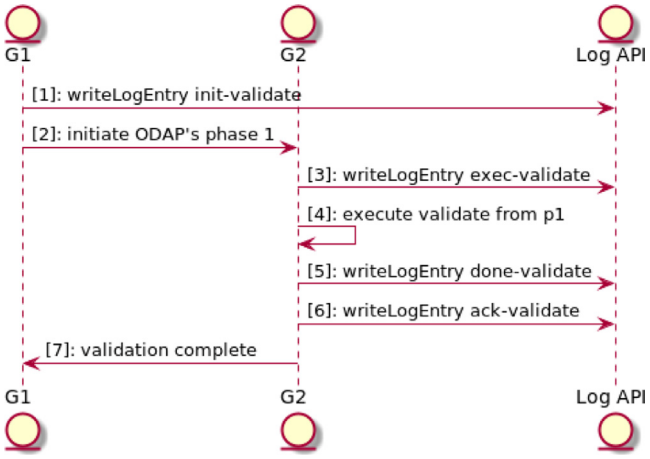


Fig. 6. Message flow regarding the validation operation, of ODAP's phase 1.

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.

4.3. ODAP-2PC: Log storage API

The log storage API allows developers to abstract operations on the log, focusing on the development of gateway-to-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 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 (thus being susceptible

to misbehavior from other gateways). This mode thus requires substantial trust in other gateways. The DLT-based repository, γ_{bc}, offers strong reliability concerning log-saving due to its immutability, transparency, and traceability [19,22]. 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 a cloud provider mediates this support, 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.

Format of log entries

The log entries' format should account for three phases, in case the gateway-to-gateway protocol is ODAP. In Section 4.2 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.

4.4. ODAP-2PC: 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.

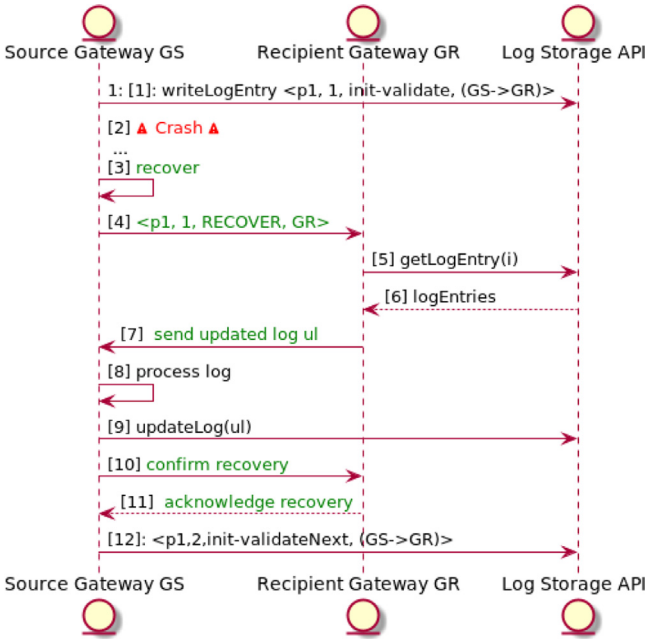


Fig. 7. $\mathcal{G}_S$ crashing before issuing init-validation to $\mathcal{G}_R$.

Overview

The protocol is crash fault-tolerant, so it does not tolerate Byzantine faults (i.e., gateways that behave arbitrarily). Gateways are trusted to operate the ODAP protocol as specified unless they crash.

ODAP-2P support two alternative fault tolerance strategies:

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 may never recover, but that timeout can detect this failure [21]; 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 operation log and executes 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. Fig. 7 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

Fig. 8. $\mathcal{G}_S$ crashing after issuing the init command to $\mathcal{G}_R$.

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, 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).

Fig. 8 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)

4.4.1. The ODAP-2PC protocol

In this section, we present the ODAP-2PC protocol itself. 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, as depicted by Figs. 7 and 8. 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 $\mathcal{G}_S \xrightarrow{vote-req} \mathcal{G}_R$ ▷ step 3.1
8 **wait until** $\mathcal{G}_R \xrightarrow{\alpha(vote-req)} \mathcal{G}_S$ ▷ step 3.2
9 ▷ Decision Phase
10 **if** $\mathcal{G}_R \xrightarrow{\alpha(vote-req)} \mathcal{G}_S = NO$ **then**
11 $\quad \mathcal{G}_S \xrightarrow{abort()} \mathcal{G}_R$ ▷ otherwise, $\mathcal{G}_R \xrightarrow{\alpha(vote-req)} \mathcal{G}_S =$ 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 $\quad \mathcal{G}_S \xrightarrow{abort()} \mathcal{G}_R$
19 $\quad \mathcal{G}_S.rollback(PO_{\mathcal{G}_S})$ ▷ undo $\mathcal{G}_S$.LOCK(a)
20 **end if**
21 $\mathcal{G}_S \xrightarrow{commit} \mathcal{G}_R$
22 $a' = \mathcal{G}_R$.CREATE_ASSET() ▷ step 3.5
23 $PO_{\mathcal{G}_R}$.append(a')
24 **wait until** $\mathcal{G}_R \xrightarrow{\alpha(commit)} \mathcal{G}_S$ ▷ step 3.6
25 **if** $\mathcal{G}_R \xrightarrow{\alpha(commit)} \mathcal{G}_S = COMMIT$ **then**
26 $\quad \mathcal{G}_S$.COMPLETE() ▷ step 3.8
27 **end if**
28 **else**
29 $\quad \mathcal{G}_S \xrightarrow{abort()} \mathcal{G}_R$ ▷ 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., $T : \mathcal{G}_S \xrightarrow{a.x} \mathcal{G}_R$. 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 rolled back 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 rolled back, 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 the counterparty gateway to recover (by assumption, it does). Upon recovery, the process depicted by steps 4–11 from Fig. 7 takes 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 (informal): 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, it can abort and stop the process if it has the timeout action triggered. 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 to 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.

4.4.2. 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 *canceling* a transaction. For example, canceling 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 canceling gateway and the counterparty gateway. The canceling gateway realizes the need to cancel one or more transactions, initiate the rollback protocol, and propagate eventual corrective measure commands to the counterparty gateway. There is a need to involve a counterparty gateway to ensure the consistency of the assets handled by the protocol.

The rollback process occurs as follows: 1) the canceling gateway undoes the transactions to be rolled back by issuing transactions with the contrary effect; 2) the same gateway sends an acknowledgment back to the counterparty gateway, and 3) counterparty gateway undoes all its pending transactions, which can lead back to step one, where the counterparty gateway serves as the canceling gateway. This recursive protocol may generate a cascade effect where several transactions from both blockchains need to be canceled. 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, 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.

5. Use case: Gateway-supported cross-jurisdiction promissory notes

This section presents a use case implementing digital asset transfers, benefiting from the gateway paradigm. The digital assets to be exchanged are defined are standardized in as an *asset profile*, which is ongoing work at the IETF [23]. 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" [23]. Asset profiles can be emitted by authorized parties, having the capability to represent real-world assets (e.g., real estate) legally.

5.1. Asset profile

The *Asset Profile Definitions for DLT Interoperability* draft presents an unambiguous manner of representing a digital asset, independently of its concrete implementation [23]. The representation of an asset via an asset profile allows for representing physical assets (called tokenization) that then can be exchanged across DLTs with Hermes. Hermes validates a given asset profile definition, allowing gateways to agree on the asset to be exchanged, in the *Transfer Initiation Flow*. An asset profile contains the following fields (from [23]):

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. It can be blank or non-applicable.
11. **Transfer Restriction**: Information about transfer restrictions (e.g., prohibited jurisdictions), 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 verify their counterparty 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.

5.2. 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, allows 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, before 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 (the asset will be transferred, and thus cannot be used), locked (asset was transferred and cannot be used), and burnt (asset was destroyed or permanently locked).

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 an additional DLT-specific operation needs 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 waits for confirmation from the counterparty gateway of such an operation. In case the protocol fails before COMMIT, a ROLLBACK is issued by $\mathcal{G}_S$, triggering an UNLOCK transaction. The UNLOCK sets the state of the pre-locked asset to *unlocked*, reverting the effect of the PRE-LOCK.

Algorithm 2: On-chain set state

```
Input: Asset a, Ledger connector c, lock level l
Result: Asset a locked at l
1 assetRepresentation = c.getStateById(a.assetCode)      ▷ DLT-specific
2 assetRepresentation.state = l      ▷ pre-lock, lock, unlocked, burnt
3 c.setState(assetRepresentation)      ▷ DLT-specific
```

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 if the logging takes place on-chain.

5.3. Using hermes to exchange promissory notes

Promissory notes are freely transferable financial instruments where issuers denote a promise to pay another party (payee) [24]. 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 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 [25]. 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" [26]. 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.

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 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.

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)

6. Discussion

Hermes can include different gateway implementations, different gateway-to-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 [27,28]. Remote attestations are particularly important since provably exposing the internal state to external parties is a crucial requirement for CC-Txs [29].

Gateways can also be leveraged for tasks other than asset transfers; they can perform the function of oracles, either centralized or decentralized [11], 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 [30]. 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.

6.1. RQ1: Reliability

Our solutions implement a strong consistency model among gateways, where there are no dirty writes or repeated reads during an atomic transaction (e.g., atomic asset exchange). This is achieved by sacrificing liveness and using one of two mechanisms: on the 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. We should point out that this strong consistency model solely applies to shared state among gateways and not shared state among blockchains. Hermes delivers eventual consistency among blockchains (either both operations eventually happen, or none does), but no stronger guarantees (e.g., strict consistency, strong consistency).

ODAP and ODAP-2PC assume a trade-off between reliability and efficiency, according to the end-to-end principle [31]. 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, determines 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 [19,22]. Thus, the log storage API serves two purposes: 1) it provides a reliable means 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, reducing the risk of counterparty fraud.

6.2. RQ2: Performance

As mentioned, a trade-off between reliability and performance exists. Storing logs in local storage typically has lower latency but delivers 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 [32], especially if one waits for 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.

6.3. 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 [22] from the jurisdiction in which the asset transfer occurs. Thus, for the legislated spaces and proper log storage support, Hermes might be 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 several techniques to decentralize gateways or 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 [11]. Smart contract-based gateways could also need to pay transaction fees in public blockchains, such as gas on Ethereum [33], 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 [15]. 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 that they should periodically publish proofs of state in an external repository [19]. 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 [27,28]. 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.

6.4. 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 [34], and use secure channels based on TLS/HTTPS [17].

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 exchanges across public blockchains, privacy might be of 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.

Another 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 represented by the underlying DLT (or at least DLT client), i.e.; it needs to convert ODAP messages to blockchain-specific transactions. Thus, the gateway has the responsibility of converting a standard representation on a DLT-specific one. If desirable, gateways can hide representation details, providing privacy regarding asset management.

7. Related work

Building dependable and trusted middleware for blockchain interoperability requires the orchestration of several disciplines. This section introduces related work on blockchain interoperability solutions, crash recovery, and other contributions.

Interoperability solutions

Blockchain interoperability solutions are diverse and numerous. However, few solutions can accommodate a seamless integration among public blockchains and non-public blockchains (private blockchains, legacy systems). Thus, we focus on the comparison of Hermes with this class of solutions. Table 2 presents existing related work with regard to several criteria (and *subcriteria)*:

- Digital Asset Support: whether the solution can transfer (1) *Utility Tokens* (non-fungible tokens, such as ERC-20 tokens [35]), or (2) *Payment Tokens* (fungible tokens, such as Bitcoin, or Ether).
- Regulation: if the solution is implemented such that it is compliant with (1) a *Legal* framework or (2) an existing or ongoing *Standard*.
- Crash Recovery: the solution supports *Crash Faults* and/or *Byzantine Faults*.

Hardjono et al. proposed a gateway-based architecture inspired by the architecture of the Internet [6], further expanded by recent work [51]. The gateway-based paradigm bootstrap the emergence of standardization efforts such as ODAP [2] at the IETF [50]. Such protocols, in which Hermes is based, aim to comply with the Travel Rule and FAFT regulations [49].

Ghaemi et al. [45] proposed a publisher-subscriber architecture for blockchain interoperability, based on connector applications that publish on the connector smart contract held by the broker blockchain. However, the connectors do not have crash recovery mechanisms and thus are not suitable for a production environment.

Hyperledger Cactus [15] is a trusted relay connecting DLTs, whereby a consortium of Cactus Nodes endorses transactions. Cactus aims to be a general-purpose interoperability solution that uses two families of software components that, in 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. However, Cactus Nodes have no crash recovery mechanism implemented and thus are not suitable for a production environment. Weaver also aims to provide general-purpose interoperability, using proofs of state of private blockchains [29], called a blockchain view [52]. On top of that, they propose interoperability RFCs [48].

Quant Overledger is a blockchain interoperability enterprise solution [36] compliant with the ongoing standardization effort from ISO [47]. Overledger exposes functionalities from different ledgers and legacy systems via a REST API. Overledger has crash recovery mechanisms and can be deployed on different clouds. However, this solution is not open source, and it is not clear if a rollback protocol is provided.

Other solutions are equally promising, but lack crash recovery capabilities, unlike Overledger, ODAP, and Hermes [38–44]. We refer readers interested in interoperability to the survey in [11], where each solution is analyzed in greater detail.

A short paper on some of the ideas presented in this paper appeared before [12]. The present paper is three times larger and presents in detail what the other barely sketches: Hermes, ODAP-2PC, etc.

Table 2
Classification of blockchain interoperability solutions connecting public and non-public blockchains. The green checkmark (✓) indicates a subcriteria is fulfilled. A red cross (✗) indicates otherwise. The gray question mark (?) indicates that the criteria may or not be fulfilled (we lack information to decide).

		Digital asset support		Regulation		Crash recovery	
Paper	Year	Payment tokens	Utility tokens	Legal	Standardized	Crash faults	Byzantine faults
Quant Overledger [36]	2018	✓	✓	✓[a]	✓[b]	✓	✗
Bifrost [37]	2019	✓	✓	✗	✗	✗	✗
Abebe et al. [38]	2019	✗	✓	✗	✗	✗	✗
Wang et al. [39]	2020	?	?	✗	✗	✗	✗
Zhao et al. [40]	2020	?	?	✗	✗	✗	✗
Cactus [15]	2020	✓	✓	✗	✗	✗	✗
Weaver [29]	2020	✓	✓	✗	✓[c]	✗	✗
Gewu et al. [41]	2020	✗	✓	✗	✗	✗	✗
SCIP [42]	2020	✗	✓	✗	✓[d]	✗	✗
Nissl et al. [43]	2020	✓	✓	✗	✗	✗	✗
Fynn et al. [44]	2020	✓	✓	✗	✗	✗	✗
ODAP [2]	2021	✓	✓	✓[e]	✓[f]	✓	✗
Ghaemi et al. [45]	2021	✗	✓	✗	✗	✗	✗
This paper	2021	✓	✓	✓[e]	✓[f]	✓	✗

[a]International standards [46].
[b]ISO/TC 307/SG 7 [47].
[c]Weaver Interop. RFCs [48].
[d]SCIP Protocol [42].
[e]Travel Rule, FAFT [49].
[f]IETF [50].

Crash recovery on cross-chain transactions

Two-phase commit was initially developed as an atomic commitment protocol that coordinates distributed transactions. Thus, it could be seen as a consensus mechanism over the global state encompassing each distributed database. Generally, 2PC is not used for blockchain consensus [53], but rather for assuring the reliability of atomic cross-chain transactions. Fynn et al. presented a Move operation that can migrate accounts and arbitrary computation across Ethereum virtual machine-based chains [44]. An atomic Move operation can be implemented with 2PC. Wang et al. [39] presented a 2PC protocol for conducting CB-Tx. A blockchain is elected as the coordinator in this scheme, managing the process between an arbitrary number of blockchains. This protocol includes a heartbeat monitoring mechanism to guarantee liveness.

However, it is unclear how ACID properties are 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. Herlihy et al. discuss the need for developing models for cross-chain transactions that evolve from traditional ACID properties when non-trusted actors are involved [54]. Gateways in Hermes are assumed to be trusted, and thus ACID properties seem reasonable to model transactions across these systems. However, and for future work, a decentralized ODAP system, where a set of mutually non-trusting gateways represents each jurisdiction, can benefit from the modeling mentioned above. A decentralized ODAP and its recovery mechanism, ODAP-3PC, could pave the way for resilience towards Byzantine faults and malicious actors.

Standardization efforts

Standardization processes typically take years from inception until publishing. While several standardization efforts are focusing on blockchain interoperability, none have been published and widely adopted. While it is likely that there will be several competing standards, the most active ones seem to be ODAP (IETF [50]), ISO 307/SG 7 [47], and the IEEE Blockchain Initiative [55].

Hermes is one of the few solutions (along with Quant Overledger) that is being built considering the existing standardization efforts.

Other contributions

Orthogonally, several contributions support the gateway paradigm: Vo et al. propose decentralized blockchain registries that can identify and address blockchain oracles [7]. Chen and Hardjono proposed an IETF draft proposing a method for identification of computer systems that act as gateways and the correct validation of the ownership of the gateway [56]. This identification occurs via DNS, where a gateway owner registers for an "Autonomous System number from ARIN, or other region networking authorities (such as RIPE NCC for Europe and APNIC for East and South Asia)" [56]. Self-sovereign based identity promotes identity portability, by deferring the authentication and authorization processes to the end-user [11,57,58]. Gateways could then be identified by a decentralized identifier and issued verifiable credentials by certified authorities that manage virtual asset providers.

8. Conclusion

In this paper, we presented Hermes, a blockchain interoperability middleware that enables gateway-to-gateway asset transfers via the Open Asset Digital Protocol. Hermes can support asset transfer across jurisdictions, contributing towards regulation-compliant, standardized, blockchain interoperability middleware. We have shown that our solution is resilient to crashes by leveraging ODAP-2PC, a distributed recovery mechanism. This implies that asset transfers are atomic, fair, and no double spending can occur. A use case on the exchange of digital promissory notes is presented, showing 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 atomic asset transfer (using ODAP-3PC), paving the way for efficient multiparty atomic swaps.

CRediT authorship contribution statement

Rafael Belchior: Conceptualization, Formal analysis, Investigation, Validation, Writing – original draft. **André Vasconcelos:** Conceptualization, Supervision, Writing – review & editing. **Miguel Correia:** Conceptualization, Investigation, Supervision, Writing – review & editing. **Thomas Hardjono:** Conceptualization, Investigation, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

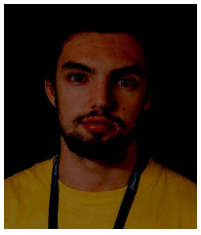
Acknowledgments

We warmly thank Benedikt Schuppli for discussions on digital promissory notes and our colleagues in the IETF's forming working group on ODAP for fruitful discussions. This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID) and 2020.06837.BD, and by the European Commission program H2020 under grant agreement 822404 (QualiChain).

References

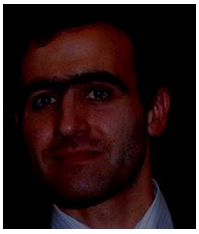
[1] C. Catalini, J.S. Gans, Some Simple Economics of the Blockchain, Working Paper Series 22952, National Bureau of Economic Research, 2016, https://doi.org/10.3386/w22952, URL http://www.nber.org/papers/w22952.
[2] M. Hargreaves, T. Hardjono, R. Belchior, Open digital asset protocol draft 02, draft-hargreaves-odap-02, 2021, Internet Engineering Task Force, URL https://datatracker.ietf.org/doc/html/draft-hargreaves-odap-02.
[3] W. Viriyasitavat, L. Da Xu, Z. Bi, V. Pungpapong, Blockchain and internet of things for modern business process in digital economy—the state of the art, IEEE Trans. Comput. Soc. Syst. 6 (6) (2019) 1420–1432.
[4] A. Pentland, A. Lipton, T. Hardjono, Time for a new, digital bretton woods, 2021, Barron's, URL https://rb.gy/yj31vq.
[5] L. Pawczuk, M. Gogh, N. Hewett, Inclusive Deployment of Blockchain for Supply Chains: A Framework for Blockchain Interoperability, Tech. Rep., World Economic Forum, 2020, URL www.weforum.org.
[6] T. Hardjono, A. Lipton, A. Pentland, Towards an interoperability architecture blockchain autonomous systems, IEEE Trans. Eng. Manag. 67 (4) (2019) 1298–1309, URL doi:10.1109/TEM.2019.2920154.
[7] H. Tam Vo, Z. Wang, D. Karunamoorthy, J. Wagner, E. Abebe, 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), 2018, pp. 1574–1581.
[8] B. Pillai, K. Biswas, Blockchain interoperable digital objects innovative applications of blockchain technology view project blockchain interoperable asset classes view project, 2019, http://dx.doi.org/10.1007/978-3-030-23404-1_6, URL https://doi.org/10.1007/978-3-030-23404-1_6.
[9] M. Borkowski, M. Sigwart, P. Frauenthaler, T. Hukkinen, S. Schulte, DeXTT: Deterministic cross-blockchain token transfers, IEEE Access 7 (2019) 111030–111042, arXiv:..
[10] S. Schulte, M. Sigwart, P. Frauenthaler, M. Borkowski, Towards blockchain interoperability, in: C. Di Ciccio, R. Gabryelczyk, L. García-Bañuelos, T. Hernaus, R. Hull, M. Indihar Štemberger, A. Kő, M. Staples (Eds.), Business Process Management: Blockchain and Central and Eastern Europe Forum, Springer International Publishing, Cham, 2019, pp. 3–10.
[11] R. Belchior, A. Vasconcelos, S. Guerreiro, M. Correia, A survey on blockchain interoperability: Past, present, and future trends, ACM Comput. Surv. 58 (8) (2022) 1–41, http://dx.doi.org/10.1145/3471140, arXiv:2005.14282.
[12] R. Belchior, A. Vasconcelos, M. Correia, T. Hardjono, Enabling cross-jurisdiction digital asset transfer, in: IEEE International Conference on Services Computing, IEEE, 2021.
[13] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.
[14] R.J. Patton, Fault-tolerant control: The 1997 situation, IFAC Proc. Vol. 30 (18) (1997) 1029–1051, http://dx.doi.org/10.1016/s1474-6670(17)42536-5.
[15] H. Montgomery, H. Borne-Pons, J. Hamilton, M. Bowman, P. Somogyvari, S. Fujimoto, T. Takeuchi, T. Kuhrt, R. Belchior, Hyperledger cactus whitepaper, 2020, URL https://github.com/hyperledger/cactus/blob/master/docs/whitepaper/whitepaper.md.
[16] T. Hardjono, A. Lipton, A. Pentland, Towards a public key management framework for virtual assets and virtual asset service providers, J. FinTech 1 (1) (2020) http://dx.doi.org/10.1142/S2705109920500017.
[17] E. Rescorla, RFC 8446 - The transport layer security (TLS) protocol version 1.3, 2014, URL https://tools.ietf.org/html/rfc8446.
[18] E. D. Hardt, RFC 6749 - The oauth 2.0 authorization framework, 2012, URL https://tools.ietf.org/html/rfc6749.
[19] B. Putz, F. Menges, G. Pernul, A secure and auditable logging infrastructure based on a permissioned blockchain, Comput. Secur. 87 (2019) 101602.
[20] D. Johnson, A. Menezes, S. Vanstone, The elliptic curve digital signature algorithm (ECDSA), Int. J. Inf. Secur. 1 (1) (2001) 36–63, http://dx.doi.org/10.1007/s102070100002.
[21] P. ALsberg, J. Day, A principle for resilient sharing of distributed resources, J. Chem. Inform. Model. (1976) arXiv:arXiv:1011.1669v3.
[22] R. Belchior, A. Vasconcelos, M. Correia, Towards Secure, Decentralized, and Automatic Audits with Blockchain, in: European Conference on Information Systems, 2020.
[23] A. Sardon, T. Hardjono, Benedikt Schuppli, Asset Profile Definitions for DLT Interoperability (draft-sardon-blockchain-interop-asset-profile-00), Tech. Rep., 2021, URL https://datatracker.ietf.org/doc/draft-sardon-blockchain-interop-asset-profile/.
[24] J.S. Waterman, The promissory note as a substitute for money, Minn. Law Rev. 14 (1929) 313.
[25] FQX, eNI™ Infrastructure - fqx.ch - Electronic Negotiable Instruments - FQX, 2020, URL https://fqx.ch/.
[26] Transatlantic Shipment of Metals Financed via FQX eNote | Treasury Management International, URL https://treasury-management.com/news/transatlantic-shipment-of-metals-financed-via-fqx-enote/.
[27] T. Hardjono, N. Smith, Towards an attestation architecture for blockchain networks, World Wide Web J. 24 (9) (2021) 1587–1615, http://dx.doi.org/10.1007/s11280-021-00869-4.
[28] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, B. Sniffen, Principles of remote attestation, Int. J. Inf. Secur. 10 (2) (2011) 63–81, http://dx.doi.org/10.1007/s10207-011-0124-7, URL https://link.springer.com/article/10.1007/s10207-011-0124-7.
[29] E. Abebe, D. Karunamoorthy, J. Yu, Y. Hu, V. Pandit, A. Irvin, V. Ramakrishna, Verifiable observation of permissioned ledgers, 2021, arXiv, arXiv:2012.07339v2.
[30] R. Mühlberger, S. Bachhofner, E. Castelló Ferrer, C. Di Ciccio, I. Weber, M. Wöhrer, U. Zdun, Foundational oracle patterns: Connecting blockchain to the off-chain world, in: Lecture Notes in Business Information Processing, Vol. 393 LNBIP, Springer Science and Business Media Deutschland GmbH, 2020, pp. 35–51, http://dx.doi.org/10.1007/978-3-030-58779-6_3, arXiv:2007.14946.
[31] J.H. Saltzer, D.P. Reed, D.D. Clark, End-to-end arguments in system design, ACM Trans. Comput. Syst. (TOCS) 2 (4) (1984) 277–288, http://dx.doi.org/10.1145/357401.357402, URL https://dl.acm.org/doi/10.1145/357401.357402.
[32] A. Bessani, M. Correia, B. Quaresma, F. Andre, P. Sousa, DepSky: Dependable and secure storage in a cloud-of-clouds, ACM Trans. Storage 9 (4) (2013) 1–33.
[33] G. Wood, Ethereum: A Secure Decentralised Generalised Transaction Ledger. Byzantium version 7e819ec, Tech. Rep., 2019, URL https://ethereum.github.io/yellowpaper/paper.pdf.
[34] Final: OpenID Connect Core 1.0 incorporating errata set 1, URL https://openid.net/specs/openid-connect-core-1_0.html.
[35] F. Vogelsteller, V. Buterin, EIP 20: ERC-20 Token standard, 2015, URL https://eips.ethereum.org/EIPS/eip-20.
[36] G. Verdian, P. Tasca, C. Paterson, G. Mondelli, Quant Overledger Whitepaper Vo.1, Tech. Rep., Quant, 2018, pp. 1–48, URL http://objects-us-west-1.dream.io/files.quant.network/Quant_Overledger_Whitepaper_v0.1.pdf.
[37] E.J. Scheid, T. Hegnauer, B. Rodrigues, B. Stiller, Bifröst: a modular blockchain interoperability API, in: IEEE 44th Conference on Local Computer Networks, Institute of Electrical and Electronics Engineers (IEEE), 2019, pp. 332–339.
[38] E. Abebe, D. Behl, C. Govindarajan, Y. Hu, D. Karunamoorthy, P. Novotny, V. Pandit, V. Ramakrishna, C. Vecchiola, Enabling enterprise blockchain interoperability with trusted data transfer, in: Proceedings of the 20th International Middleware Conference Industrial Track, Association for Computing Machinery, 2019, pp. 29–35.
[39] X. Wang, O.T. Tawose, F. Yan, D. Zhao, Distributed nonblocking commit protocols for many-party cross-blockchain transactions, 2020, arXiv, arXiv:2001.01174.
[40] D. Zhao, T. Li, Distributed cross-blockchain transactions, 2020, arXiv, arXiv:2002.11771v1.
[41] G. Bu, R. Haouara, T.S.L. Nguyen, M. Potop-Butucaru, Cross hyperledger fabric transactions, in: CRYBLOCK 2020 - Proceedings of the 3rd Workshop on Cryptocurrencies and Blockchains for Distributed Systems, Part of MobiCom 2020, Association for Computing Machinery, 2020, pp. 35–40, http://dx.doi.org/10.1145/3410699.3413796.
[42] G. Falazi, U. Breitenbücher, F. Daniel, A. Lamparelli, F. Leymann, 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, Vol. 12127 LNCS, 2020, pp. 134–149.
[43] M. Nissl, E. Sallinger, S. Schulte, M. Borkowski, Towards cross-blockchain smart contracts, 2020, arXiv:2010.07352.
[44] E. Fynn, F. Pedone, B. Alysson, Smart contracts on the move, in: Dependable Systems and Networks, 2020.
[45] S. Ghaemi, S. Rouhani, R. Belchior, R.S. Cruz, H. Khazaei, P. Musilek, A pub-sub architecture to promote blockchain interoperability, 2021, arXiv:2101.12331.
[46] Quant, Overledger Enterprise, URL https://www.quant.network/overledger-enterprise.
[47] ISO, ISO - ISO/TC 307 - Blockchain and distributed ledger technologies, URL https://www.iso.org/committee/6266604.html.

[48] Weaver, Weaver Interoperability RFCs, URL https://github.com/hyperledger-labs/weaver-dlt-interoperability/tree/main/rfcs.
[49] T. Hardjono, A. Lipton, A. Pentland, Towards a Public Key Management Framework for Virtual Assets and Virtual Asset Service Providers, Tech. Rep., 2019, arXiv:1909.08607v1.
[50] IETF, IETF | Internet Engineering Task Force, URL https://www.ietf.org/.
[51] T. Hardjono, Blockchain gateways, bridges and delegated hash-locks, 2021, arXiv:2102.03933.
[52] R. Belchior, S. Guerreiro, A. Vasconcelos, M. Correia, A survey on business process view integration, 2020, arXiv:2011.14465.
[53] J. Nijsse, A. Litchfield, A taxonomy of blockchain consensus methods, Cryptography 4 (4) (2020) 32.
[54] M. Herlihy, B. Liskov, L. Shrira, Cross-chain deals and adversarial commerce, PVLDB, in: Cross-Chain Deals and Adversarial Commerce, vol. 13, (2) 2019, pp. 100–113, http://dx.doi.org/10.14778/3364324.3364326, URL https://doi.org/10.14778/3364324.3364326.
[55] IEEE, P3205 - Standard for Blockchain Interoperability - Data Authentication and Communication Protocol, URL https://standards.ieee.org/project/3205.html.
[56] S. Chen, T. Hardjono, Gateway identification and discovery for decentralized ledger networks, 2021, Internet-Draft draft-chen-dlt-gateway-identification-00, Internet Engineering Task Force Work in Progress, URL https://datatracker.ietf.org/doc/html/draft-chen-dlt-gateway-identification-00.
[57] R. Belchior, B. Putz, G. Pernul, M. Correia, A. Vasconcelos, S. Guerreiro, SSI-BAC : SElf-sovereign identity based access control, in: The 3rd International Workshop on Blockchain Systems and Applications, IEEE, 2020.
[58] M.S. Ferdous, F. Chowdhury, M.O. Alassafi, In search of self-sovereign identity leveraging blockchain technology, IEEE Access 7 (2019) 103059–103079, http://dx.doi.org/10.1109/access.2019.2931173.



Eng. Rafael Belchior is a researcher at INESC-ID in the Distributed Systems Group and Ph.D. student at Técnico Lisboa, where he started lecturing in 2018.

Studying the intersection of blockchain interoperability and security, he contributes for Hyperledger Cactus, ODAP's forming group at IETF, and European Commission projects Qualichain and DE4A. He worked as a blockchain research engineer at the Portuguese government, Linux Foundation, and Quant. Rafael also volunteers as a speaker and mentor (invited classes, Hyperledger Summer Internship program).



André is Assistant Professor (Professor Auxiliar) in the Department of Computer Science and Engineering, Instituto Superior Técnico, Lisbon University, and researcher in Information and Decision Support Systems Lab at INESC-ID, in Enterprise Architecture domains namely representation, and modeling of Architectures of Information Systems, and Evaluation of Information Systems Architectures.

He has over 15 years of experience in advising organizations in information systems and enterprise architectures in eGovernment and private sector projects. Head of teams accountable for delivering state of art Information and Communication Technology (ICT) innovation aligned with business needs, in distinctive projects.

Miguel Correia is a Full Professor at Instituto Superior Técnico (IST), Universidade de Lisboa, in Lisboa, Portugal. He is vice-president for faculty at DEI. He is coordinator of the Doctoral Program in Information Security at IST. He is a senior researcher at INESC-ID, and member of the Distributed Systems Group (GSD). He is co-chair of the European Blockchain Partnership that is designing the European Blockchain Services Infrastructure (EBSI). He is a member of the Board of Técnico+ and a non-executive member of the Board of Associação .PT. He is Associate Editor for IEEE Transactions on Computers. He has a Ph.D. in Computer Science from the Universidade de Lisboa Faculdade de Ciências. He has been involved in several international and national research projects related to cybersecurity, including the DE4A, BIG, QualiChain, SPARTA, SafeCloud, PCAS, TCLOUDS, ReSIST, CRUTIAL, and MAFTIA European projects. He has more than 200 publications and is Senior Member of the IEEE.

Dr. Thomas Hardjono is currently the CTO of Connection Science and Technical Director of the MIT Trust-Data Consortium, located at MIT in Cambridge, MA. For several years prior to this he was the Executive Director of the MIT Kerberos Consortium, helping make the Kerberos protocol to become the most ubiquitously deployed authentication protocol in world today. Over the past two decades Thomas he has held various industry technical leadership roles, including Distinguished Engineer at Bay Networks, Principal Scientist at VeriSign PKI, and CTO roles at several start-ups. He has been at the forefront of several industry initiatives around identity, data privacy, trust, applied cryptography, and cybersecurity.