Workgroup: Network Working Group
Internet-Draft: draft-hedera-charge-00
Published: 20 May 2026
Intended Status: Informational
Expires: 21 November 2026
Authors: T. Rowbotham L. Walker
Hedera / Hashgraph

Hedera Charge Intent for HTTP Payment Authentication

Abstract

This document defines the "charge" intent for the "hedera" payment method within the Payment HTTP Authentication Scheme [I-D.httpauth-payment]. The client constructs and signs a native Hedera Token Service (HTS) transfer; the server verifies the payment via the Mirror Node REST API and presents the transaction ID as proof of payment.

Two credential types are supported: `type="hash"` (default), where the client broadcasts the transaction itself and presents the transaction ID for server verification, and `type="transaction"` (pull mode), where the client signs and serializes the transaction for the server to broadcast.

Table of Contents

1. Introduction

HTTP Payment Authentication [I-D.httpauth-payment] defines a challenge-response mechanism that gates access to resources behind payments. This document registers the "charge" intent for the "hedera" payment method.

Hedera is a distributed ledger with asynchronous Byzantine Fault Tolerant (aBFT) consensus, deterministic finality in 3-5 seconds, and fixed transaction fees [HEDERA-DOCS]. This specification supports payments in Hedera Token Service (HTS) tokens, including Circle USDC [CIRCLE-USDC-HEDERA], making it suitable for micropayment use cases where fast confirmation and predictable costs are important.

Challenge binding and replay protection are achieved through an Attribution memo embedded in the transaction's native memo field (see Section 6).

1.1. Push Mode (Default)

The default flow, called "push mode", uses `type="hash"` credentials. The client "pushes" the transaction to the Hedera network itself and presents the confirmed transaction ID:

```
Client                Server         Hedera Network
   |                     |                   |
   | (1) GET /resource   |                   |
   |-------------------> |                   |
   |                     |                   |
   | (2) 402 Payment     |                   |
   |     Required        |                   |
   |     (recipient,     |                   |
   |      amount, memo)  |                   |
   |<------------------- |                   |
   |                     |                   |
   | (3) Build tx with   |                   |
   |     Attribution     |                   |
   |     memo, sign      |                   |
   |                     |                   |
   | (4) Execute tx      |                   |
   |--------------------------------------> |
   | (5) Receipt         |                   |
   |<-------------------------------------- |
   |                     |                   |
   | (6) Authorization:  |                   |
   |     Payment         |                   |
   |     <credential>    |                   |
   |     (transaction ID)|                   |
   |-------------------> |                   |
   |                     | (7) Mirror Node   |
   |                     |     GET /api/v1/  |
   |                     |     transactions/ |
   |                     |     {txId}        |
   |                     |-----------------> |
   |                     | (8) Tx data       |
   |                     |<----------------- |
   |                     |                   |
   | (9) 200 OK +Receipt |                   |
   |<------------------- |                   |
   |                     |                   |
```

This flow is useful when the client has its own Hedera account and operator key. The server verifies the payment by querying the Mirror Node REST API [MIRROR-NODE].

1.2. Pull Mode

The pull mode flow uses `type="transaction"` credentials. The client signs the transaction and the server "pulls" it for broadcast to the Hedera network:

```
 Client                    Server         Hedera Network
    |                         |                     |
    |  (1) GET /resource      |                     |
    |-------------------->    |                     |
    |                         |                     |
    |  (2) 402 Payment        |                     |
    |      Required           |                     |
    |      (recipient,        |                     |
    |       amount)           |                     |
    |<--------------------    |                     |
    |                         |                     |
    |  (3) Build tx with      |                     |
    |      Attribution        |                     |
    |      memo, freeze,      |                     |
    |      sign               |                     |
    |                         |                     |
    |  (4) Authorization:     |                     |
    |      Payment            |                     |
    |      <credential>       |                     |
    |      (serialized tx)    |                     |
    |-------------------->    |                     |
    |                         |  (5) Verify memo,   |
    |                         |      execute tx     |
    |                         |------------------>  |
    |                         |  (6) Receipt        |
    |                         |<------------------  |
    |                         |                     |
    |  (7) 200 OK +Receipt    |                     |
    |<--------------------    |                     |
    |                         |                     |
```

In this model the server controls transaction broadcast, enabling server-side retry logic and future fee delegation (see Section 13.8).

1.3. Relationship to the Charge Intent

This document inherits the shared request semantics of the "charge" intent from [I-D.payment-intent-charge]. It defines only the Hedera-specific `methodDetails`, `payload`, and verification procedures for the "hedera" payment method.

2. Requirements Language

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

Transaction ID

A unique identifier for a Hedera transaction in the format `shard.realm.num@seconds.nanoseconds` (e.g., `0.0.12345@1681234567.123456789`). Composed of the payer account ID and the transaction's valid-start timestamp.

Account ID
: A Hedera account identifier in the format `shard.realm.num` (e.g., `0.0.12345`). The shard and realm are typically `0.0` on the public Hedera network.

Token ID
: A Hedera Token Service (HTS) token identifier in the format `shard.realm.num` (e.g., `0.0.456858` for Circle USDC on mainnet). Uniquely identifies a fungible or non-fungible token on the Hedera network.

Token Association
: A one-time operation that associates a Hedera account with an HTS token, enabling the account to hold and receive that token. Unlike Solana's Associated Token Accounts, token association is a single on-chain operation that does not create a separate account.

Base Units
: The smallest transferable unit of an HTS token, determined by the token's decimal precision. For example, USDC uses 6 decimals, so 1 USDC = 1,000,000 base units.

Mirror Node
: A read-only node that archives Hedera network data and exposes it via a REST API [MIRROR-NODE]. Used by servers to verify transaction details after consensus.

Attribution Memo
: A 32-byte challenge-bound memo embedded in the Hedera transaction's native memo field. Encodes the MPP tag, version, server identity, optional client identity, and a challenge-specific nonce. See Section 6 for the full byte layout.

Push Mode
: The default settlement flow where the client broadcasts the transaction itself and presents the confirmed transaction ID (`type="hash"`). The client "pushes" the transaction to the network directly.

Pull Mode
: The alternative settlement flow where the client signs and serializes the transaction and the server broadcasts it (`type="transaction"`). The server "pulls" the signed transaction from the credential.

4. Intent Identifier

The intent identifier for this specification is "charge". It **MUST** be lowercase.

5. Intent: "charge"

The "charge" intent represents a one-time payment gating access to a resource. The client builds and signs a Hedera `TransferTransaction` with an Attribution memo, then either broadcasts the transaction itself and sends the transaction ID (`type="hash"`) or sends the serialized signed transaction bytes to the server for broadcast (`type="transaction"`). The server verifies the transfer details and returns a receipt.

6. Attribution Memo

Every Hedera charge transaction **MUST** include an Attribution memo in the transaction's native memo field. The memo provides challenge binding (replay protection) and server identity verification.

6.1. Byte Layout

The Attribution memo is exactly 32 bytes, stored in the Hedera transaction memo as a `0x`-prefixed hex string (66 characters: `0x` + 64 hex digits = 66 bytes UTF-8). This fits well within Hedera's 100-byte memo limit.

```
Offset  Size  Field
------  ----  -----------------------------------
0..3    4     TAG = keccak256("mpp")[0..3]
4       1     VERSION = 0x01
5..14   10    SERVER_ID =
                keccak256(realm)[0..9]
15..24  10    CLIENT_ID =
                keccak256(clientId)[0..9]
                or zero bytes if anonymous
25..31  7     NONCE =
                keccak256(challengeId)[0..6]
```

TAG (bytes 0-3)   The first 4 bytes of `keccak256("mpp")`. Identifies this memo as an MPP attribution memo. Implementations **MUST** reject memos where these bytes do not match.

VERSION (byte 4)   Protocol version. **MUST** be `0x01` for this specification. Implementations **MUST** reject memos with an unrecognized version.

SERVER_ID (bytes 5-14)   The first 10 bytes of `keccak256(realm)`, where `realm` is the challenge's `realm` auth-param. Binds the memo to a specific server. Servers **MUST** verify this fingerprint matches their own realm.

CLIENT_ID (bytes 15-24)   The first 10 bytes of `keccak256(clientId)`, where `clientId` is an optional client identifier. If the client is anonymous, all 10 bytes **MUST** be zero.

NONCE (bytes 25-31)   The first 7 bytes of `keccak256(challengeId)`, where `challengeId` is the challenge `id` auth-param from the `WWW-Authenticate` header. Binds the memo to a specific challenge instance, preventing replay.

6.2. Memo Encoding

The 32-byte memo **MUST** be hex-encoded with a `0x` prefix and stored as the Hedera transaction memo via `setTransactionMemo()`. The resulting string is exactly 66 characters (`0x` + 64 hex digits) and 66 bytes UTF-8, which is within Hedera's 100-byte memo limit.

Example memo (hex):

```
0xef1ed71201a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7
  f8a9b0c1d2e3f4a5b6c7
```

6.3. Compatibility

This byte layout is identical to the attribution memo used by the Tempo payment method, ensuring compatibility across the MPP ecosystem. The only difference is the transport: Tempo embeds the memo in a smart contract call (`transferWithMemo`), while Hedera uses the native transaction memo field.

7. Encoding Conventions

All JSON [RFC8259] objects carried in auth-params or HTTP headers in this specification **MUST** be serialized using the JSON Canonicalization Scheme (JCS) [RFC8785] before encoding. JCS produces a deterministic byte sequence, which is required for any digest or signature operations defined by the base spec [I-D.httpauth-payment].

The resulting bytes **MUST** then be encoded using base64url [RFC4648] Section 5 without padding characters (`=`). Implementations **MUST NOT** append `=` padding when encoding, and **MUST** accept input with or without padding when decoding.

This encoding convention applies to: the `request` auth-param in `WWW-Authenticate`, the credential token in `Authorization`, and the receipt token in `Payment-Receipt`.

8. Request Schema

8.1. Shared Fields

The `request` auth-param of the `WWW-Authenticate: Payment` header contains a JCS-serialized, base64url-encoded JSON object (see Section 7). The following shared fields are included in that object:

amount
: **REQUIRED**. The payment amount in base units, encoded as a decimal string. For HTS tokens, the amount is in the token's smallest unit (e.g., for USDC with 6 decimals, "1000000" represents 1 USDC). The value **MUST** be a positive integer that fits in a 64-bit signed integer (max 9,223,372,036,854,775,807), consistent with Hedera's `int64` transfer amounts.

currency
: **REQUIRED**. The HTS token ID string identifying the payment asset (e.g., `"0.0.456858"` for Circle USDC on mainnet). The token ID uniquely identifies the token on the Hedera network and is used by the client to construct the `TransferTransaction`. **MUST** be a valid Hedera entity ID in the format `shard.realm.num`.

description **OPTIONAL**. A human-readable memo describing the resource or service being paid for. **MUST NOT** exceed 256 characters.

recipient **REQUIRED**. The Hedera account ID of the account receiving the payment (e.g., `"0.0.12345"`). **MUST** be a valid Hedera account ID in the format `shard.realm.num`.

externalId **OPTIONAL**. Merchant's reference (e.g., order ID, invoice number), per [I-D.payment-intent-charge]. May be used for reconciliation or idempotency. **MUST NOT** exceed 34 bytes (100-byte Hedera memo limit minus the 66-byte Attribution memo). When the Attribution memo is present, there is no remaining memo capacity for an on-chain external ID; the `externalId` is therefore carried only in the credential's challenge echo and is not written on-chain.

splits **OPTIONAL**. An array of at most 9 additional payment splits. Each entry is a JSON object with the following fields:

- `recipient` (**REQUIRED**): Hedera account ID of the split recipient (e.g., `"0.0.67890"`).
- `amount` (**REQUIRED**): Amount in the same base units and token as the primary `amount`.

When present, the client **MUST** include a token transfer entry for each split in addition to the primary transfer to `recipient`. All splits use the same token as the primary payment (the `currency` token ID).

Hedera's `TransferTransaction` natively supports atomic multi-party transfers (up to 10 token transfer entries per transaction), making splits straightforward: the client adds one debit from the payer and one credit per recipient in a single atomic transaction.

The top-level `amount` is the total the client pays. The sum of all split amounts **MUST NOT** exceed `amount`. The primary `recipient` receives `amount` minus the sum of all split amounts; this remainder **MUST** be greater than zero. Servers **MUST** reject challenges where splits consume the entire amount. Servers **MUST** verify each split transfer on-chain during credential verification. If the same recipient appears more than once in `splits`, each occurrence is a distinct payment leg and **MUST** be verified separately; servers **MUST NOT** implicitly aggregate such entries.

This mechanism is a Hedera-specific extension to the base `charge` intent. It can be used for platform fees, revenue sharing, or referral commissions.

Note: The `splits` field is at the top level of the request object (alongside `amount`, `currency`, `recipient`, etc.), not nested under `methodDetails`. The mppx framework's schema transform outputs `splits` at the top level.

8.2. Method Details

The following fields are nested under `methodDetails` in the request JSON:

chainId

OPTIONAL. The EIP-155 chain ID for the Hedera network: 295 for mainnet, 296 for testnet. Implementations **SHOULD** document their default network. The reference implementation defaults to testnet (296) for safety. Clients **MUST** reject challenges whose `chainId` does not match their configured network.

8.3. Client Configuration Fields

The following fields are used during request construction by the mppx framework's schema transform but are NOT present in the serialized wire-format challenge. They are consumed by `parseUnits()` to convert human-readable amounts to base units before the request is serialized.

decimals **OPTIONAL**. The number of decimal places for the token (0-18). Used by `parseUnits()` during request construction to convert a human-readable amount (e.g., "1.00") into base units (e.g., "1000000"). This field is consumed by the schema transform and does NOT appear in the serialized challenge sent over the wire. Clients that construct requests manually **MUST** provide `amount` in base units directly and do not need this field.

8.3.1. HTS Token Example

```
{
  "amount": "1000000",
  "currency": "0.0.456858",
  "recipient": "0.0.12345",
  "description": "Weather API access",
  "methodDetails": {
    "chainId": 295
  }
}
```

This requests a transfer of 1 USDC (1,000,000 base units) on Hedera mainnet (chain ID 295).

8.3.2. Testnet Example

```
{
  "amount": "500000",
  "currency": "0.0.5449",
  "recipient": "0.0.67890",
  "description": "Premium API call",
  "methodDetails": {
    "chainId": 296
  }
}
```

This requests a transfer of 0.50 USDC on Hedera testnet (chain ID 296). Note that `decimals` is not present in the wire format; it is only used during request construction by the mppx schema transform.

8.3.3. Payment Splits Example

```
{
  "amount": "1050000",
  "currency": "0.0.456858",
  "recipient": "0.0.12345",
  "description": "Marketplace purchase",
  "splits": [
    {
      "recipient": "0.0.67890",
      "amount": "50000"
    }
  ],
  "methodDetails": {
    "chainId": 295
  }
}
```

This requests a total payment of 1.05 USDC. The platform receives 0.05 USDC and the primary recipient (seller) receives 1.00 USDC.

9. Credential Schema

The `Authorization` header carries a single base64url- encoded JSON token (no auth-params). The decoded object contains the following top-level fields:

challenge **REQUIRED**. An echo of the challenge auth-params from the `WWW-Authenticate` header: `id`, `realm`, `method`, `intent`, `request`, and (if present) `expires`. This binds the credential to the exact challenge that was issued.

source **OPTIONAL**. A payer identifier string, as defined by [I-D.httpauth-payment]. Hedera implementations **MAY** use a DID in the format `did:pkh:hedera:{network}:{accountId}`.

payload **REQUIRED**. A JSON object containing the Hedera-specific credential fields. The `type` field determines which additional fields are present. Two payload types are defined: `"hash"` (default) and `"transaction"` (pull mode).

9.1. Hash Payload -- Push Mode

In push mode (`type="hash"`), the client has already broadcast the transaction to the Hedera network. The `transactionId` field contains the Hedera transaction ID for the server to verify via the Mirror Node.

Field	Type	Req	Description
`type`	string	Y	`"hash"`

Field	Type	Req	Description
`transactionId`	string	Y	Hedera transaction ID

Table 1

The `transactionId` **MUST** be in the standard Hedera format `shard.realm.num@seconds.nanoseconds` (e.g., `"0.0.12345@1681234567.123456789"`).

Example (decoded):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "hedera",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "hash",
    "transactionId":
      "0.0.12345@1681234567.123456789"
  }
}
```

9.2. Transaction Payload -- Pull Mode

In pull mode (`type="transaction"`), the client sends the signed transaction bytes to the server for broadcast. The `transaction` field contains the base64-encoded serialized signed transaction.

Field	Type	Req	Description
`type`	string	Y	`"transaction"`
`transaction`	string	Y	Base64-encoded signed tx bytes

Table 2

The transaction **MUST** be a valid Hedera transaction that has been frozen and signed by the payer. The server deserializes the transaction via `Transaction.fromBytes()`, verifies the Attribution memo, and executes it.

Example (decoded):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "hedera",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "transaction",
    "transaction": "CgMA...base64-encoded..."
  }
}
```


10. Verification Procedure

Upon receiving a request with a credential, the server **MUST**:

1. Decode the base64url credential and parse the JSON.
2. Verify that `payload.type` is present and is either `"hash"` or `"transaction"`.
3. Look up the stored challenge using `credential.challenge.id`. If no matching challenge is found, reject the request.
4. Verify that all fields in `credential.challenge` exactly match the stored challenge auth-params.
5. Proceed with type-specific verification:
   - For `type="hash"`: see Section 10.1.
   - For `type="transaction"`: see Section 10.2.

10.1. Push Mode Verification

For credentials with `type="hash"`:

1. Verify that `payload.transactionId` is present and is a valid Hedera transaction ID string.
2. Verify the transaction ID has not been previously consumed (see Section 10.4).
3. Fetch the transaction from the Mirror Node REST API at `/api/v1/transactions/{txId}`, where `{txId}` is the transaction ID with `@` replaced by `-` and `.` in the timestamp replaced by `-` (Mirror Node URL format). The server **MUST** poll with retry to account for the 3-5 second lag between consensus and Mirror Node indexing (see Section 10.5).
4. Verify the transaction was successful: the `result` field **MUST** be `"SUCCESS"`.
5. Verify the Attribution memo: decode the `memo_base64` field from the Mirror Node response (base64 to UTF-8 to hex string), then verify:
   - The memo is a valid MPP attribution memo (TAG and VERSION match).
   - The SERVER_ID fingerprint matches the server's realm.
   - The NONCE matches `keccak256(challengeId)[0..6]`.

6. Verify the token transfers match the challenge request (see Section 10.3).
7. Mark the transaction ID as consumed to prevent replay.
8. Return the resource with a `Payment-Receipt` header.

10.2. Pull Mode Verification

For credentials with `type="transaction"`:

1. Decode the base64 `payload.transaction` value.
2. Deserialize the transaction using `Transaction.fromBytes()`.
3. Extract the transaction memo and verify it is a valid MPP attribution memo:
   - The memo string starts with `0x` and is 66 characters.
   - TAG and VERSION match.
   - SERVER_ID fingerprint matches the server's realm.
   - NONCE matches `keccak256(challengeId)[0..6]`.
4. Verify the serialized transaction bytes have not been previously submitted (see Section 10.4).
5. Execute the transaction on the Hedera network using the server's operator credentials.
6. Verify the transaction receipt status is `SUCCESS`.
7. Fetch the transaction from the Mirror Node and verify the token transfers match the challenge request (see Section 10.3).
8. Mark the transaction ID as consumed to prevent replay.
9. Return the resource with a `Payment-Receipt` header.

10.3. Transfer Verification

For all credential types, the server **MUST** verify the token transfers from the Mirror Node response:

1. Compute the primary payment amount as the top-level `amount` minus the sum of all `splits`, if any.
2. Locate a token transfer entry in the Mirror Node response's `token_transfers` array where:
   - `token_id` matches the `currency` from the challenge request.
   - `account` matches the top-level `recipient`.
   - `amount` is greater than or equal to the computed primary payment amount.
3. For each split in `splits`, if any, locate an additional token transfer entry where:
   - `token_id` matches the `currency`.
   - `account` matches the split `recipient`.
   - `amount` is greater than or equal to the split `amount`.

> Each required payment leg **MUST** be matched to a distinct token transfer entry. A single entry **MUST NOT** satisfy more than one required payment leg, even if multiple legs share the same recipient.

If any required token transfer entry is missing, the server **MUST** reject the credential.

10.4. Replay Protection

Servers **MUST** maintain a set of consumed transaction identifiers. Before accepting a credential, the server **MUST** check whether the identifier has already been consumed. After successful verification, the server **MUST** atomically mark the identifier as consumed.

For `type="hash"` credentials, the transaction ID is provided directly by the client. For `type="transaction"` credentials, the transaction ID is derived after the server executes the transaction.

The Attribution memo's NONCE field provides an additional layer of replay protection: even if a transaction ID were somehow reusable, the challenge-bound nonce ensures the memo can only satisfy the specific challenge it was created for.

A transaction ID that has been consumed **MUST NOT** be accepted again, even if presented with a different challenge ID.

10.5. Mirror Node Lag

Hedera achieves consensus in approximately 3-5 seconds, but the Mirror Node REST API may take an additional 3-5 seconds to index the transaction. Servers **MUST** implement retry logic when fetching transactions from the Mirror Node:

- Servers **SHOULD** retry up to 10 times with a 2-second delay between attempts.
- A 404 response from the Mirror Node during the retry window is expected and **MUST NOT** be treated as a permanent failure.
- After exhausting retries, the server **MUST** reject the credential with a `verification-failed` error.

11. Settlement Procedure

Two settlement flows are supported, corresponding to the two credential types.

11.1. Push Mode Settlement (type="hash")

For `type="hash"` credentials, the client broadcasts the transaction and presents the transaction ID:

```
 Client                 Server         Mirror Node
    |                      |                 |
    | (1) Build tx with    |                 |
    |     Attribution      |                 |
    |     memo, sign,      |                 |
    |     execute          |                 |
    |                      |                 |
    | (2) Authorization:   |                 |
    |     Payment          |                 |
    |     <credential>     |                 |
    |     (transaction ID)|                 |
    |--------------------> |                 |
    |                      |                 |
    |                      | (3) GET         |
    |                      |  /api/v1/       |
    |                      |  transactions/  |
    |                      |  {txId}         |
    |                      |  (with retry)   |
    |                      |---------------> |
    |                      | (4) Tx data     |
    |                      |<--------------- |
    |                      |                 |
    |                      | (5) Verify:     |
    |                      |  - memo         |
    |                      |  - transfers    |
    |                      |  - result       |
    |                      |                 |
    | (6) 200 OK +Receipt  |                 |
    |<-------------------- |                 |
    |                      |                 |
```

1. Client builds a `TransferTransaction` with the Attribution memo, signs it, and executes it on the Hedera network.
2. Client presents the transaction ID as the credential.
3. Server fetches the transaction from the Mirror Node REST API, retrying to account for indexing lag.
4. Server verifies the Attribution memo (challenge binding, server identity) and token transfers (amount, recipient, splits).
5. Server records the transaction ID as consumed and returns the resource with a `Payment-Receipt` header.

11.2. Pull Mode Settlement (type="transaction")

For `type="transaction"` credentials, the client signs the transaction and sends it to the server:

```
  Client                  Server         Hedera Network
     |                      |                    |
     |  (1) Authorization:  |                    |
     |      Payment         |                    |
     |      <credential>    |                    |
     |      (signed tx      |                    |
     |       bytes)         |                    |
     |--------------------> |                    |
     |                      |                    |
     |                      |  (2) Deserialize,  |
     |                      |      verify memo   |
     |                      |                    |
     |                      |  (3) Execute tx    |
     |                      |------------------> |
     |                      |  (4) Receipt       |
     |                      |<------------------ |
     |                      |                    |
     |                      |  (5) Mirror Node   |
     |                      |      verify        |
     |                      |      transfers     |
     |                      |                    |
     |  (6) 200 OK +Receipt |                    |
     |<-------------------- |                    |
     |                      |                    |
```

1. Client submits credential containing signed transaction bytes.
2. Server deserializes the transaction, verifies the Attribution memo (challenge binding, server identity).
3. Server executes the transaction on the Hedera network.
4. Server verifies the receipt status is `SUCCESS`.
5. Server fetches the transaction from the Mirror Node and verifies token transfers match the challenge request.
6. Server records the transaction ID as consumed and returns the resource with a `Payment-Receipt` header.

11.3. Client Transaction Construction

The client **MUST** construct a `TransferTransaction` with:

1. A debit of the full `amount` from the client's account for the specified `currency` token.
2. A credit of the primary payment amount (total `amount` minus sum of splits) to the `recipient` account for the `currency` token.
3. For each split, a credit of the split `amount` to the split `recipient` for the `currency` token.
4. The Attribution memo set via `setTransactionMemo()` (see Section 6).

All debit and credit entries **MUST** sum to zero within the `TransferTransaction`, as required by Hedera's transfer semantics.

The recipient account(s) **MUST** have previously associated with the `currency` token. Unlike Solana's Associated Token Accounts, Hedera token association is a one-time operation and does not require rent or account creation by the payer. If the recipient has not associated with the token, the transaction will fail with `TOKEN_NOT_ASSOCIATED_TO_ACCOUNT`.

11.4. Finality

Hedera provides asynchronous Byzantine Fault Tolerant (aBFT) consensus with deterministic finality in approximately 3-5 seconds. Once a transaction reaches consensus, it cannot be rolled back or reversed.

This is in contrast to probabilistic finality models (e.g., proof-of-work chains) where transactions can theoretically be reversed. Hedera's deterministic finality means that once the Mirror Node reports a transaction as `SUCCESS`, the payment is irreversible.

Servers **MAY** accept the credential immediately upon Mirror Node confirmation without waiting for additional confirmations.

11.5. Receipt Generation

Upon successful verification, the server **MUST** include a `Payment-Receipt` header in the 200 response.

The receipt payload for Hedera charge:

Field	Type	Description
`method`	string	`"hedera"`
`reference`	string	The transaction ID
`status`	string	`"success"`
`timestamp`	string	[RFC3339] time

Table 3

Example (decoded):


```
{
  "method": "hedera",
  "reference":
    "0.0.12345@1681234567.123456789",
  "status": "success",
  "timestamp": "2026-03-10T21:00:00Z"
}
```

12. Error Responses

When rejecting a credential, the server **MUST** return HTTP 402 (Payment Required) with a fresh `WWW-Authenticate: Payment` challenge per [I-D.httpauth-payment]. The server **SHOULD** include a response body conforming to RFC 9457 [RFC9457] Problem Details, with `Content-Type: application/problem+json`. Servers **MUST** use the standard problem types defined in [I-D.httpauth-payment]: `malformed-credential`, `invalid-challenge`, and `verification-failed`. The `detail` field **SHOULD** contain a human-readable description of the specific failure (e.g., "Transaction not found on Mirror Node", "Attribution memo mismatch", "Transaction ID already consumed").

All error responses **MUST** include a fresh challenge in `WWW-Authenticate`.

Example error response body:

```
{
  "type": "https://paymentauth.org/problems/verification-failed",
  "title": "Attribution Memo Mismatch",
  "status": 402,
  "detail": "Memo challenge nonce does not match"
}
```

13. Security Considerations

13.1. Transport Security

All communication **MUST** use TLS 1.2 or higher. Hedera credentials **MUST** only be transmitted over HTTPS connections.

13.2. Replay Protection Considerations

Servers **MUST** track consumed transaction IDs and reject any transaction ID that has already been accepted. The check-and-consume operation **MUST** be atomic to prevent race conditions where concurrent requests present the same transaction ID.

The Attribution memo's NONCE field (derived from the challenge ID) provides cryptographic challenge binding: even if an attacker obtains a valid transaction ID, they cannot construct a valid credential without the matching challenge. However, the consumed-set check remains essential because a single transaction could theoretically match multiple challenges with identical terms.

13.3. Attribution Memo Security

The Attribution memo provides challenge binding but is not a cryptographic signature over the challenge parameters. It binds the transaction to a specific challenge ID and server realm via keccak256 fingerprints, which provides collision resistance (~2^56 for the 7-byte nonce, ~2^80 for the 10-byte server and client fingerprints).

An attacker would need to find a challenge ID whose keccak256 prefix collides with the target nonce to forge a memo. At 7 bytes (56 bits), this requires approximately 2^56 hash operations, which is computationally infeasible for real-time attacks.

13.4. Client-Side Verification

Clients **MUST** verify the challenge before signing:

1. `amount` is reasonable for the service.
2. `currency` matches the expected token ID.
3. `recipient` is the expected party.
4. `splits`, if present, contain expected recipients and amounts -- malicious servers could add splits to redirect funds.
5. The `chainId` matches the client's configured network.

Malicious servers could request excessive amounts, direct payments to unexpected recipients, or add hidden splits.

13.5. Mirror Node Trust

The server relies on the Hedera Mirror Node REST API to provide accurate transaction data for on-chain verification. A compromised Mirror Node could return fabricated transaction data, causing the server to accept payments that were never made. Servers **SHOULD** use trusted Mirror Node providers or run their own Mirror Node instance.

13.6. Front-running (Push Mode)

In push mode, the client broadcasts the transaction before presenting the credential, making it visible on the Hedera network. A party monitoring the network could attempt to present the same transaction ID to the server. The challenge binding (the credential echoes the challenge `id`, which is HMAC-verified by the server) and the Attribution memo (which binds the transaction to a specific challenge nonce) mitigate this: only the party that received the challenge can construct a valid credential with a matching memo.

Unlike the Solana method's push mode, Hedera's Attribution memo provides stronger on-chain challenge binding. The memo's NONCE field cryptographically ties the transaction to a specific challenge instance, preventing a single transaction from satisfying multiple challenges even if they have identical terms.

13.7. Transaction Payload Security (Pull Mode)

In pull mode, the server receives raw transaction bytes from the client. A malicious client could craft a transaction that performs unexpected operations.

Servers **MUST** verify that the deserialized transaction: - Contains only the expected token transfer entries. - Has a valid Attribution memo bound to the current challenge. - Does not include unexpected operations beyond the token transfer.

13.8. Fee Delegation (Future)

Hedera natively supports fee delegation via the `feePayerAccountId` field on transactions. This allows a third party (e.g., the server) to pay the transaction fee on behalf of the client.

This specification does not define fee delegation semantics in this version. A future revision **MAY** add `feePayer` and `feePayerAccountId` fields to `methodDetails`, following a pattern similar to the Solana method's fee sponsorship mechanism. When implemented, fee delegation would pair naturally with pull mode (`type="transaction"`), where the server can add its fee payer signature before broadcasting.

14. IANA Considerations

14.1. Payment Method Registration

This document requests registration of the following entry in the "HTTP Payment Methods" registry established by [I-D.httpauth-payment]:

Method Identifier	Description	Reference
`hedera`	Hedera Token Service (HTS) token transfer	This document

Table 4

14.2. Payment Intent Registration

This document requests registration of the following entry in the "HTTP Payment Intents" registry established by [I-D.httpauth-payment]:

Intent	Applicable Methods	Description	Reference
`charge`	`hedera`	One-time HTS token transfer	This document

Table 5

15. References

15.1. Normative References

[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>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/info/rfc8785>.

[RFC9457] Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023, <https://www.rfc-editor.org/info/rfc9457>.

[I-D.payment-intent-charge] Moxey, J., Ryan, B., and T. Meagher, "'charge' Intent for HTTP Payment Authentication", 2026, <https://datatracker.ietf.org/doc/draft-payment-intent-charge/>.

[I-D.httpauth-payment] Moxey, J., "The 'Payment' HTTP Authentication Scheme", January 2026, <https://datatracker.ietf.org/doc/draft-ietf-httpauth-payment/>.

15.2. Informative References

[HEDERA-DOCS] Hedera, "Hedera Documentation", 2026, <https://docs.hedera.com>.

[HIP-218] Hedera, "HIP-218: Smart Contract Verification", 2022, <https://hips.hedera.com/hip/hip-218>.

[HIP-376] Hedera, "HIP-376: Approve/Allowance API for Tokens", 2022, <https://hips.hedera.com/hip/hip-376>.

[MIRROR-NODE] Hedera, "Hedera Mirror Node REST API", 2026, <https://docs.hedera.com/hedera/sdks-and-apis/rest-api>.

[CIRCLE-USDC-HEDERA] Circle, "Circle USDC on Hedera", 2026, <https://www.circle.com/multi-chain-usdc/hedera>.

Appendix A. Examples

The following examples illustrate the complete HTTP exchange for each flow. Base64url values are shown with their decoded JSON below.

A.1. USDC Charge (Push Mode)

A 1 USDC charge for weather API access on mainnet.

1. Challenge (402 response):

```
HTTP/1.1 402 Payment Required
WWW-Authenticate: Payment
  id="kM9xPqWvT2nJrHsY4aDfEb",
  realm="api.example.com",
  method="hedera",
  intent="charge",
  request="<base64url request>",
  expires="2026-03-15T12:05:00Z"
Cache-Control: no-store
```

Decoded `request`:

```
{
  "amount": "1000000",
  "currency": "0.0.456858",
  "recipient": "0.0.12345",
  "description": "Weather API access",
  "methodDetails": {
    "chainId": 295
  }
}
```

2. Credential (retry with transaction ID):

```
GET /weather HTTP/1.1
Host: api.example.com
Authorization: Payment <base64url credential>
```

Decoded credential:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "hedera",
    "intent": "charge",
    "request": "<base64url request>",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "hash",
    "transactionId":
      "0.0.12345@1681234567.123456789"
  }
}
```

3. Response (with receipt):

```
HTTP/1.1 200 OK
Payment-Receipt: <base64url receipt>
Content-Type: application/json

{"temperature": 72, "condition": "sunny"}
```

Decoded receipt:

```
{
  "method": "hedera",
  "reference":
    "0.0.12345@1681234567.123456789",
  "status": "success",
  "timestamp": "2026-03-15T12:04:58Z"
}
```

A.2. Pull Mode (type="transaction")

The client signs and serializes the transaction; the server deserializes, verifies, and executes it.

Decoded credential:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "hedera",
    "intent": "charge",
    "request": "<base64url request>",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "transaction",
    "transaction": "CgMA...base64-encoded..."
  }
}
```

A.3. Payment Splits

A marketplace charge of 1.05 USDC where 0.05 USDC goes to the platform as a fee.

Decoded `request`:

```
{
  "amount": "1050000",
  "currency": "0.0.456858",
  "recipient": "0.0.12345",
  "description": "Marketplace purchase",
  "splits": [
    {
      "recipient": "0.0.67890",
      "amount": "50000"
    }
  ],
  "methodDetails": {
    "chainId": 295
  }
}
```

The client builds a `TransferTransaction` with three token transfer entries: - Debit 1,050,000 from the payer (`0.0.PAYER`) - Credit 1,000,000 to the seller (`0.0.12345`) - Credit 50,000 to the platform (`0.0.67890`)

All three entries are atomic within a single transaction, leveraging Hedera's native multi-party transfer support.

Appendix B. Acknowledgements

The author thanks the Tempo team for the MPP attribution memo design and the mppx ecosystem architecture that this specification builds upon.

Authors' Addresses

Tom Rowbotham
Email: tom@xeno.money

Lindsay Walker
Hedera / Hashgraph
Email: lindsay.w@swirldslabs.com