Workgroup:	Network Working Group
Internet-Draft:	draft-xrpl-charge-00
Published:	9 September 2026
Intended Status:	Informational
Expires:	13 March 2027
Author:	M. Dienger *Ripple*

XRP Ledger Charge Intent for HTTP Payment Authentication

Abstract

This document defines the "charge" intent for the "xrpl" payment method within the Payment HTTP Authentication Scheme. A charge settles as a single XRP Ledger Payment transaction carrying XRP, an issued currency, or a Multi-Purpose Token.

Two credential types are supported. In `type="transaction"` (the default) the client signs a Payment and hands the serialized blob to the server, which submits it and observes the result. In `type="hash"` the client submits the transaction itself and presents the transaction hash as proof.

The default is the signed blob rather than the hash, which is the inverse of several other methods: a payer who broadcasts first has already parted with funds before learning whether the server will honour them.

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 13 March 2027.

Copyright Notice

Table of Contents

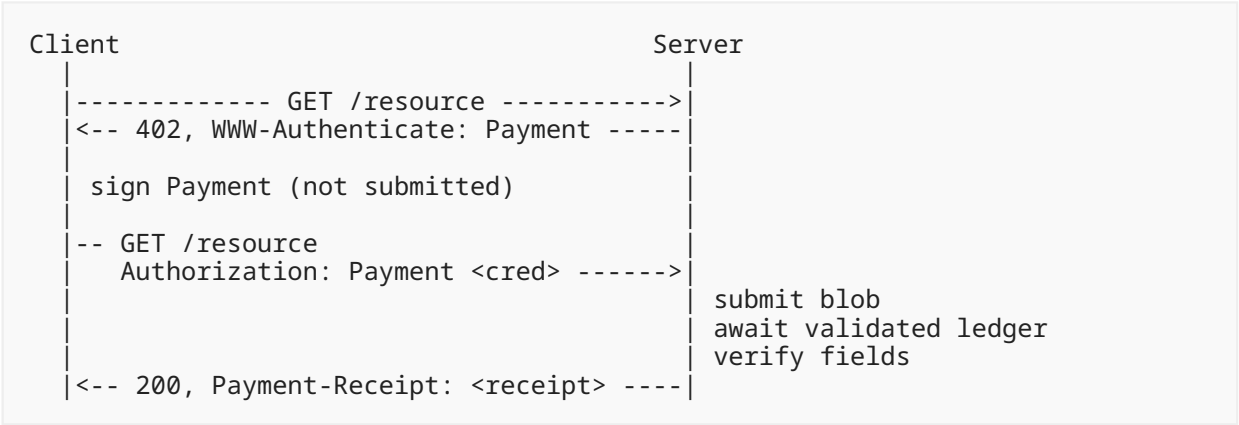
1. Introduction

The XRP Ledger settles payments in three to five seconds under a consensus protocol with deterministic finality: a transaction in a validated ledger cannot be reordered or reversed [XRPL-FINALITY]. There is no probabilistic confirmation depth to reason about, which makes it a natural fit for the synchronous request-response shape of [I-D.httpauth-payment] -- a server can decide within one HTTP round trip whether payment has settled.

This document specifies how the "charge" intent [I-D.payment-intent-charge] maps onto an XRP Ledger Payment transaction [XRPL-PAYMENT].

1.1. Pull Mode (Default)

The client signs a Payment transaction but does not submit it. The serialized blob travels in the credential; the server submits it and watches for validation.



Pull mode is the default for a reason worth stating plainly. If the client broadcasts first, it has irrevocably spent funds before knowing whether the server will deliver: a server that crashes, rejects the credential, or simply never replies leaves the payer with an on-chain debit and nothing to show for it. In pull mode the payer's exposure begins only once the server has taken custody of the blob and committed to submitting it.

Pull mode also gets replay protection for free from the ledger. A Payment consumes the sender's account sequence number, so a resubmitted blob fails with `tefPAST_SEQ`. This is a property of the ledger, not of the server's bookkeeping, and it holds even if the server's replay store is lost.

1.2. Push Mode

The client submits the transaction itself and sends only the 64-hex transaction hash. The server looks the transaction up and verifies it.

Push mode suits clients that already have a signing and broadcast pipeline and do not wish to hand a signed blob to a counterparty. It carries the exposure described above, and it has no sequence-number protection: a hash of an already-validated transaction can be presented repeatedly, and to any server. Section 9.4 therefore makes the challenge binding mandatory in push mode.

1.3. Relationship to the Charge Intent

This document constrains [I-D.payment-intent-charge]; it does not redefine it. Fields defined there keep their meaning. Everything introduced here lives under `methodDetails`, except where this document narrows the permitted values of a shared field.

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

Drops: The indivisible unit of XRP. One XRP is 10^6 drops. All XRP amounts on the wire are integer drop counts expressed as decimal strings. Implementations **MUST NOT** represent drop counts as IEEE-754 doubles: the total supply is 10^17 drops, which exceeds the 2^53 range of exactly representable integers.

Classic address: An XRPL account identifier in base58, beginning with `r`.

Issued currency: A token on the XRP Ledger denominated by a currency code and the classic address of its issuer [XRPL-CURRENCY], held through a trustline [XRPL-TRUSTLINES].

Multi-Purpose Token (MPT): A token type defined by [XLS-33], identified by an `mpt_issuance_id` naming an `MPTokenIssuance` entry, and held through an MPToken object rather than a trustline.

InvoiceID: An optional 256-bit field on a Payment transaction, covered by the transaction signature and indexed by the ledger. This document uses it to bind a payment to one challenge (Section 9.4).

Validated ledger: A ledger version agreed by consensus. A transaction reported inside a validated ledger is final.

4. Method Identifier

The method identifier is the string `xrpl`.

Servers advertising this method in a challenge **MUST** use exactly this value. The identifier names the ledger, not any particular asset on it: XRP, issued currencies and MPTs are all carried by this one method and distinguished by the `currency` field.

Which of them a given network carries may be gated by amendments [XRPL-AMENDMENTS]; MPT support in particular arrives with [XLS-33]. A server **SHOULD** determine the assets it can accept from the network it is connected to rather than from this document.

5. Intent: "charge"

The `charge` intent settles a single Payment transaction. It is the only intent defined by this document; the `session` intent for the same method is specified separately in draft-xrpl-session.

6. Encoding Conventions

Amounts: XRP amounts are integer drop counts as decimal strings, for example `"1000000"` for 1 XRP. Issued-currency and MPT amounts are decimal strings in the token's own units. Implementations **MUST** compare amounts as exact decimal values, never as floating-point numbers.

Credentials and challenges are JSON [RFC8259].

Hex fields: `InvoiceID` and transaction hashes are 64 hexadecimal characters. The ledger reports them uppercase, and implementations **MUST** emit uppercase, but hex is case-insensitive as a value and an implementation **MUST** accept either casing on input.

Case **MUST NOT** be load-bearing anywhere downstream. Comparisons and any key derived from such an identifier **MUST** be canonicalised first. A store keyed on the raw string while the signature check and the ledger lookup both accept either casing is a replay hole: the same credential resubmitted with the casing changed verifies, resolves, and finds a different key holding nothing.

Addresses: Classic addresses only. X-addresses (which pack a destination tag into the address) **MUST NOT** appear in the `recipient` field; a destination tag is carried in `methodDetails.destinationTag` so that it remains independently verifiable.

7. Request Schema

7.1. Shared Fields

The `charge` intent's shared fields apply, with these constraints:

Field	Type	Required	Constraint
`amount`	string	yes	positive decimal, no sign, no exponent
`currency`	string	yes	see below
`recipient`	string	yes	classic address
`description`	string	no	display only, see Section 13.8
`externalId`	string	no	merchant reconciliation handle

Table 1

The `currency` field is `"XRP"` for the native asset. For an issued currency it is a JSON object with `currency` and `issuer`. For an MPT it is a JSON object with `mpt_issuance_id`. Servers **MUST** reject a credential whose settled asset does not match the challenged `currency` exactly, including the issuer: an amount of the right size in the wrong issuer's token is not payment.

7.2. Method Details

Field	Type	Required	Meaning
`network`	string	no	`mainnet`, `testnet` or `devnet`
`reference`	string	no	server-generated correlation ID
`invoiceId`	64-hex	no	explicit challenge binding, Section 9.4
`destinationTag`	uint32	no	enforced on verification when present
`sourceTag`	uint32	no	enforced on verification when present
`memos`	array	no	UTF-8 memo entries embedded in the Payment

Table 2

Servers **SHOULD** set `network` explicitly. This document defines no default: an implementation that assumes one risks disagreeing with a counterparty that assumed the other, and the mistake is only visible after a payment has settled on the wrong ledger.

Clients **SHOULD** refuse to sign when the challenge's network does not match the ledger they are connected to. A testnet payment presented against a mainnet challenge is worthless, and the mismatch is detectable before signing rather than after.

7.3. Recipient Prerequisites

A charge in XRP needs nothing of the recipient beyond a funded account. The other two asset types do, and the requirement falls on the recipient before any client is ever challenged.

For an issued currency, a recipient that is not the issuer **MUST** already hold a trustline [XRPL-TRUSTLINES] to the issuer for that currency code. For an MPT, a recipient that is not the issuer **MUST** already have opted in, which creates its `MPToken` object; a balance is not required. Where the issuance was created requiring authorization, the issuer **MUST** additionally have authorized that holder.

The issuer is exempt in both cases: a payment returning a token to the account that issued it redeems the token rather than transferring it, and redemption requires nothing on the issuer's side.

Neither prerequisite is something a payer can supply. A server advertising a charge it cannot receive produces a payment that fails on submission, after the client has signed and, in push mode, after it has already parted with the funds. A server **SHOULD** therefore confirm the prerequisite once at startup rather than per challenge: the condition is a property of the recipient account, not of any particular payment.

How it is established -- `TrustSet` for a trustline, `MPTokenAuthorize` for an MPT holding -- is ordinary ledger operation and outside this document. What matters here is that a charge advertised without it is a charge that cannot settle.

8. Credential Schema

The credential payload is a discriminated union on `type`.

8.1. Transaction Payload -- Pull Mode

```
{
  "type": "transaction",
  "blob": "120000228000000024012E..."
}
```

`blob` is the hex-encoded, signed but unsubmitted Payment transaction.

8.2. Hash Payload -- Push Mode

```
{
  "type": "hash",
  "hash": "BE3DE95F52CC58E22F78CD5D2F7DE9084F596D88B390D479259D7DEC62EBDB49"
}
```

9. Verification Procedure

A server **MUST** perform every check in this section before returning a receipt. The order given is normative where one step guards another; in particular, cheap local checks precede any network call so that an unauthenticated caller cannot use verification as an amplifier.

9.1. Challenge Freshness

The server **MUST** reject a credential whose challenge carries no `expires`, one whose `expires` is not a valid [RFC3339] timestamp, and one whose `expires` has passed. `expires` is covered by the challenge's integrity protection and is the only authenticated statement of time the challenge carries; treating its absence as "unbounded" would let a credential be presented indefinitely.

9.2. Single Use

The server **MUST** record, atomically, that a given challenge has been answered, and reject a second credential for the same challenge.

The record **MUST** be created by a compare-and-set that fails if the key already exists, not by a read followed by a write. Two replicas presented with the same credential concurrently will both observe an empty store on a read-then-write, and both will accept. The store **MUST** therefore be shared across every process serving the realm, and **MUST** be durable: a store lost on restart re-opens every window it was protecting.

9.3. Transaction Field Verification

The server **MUST** verify, against the transaction as recorded on the ledger rather than as presented by the client:

1. `TransactionType` is `Payment`. An `EscrowCreate` or a `PaymentChannelCreate` can carry the right destination and amount while delivering nothing: an escrow can be cancelled back to the sender and a channel deposit reclaimed after its settle delay.
2. `Destination` equals the challenged `recipient`.
3. The delivered amount equals the challenged `amount`. The server **MUST** use `delivered_amount` from the transaction metadata where it is present, not the `Amount` field, which is an upper bound.
4. The asset matches the challenged `currency`, including issuer for an issued currency and `mpt_issuance_id` for an MPT.
5. The `tfPartialPayment` flag is not set [XRPL-PARTIAL]. With that flag a Payment may deliver less than `Amount` and still succeed.
6. `Account` matches the address encoded in the credential's `source` DID. Without this check a client can present a third party's transaction as its own.
7. `DestinationTag` and `SourceTag` match the challenge where the challenge specifies them.

9.4. Challenge Binding

Field verification alone does not tie a payment to a challenge. Any earlier payment by the same account, for the same amount, to the same recipient satisfies every check in Section 9.3. The binding closes this.

The expected `InvoiceID` is either the value the server placed in `methodDetails.invoiceId`, or -- absent that -- a value derived from the challenge identifier. When derived, it is the SHA-512Half of the challenge identifier, rendered as 64 uppercase hexadecimal characters. SHA-512Half is the ledger's own digest convention.

Because the challenge identifier is itself integrity-protected over the whole challenge, binding to it transitively binds the payment to the amount, recipient, currency and expiry the server issued.

A server **MUST** require a matching `InvoiceID` when the credential is a `hash` payload, and when the challenge carried an explicit `invoiceId`. On the `transaction` path the binding **SHOULD** be verified when present but **MAY** be absent, since sequence-number consumption already prevents reuse there; this keeps clients predating the field working.

9.5. Finality

A `tesSUCCESS` result is not settlement [XRPL-FINALITY]. Nodes report metadata for transactions in the open ledger, which can still be reordered or dropped. The server **MUST** confirm the transaction is reported in a validated ledger before returning a receipt, and **MAY** require the validated ledger to be buried under further closed ledgers as defence against a single node reporting a validation its peers have not seen.

9.6. Transaction Age

The server **SHOULD** reject a transaction that settled before the challenge could plausibly have been issued. Combined with the binding, this narrows the window in which any transaction can be offered as proof.

10. Settlement Procedure

10.1. Pull Mode

The server deserializes the blob, verifies it, submits, and polls until the transaction appears in a validated ledger or the submission window closes.

Verification **MUST** happen twice, against two different objects.

First against the decoded blob, before submission. A blob that fails a check is rejected without being broadcast, so a malformed or mistargeted credential costs the payer nothing.

Then again against the transaction as the ledger recorded it. The two are normally identical, and checking only the first would trust the server's own decode of client-supplied bytes over what actually settled. The second pass is also the only one that can read `delivered_amount`, which exists only in the metadata.

10.2. Push Mode

The server looks the hash up. A node that does not yet know the transaction is not evidence of absence -- propagation takes time -- so the server **SHOULD** retry a small number of times before treating a hash as unknown, and **MUST NOT** hold the request open for the full poll budget on a hash that is simply fabricated.

10.3. Failure Handling

An XRPL result code [XRPL-TX-RESULTS] is not an error shape a client can reason about. Servers **MUST** map result codes to the error vocabulary below and **MUST NOT** surface raw ledger strings.

Result	Condition
`tecUNFUNDED_PAYMENT`	sender cannot cover amount plus reserve [XRPL-RESERVES]
`tecNO_DST`	destination account does not exist
`tecPATH_DRY`	no usable path for the issued currency
`tecPATH_PARTIAL`	path cannot deliver the full amount
`tecNO_LINE`	no trustline for the issued currency
`tecNO_AUTH`	trustline exists but is not authorized
`tecFROZEN`	trustline or issuer is frozen
`tecMPT_NOT_AUTHORIZED`	holder not authorized for the MPT
`tecINSUFFICIENT_RESERVE`	reserve requirement unmet
`tefPAST_SEQ`	sequence consumed; blob already settled
`tecNO_PERMISSION`	context-dependent: MPT holder not authorized, or the recipient requires deposit authorization [XRPL-DEPOSITAUTH]
`temBAD_AMOUNT`	amount malformed or zero
`tecMPT_LOCKED`	MPT holding is locked
`tecINSUFF_FEE`, `terINSUF_FEE_B`	fee below the current load-scaled minimum

Result	Condition
`tefBAD_AUTH`, `tefMASTER_DISABLED`	signing key not valid for the account

Table 3

11. Receipts

A receipt for a settled charge carries the base fields [I-D.httpauth-payment] defines, and the following in addition:

Field	Type	Required	Meaning
`txHash`	string	**REQUIRED**	Hash of the settled transaction, 64 uppercase hexadecimal characters
`ledgerIndex`	number	**OPTIONAL**	Index of the validated ledger it settled in

Table 4

The base `reference` **MUST** also carry the transaction hash, which is what the core specification asks of a method-specific reference. The named field exists because one opaque value cannot be read without method knowledge: a consumer looking for a transaction hash finds no field called one, and confirming the value means testing whether it resolves on the ledger.

Servers **SHOULD** emit both. Clients **SHOULD** read `txHash` and fall back to `reference`.

12. Error Responses

Errors are Problem Details [RFC9457] carried on a `402` response. Every payment failure is a `402`; `401` is reserved for non-payment authentication failures.

The type URIs are those [I-D.httpauth-payment] already defines, under its base URI `https://paymentauth.org/problems/`. A malformed or unparseable credential is `malformed-credential`; a challenge that is unknown, expired or already spent is `invalid-challenge`; every other failure of the procedure in Section 9 is `verification-failed`. This document defines no type of its own.

A server **MUST NOT** disclose, in an error, whether a given challenge was previously used by a different caller, nor any part of the transaction beyond what the caller supplied.

13. Security Considerations

13.1. Transport Security

The server's ledger connection carries the evidence on which payment decisions rest. Implementations **MUST** use TLS (`wss://` or `https://`) for any non-loopback node, and **MUST NOT** accept a plaintext node address outside a development configuration that says so explicitly.

13.2. Replay Protection

Section 9.2 is the load-bearing requirement. Three properties are each necessary and none is sufficient alone: the check must be atomic, the store must be shared across replicas, and it must be durable.

Retention is not optional either. A record deleted while its challenge is still presentable re-opens the window it existed to close, so a record **MUST** be retained at least until the challenge expires plus the longest time verification may take.

13.3. Push Mode Exposure

A transaction hash is public the moment it is validated. Anyone who observes the ledger can present another party's hash. Two checks make this ineffective: the sender must match the credential's DID (Section 9.3), and the binding must match (Section 9.4). A server that relaxes either on the push path has no defence.

13.4. Amount Precision

The total XRP supply is 10^17 drops, beyond the 2^53 integers exactly representable in IEEE-754. An implementation that converts drops to a floating-point XRP figure -- for display, comparison, or signing -- loses precision above 2^53 drops, which is 9,007,199,254 XRP, or about nine percent of the total supply.

That threshold is far above any plausible single payment, so the practical exposure is small. It is stated as a **MUST** regardless, because the failure is silent: the converted value differs from the intended one by a few drops, and a signature computed over it will not verify against the amount actually submitted. Implementations **MUST** use exact integer arithmetic throughout.

13.5. Settlement Is Final, Token Balances Are Not

A validated Payment cannot be reversed. That is a property of the ledger, and it is what lets a server decide within one round trip.

It does not follow that the value received is permanent. For an issued currency whose issuer has enabled clawback [XLS-39], the issuer may reclaim the tokens from any holder afterwards with a `Clawback` transaction. The payment settled; the balance later left.

This is a property of the asset, not of this method, and no verification step can detect it after the fact. A server accepting an issued currency is extending trust to its issuer, and **SHOULD** decide which issuers it accepts rather than accepting any token that arrives with the right code. Servers that cannot make that judgement **SHOULD** charge in XRP, which has no issuer.

13.6. Deposit Authorization Can Refuse the Payment

An account may set Deposit Authorization [XRPL-DEPOSITAUTH], after which it receives no payment from an unauthorized sender. A recipient configured this way rejects every charge until each payer is preauthorized, which is a deployment error rather than an attack, but it presents as a payment that cannot be made rather than as a misconfiguration.

A server **MAY** confirm before advertising a charge that its recipient can receive from arbitrary senders. Whether that check is worth a round trip per challenge is a deployment question: the condition is static, so checking once at startup is usually enough.

What is not optional is the reporting. `tecNO_PERMISSION` covers more than one cause and the result code alone does not say which, so a server **MUST NOT** report it as though it did. What the transaction carried does narrow it: on a payment moving an MPT the cause is an unauthorized holder, and a server **SHOULD** say so. Otherwise the server **SHOULD** name the destination's refusal, and **MAY** name deposit authorization as the usual cause without asserting it -- only reading the destination's account flags settles that. A configuration condition on the recipient, reported as a failure of the payment, sends the operator looking at the wrong account.

13.7. The Challenge Must Match the Resource

A verifier reads what to expect from the challenge the credential carries. That is sound only while the challenge is known to be the one this resource issued. A server **MUST** confirm that the terms it is about to verify are the terms the requested resource charges, and **MUST** refuse the credential otherwise.

The check covers every term the verifier acts on, not only the priced ones. Amount, currency and recipient decide what is owed; a destination tag decides which sub-account the funds land in, and an invoice identifier decides what the payment is bound to. A term the resource sets and the verifier honours is a term that **MUST** be confirmed. Terms the resource leaves unset are not demands and **MUST NOT** be treated as such.

Without this a server offering more than one priced resource under one challenge-issuing key accepts a challenge minted for the cheaper or less-constrained one against the stricter one. Where a scheme re-derives the resource's own challenge before dispatch, that binding already holds for the fields the derivation covers; a verifier reachable outside that path has no such protection, and embedded and script callers routinely are.

Identifier comparisons here follow the same rule as everywhere else in this document: hex is compared as a value, not as a string.

13.8. Display Fields Are Not Decisions

`description` and `externalId` are attacker-influenced strings that travel through the challenge. They **MUST NOT** participate in any authorization decision, and **MUST** be treated as untrusted input by anything that renders them.

13.9. Key Material

A charge requires the payer's signing key. Implementations **SHOULD** keep the key out of any value that can be serialized, logged or included in an error, and **SHOULD** prefer injecting a signing capability over a raw seed.

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
`xrpl`	XRP Ledger payments: XRP, issued currencies, MPTs	This document

Table 5

Contact: Maxime Dienger (maximed@ripple.com)

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`	`xrpl`	One-time XRP, issued currency or MPT transfer	This document

Table 6

14.3. Problem Types

This document registers no problem type URI. Every condition in Section 12 is reported with a type [I-D.httpauth-payment] already establishes.

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>.

[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>.

[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-ryan-httpauth-payment/>.

15.2. Informative References

[XLS-33] XRP Ledger Standards, "XLS-33: Multi-Purpose Tokens", 2024, <https://github.com/XRPLF/XRPL-Standards/tree/master/XLS-0033-multi-purpose-tokens>.

[XRPL-PAYMENT] XRP Ledger Foundation, "Payment Transaction", 2026, <https://xrpl.org/docs/references/protocol/transactions/types/payment>.

[XRPL-PARTIAL] XRP Ledger Foundation, "Partial Payments", 2026, <https://xrpl.org/docs/concepts/payment-types/partial-payments>.

[XRPL-CURRENCY] XRP Ledger Foundation, "Currency Formats", 2026, <https://xrpl.org/docs/references/protocol/data-types/currency-formats>.

[XRPL-TRUSTLINES] XRP Ledger Foundation, "Trust Lines and Issuing", 2026, <https://xrpl.org/docs/concepts/tokens/fungible-tokens>.

[XRPL-RESERVES] XRP Ledger Foundation, "Reserves", 2026, <https://xrpl.org/docs/concepts/accounts/reserves>.

[XRPL-FINALITY] XRP Ledger Foundation, "Finality of Results", 2026, <https://xrpl.org/docs/concepts/transactions/finality-of-results>.

[XLS-39] XRP Ledger Standards, "XLS-39: Clawback", 2023, <https://github.com/XRPLF/XRPL-Standards/tree/master/XLS-0039-clawback>.

[XRPL-DEPOSITAUTH] XRP Ledger Foundation, "Deposit Authorization", 2026, <https://xrpl.org/docs/concepts/accounts/depositauth>.

[XRPL-AMENDMENTS] XRP Ledger Foundation, "Known Amendments", 2026, <https://xrpl.org/resources/known-amendments>.

[XRPL-TX-RESULTS] XRP Ledger Foundation, "Transaction Results", 2026, <https://xrpl.org/docs/references/protocol/transactions/transaction-results>.

Appendix A. Examples

A.1. Challenge

The `402` carries the challenge in a `WWW-Authenticate` header, with the request object base64url-encoded in the `request` parameter, as [I-D.httpauth-payment] defines:

```
HTTP/1.1 402 Payment Required
WWW-Authenticate: Payment id="PgHorYGpATG5ifG-QKCa2OVPjOku...",
  realm="api.example.com", method="xrpl", intent="charge",
  request="eyJhbW91bnQiOiIxMDAwMDAwIiwiY3VycmVuY3kiOiJYUlAiLC...",
  expires="2026-08-21T10:32:00Z"
```

The `request` parameter decodes to the object this document specifies in Section 7:

```
{
  "amount": "1000000",
  "currency": "XRP",
  "methodDetails": {
    "network": "testnet",
    "reference": "3f7a1c02-9e44-4b1e-8a10-0c2b5d6e7f80"
  },
  "recipient": "rhewi79quXUDwcqjkpj4bXuw3cuHYC9fwv"
}
```

The header parameters and the request object are distinct: `method`, `intent` and `expires` are the scheme's, while `amount`, `currency` and `recipient` belong to the encoded request. The line breaks above are for presentation.

A.2. Settled Transaction

The Payment produced by the challenge above, as recorded on the XRP Ledger testnet. `InvoiceID` is the derived binding; `SourceTag` identifies the originating SDK.

```
{
  "TransactionType": "Payment",
  "Account": "rBkRQZrL4K8Rg2Bg2UzyQUSzYyNeBAK95Z",
  "Destination": "rhewi79quXUDwcqjkpj4bXuw3cuHYC9fwv",
  "Amount": "1000000",
  "Fee": "12",
  "Sequence": 19831823,
  "InvoiceID":
    "0B4F21A0C9A47D3351CD611AD7D41390AA64C22E7D61CFC78B52A814D0359CCB",
  "SourceTag": 593184257,
  "LastLedgerSequence": 19831855
}
```

Validated in ledger 19831837 with result `tesSUCCESS`.

A.3. Issued Currency Challenge

```
{
  "method": "xrpl",
  "intent": "charge",
  "amount": "10",
  "currency": "{\"currency\":\"USD\",\"issuer\":
\"rwzRswng9sqR9Buw2T8FG18K4n8xdd1dCa\"}",
  "recipient": "rhewi79quXUDwcqjkpj4bXuw3cuHYC9fwv",
  "expires": "2026-08-21T10:32:00Z",
  "methodDetails": { "network": "testnet" }
}
```

Settling this requires a trustline to the issuer on the recipient side and rippling enabled on the issuer. Absent either, the ledger answers `tecPATH_DRY` and no payment occurs.

Author's Address

Maxime Dienger
Ripple
Email: maximed@ripple.com