Secure Asset Transfer Protocol
Internet-Draft
Intended status: Informational
Expires: 31 July 2026

R. Belchior
INESC-ID, Técnico Lisboa, Blockdaemon
M. Correia
A. Augusto
INESC-ID, Técnico Lisboa
T. Hardjono
MIT

27 January 2026

Secure Asset Transfer Protocol (SATP) Gateway Crash Recovery Mechanism
draft-belchior-satp-gateway-recovery-04

Abstract

This memo describes the crash recovery mechanism for the Secure Asset Transfer Protocol (SATP). The goal of this draft is to specify the message flow that implements a crash recovery mechanism, composed of self-healing and rollback sub-protocols. The mechanism assures that gateways running SATP are able to recover faults, enforcing ACID properties for asset transfers across ledgers (i.e., double spend does not occur).

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://ietf-satp.github.io/draft-belchior-satp-gateway-recovery/draft-belchior-satp-gateway-recovery.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-belchior-satp-gateway-recovery/.

Discussion of this document takes place on the Secure Asset Transfer Protocol Working Group mailing list (mailto:sat@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/sat/. Subscribe at https://www.ietf.org/mailman/listinfo/sat/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-satp/draft-belchior-satp-gateway-recovery.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Copyright Notice

Table of Contents

1. Introduction

Gateway systems that perform digital asset transfers among networks 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 networks.

The crash recovering mechanism is applied to a version of SATP [I-D.draft-ietf-satp-core] using either 2PC or 3PC, which are atomic commitment protocol (ACP). 2PC and 3PC considers two roles: a coordinator who manages the protocol's execution and participants who manage the resources that must be kept consistent. The origin gateway plays the ACP role of Coordinator, and the destination Gateway plays the Participant role in relay mode. Gateways exchange messages corresponding to the protocol execution, generating log entries for each one. The crash recovery draft does not depend on the specific SATP messages, but defines procedures to recover from crashes and for rollbacks, independently of the specific protocol phase being executed.

Log entries are organized into logs. Logs enable either the same or other backup gateways to resume any phase of SATP. This log can also serve as an accountability tool in case of disputes. 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).

This memo proposes:

(i) the logging model of the crash recovery mechanism; (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 collection of services which connects to a minimum of one network or system, and which implements the secure asset transfer protocol.

* Primary Gateway: The node of a network that has been selected or elected to act as a gateway in an asset transfer.

* Backup Gateway: The node of a network 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.

* Origin Gateway: The gateway that initiates the transfer protocol. Acts as a coordinator of the ACP and mediates the message flow.

* Destination Gateway: The gateway that is the target of an asset transfer. It follows instructions from the origin Gateway.

* Log: Set of log entries such that those are ordered by the time of its creation.

* Public (or Shared) Log: log where several gateways can read and write from it.

* Private Log: log where only one gateway 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 taken from a network are persisted into the other network. 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: the models allowing a system to keep operating correctly despite having a set of faulty components.

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 SATP, 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).

The following example shows a simplified version log referring to the transfer initiation flow SATP 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 4).

3.1. Example

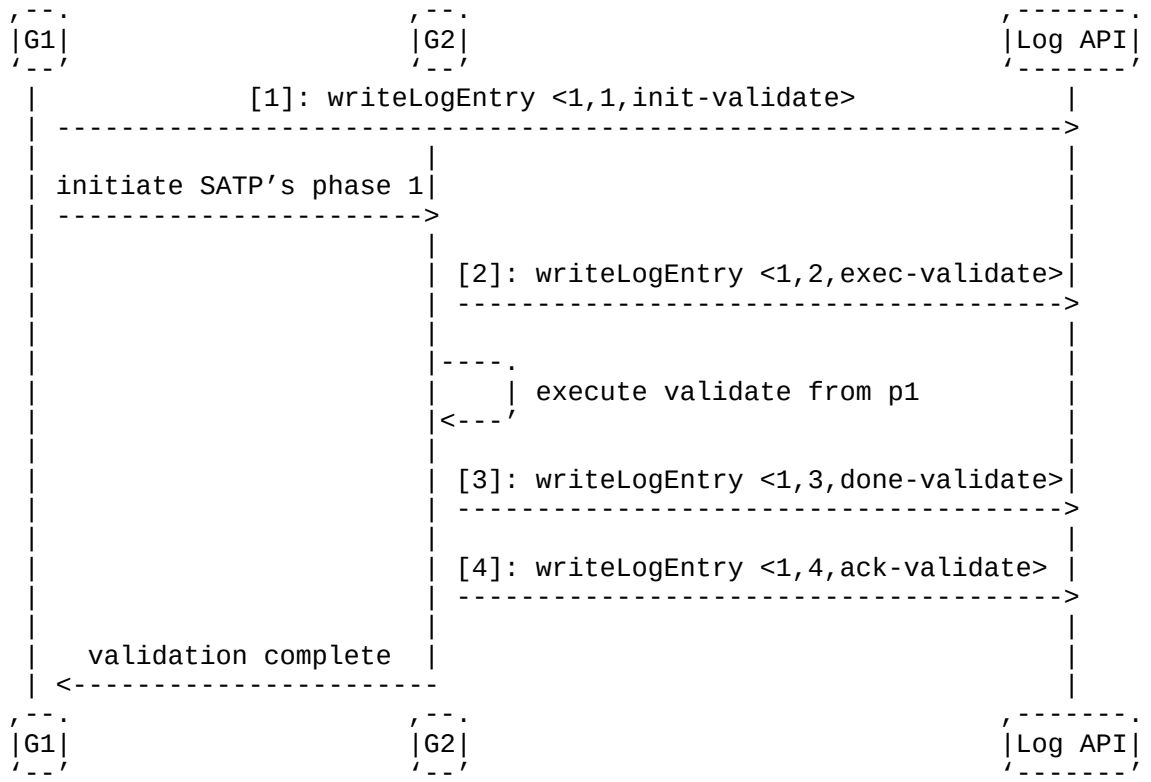


Figure 1

This example shows the sequence of logging operations over part of the first phase of SATP (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).

Without loss of generality, the above logging model applies to all phases of SATP.

3.2. SATP Example

This example showcases the logging procedure step 2.4 of SATP (lock-assertion) by both gateways.

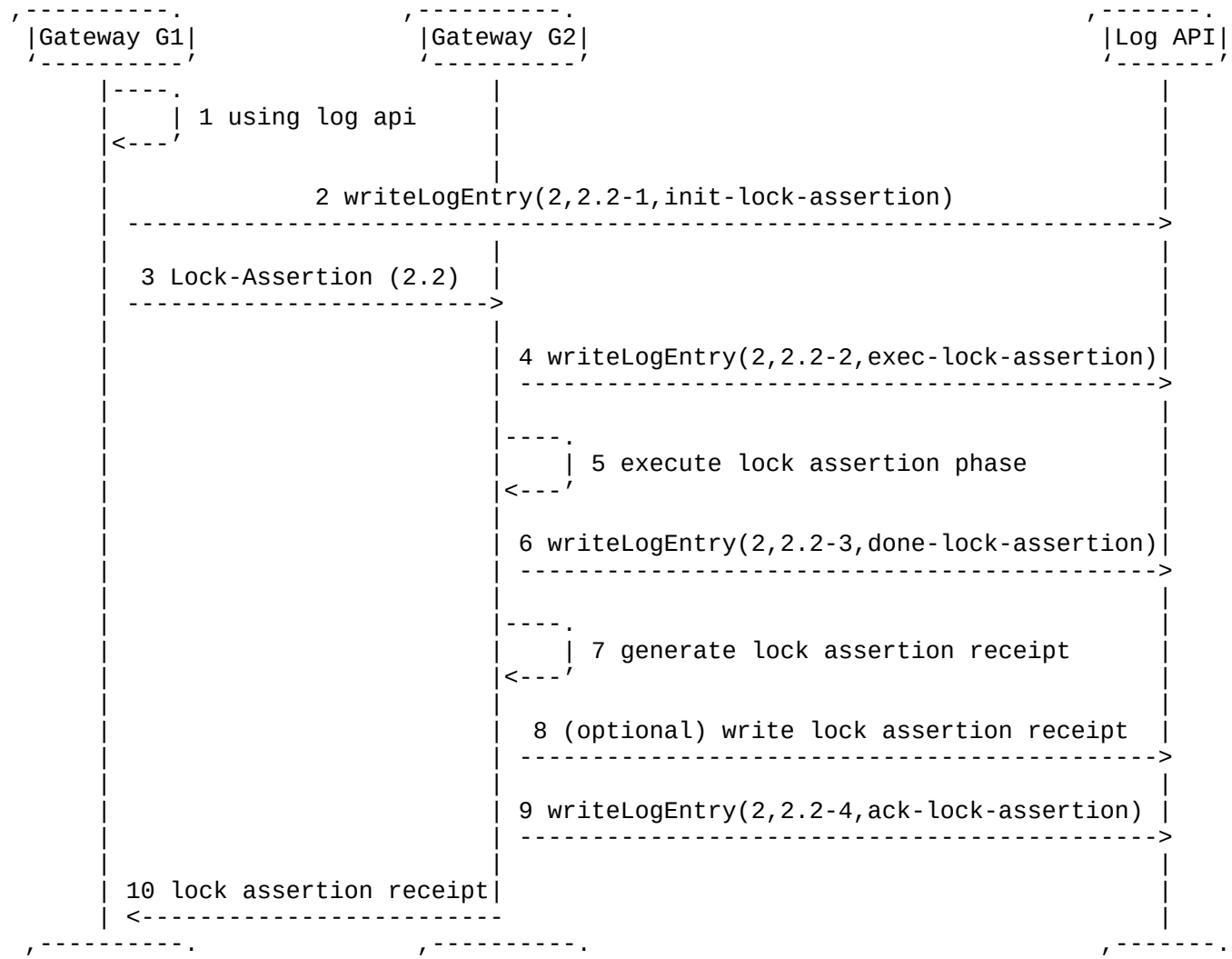


Figure 2

3.3. Log Storage Modes

Gateways store state that is captured by logs. Gateways have private logs recording enterprise-sensitive data that can be used, for instance, for analytics. 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 log access mode.

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 leak of information which 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 exists. In particular, this choice leads to lower availability guarantees since a gateway needs to wait for the counterparty 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 counterparty 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.

4. Private decentralized log. Each gateway stores logs in a private blockchain, and are shared with other gateways by default.

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 counterparty 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.

3.4. 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	Endpoint
Append log entry	logId - log entry to be appended	POST / writeLogEntry/:logId Host: example.org Accept: application/json
Obtains a log entry	id - log entry id	GET getLogEntry/:id Host: example.org
Obtains the length of the log	None	GET getLogLength Host: example.org
Obtains the difference between a given log and a current log	log - log to be compared	POST /getLogDiff/:log Host: example.org
Obtains the last log entry	None	GET getLastEntry Host: example.org
Obtains the whole log	None	GET getLog Host: example.org

Figure 3

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: Mon, 02 Mar 2020 05:07:35 GMT Content-Type: application/json { "success": true, "response_data":"2" }
A log entry	HTTP/1.1 200 OK Cache-Control: private Date: Mon, 02 Mar 2020 05:07:35 GMT Content-Type: application/json { "success": true, "response_data": {...} }

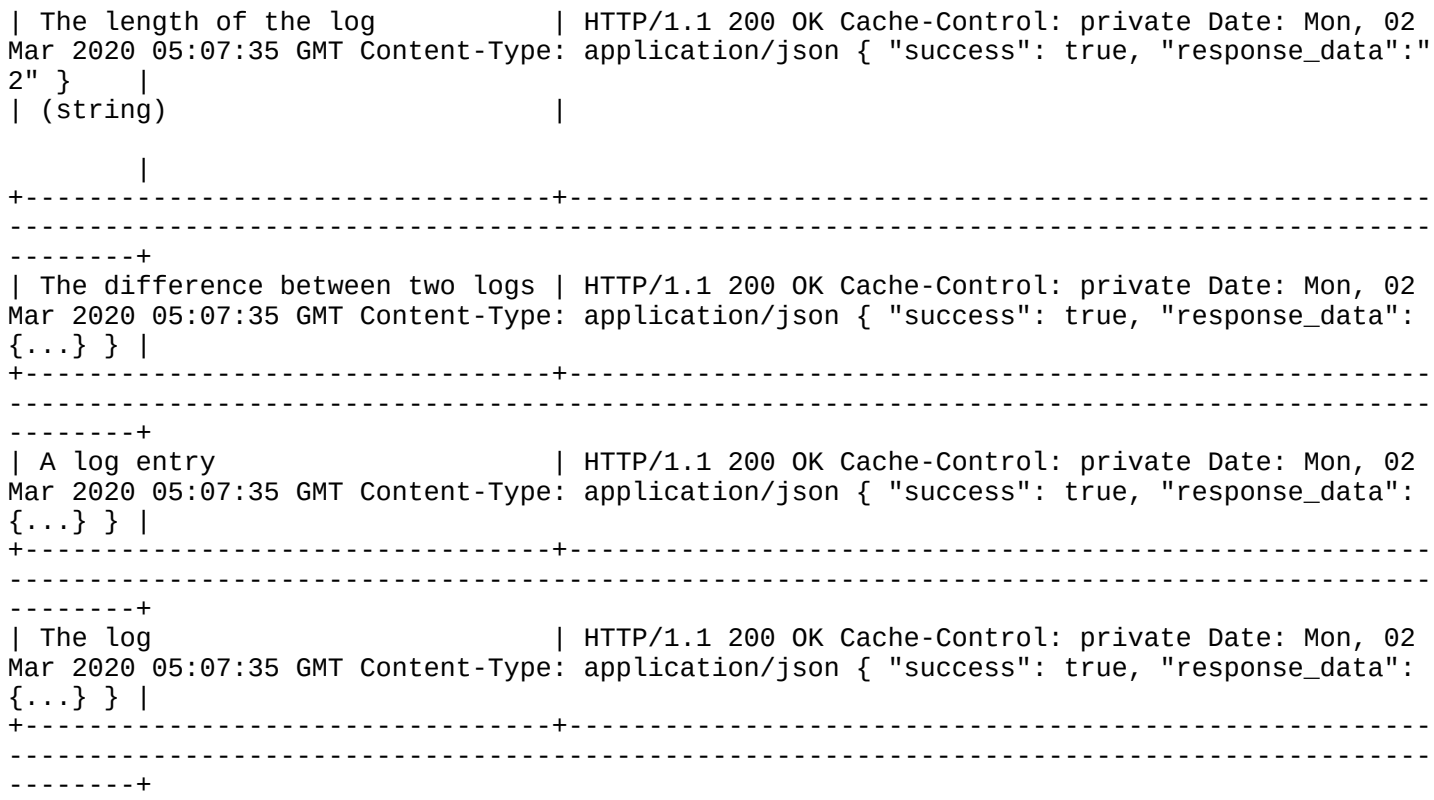


Figure 4

3.4.1. 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 SATP 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 SATP gateway or an underlying network. Although the recommended format is JSON, other formats can be used (e.g., XML).

The mandatory fields of a log entry, that SATP generates, are:

* Version: SATP protocol Version (major, minor).

* Session ID: a unique identifier (UUIDv2) representing a session.

* Context ID: a unique identifier (UUIDv2) representing a session context [I-D.draft-avrilionis-satp-setup-stage-01].

* Sequence Number: monotonically increasing counter that uniquely represents a message from a session.

* SATP Phase: current SATP 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 the 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 network transactions. The payload is specific to the current SATP phase.

* Payload Hash: hash of the current message payload.

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

* timestamp REQUIRED: timestamp referring to when the log entry was generated (UNIX format).

* origin_gateway_pubkey REQUIRED: the public key of the gateway initiating a transfer.

* origin_gateway_system REQUIRED: the ID of the source network.

* destination_gateway_pubkey REQUIRED: the public key of the gateway involved in a transfer.

* destination_gateway_system REQUIRED: the ID of the destination Gateway involved in a transfer.

* logging_profile REQUIRED: contains the profile regarding the logging procedure. Default is a local store.

* Message_signature REQUIRED: Gateway ECDSA 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 that 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 executes a step of the protocol; Operation ack- refers to when a node acknowledges a message received from another (e.g., the command executed); Operation fail- occurs when an agent fails to execute a specific step.

Optional field entries are:

* recovery message: the type of recovery message, if the 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 SATP protocol):


```
{
  "Version": "1.0",
  "Session ID": "123e4567-e89b-12d3-a456-426655440000",
  "Sequence Number": 1,
  "SATP Phase": "Initialization",
  "Resource URL": "http://myresource.com",
  "Developer URN": "urn:myapp:developerid",
  "Action/Response": "POST /myresource",
  "Credential Profile": "OAuth",
  "Credential Block": "ABC123TOKEN",
  "Payload Profile": "ProvenanceProfile1",
  "Application Profile": "AppProfile1",
  "Payload": "{ 'key1': 'value1', 'key2': 'value2' }",
  "Payload Hash": "abc123def456",
  "timestamp": 1646176142,
  "origin_gateway_pubkey": "abc123",
  "origin_gateway_system": "system1",
  "destination_gateway_pubkey": "def456",
  "destination_gateway_system": "system2",
  "logging_profile": "Local Store",
  "Message_signature": "ecdsa_signature_here",
  "Last_entry_hash": "hash_of_last_entry",
  "Access_control_profile": "GatewayOnly",
  "Operation": "init",
  "recovery message": "recovery_message_here",
  "recovery payload": "recovery_payload_here"
}
```


Figure 5

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

```
{
    "sessionId": "4eb424c8-aead-4e9e-a321-a160ac3909ac",
    "contextId": "5eb424c8-aead-4e9e-a321-a160ac3909ac",
    "seqNumber": 7,
    "phaseId": "lock",
    "originGatewayId": "5.47.165.186",
    "originNetworkId": "Hyperledger-Fabric-JusticeChain",
    "destinationGatewayId": "192.47.113.116",
    "destinationNetworkId": "Ethereum",
    "timestamp": "1606157333",
    "payload": {
    "messageType": "2pc-log",
    "message": "LOCK_ASSET_ACK",
    "votes": "none"
}
```


Figure 6

5. Crash Recovery Procedure

This section defines general considerations about crash recovery for the self-healing mode. Note that the procedure for the primary-backup mode is the same, but first has a session resumption process.

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-UPDATE-ACK), and the respective acknowledgment is sent by the counterparty gateway (RECOVERY-SUCCESS).

Finally, the gateway resumes the normal execution of SATP (session resumption).

5.2. Recovery Procedure

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

5.2.1. Transfer Initiation Flow

This phase of SATP follows the Crash Recovery Model from Section 6.1.

5.2.2. Lock-Evidence Flow

This phase of SATP 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

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 to keep track of which transaction may be rolled back. The crash recovery protocol for the Stage 2 (lock) and Stage 3 (mint) is as follows:

1. Rollback lists for all the gateways involved are initialized.

2. On step 2.1A, add a pre-lock transaction to the origin gateway rollback list.

3. On step 2.1B, if the request is denied, abort the transaction and apply rollbacks on the origin gateway.

4. On step 3.4A, add a lock transaction to the origin gateway rollback list.

5. On step 3.4B, if the commit fails, abort the transaction and apply rollbacks on the origin gateway.

6. On step 3.6A, add a create asset transaction to the rollback list of the destination gateway.

7. On step 3.8, if the commit is successful (confirmed by the ack final receipt), SATP terminates (3.9).

8: Otherwise, if the last commit is unsuccessful, then abort the transaction and apply rollbacks to both gateways.

5.3. Recovery Messages

SATP-2PC messages are used to recover from crashes at the several SATP phases. These messages inform gateways of the current state of a recovery procedure. SATP-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 SATP log.

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

* Session ID: a unique identifier (UUIDv2) representing a session.

* Context ID: a unique identifier (UUIDv2) representing a session context [I-D.draft-avrilionis-satp-setup-stage-01].

* Message Type REQUIRED: urn:ietf:SATP-2pc:msgtype:recover-msg.

* SATP phase: latest SATP phase registered.

* Sequence number: latest sequence number registered.

* Is_Backup REQUIRED: indicates whether the sender is a backup gateway or not.

* New Identity Public Key: The public key of the sender if it is a backup gateway.

* Last_entry_timestamp REQUIRED: Timestamp of last known log entry.

* Sender Signature REQUIRED. The digital signature of the sender.

5.3.2. RECOVER-UDPDATE

The counterparty gateway sends the recover update message 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 consist of the following:

* Session ID: a unique identifier (UUIDv2) representing a session.

* Context ID: a unique identifier (UUIDv2) representing a session context [I-D.draft-avrilionis-satp-setup-stage-01].

* Message Type REQUIRED: urn:ietf:SATP-2pc:msgtype:recover-update-msg.

* Hash Recover Message REQUIRED. The hash of previous message.

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

* Sender Signature REQUIRED. The digital signature of the sender.

5.3.3. RECOVER-SUCCESS

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

The counterparty gateway sends this message to sinalize the recovered gateway acknowledging that the state is synchronized.

The parameters of this message consist of the following:

* Session ID: a unique identifier (UUIDv2) representing a session.
* Context ID: a unique identifier (UUIDv2) representing a session context [I-D.draft-avrilionis-satp-setup-stage-01].
* Message Type REQUIRED: urn:ietf:SATP-2pc:msgtype:recover-update-ack-msg.
* Hash Recover Update Message REQUIRED. The hash of previous message.
* success: true/false.
* entries changed: list of hashes of log entries that were appended to the recovered gateway log.
* Sender Signature REQUIRED. The digital signature of the sender.

5.3.4. ROLLBACK

A rollback message is sent by a gateway that initiates a rollback. The parameters of this message consist of the following:

* Session ID: a unique identifier (UUIDv2) representing a session.
* Context ID: a unique identifier (UUIDv2) representing a session context [I-D.draft-avrilionis-satp-setup-stage-01].
* Message Type REQUIRED: urn:ietf:SATP-2pc:msgtype:rollback-msg.
* success: true/false.
* actions performed: actions performed to rollback a state (e.g., UNLOCK; BURN).
* proofs: a list of proofs specific to the network

* Sender Signature REQUIRED. The digital signature of the sender.

5.3.5. ROLLBACK-ACK

The counterparty gateway sends the rollback-ack message to the recovered gateway acknowledging that the rollback has been performed successfully.

The parameters of this message consist of the following:

* Session ID: a unique identifier (UUIDv2) representing a session.

* Context ID: a unique identifier (UUIDv2) representing a session context [I-D.draft-avrilionis-satp-setup-stage-01].

* Message Type REQUIRED: urn:ietf:SATP-2pc:msgtype:rollback-ack-msg.

* success: true/false.

* actions performed: actions performed to rollback a state (e.g., UNLOCK; BURN).

* proofs: a list of proofs specific to the network

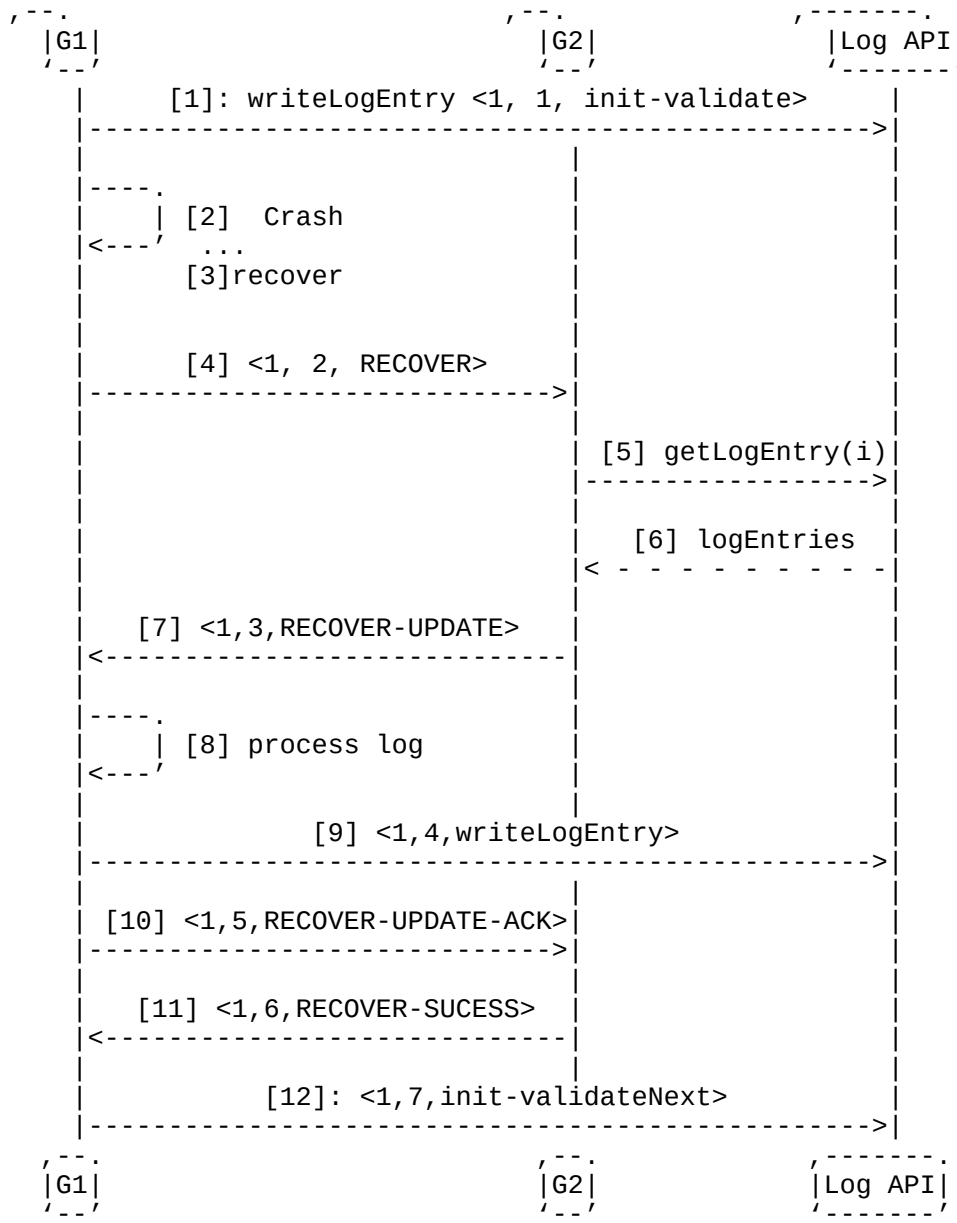
* Sender Signature REQUIRED. The digital signature of the sender.

5.4. Examples

There are several situations when a crash may occur.

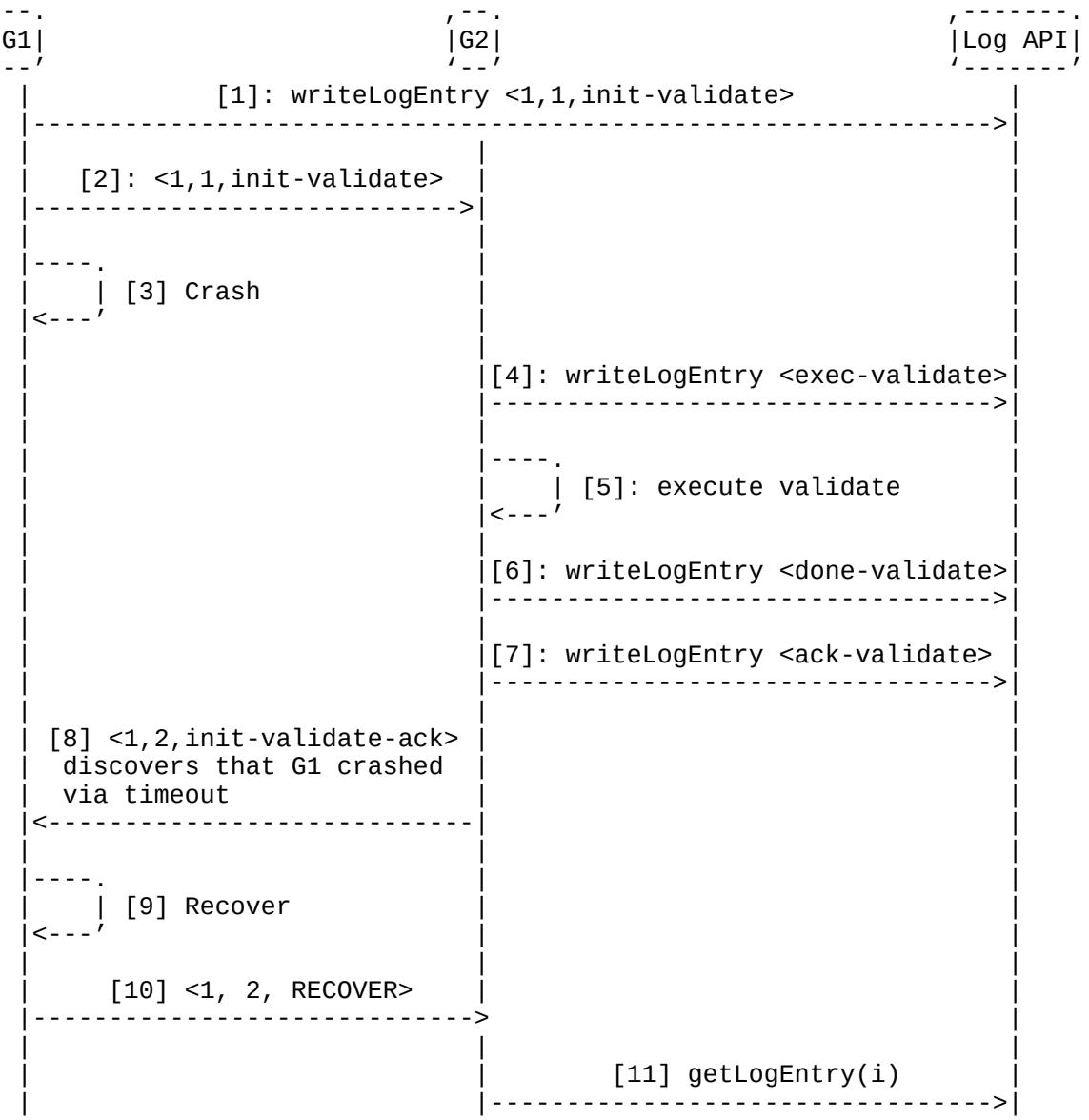
5.4.1. Crashing before issuing a command to the counterparty gateway

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


G1
G2
Log API
[1]: writeLogEntry <1, 1, init-validate>
[2] Crash
...
[3]recover
[4] <1, 2, RECOVER>
[5] getLogEntry(i)
[6] logEntries
[7] <1,3,RECOVER-UPDATE>
[8] process log
[9] <1,4,writeLogEntry>
[10] <1,5,RECOVER-UPDATE-ACK>
[11] <1,6,RECOVER-SUCESS>
[12]: <1,7,init-validateNext>
G1
G2
Log API

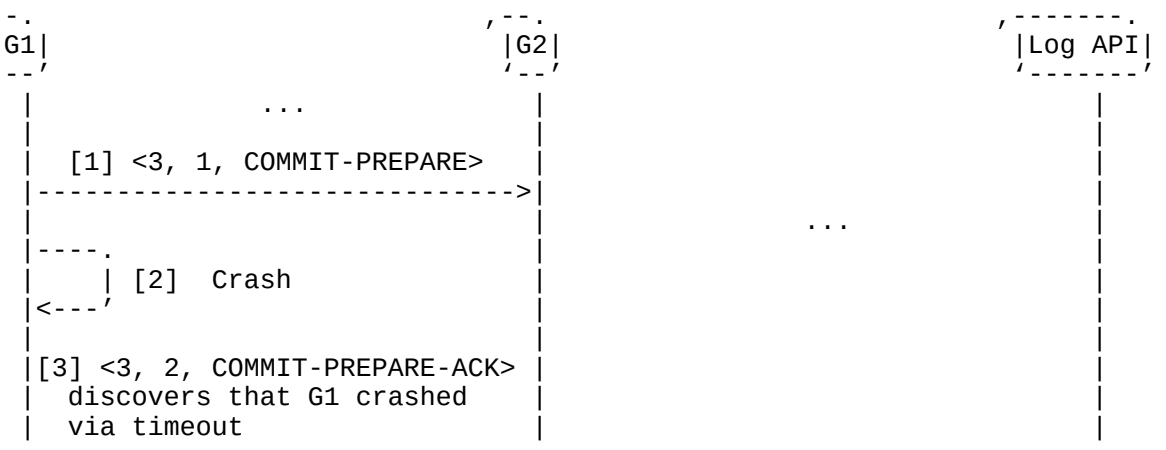
5.4.2. Crashing after issuing a command to the counterparty gateway

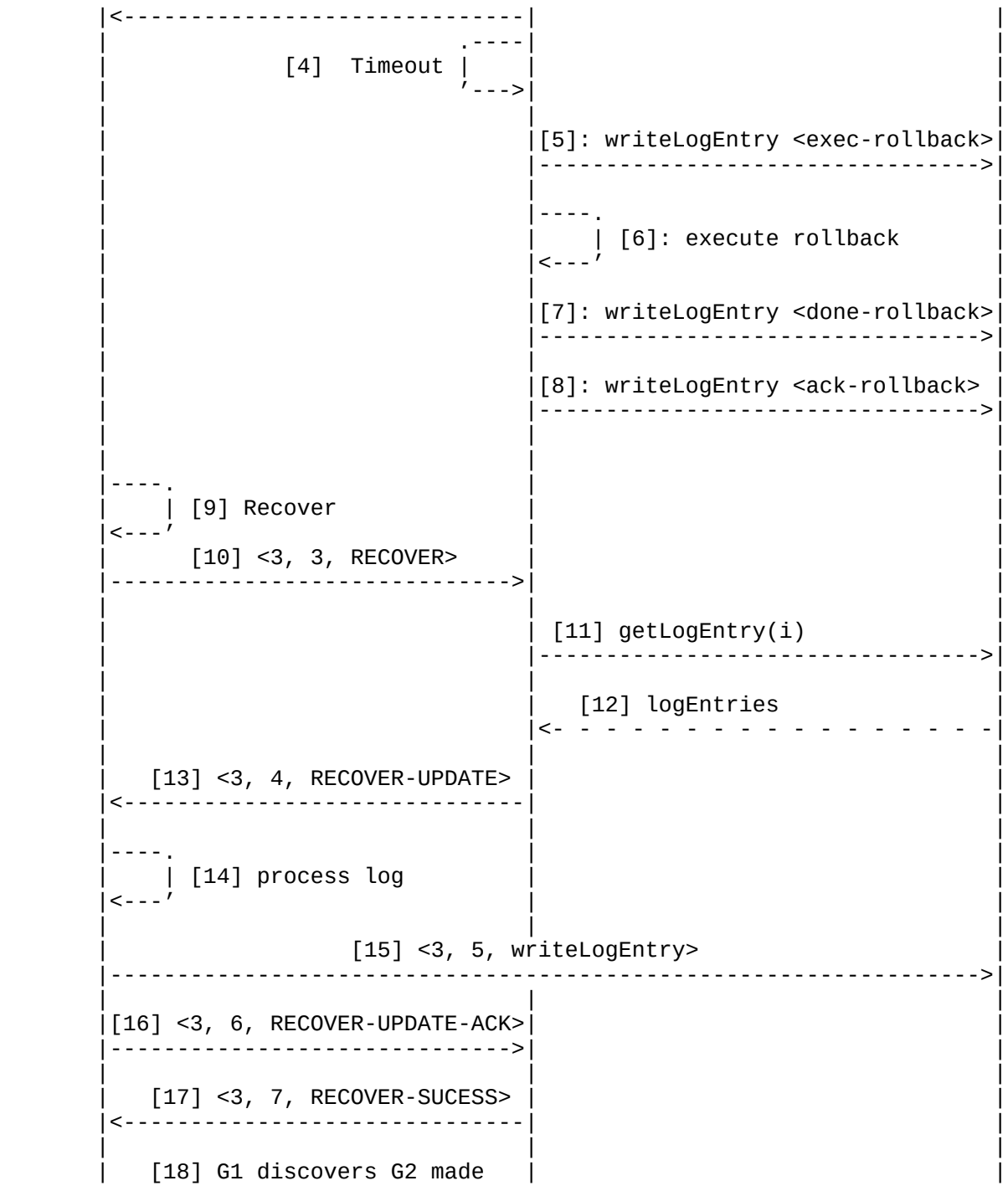
The second scenario requires further synchronization (figure below). At the retrieval of the latest log entry, G1 notices its log is outdated. It updates it upon necessary validation and then communicates its recovery to G2. The process then continues as defined.

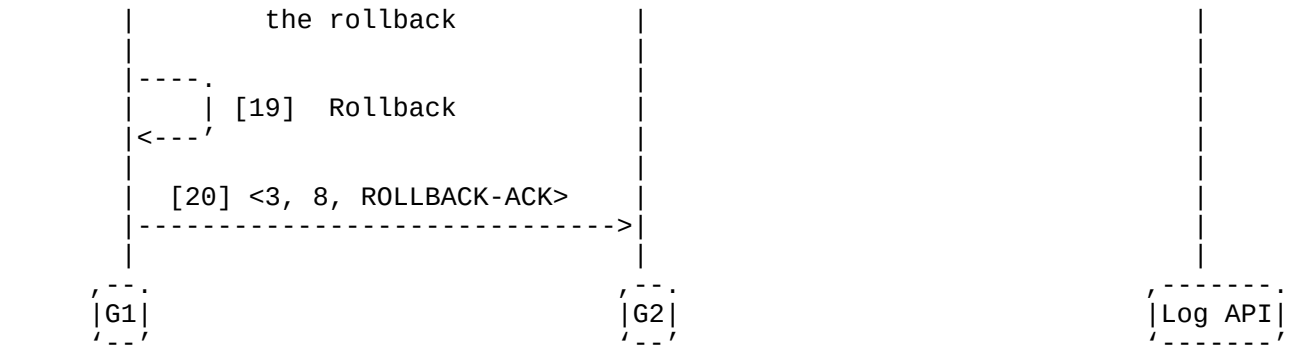


5.4.3. Rollback after counterparty gateway crash

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.




[4] Timeout
[5]: writeLogEntry <exec-rollback>
[6]: execute rollback
[7]: writeLogEntry <done-rollback>
[8]: writeLogEntry <ack-rollback>
[9] Recover
[10] <3, 3, RECOVER>
[11] getLogEntry(i)
[12] logEntries
[13] <3, 4, RECOVER-UPDATE>
[14] process log
[15] <3, 5, writeLogEntry>
[16] <3, 6, RECOVER-UPDATE-ACK>
[17] <3, 7, RECOVER-SUCESS>
[18] G1 discovers G2 made



6. Session Resumption

This section explains how the primary-backup mode works for crash recovery. First, there is a session resumption phase. After that, the gateways perform the protocol specified in Section 5.

6.1. Gateway Replacement

The gateway replacemenet protocol introduces an assumption. We assume every gateway has a valid X.509 certificate that was issued by its owner, which is the entity legally responsible for the gateway. Moreover, in the extensions field of the certificate, there is a list containing the hash of the authorized backup gateways. When the primary gateway crashes, a replacement is bootstrapped with the latest version of the local state, and it engages in a protocol with the counterparty gateway. This protocol aims to establish trust between gateways and the creation of a new TLS session:

1. Validate the backup gateway certificate by running a certification path algorithm, which includes validating all the intermediate certificates up to a trusted root (can be the VASPs CA).

2. The counterparty gateway verifies if the parent certificate of the crashed gateway and the backup gateway is the same (proving they belong to the same authority).

3. Verify if the backup gateway certificate hash belongs to the list specified in the crashed gateway certificate extensions.

The backup gateway, on its turn, defines gateways to replace it in case of a crash (X.509 certificate extensions).

7. 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. 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. 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 decentralized). 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. decentralized access control).

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.

8. Performance Considerations

After the session setup using asymmetric-cryptography, the authenticated messages in the TLS Record Protocol utilize symmetric-key operations (using the session key). Since symmetric-key operations are much faster than public-key operations, a persistent TLS connection delivers performance suitable for quickly exchange of log entries across gateways. Upon a crash, gateways might employ their best effort for resuming the crashed session.

9. Assumptions

For the protocol to work correctly, a few assumptions are taken: i) the crashed gateways eventually recover, at most for a fixed time (or are replaced); ii) The Log API is reliable - all requests are served up to a pre-defined time bound.

10. References

10.1. Normative References

[HTTP2] Thomson, M., Ed. and C. Benfield, Ed., "HTTP/2", RFC 9113, DOI 10.17487/RFC9113, June 2022, <https://www.rfc-editor.org/rfc/rfc9113>.

[TLS] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

10.2. Informative References

[AD76] Alsberg, P. and D. Day, "A principle for resilient sharing of distributed resources", 1976, <http://openid.net/specs/openid-connect-core-1_0.html>.

[I-D.draft-avrilionis-satp-setup-stage-01] Avrilionis, D. and T. Hardjono, "SATP Setup Stage", Work in Progress, Internet-Draft, draft-avrilionis-satp-setup-stage-01, 16 December 2024, <https://datatracker.ietf.org/doc/html/draft-avrilionis-satp-setup-stage-01>.

[I-D.draft-ietf-satp-core] Hargreaves, M., Hardjono, T., Belchior, R., Ramakrishna, V., and A. Chiriac, "Secure Asset Transfer Protocol (SATP) Core", Work in Progress, Internet-Draft, draft-ietf-satp-core-12, 2 November 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-satp-core-12>.

[OIDC] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.

Authors' Addresses

Rafael Belchior
INESC-ID, T迆cnico Lisboa, Blockdaemon

Email: rafael.belchior@tecnico.ulisboa.pt

Miguel Correia
INESC-ID, T迆cnico Lisboa
Email: miguel.p.correia@tecnico.ulisboa.pt

Andr迆 Augusto
INESC-ID, T迆cnico Lisboa
Email: andre.augusto@tecnico.ulisboa.pt

Thomas Hardjono
MIT
Email: hardjono@mit.edu