

Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: 6 April 2022

R. Belchior
M. Correia
INESC-ID, Instituto Superior Técnico
T. Hardjono
MIT
3 October 2021

DLT Gateway Crash Recovery Mechanism
draft-belchior-gateway-recovery-03

Abstract

This memo describes the crash recovery mechanism for the Open Digital Asset Protocol (ODAP), called ODAP-2PC. The goal is to assure gateways running ODAP to be able to recover from crashes, and thus preserve the consistency of an asset across ledgers (i.e., double spend does not occur). This draft includes the description of the messaging and logging flow necessary for the correct functioning of ODAP-2PC.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 6 April 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components

Internet-Draft

Gateway Crash Recovery

October 2021

extracted from this document must include Simplified BSD License text as described in Section 4.e of the [Trust Legal Provisions](#) and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	2
2.	Terminology	3
3.	Logging Model	4
3.1.	Example	5
3.2.	Log Storage Modes	7
3.3.	Log Storage API:	8
3.3.1.	Response Codes	10
4.	Format of log entries	10
5.	ODAP-2PC	13
5.1.	Crash Recovery Model	13
5.2.	Recovery Procedure	14
5.2.1.	Transfer Initiation Flow	14
5.2.2.	Lock-Evidence Flow	14
5.2.3.	Commitment Establishment Flow	15
5.3.	ODAP-2PC Messages	15
5.3.1.	RECOVER	15
5.3.2.	RECOVER-UDPDATE	16
5.3.3.	RECOVER-UPDATE ACK	16
5.3.4.	RECOVER-SUCCESS	16
5.3.5.	ROLLBACK	17
5.4.	Examples	17
5.4.1.	Crashing before issuing a command to the counterparty gateway	17
5.4.2.	Crashing after issuing a command to the counterparty gateway	19
6.	Security Considerations	20
7.	References	21
7.1.	Normative References	21
7.2.	Informative References	21
	Authors' Addresses	22

[1.](#) Introduction

Gateway systems that perform virtual asset transfers among DLTs must possess a degree of resiliency and fault tolerance in the face of possible crashes. Accounting for the possibility of crashes is particularly important to guarantee asset consistency across DLTs.

ODAP-2PC [[HERMES](#)] uses 2PC, an atomic commitment protocol (ACP). 2PC considers two roles: a coordinator who manages the protocol's execution and participants who manage the resources that must be kept consistent. The source gateway plays the ACP role of Coordinator,

and the recipient gateway plays the Participant role in relay mode. Gateways exchange messages corresponding to the protocol execution, generating log entries for each one.

Log entries are organized into logs. Logs enable either the same or other backup gateways to resume any phase of ODAP. This log can also serve as an accountability tool in case of disputes. Another key component is an atomic commit protocol (ACP) that guarantees that the source and target DLTs are modified consistently (atomicity) and permanently (durability), e.g., that assets that are taken from the source DLT are persisted into the recipient DLT.

Log entries are then the basis satisfying one of the key deployment requirements of gateways for asset transfers: a high degree of availability. In this document, we consider two common strategies to increase availability: (1) to support the recovery of the gateways (self-healing model) and (2) to employ backup gateways with the ability to resume a stalled transfer (primary-backup model) [[HERMES](#)].

This memo proposes: (i) the logging model of ODAP-2PC; (ii) the log storage types; (iii) the log storage API; (iv) the log entry format; (v) the recovery and rollback procedures.

[2.](#) Terminology

There following are some terminology used in the current document:

- * Gateway: The nodes of a DLT system that are functionally capable of handling an asset transfer with another DLT. Gateway nodes implement the gateway-to-gateway asset transfer protocol.
- * Primary Gateway: The node of a DLT system that has been selected or elected to act as a gateway in an asset transfer.
- * Backup Gateway: The node of a DLT system that has been selected or elected to act as a backup gateway to a primary gateway.

- * **Message Flow Parameters:** The parameters and payload employed in a message flow between a sending gateway and receiving gateway.
- * **Source Gateway (or G1):** The gateway that initiates the transfer protocol. Acts as a coordinator of the ACP and mediates the message flow.
- * **Recipient Gateway (or G2):** The gateway that is the target of an asset transfer. It follows instructions from the source gateway.
- * **Source DLT:** The DLT of the source gateway.

- * **Recipient DLT:** The DLT of the recipient gateway.
- * **Log:** Set of log entries such that those are ordered by the time of its creation.
- * **Public (or Shared) Log:** log where several nodes can read and write from it.
- * **Private Log:** log where only one node can read and write from it.
- * **Log data:** The log information is retained by a gateway connected to an exchanged message within an asset transfer protocol.
- * **Log entry:** The log information generated and persisted by a gateway regarding one specific message flow step.
- * **Log format:** The format of log-data generated by a gateway.
- * **Atomic commit protocol (ACP):** A protocol that guarantees that assets that are taken from a DLT are persisted into the other DLT. Examples are two and three-phase commit protocols (2PC, 3PC, respectively) and non-blocking atomic commit protocols.
- * **Fault:** A fault is an event that alters the expected behavior of a system.
- * **Crash-fault tolerant models:** models allowing a system to keep operating correctly despite having a set of faulty components.

- * Digital asset: a form of digital medium record used as a digital representation of a tangible or intangible asset.

3. Logging Model

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 (the highest index). For each protocol step a gateway performs, a log entry is created immediately before executing and immediately after executing a given operation.

To manipulate the log, we define a set of log primitives that translate log entry requests from a process into log entries, realized by the log storage API (for the context of ODAP, [Section 3.5](#)):

- * `writeLogEntry(e,L)` (WRITE) - appends a log entry `e` in the log `L` (held by the corresponding Log Storage Support).

- * `getLogEntry(i,L)` (READ) - retrieves a log entry with index `i` from log `L`.

From these primitives, other functions can be built:

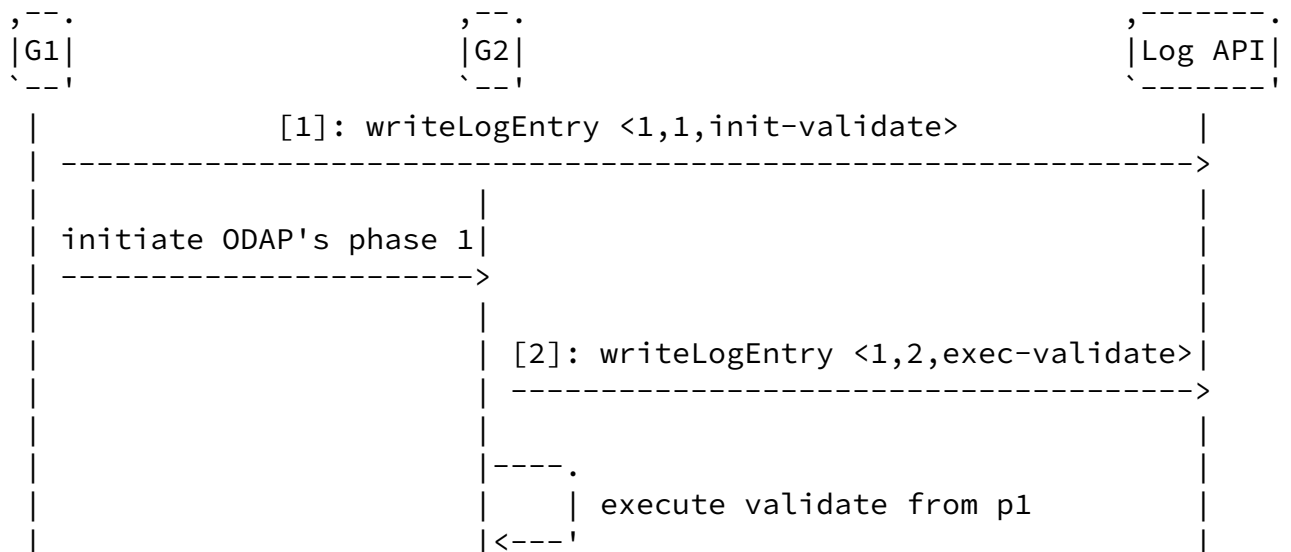
- * `getLogLength(L)` (READ) - obtains the number of log entries from log `L`.
- * `getLogDiff(l1,l2)` (READ) - obtains the difference between two logs.
- * `getLastEntry(L)`: obtains the last log entry from log `L`.
- * `getLog(L)`: retrieves the whole log `L`.
- * `updateLog(l1,l2)`: updates `l1` based on `l2` (uses `getLogDiff` and `writeLogEntry`).

Example 3.1 shows a simplified version log referring to the transfer initiation flow ODAP phase. Each log entry (simplified, see the definition in [Section 3](#)) is composed of metadata (phase, sequence number) and one attribute from the payload (operation). Operations

map behavior to state (see [Section 3](#)).

The following table illustrates the log storage API. The Function describes the primitive supported by the log storage API. The Parameters column specifies the parameters given to the endpoint as query parameters. Endpoint specifies the endpoint mapping a specific log primitive. The column Returns specifies what the contents of "response_data" mean. The column Response Example illustrates this last field.

[3.1](#). Example



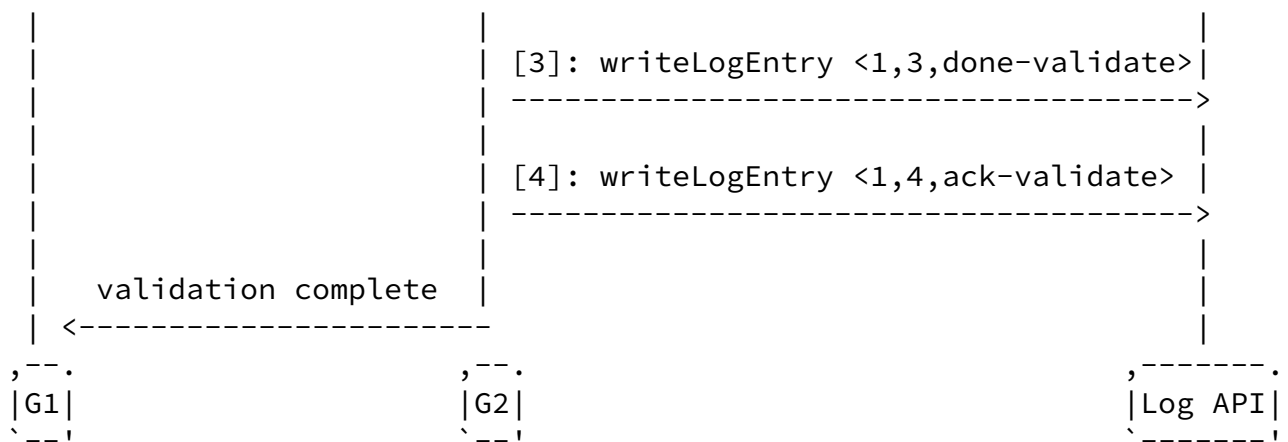


Figure 1

Example 2.1 shows the sequence of logging operations over part of the first phase of ODAP (simplified):

- * 1. At step 1, G1 writes an init-validate operation, meaning it will require G2 to initiate the validate function: This step generates a log entry (p1, 1, init-validate).
- * 2. At step 2, G2 writes an exec-validate operation, meaning it will try to execute the validate function: This step generates a log entry (p1, 2, exec-validate).
- * 3. At step 3, G2 writes a done-validate operation, meaning it successfully executed the validate function: This step generates a log entry (p1, 3, done-validate).

- * 4. At step 4, G2 writes an ack-validate operation, meaning it will send an acknowledgment to G1 regarding the done-validate: This step generates a log entry (p1, 4, ack-validate).

[3.2.](#) Log Storage Modes

Gateways store state mapped by logs. Gateways have private logs recording enterprise-sensitive data that can be used, for instance,

for analytics (enterprise log). Entries can include end-to-end cross-jurisdiction transaction latency and throughput.

Apart from the enterprise log, a state log can be public or private, centralized or decentralized. This log is meant to be shared with everyone with an Internet connection (public) or only within the gateway consortium (private). Logs can be stored locally or in a cloud service, per gateway (centralized), or in a decentralized infrastructure (i.e., decentralized ledger, decentralized database). We call the latter option decentralized log storage. The type of the state log depends on the trust assumptions among gateways and the access mode [[ODAP](#)].

In greater detail:

- * 1. Public decentralized log: log entries are stored on a decentralized public log (e.g., Ethereum blockchain, IPFS). Each gateway writes non-encrypted log entries to a decentralized log storage. Although this is the best option for providing accountability of gateways, availability, and integrity of the logs, leading to shorter dispute resolution, this can lead to privacy issues. The integrity of the log can be asserted by hashing the entries and comparing it to each stored hash on the decentralized log storage. A solution to the privacy problems could be given by gateways publishing a hash of the log entry plus metadata to the decentralized log storage instead of the log entries. Although this is a first step towards resolving privacy issues, a tradeoff with data availability is done. In particular, this choice leads to lower availability guarantees since a gateway needs to wait for the counter-party gateway to deliver the logs in case logs need to be shared. In this case, the decentralized log storage acts as a notarizing service. This mode is recommended when gateways operate in the Relay Mode: Client-initiated Gateway to Gateway. This mode can also be used by the Direct Mode: Client to Multiple Gateway access mode because gateways may need to share state between themselves. Note: the difference between the mentioned modes is that in Direct Mode: Client to Multiple Gateway, a single client/organization controls all the gateways, whereas, in the Relay Mode, gateways are controlled by different organizations.

- * 2. Public centralized log: log entries are published in a

bulletin that more organizations control. That bulletin can be updated or removed at any time. Accountability is only guaranteed provided that there are multiple copies of such bulletin by conflicting parties. Availability and integrity can be obtained via redundancy.

- * 3. Private centralized log. Each gateway stores logs locally or in a cloud in the private log storage mode but does not share them by default with other gateways. If needed, logs are requested from the counter-party gateway. Saving logs locally is faster than saving them on the respective ledger since issuing a transaction is several orders of magnitude slower than writing on a disk or accessing a cloud service. Nonetheless, this model delivers weaker integrity and availability guarantees.

Each log storage mode provides a different process to recover the state from crashes. In the private log, a gateway requires the most recent log from the counter-party gateway. This mode is the one where the most trust is needed. The gateway publishes hashes of log entries and metadata on a decentralized log storage in the centralized public log. Gateways who need the logs request them from other gateways and perform integrity checks of the received logs. In the public decentralized mode, the gateways publish the plain log entries on decentralized log storage. This is the most trustless and decentralized mode of operation.

By default, if there are gateways from different institutions involved in an asset transfer, the storage mode should be a decentralized log storage. The decentralized log storage can provide a common source of truth to solve disputes and maintain a shared state, alleviating trust assumptions between gateways.

[3.3.](#) Log Storage API:

The log storage API allows developers to be abstracted from the log storage support, providing a standardized way to interact with logs (e.g., relational vs. non-relational, local vs. on-chain). It also handles access control if needed.

Function	Parameters	En
Append log entry	logId - log entry to be appended	PO
Obtains a log entry	id - log entry id	GE
Obtains the length of the log	None	GE
Obtains the difference between a given log and a current log	log - log to be compared	P
Obtains the last log entry	None	GE
Obtains the whole log	None	GE

Figure 2

The following table maps the respective return values and response examples:

Returns	Response Example
The entry index of the last log (string)	HTTP/1.1 200 OK Cache-Control: private Date
A log entry	HTTP/1.1 200 OK Cache-Control: private Date
The length of the log (string)	HTTP/1.1 200 OK Cache-Control: private Date
The difference between two logs	HTTP/1.1 200 OK Cache-Control: private Date
A log entry	HTTP/1.1 200 OK Cache-Control: private Date
The log	HTTP/1.1 200 OK Cache-Control: private Date

Figure 3

[3.3.1.](#) Response Codes

The log storage API MUST respond with return codes indicating the failure (error 5XX) or success of the operation (200). The application may carry out a further operation in the future to determine the ultimate status of the operation.

The log storage API response is in JSON format and contains two fields: 1) `success`: true if the operation was successful, and 2) `response_data`: contains the payload of the response generated by the log storage API.

[4.](#) Format of log entries

A gateway stores the log entries in its log, and they capture gateways operations. Entries account for the current status of one of the three ODAP flows: Transfer Initiation flow, Lock-Evidence flow, and Commitment Establishment flow.

The recommended format for log entries is JSON, with protocol-specific mandatory fields supporting a free format field for plaintext or encrypted payloads directed at the DLT gateway or an underlying DLT. Although the recommended format is JSON, other formats can be used (e.g., XML).

The mandatory fields of a log entry, that are generated by ODAP, are:

- * `Version`: ODAP protocol Version (major, minor).
- * `Session ID`: unique identifier (UUIDv2) representing a session.
- * `Sequence Number`: monotonically increasing counter that uniquely represents a message from a session.
- * `ODAP Phase`: current ODAP phase.
- * `Resource URL`: Location of Resource to be accessed.
- * `Developer URN`: Assertion of developer / application identity.

- * Action/Response: GET/POST and arguments (or Response Code).
- * Credential Profile: Specify type of auth (e.g. SAML, OAuth, X.509).
- * Credential Block: Credential token, certificate, string.
- * Payload Profile: Asset Profile provenance and capabilities.

- * Application Profile: Vendor or Application specific profile.
- * Payload: Payload for POST, responses, and native DLT txns. The payload is specific to the current ODAP phase.
- * Payload Hash: hash of the current message payload.

In addition to the attributes that belong to ODAP s schema, each log entry REQUIRES the following attributes:

- * timestamp REQUIRED: timestamp referring to when the log entry was generated (UNIX format).
- * source_gateway_pubkey REQUIRED: the public key of the gateway initiating a transfer.
- * source_gateway_dlt_system REQUIRED: the ID of the source DLT.
- * recipient_gateway_pubkey REQUIRED: the public key of the gateway involved in a transfer.
- * recipient_gateway_dlt_system REQUIRED: the ID of the recipient gatewayinvolved in a transfer.
- * logging_profile REQUIRED: contains the profile regarding the logging procedure. Default is local store.
- * Message_signature REQUIRED: Gateway EDCSA signature over the log entry.
- * Last_entry_hash REQUIRED: Hash of previous log entry.
- * Access_control_profile REQUIRED: the profile regarding the

confidentiality of the log entries being stored. Default is only the gateway that created the logs can access them.

- * **Operation:** the high level operation being executed by the gateway on that step. There are five types of operations: Operation init- states the intention of a node to execute a particular operation; Operation exec- expresses that the node is executing the operation; Operation done- states when a node successfully executed a step of the protocol; Operation ack- refers to when a node acknowledges a message received from another (e.g., command executed); Operation fail- occurs when an agent fails to execute a specific step.

Optional field entries are:

- * **source_gateway_uid** OPTIONAL: the uid of the source gateway involved in a transfer.
- * **recipient_gateway_uid** : the uid of the recipient gateway involved in a transfer.
- * **recovery message:** the type of recovery message, if gateway is involved in a recovery procedure.
- * **recovery payload:** the payload associated with the recovery message.

Example of a log entry created by G1, corresponding to locking an asset (phase 2.3 of the ODAP protocol) :

```
{
"sessionId": "4eb424c8-ae4d-4e9e-a321-a160ac3909ac",
"seqNumber": 6,
"phaseId": "lock",
"sourceGatewayId": "5.47.165.186",
"sourceDltId": "Hyperledger-Fabric-JusticeChain",
"targetGatewayId": "192.47.113.116",
"targetDltId": "Ethereum",
"timestamp": "1606157330",
```

```

"payload": {
  "messageType": "2pc-log",
  "message": "LOCK_ASSET",
  "votes": "none"
},
"payloadHash": "80BCF1C7421E98B097264D1C6F1A514576D6C9F4EF04955FA3AEF1C0664B34E",
"logEntryHash": "[...]"
}

```

Figure 4

Example of a log entry created by G2, acknowledging G1 locking an asset (phase 2.4 of the ODAP protocol) :

```

{
  "sessionId": "4eb424c8-aead-4e9e-a321-a160ac3909ac",
  "seqNumber": 7,
  "phaseId": "lock",
  "sourceGatewayId": "5.47.165.186",
  "sourceDltId": "Hyperledger-Fabric-JusticeChain",
  "targetGatewayId": "192.47.113.116",
  "targetDltId": "Ethereum",
  "timestamp": "1606157333",
  "payload": {
    "messageType": "2pc-log",
    "message": "LOCK_ASSET_ACK",
    "votes": "none"
  }
}

```

Figure 5

This section defines general considerations about crash recovery. ODAP-2PC is the application of the gateway crash recovery mechanism to asset transfers across all ODAP phases.

[5.1.](#) Crash Recovery Model

Gateways can fail by crashing (i.e., becoming silent). In order to be able to recover from these crashes, gateways store log entries in a persistent data storage. Thus, gateways can recover by obtaining the latest successful operation and continuing from there. We consider two recovery models:

- * 1. Self-healing mode: assumes that after a crash, a gateway eventually recovers. The gateway does not lose its long-term keys (public-private key pair) and can reestablish all TLS connections.
- * 2. Primary-backup mode assumes that a gateway may never recover after a crash but that this failure can be detected by timeout [[AD76](#)]. If the timeout is exceeded, a backup gateway detects that failure unequivocally and takes the role of the primary gateway. The failure is detected using heartbeat messages and a conservative period.

In both modes, after a gateway recovers, the gateways follow a general recovery procedure (in [Section 6.2](#) explained in detail for each phase):

- * 1. Crash communication: using the self-healing or primary-backup modes, a node recovers. After that, it sends a message RECOVER to the counterparty gateways.
- * 2. State update: The gateway syncs its state with the latest state, either by requesting it from the decentralized log storage or other gateways (depending on the log storage mode). If a decentralized log storage is available, the crashed gateway attempts to update its local log, using getLogDiff from the shared log. If there is no shared log, the crashed gateway needs to synchronize itself with the counterparty gateway by querying the counterparty gateway with a recovery message RECOVER containing

the latest log before the crash. The counterparty gateway sends back a RECOVER-UPDATE message with its log. The recovered gateway can now reconstruct the updated log via getLogDiff, and derive the current state of the asset transfer. The gateways now share the same state and can proceed with its operation.

- * 3. Recovery communication: The gateway and informs other gateways of the recovery with a recovery confirmation message is sent (RECOVERY-CONFIRM), and the respective acknowledgment is sent by the counterparty gateway (RECOVERY-ACK).

Finally, the gateway resumes the normal execution of ODAP.

[5.2.](#) Recovery Procedure

The previous section explained the general procedure that gateways follow upon crashing. In more detail, for each ODAP phase, we define the recovery procedure called ODAP-2PC:

[5.2.1.](#) Transfer Initiation Flow

This phase of ODAP follows the Crash Recovery Model from [Section 6.1](#).

[5.2.2.](#) Lock-Evidence Flow

This phase of ODAP follows the Crash Recovery Model from [Section 6.1](#). Note that, in this phase, distributed ledgers were changed by gateways. The crash gateways' recovery should take place in less than the timeout specified for the asset transfer. Otherwise, the rollback protocol present in the next section is applied.

[5.2.3.](#) Commitment Establishment Flow

This phase of ODAP follows the Crash Recovery Model from [Section 6.1](#) and extra steps because in the third phase, distributed gateways changed ledgers. As transactions cannot be undone on blockchains,

reverting a transaction includes issuing new transactions (with the contrary effect of the ones to be reverted). We use a rollback list [[HERMES](#)] to keep track of which transaction may be rolled back. The crash recovery protocol for the Commitment Establishment Flow is as follows (steps according to Figure 4 [[HERMES](#)]):

- * 1. Rollback lists for all the gateways involved are initialized.
- * 2. On step 2.3, add a pre-lock transaction to the source gateway rollback list.
- * 3. On step 3.2, if the request is denied, abort the transaction and apply rollbacks on the source gateway.
- * 4. On step 3.3, add a lock transaction to the source gateway rollback list.
- * 5. On step 3.4, if the commit fails, abort the transaction and apply rollbacks on the source gateway.
- * 6. On step 3.5, add a create asset transaction to the rollback list of the recipient gateway.
- * 7. On step 3.8, if the commit is successful, ODAP terminates.
- * 8: Otherwise, if the last commit is unsuccessful, then abort the transaction and apply rollbacks to both gateways.

[5.3.](#) ODAP-2PC Messages

ODAP-2PC messages are used to recover from crashes at the several ODAP phases. These messages inform gateways of the current state of a recovery procedure. ODAP-2PC messages follow the log format from [Section 4](#).

[5.3.1.](#) RECOVER

A recover message is sent from the crashed gateway to the counterparty gateway, sending its most recent state. This message type is encoded on the recovery message field of an ODAP log.

The parameters of the recovery message payload consists of the following:

- * ODAP phase: latest ODAP phase registered.
- * Sequence number: latest sequence number registered.
- * Last_entry_hash REQUIRED: Hash of previous log entry.

[5.3.2.](#) RECOVER-UPDATE

The recover update message is sent by the counterparty gateway after receiving a recover message from a recovered gateway. The recovered gateway informs of its current state (via the current state of the log). The counterparty gateway now calculates the difference between the log entry corresponding to the received sequence number from the recovered gateway and the latest sequence number (corresponding to the latest log entry). This state is sent to the recovered gateway.

The parameters of the recover update payload consists of the following:

- * recovered logs: the list of log messages that the recovered gateway needs to update.

[5.3.3.](#) RECOVER-UPDATE ACK

The recover-update ack message (response to RECOVER-UPDATE) states if the recovered gateway's logs has been successfully updated. If inconsistencies are detected, the recovered gateway answers with initiates a dispute (RECOVER-DISPUTE message).

The parameters of this message consists of the following:

- * success: true/false.
- * entries changed: list of hashes of log entries that were appended to the recovered gateway log.

[5.3.4.](#) RECOVER-SUCCESS

The recover-ack message is sent by the counterparty gateway to the recovered gateway acknowledging that the state is synchronized.

The parameters of this message consists of the following:

- * success: true/false.

Internet-Draft

Gateway Crash Recovery

October 2021

[5.3.5.](#) ROLLBACK

A rollback message is sent by a gateway that initiated a rollback as defined by ODAP-2PC.

The parameters of this message consists of the following:

- * success: true/false.
- * actions performed: actions performed to rollback a state (e.g., UNLOCK; BURN).
- * proofs: TBD.

[5.4.](#) Examples

There are several situations when a crash may occur.

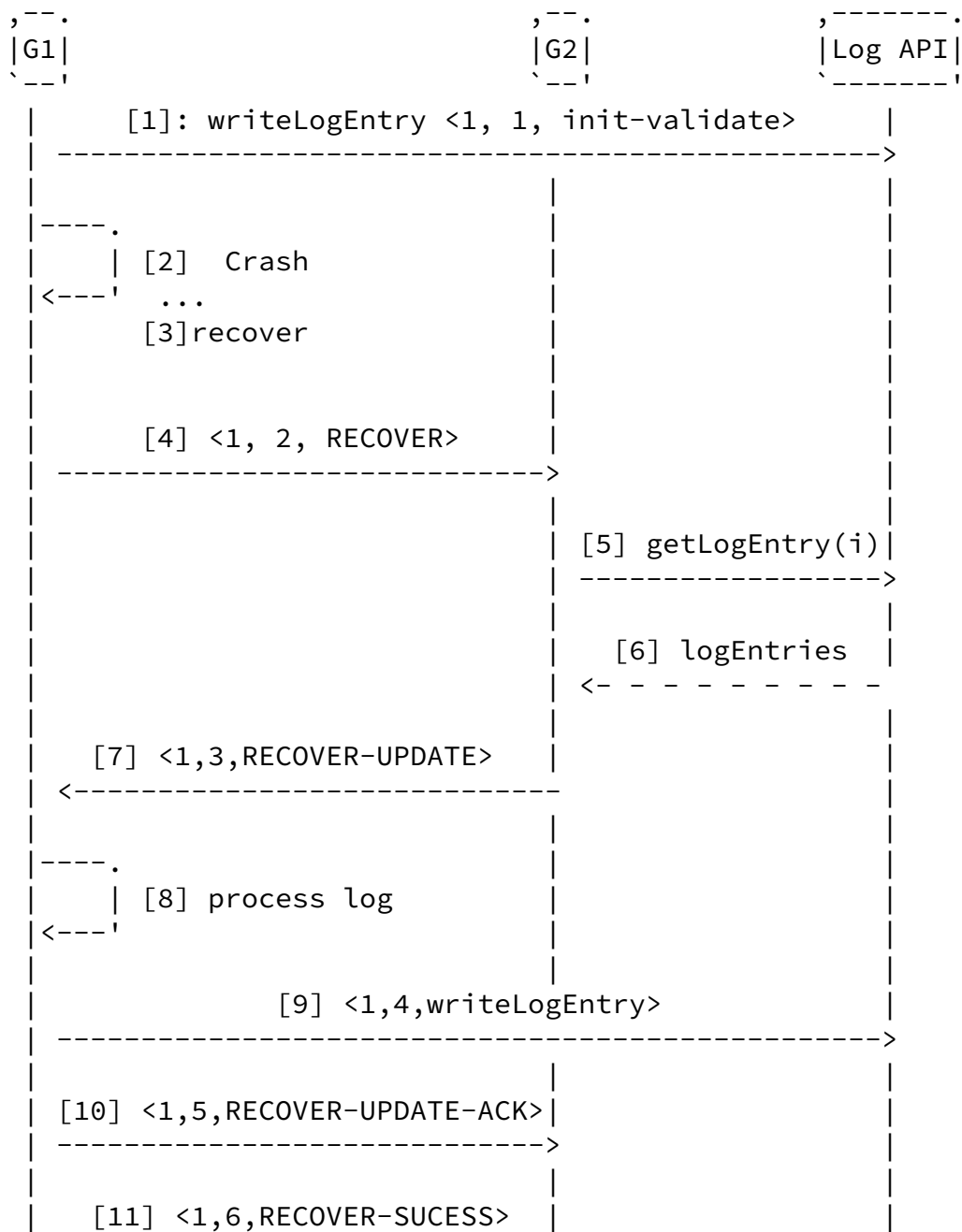
[5.4.1.](#) Crashing before issuing a command to the counterparty gateway

The following figure represents the source gateway (G1) crashing before it issued an init command to the recipient gateway (G2).

Internet-Draft

Gateway Crash Recovery

October 2021



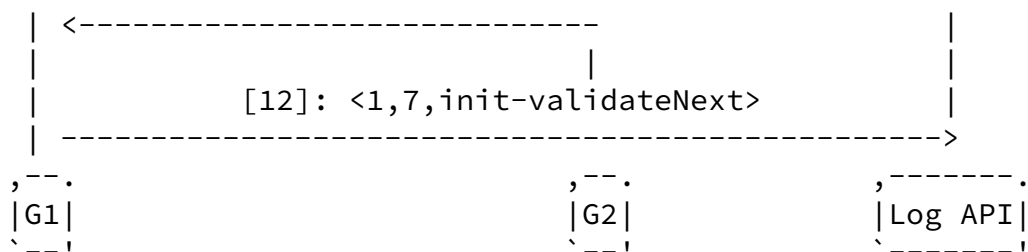
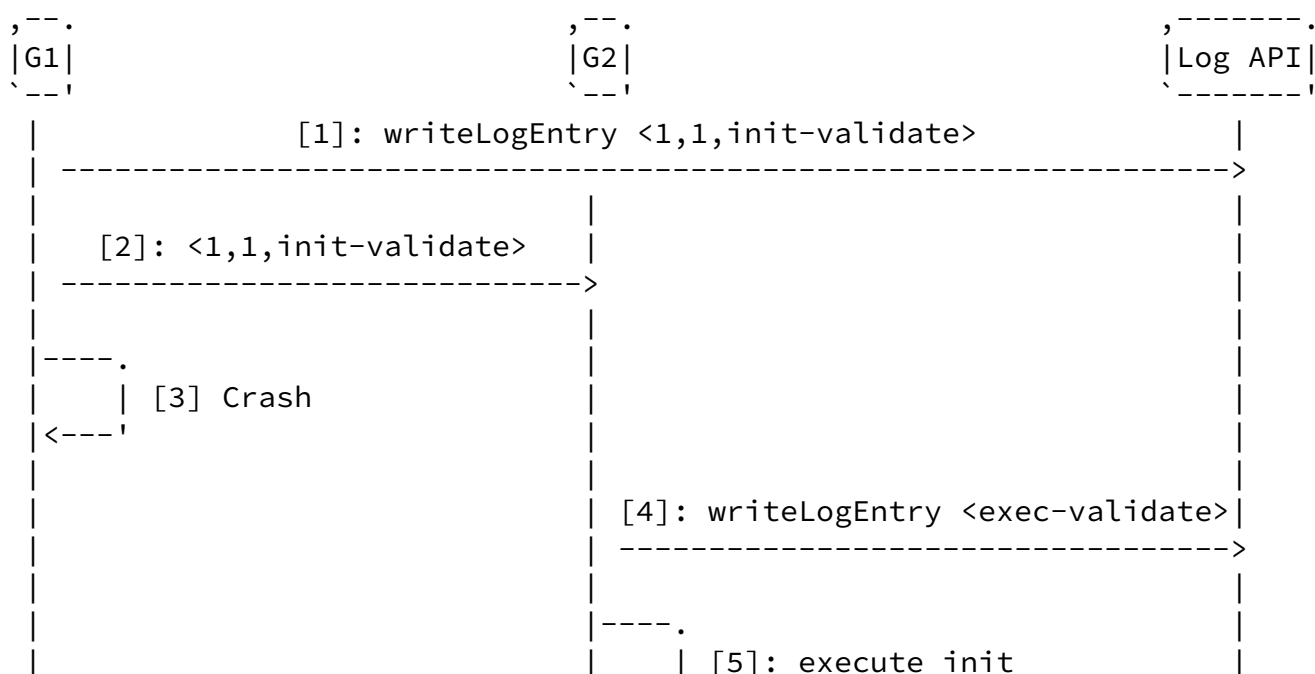
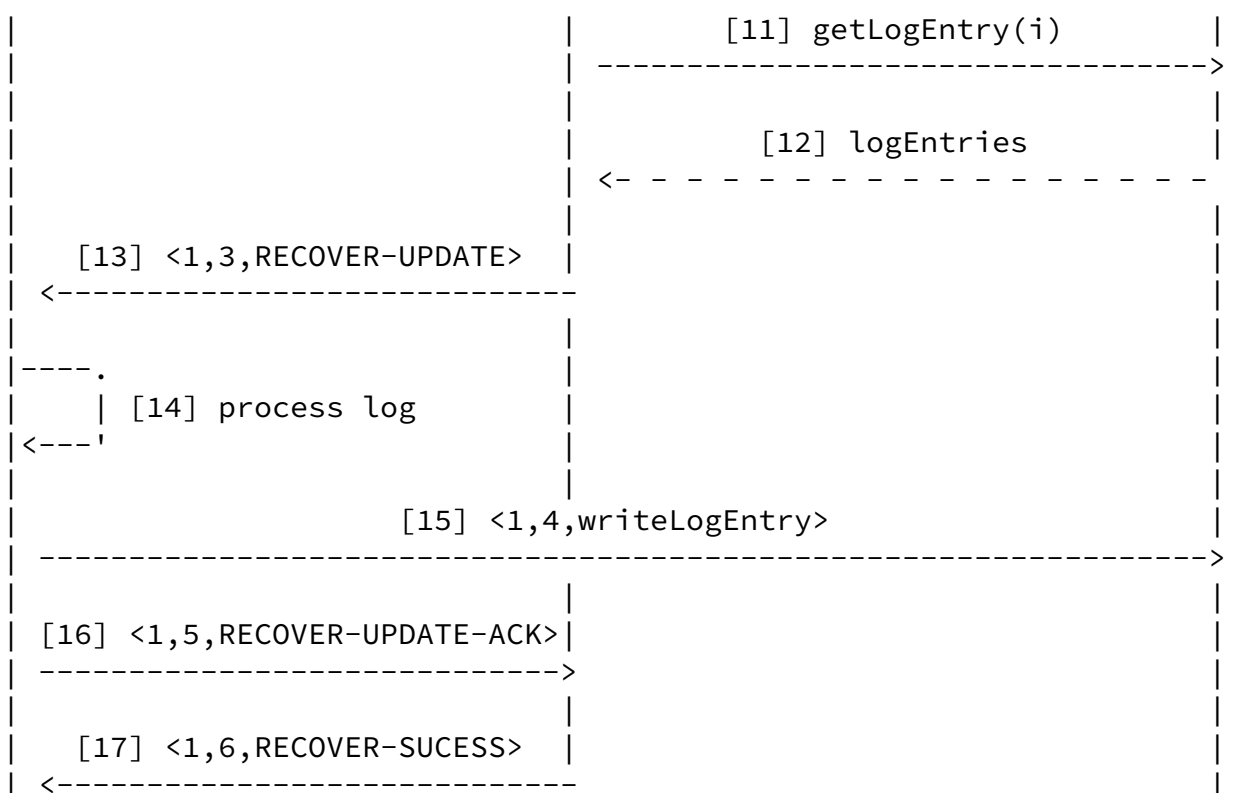
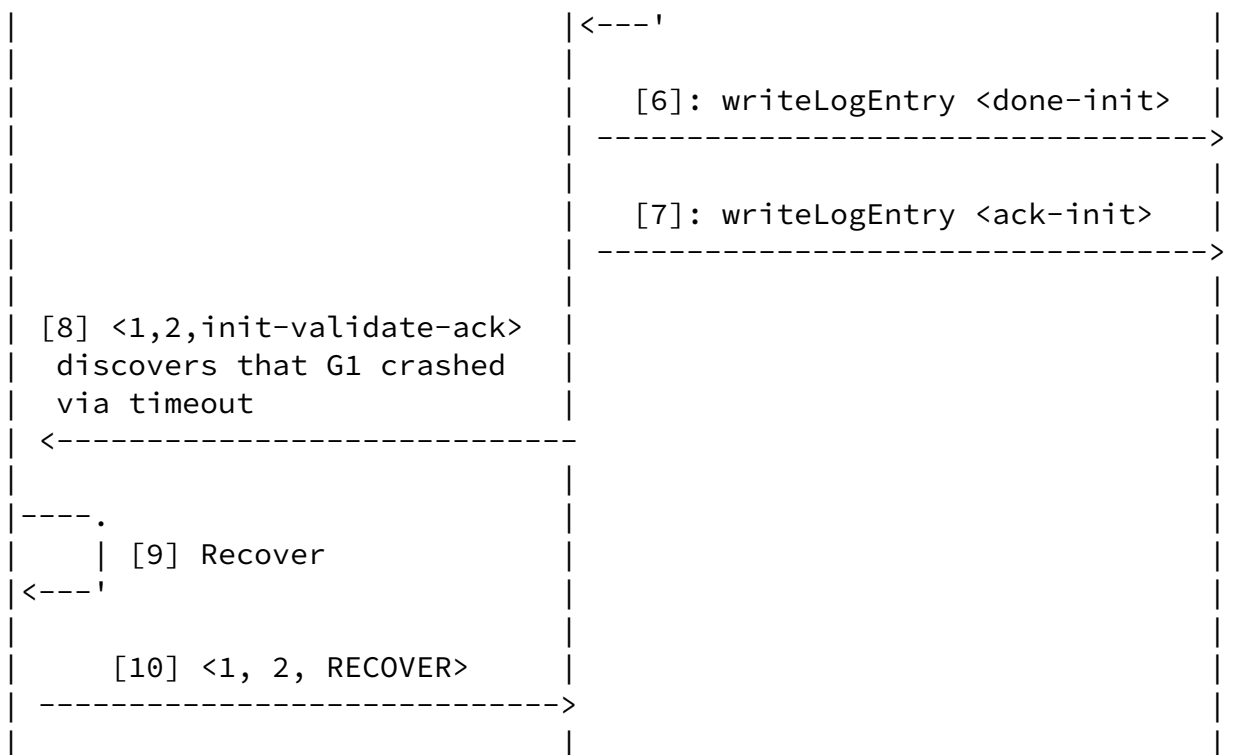


Figure 6

[5.4.2.](#) Crashing after issuing a command to the counterparty gateway

The second scenario requires further synchronization (figure below). At the retrieval of the latest log entry, G1 notices its log is outdated. It updates it upon necessary validation and then communicates its recovery to G2. The process then continues as defined.





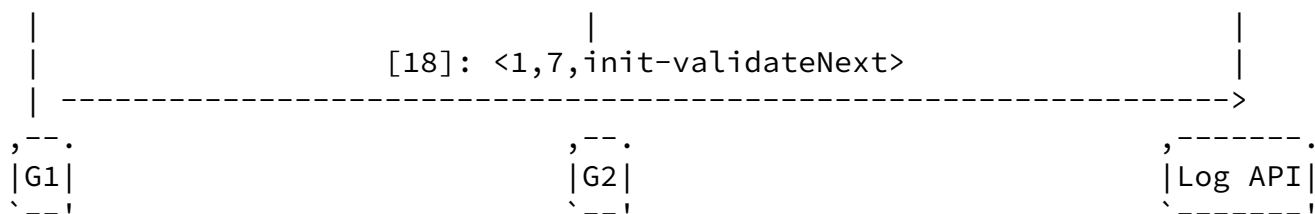


Figure 7

6. Security Considerations

We assume a trusted, authenticated, secure, reliable communication channel between gateways (i.e., messages cannot be spoofed and/or altered by an adversary) using TLS/HTTPS [TLS]. Clients support acceptable credential schemes such as OAuth2.0. We assume the storage service used provides the means necessary to assure the logs' confidentiality and integrity, stored and in transit. The service must provide an authentication and authorization scheme, e.g., based on OAuth and OIDC [OIDC], and use secure channels based on TLS/HTTPS [TLS]. The present protocol is crash fault-tolerant, meaning that it handles gateways that crash for several reasons (e.g., power outage). The present protocol does not support Byzantine faults, where gateways can behave arbitrarily (including being malicious). This implies that both gateways are considered trusted. We assume logs

are not tampered with or lost. Log entries need integrity, availability, and confidentiality guarantees, as they are an attractive point of attack [BVC19]. 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 its creator. Availability is guaranteed by the usage of the log storage API that connects a gateway to a 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 who can access the log entry 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+password authentication in local storage vs. blockchain-based access control in a DLT). For extra guarantees, the nodes running the log storage API (or the gateway nodes themselves) can be protected by hardening technologies such as Intel SGX [CD16].

7. References

7.1. Normative References

- [ODAP] Hargreaves, M. and T. Hardjono, "Open Digital Asset Protocol, October 2020, IETF, [draft-hargreaves-odap-00](#).", October 2020, <<https://datatracker.ietf.org/doc/draft-hargreaves-odap/>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [TLS] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3?", [RFC 8446](#).", 2018, <<https://tools.ietf.org/rfc/rfc8446>>.

7.2. Informative References

- [AD76] Alsberg, P. and D. Day, "A principle for resilient sharing of distributed resources. In Proc. of the 2nd Int. Conf. on Software Engineering", 1976, <978-0-201-10715-9>.
- [BHG87] Bernstein, P., Hadzilacos, V., and N. Goodman, "Concurrency Control and Recovery in Database Systems, Chapter 7. Addison Wesley Publishing Company", 1987, <<https://doi.org/10.3389/fbloc.2019.00024>>.

- [BVC19] Belchior, R., Vasconcelos, A., and M. Correia, "Towards Secure, Decentralized, and Automatic Audits with Blockchain. European Conference on Information Systems", 2019, <https://aisel.aisnet.org/ecis2020_rp/68/>.
- [Clar88] Clark, D., "The Design Philosophy of the DARPA Internet Protocols, ACM Computer Communication Review, Proc SIGCOMM

88, vol. 18, no. 4, pp. 106–114", August 1988.

- [HERMES] Belchior, R., Vasconcelos, A., Correia, M., and T. Hardjono, "HERMES: Fault-Tolerant Middleware for Blockchain Interoperability", 2021, <https://www.techrxiv.org/articles/preprint/HERMES_Fault-Tolerant_Middleware_for_Blockchain_Interoperability/14120291>.
- [HS2019] Hardjono, T. and N. Smith, "Decentralized Trusted Computing Base for Blockchain Infrastructure Security, Frontiers Journal, Special Issue on Blockchain Technology, Vol. 2, No. 24", December 2019, <<https://doi.org/10.3389/fbloc.2019.00024>>.
- [OIDC] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.
- [SRC84] Saltzer, J., Reed, D., and D. Clark, "End-to-End Arguments in System Design, ACM Transactions on Computer Systems, vol. 2, no. 4, pp. 277–288", November 1984.

Authors' Addresses

Rafael Belchior
INESC-ID, Instituto Superior Técnico

Email: rafael.belchior@tecnico.ulisboa.pt

Miguel Correia
INESC-ID, Instituto Superior Técnico

Email: miguel.p.correia@tecnico.ulisboa.pt

Thomas Hardjono
MIT

Email: hardjono@mit.edu