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

XRP Ledger Session Intent for HTTP Payment Authentication

Abstract

This document defines the "session" intent for the "xrpl" payment method within the Payment HTTP Authentication Scheme. A session is carried by an XRP Ledger Payment Channel: the client locks XRP on-chain once, then authorises a series of off-chain claims, each a signature over a cumulative total. The server redeems the final claim in a single closing transaction.

Two on-chain transactions therefore settle an unbounded number of payments, which is what makes per-request and per-token billing viable at amounts where a transaction fee would otherwise dominate.

The "session" intent is experimental. It is defined here rather than in a standalone intent document because it is not yet formalized in the intent registry.

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

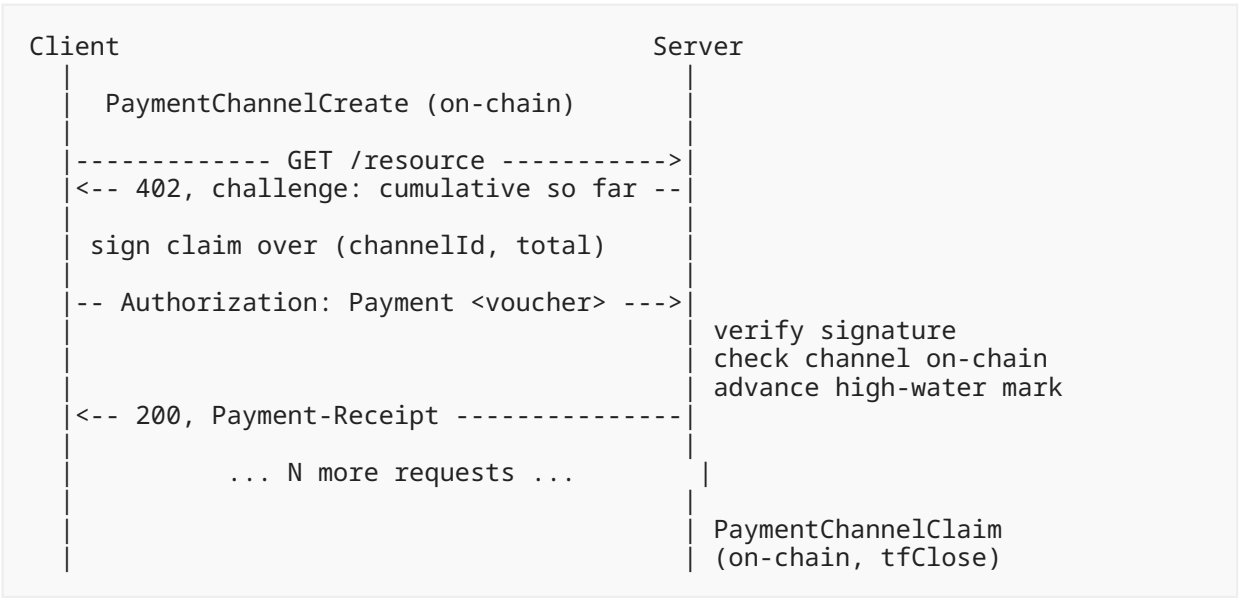
1. Introduction

The charge intent [I-D.xrpl-charge] settles every payment on-chain. That is correct and final, but it costs a transaction and several seconds each time, which rules out the cases this intent exists for: paying per API call, per inference token, or per streamed chunk.

An XRP Ledger Payment Channel [XRPL-PAYCHAN] decouples authorisation from settlement. The funder locks XRP in a channel naming a destination, which also locks an owner reserve for the new ledger entry [XRPL-RESERVES]. Thereafter it signs claims off-chain, each stating a *cumulative* total rather than an increment. The destination may redeem the highest claim it holds at any time, in one transaction.

Cumulative rather than incremental is the property that makes this safe with no coordination: a lost or reordered claim costs nothing, because the next one supersedes it. The server need only retain the largest.

1.1. Session Flow



1.2. Channels Are XRP-Only

Payment Channels carry XRP exclusively. Issued currencies and MPTs cannot fund a channel, so every amount in this document is an integer drop count. A server needing off-chain settlement in another asset must use a different mechanism; this intent does not provide one.

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

Channel: A `PayChannel` ledger entry [XRPL-PAYCHAN-OBJECT] created by `PaymentChannelCreate` [XRPL-CHAN-CREATE], identified by a 256-bit channel ID, naming an `Account` (the funder), a `Destination`, an `Amount` deposited, a `Balance` already redeemed, a `PublicKey` authorised to sign claims, and a `SettleDelay`.

Claim (voucher): A signature by the channel's `PublicKey` over the tuple of channel ID and a cumulative drop amount. Not a ledger transaction.

Cumulative amount: The running total authorised since the channel opened [XRPL-BASIC-TYPES], not the amount owed for one request.

High-water mark: The largest cumulative amount a server has accepted for a channel.

SettleDelay: Seconds the funder must wait, after initiating closure, before the channel can be destroyed. The window in which the destination can still redeem.

4. Method Identifier

The method identifier is `xrpl`, as in [I-D.xrpl-charge], carried in the challenge and credential fields the Payment HTTP Authentication Scheme [I-D.httpauth-payment] defines. This document defines only the `session` intent for it.

5. Intent: "session"

`session` is the intent identifier on the wire. An implementation **MAY** expose a local alias for its own API, but **MUST** advertise and accept `session` in challenges and credentials.

6. Encoding Conventions

The encoding rules of [I-D.xrpl-charge] apply to this document unchanged. They are inherited rather than restated: the rule that case must not be load-bearing in a hex identifier is the one a divergent copy would quietly break, and one statement cannot diverge from itself.

What this intent adds is narrower.

Amounts: Every amount is an integer count of drops, as a decimal string, with no fractional part. A channel carries XRP alone, so the token units of [I-D.xrpl-charge] do not arise here.

Cumulative amounts: A voucher states the running total authorised over the channel's life, not the increment it adds. It is an integer drop count on the same terms as any other amount here, and it is compared and accumulated as an exact integer -- see Section 12.5 for why, and for the boundary that makes it a **MUST**.

Channel identifiers: A `channelId` is 64 hexadecimal characters, carrying the case rules [I-D.xrpl-charge] states. A claim signature is verified over the hex-decoded identifier, so two spellings of one channel are one channel, and anything derived from it -- a high-water mark above all -- **MUST** be keyed on a canonical form.

7. Request Schema

Field	Type	Required	Meaning
`amount`	string	yes	increment charged for this request, in drops
`channelId`	string	yes	64-hex channel ID, or `""` on an open
`recipient`	string	yes	classic address the channel must pay
`currency`	string	no	always `"XRP"` when present
`description`	string	no	display only

Table 1

`amount` is the increment for this request, and the cumulative total is the server's to state where it can. A challenge that names a channel **SHOULD** report the mark for it, so a client resumes from server state rather than its own bookkeeping.

A challenge that names no channel cannot: there is nothing to look the mark up by, and reporting zero is all it can do. A client **MUST** therefore track the highest cumulative it has signed, per network and channel together, and sign above that, taking whichever of the two is greater. Signing from a reported zero alone re-sends an accepted cumulative, which the server **MUST** refuse as a replay -- see Section 9.8.

Per network as well as per channel because a channel ID does not identify a channel on its own: it derives from the funder, the destination and a sequence number, and one seed controls the same address on every network, so the same funder opening to the same destination from a fresh account produces the same ID twice. A mark shared between them makes the second network sign above what it was asked for.

`channelId` is empty when the server names no channel, for either of two reasons. On an open-action challenge the channel does not exist yet: its ID cannot be read until the creating transaction is validated. Otherwise, a server accepting callers it has not met has no channel to

name, because a client learns its channel ID from its own `PaymentChannelCreate`. In both cases the client supplies the channel. A `voucher` or `close` payload **MUST** carry a full 64-hex channel ID; the empty form is confined to the challenge. An open payload carries none, and cannot: the channel does not exist until the transaction it carries has been validated, and the server reads the ID from that transaction's metadata.

7.1. Network

Servers **MUST** set `network` in `methodDetails`, and a server **MUST** refuse a credential whose challenge omits it or names a network other than the one the server settles on. A charge only recommends it [I-D.xrpl-charge]; a session requires it.

The reason is that a session spans requests. Absent the field, each side falls back to a default of its own, and those can differ silently: a claim signed for a channel the client believes is on one ledger, verified and redeemed by a server on another. It also leaves a client that pinned a network nothing to compare against, so that guard cannot fire. A challenge naming a network the server does not settle on is the same divergence stated out loud, and is reachable where two deployments share a secret and settle on different ledgers.

Clients **SHOULD** refuse a challenge naming a network other than the one they were configured for. The stake is higher here than on a charge: answering an open-action challenge means submitting a `PaymentChannelCreate`, so a client that follows the challenge deposits real XRP on a ledger its operator did not choose. The same seed controls the same address on every network, so the deposit comes from a funded account whatever the client believed it was configured for.

8. Credential Schema

Credentials and challenges are JSON [RFC8259]; timestamps such as a challenge's `expires` are [RFC3339].

The payload is discriminated by `action`.

8.1. action = "open"

```
{
  "action": "open",
  "transaction": "1200...",
  "amount": "100000",
  "signature": "304402..."
}
```

`transaction` is a signed but unsubmitted `PaymentChannelCreate` [XRPL-CHAN-CREATE]. The server broadcasts it, reads the channel ID from the validated metadata, then treats `amount` and `signature` as the first claim.

This folds channel establishment into the 402 exchange, so no out-of-band endpoint is needed. A server **MAY** instead require the client to open the channel itself and supply the ID by other means.

A client **SHOULD** take the transaction's `Destination` from the challenge's `recipient`, which makes the exchange self-contained: nothing about the server has to be known before the resource is asked for. The deposit and the `SettleDelay` are not in the challenge and **MUST NOT** be inferred from one -- they bound what the funder stands to lose and how long it waits to recover it, so they belong to the funder.

8.2. action = "voucher"

```
{
  "action": "voucher",
  "channelId":
"2D398F9458B0CF96284E3602E57A83E787C1A01658512F972CEDDB1819607E89",
  "amount": "200000",
  "signature": "304402..."
}
```

`amount` is the new cumulative total, not the increment.

8.3. action = "close"

```
{
  "action": "close",
  "channelId":
"2D398F9458B0CF96284E3602E57A83E787C1A01658512F972CEDDB1819607E89",
  "amount": "500000",
  "signature": "304402..."
}
```

Asks the server to redeem and close. The fields are those of a `voucher`, and carry the same meaning: `amount` is the final cumulative total, not an increment, and the signature covers it.

A server **MUST** verify a `close` payload exactly as it verifies a `voucher` before acting on it. The request is a claim like any other, and being the last one confers no standing: a `close` carrying a cumulative below the mark, or a signature that does not verify, **MUST** be refused on the same terms.

Redemption itself is the server's to perform and its timing the server's to choose -- see Section 10.1. A client cannot compel a close by asking for one, and a server **MAY** treat the action as a voucher that also signals the session is over.

9. Verification Procedure

The order is normative. Cheap local checks precede network calls so that an unauthenticated caller cannot use verification to generate ledger traffic.

9.1. Size and Shape

The server **MUST** bound the credential size before parsing, and reject a payload whose fields do not match the schema.

9.2. Signature

The server **MUST** verify the claim signature over the tuple of channel ID and cumulative amount, against the channel's authorised public key [XRPL-KEYS].

That key is a property of the channel, not of the server. A funder chooses it in its own `PaymentChannelCreate` [XRPL-CHAN-CREATE], so a server accepting channels from callers it has not met cannot know it in advance and **MUST** read it from the channel.

The signature check therefore follows the read rather than preceding it. A consequence worth stating: a forged claim on a channel identifier the server has not seen costs it one read. Servers **SHOULD** cache channel state per channel, which reduces this to established channels costing nothing, and **SHOULD** rate-limit ahead of verification.

Verification **MAY** be performed locally or through the ledger's `channel_verify` method [XRPL-CHANNEL-VERIFY]. Local verification is preferred: it costs no round trip, and it does not disclose to a node which channels a server is being paid through.

A matching `channel_authorize` [XRPL-CHANNEL-AUTHORIZE] exists for producing claim signatures, but it is an admin method and takes a secret, so a client signs locally in practice.

Ordering matters. A signature check is local arithmetic; a channel lookup is a network round trip. Verifying the signature first means a caller supplying random channel IDs is rejected without the server making a request on its behalf.

The claim signs an XRP-denominated figure. Implementations **MUST** derive it from the drop count by exact integer arithmetic. A conversion through a binary floating-point value is lossy above 2^53 drops and, where it is lossy, produces a signature over a figure differing from the amount that will be submitted, which then fails to verify on-chain. See Section 12.5.

9.3. Sender Binding

The channel's `Account` **MUST** match the address in the credential's `source` DID. Without this, a claim can be replayed under another party's identity.

The comparison **MUST NOT** be made against an address derived from the channel's authorised public key. A channel may name any valid key [XRPL-PAYCHAN-OBJECT], and a funder is well advised to dedicate a key pair to the channel so that its loss costs that channel alone; the address derived from such a key belongs to nobody. Deriving the funder from the key therefore rejects the funders that hold their keys most carefully, and where it does succeed it establishes only that the account and the channel key coincide.

A server therefore has to have read the channel before it can make this comparison, which is one more reason the read is not optional.

9.4. Channel State

The server **MUST** confirm, against the ledger, that:

1. the channel exists;
2. its `Destination` is the recipient this server is charging for -- a funder can otherwise open a channel to an address of its own choosing and receive service against claims this server can never redeem;
3. its `PublicKey` is the key the claim signature was verified against;
4. its `SettleDelay` is at least the server's configured minimum;
5. the cumulative claimed is greater than `Balance` and no greater than `Amount`;
6. the channel is not expired, and not within the settlement margin of expiry.

`Amount` is everything the channel holds and `Balance` is what it has already delivered, so a cumulative claim is bounded by `Amount` alone. Subtracting `Balance` from it would reject valid claims: a channel holding 1,000,000 drops that has delivered 500,000 still honours a claim for 600,000.

The drops a claim delivers on redemption are `claimed - Balance`. A server tracking what it has earned **SHOULD** compute the increment against `max(Balance, mark)` rather than against its own high-water mark alone, since a claim redeemed outside this exchange advances `Balance` without the server observing it.

A server **MAY** cache this state briefly, but the two fields that can move against it need care of different kinds.

`Amount` only ever grows, since a funder may add to a channel and cannot withdraw from it. A stale value is therefore pessimistic: it under-reports the deposit and can only cause an unnecessary refusal, which a single re-read resolves.

`Expiration` is not monotone. A funder may set or shorten it at any time, so a stale absence of an expiry is optimistic in exactly the wrong direction. A cache lifetime as long as the settlement margin of Section 9.6 spends that whole margin on staleness: the channel may have entered its closing window a full lifetime ago, leaving no real time to redeem. Implementations **MUST** keep the cache lifetime materially below the margin, or read `Expiration` fresh.

9.5. Settle Delay Floor

The server **MUST** reject a channel whose `SettleDelay` is below a configured minimum, and that minimum **SHOULD** be no less than one hour.

The delay is the whole of the destination's protection. Once the funder initiates closure, the destination has exactly `SettleDelay` to submit its claim. A channel with a delay of sixty seconds lets a funder consume service and close before any realistic operator can detect and respond, and the unredeemed value returns to the funder.

9.6. Closing Window

The server **MUST** refuse a voucher when the channel is within a configured margin of `Expiration` or `CancelAfter`.

One ledger amendment [XRPL-AMENDMENTS] bears directly on this window. `fixPayChanCancelAfter` makes `PaymentChannelCreate` fail when `CancelAfter` is already in the past; without it a channel could be created that was unusable from the moment it existed. A server **MUST NOT** assume the amendment is enabled on the network it is talking to. It does not need to: reading `CancelAfter` from the channel, which Section 9.4 requires anyway, settles the question for that channel whatever the network has enabled.

Accepting a claim in that window earns value the server has no time left to redeem. Treating those fields as advisory -- reporting them while still accepting the claim -- is not sufficient: after `CancelAfter` anyone may delete the channel and the deposit returns to the funder.

9.7. State Keys

Every piece of state a server keeps for a channel -- its high-water mark, any cached ledger metadata, and any record that the channel is finalized -- **MUST** be keyed on the pair of network and channel ID, not on the channel ID alone.

A channel ID does not identify a channel on its own. It derives from the funder, the destination and a sequence number, and one seed controls the same address on every network, so the same funder opening to the same destination from a fresh account produces the same ID twice. Keyed on the ID alone, testnet activity moves a mainnet channel's mark, and a channel finalized on one network is refused on the other.

The channel ID **MUST** be canonicalised before use as a key. Hex is case-insensitive as a value, and both layers beneath the store treat it that way: a claim signed over one casing verifies against another, and the ledger resolves either. A key built on the raw string is therefore not one key but one per casing, and the same voucher can be spent once for each -- unbounded in practice, since a 64-character identifier has as many casings as it has letters.

9.8. Monotonicity

The server **MUST** reject a cumulative amount that is not strictly greater than its high-water mark for that channel, and **MUST** perform the comparison and the update as one atomic operation.

Three outcomes are distinct and **MUST** be distinguished:

Condition	Meaning
cumulative equals the mark	replay of an accepted claim
cumulative below the mark	attempt to roll back
cumulative above, but increment below what was requested	underpayment

Table 2

The third is the one most easily missed. A first claim on a fresh channel has no previous mark to exceed, so a check written only against the mark accepts any positive amount -- one drop satisfies a one-XRP request.

The update **MUST** be a compare-and-set, shared across every process serving the channel and durable across restarts. A read followed by a write lets two replicas accept the same claim concurrently, and a mark lost on restart lets every claim be replayed.

9.9. Finalized Channels

Once the server has closed a channel, it **MUST** record that and reject later claims against it. A closed channel cannot be redeemed again, so a claim accepted afterwards is service given away.

10. Settlement Procedure

10.1. Redemption

The server submits `PaymentChannelClaim` [XRPL-CHAN-CLAIM] carrying the highest cumulative amount it holds and the matching signature.

The `tfClose` flag is accepted from the source and from the destination alike, and its effect differs by sender. From the destination the channel closes at once: the claim settles, the entry is deleted, and the unspent deposit returns to the funder. From the source it schedules closure for once `SettleDelay` has elapsed, which is what preserves the destination's window described in Section 9.5.

A server ending a session **SHOULD** set it. One transaction then both collects what was earned and releases the funder's deposit and owner reserve, where a claim without it leaves the entry in place holding both.

Implementations **MUST** verify which flag they set. `tfClose` and `tfRenew` are adjacent values, and `tfRenew` clears the channel's `Expiration` rather than closing anything. Substituting one for the other fails silently: the claim still settles and the transaction still succeeds, so only reading the channel entry afterwards distinguishes a close from a renewal.

The submitted `Balance` **MUST** be the exact drop count the signature covers. If the two disagree the ledger rejects the signature and the earned value becomes unredeemable.

10.2. Idle Channels

A funder may extend a channel's deposit or expiry with `PaymentChannelFund` [XRPL-CHAN-FUND], but is under no obligation to. A client that disappears mid-session leaves value authorised but unclaimed, and the funder may begin closure at any time. A server **SHOULD** therefore redeem proactively rather than waiting for a client that may not return.

Where `fixPayChanRecipientOwnerDir` [XRPL-AMENDMENTS] is enabled, a channel is listed in the recipient's owner directory as well as the funder's, so a server can enumerate the channels paying it rather than relying solely on its own records. The owner reserve for the entry stays with the funder, who created it; being listed costs the recipient nothing.

10.3. Receipts

A receipt for a session payment identifies the claim, not a transaction. Beyond the base fields [I-D.httpauth-payment] defines:

Field	Type	Open	Voucher	Close	Meaning
`channelId`	string	**REQUIRED**	**REQUIRED**	**REQUIRED**	Channel the payment went through
`cumulative`	string	**OPTIONAL**	**REQUIRED**	**REQUIRED**	Drop total authorised after this claim
`txHash`	string	**REQUIRED**	absent	conditional	Transaction the server submitted

Table 3

A voucher receipt carries no `txHash`, and **MUST NOT** invent one: the claim settles nothing by itself, and no transaction exists until the channel is closed. An open receipt does carry one, for the `PaymentChannelCreate` the server submitted.

A `close` receipt carries one when the server redeemed in the course of answering, and none when it accepted the claim and deferred redemption, which Section 10.1 permits. Its presence is therefore what tells a client whether settlement has happened, and a client **MUST NOT** infer settlement from the action alone.

The base `reference` remains method-specific and **MAY** carry these values in a composite form. A server **MUST NOT** rely on a client parsing one: a composite cannot be read without method knowledge, which is what the named fields are for.

11. Error Responses

Problem Details [RFC9457] on a `402`. Ledger result codes [XRPL-TX-RESULTS] **MUST NOT** be surfaced raw. Distinct conditions **MUST** be distinguishable by the client, and a server **SHOULD** report each with the type named here:

Condition	Meaning	Problem type
channel not found	no such channel on the ledger	`session/channel-not-found`
destination mismatch	channel does not pay this server	`verification-failed`
settle delay too short	below the server's floor	`verification-failed`
channel expired	past `Expiration`/`CancelAfter`	`session/channel-finalized`
channel closing	inside the settlement margin	`session/channel-finalized`
channel exhausted	claim exceeds the deposit left	`session/amount-exceeds-deposit`
invalid signature	claim does not verify	`session/invalid-signature`
replay detected	cumulative not above the mark	`verification-failed`

Table 4

The types are relative to `https://paymentauth.org/problems/`, the base URI [I-D.httpauth-payment] establishes. Those in the `session` namespace are the ones payment-channel methods already share; this document adds none. Three conditions carry `verification-failed` because they are refusals of the procedure in Section 9 rather than states of the channel, and the detail field distinguishes them.

12. Security Considerations

12.1. The Server Bears the Settlement Risk

The asymmetry is structural and worth stating plainly. The funder's exposure is bounded by the deposit. The server's exposure is every claim it has accepted but not yet redeemed, and it can lose that value in three ways: the funder closes and the delay elapses unnoticed, `CancelAfter` passes, or the server's high-water record is lost before redemption. Only a redemption in a validated ledger settles the matter [XRPL-FINALITY].

Sections 7.5, 7.6, 7.7 and 8.2 each close one of these. None is optional.

12.2. Signature Malleability

For secp256k1 keys an ECDSA signature has two valid encodings differing in the sign of S. An implementation **MUST NOT** treat the two as distinct claims -- a distinct encoding of an accepted claim is still that claim, and accepting it as new would credit the funder twice.

Enforcing canonical low-S form on verification is the direct defence. Implementations **SHOULD** confirm their verifier does so rather than assume it.

12.3. Cache Staleness

Caching channel state trades a round trip for a window in which the server acts on a stale view. The deposit is safe to cache because it only grows. The expiry is not, and a cache lifetime at or above the settlement margin makes that margin nominal -- see Section 9.6.

12.4. Replay Store Durability

The high-water mark is the only record that a claim has been spent. It is not reconstructible from the ledger, which sees only the final redemption. A store lost on restart therefore does not degrade gracefully: every claim ever issued becomes replayable.

12.5. Amount Precision

Drop counts up to the total supply exceed the 2^53 integers exactly representable in IEEE-754; the boundary is 9,007,199,254 XRP. Above it, a conversion through a floating-point value yields a signature over a figure differing from the submitted amount, and the ledger rejects it. The threshold is far above any plausible channel, but the failure is silent and the arithmetic is not hard to get right, so exact integer handling is a **MUST**.

12.6. The Challenge Must Match the Resource

A verifier reads the increment, the channel and the recipient from the challenge the credential carries, so the requirement in "The Challenge Must Match the Resource" of [I-D.xrpl-charge] applies here unchanged: 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. A session compounds the consequence, because a challenge accepted against the wrong resource moves that resource's high-water mark for every voucher that follows.

12.7. Transport Security

As [I-D.xrpl-charge]: TLS for any non-loopback ledger connection.

13. IANA Considerations

The `xrpl` payment method is registered in the "HTTP Payment Methods" registry by [I-D.xrpl-charge]; this document does not register it again.

13.1. 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
`session`	`xrpl`	Off-ledger payment channel vouchers	This document

Table 5

Contact: Maxime Dienger (maximed@ripple.com)

`session` is an experimental intent, defined by method documents rather than by an intent document of its own. Should it be formalized, this document should be updated to reference that document rather than define the intent here.

13.2. Problem Types

This document registers no problem type URI. The conditions in Section 11 are reported with types already established: the core types of [I-D.httpauth-payment], and the `session` namespace that payment-channel methods share.

14. References

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

[I-D.xrpl-charge] Dienger, M., "XRP Ledger Charge Intent for HTTP Payment Authentication", 2026, <https://datatracker.ietf.org/doc/draft-xrpl-charge/>.

14.2. Informative References

[XRPL-PAYCHAN] XRP Ledger Foundation, "Payment Channels", 2026, <https://xrpl.org/docs/concepts/payment-types/payment-channels>.

[XRPL-PAYCHAN-OBJECT] XRP Ledger Foundation, "PayChannel Ledger Entry", 2026, <https://xrpl.org/docs/references/protocol/ledger-data/ledger-entry-types/paychannel>.

[XRPL-CHAN-CREATE] XRP Ledger Foundation, "PaymentChannelCreate Transaction", 2026, <https://xrpl.org/docs/references/protocol/transactions/types/paymentchannelcreate>.

[XRPL-CHAN-CLAIM] XRP Ledger Foundation, "PaymentChannelClaim Transaction", 2026, <https://xrpl.org/docs/references/protocol/transactions/types/paymentchannelclaim>.

[XRPL-CHAN-FUND] XRP Ledger Foundation, "PaymentChannelFund Transaction", 2026, <https://xrpl.org/docs/references/protocol/transactions/types/paymentchannelfund>.

[XRPL-CHANNEL-VERIFY] XRP Ledger Foundation, "channel_verify Method", 2026, <https://xrpl.org/docs/references/http-websocket-apis/public-api-methods/payment-channel-methods/channel_verify>.

[XRPL-CHANNEL-AUTHORIZE] XRP Ledger Foundation, "channel_authorize Method", 2026, <https://xrpl.org/docs/references/http-websocket-apis/admin-api-methods/signing-methods/channel_authorize>.

[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>.

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

[XRPL-KEYS] XRP Ledger Foundation, "Cryptographic Keys", 2026, <https://xrpl.org/docs/concepts/accounts/cryptographic-keys>.

[XRPL-BASIC-TYPES] XRP Ledger Foundation, "Basic Data Types", 2026, <https://xrpl.org/docs/references/protocol/data-types/basic-data-types>.

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

Appendix A. Examples

A.1. Voucher Challenge

As on a charge, the challenge travels as `WWW-Authenticate` parameters with the request object base64url-encoded in `request`:

```
HTTP/1.1 402 Payment Required
WWW-Authenticate: Payment id="qp9htNPwjAcnwDQqNTfHEulIIuAd...",
  realm="api.example.com", method="xrpl", intent="session",
  request="eyJhbW91bnQiOiIxMDAwMDAiLCJjaGFubmVsSWQiOiIyRDM5OEY5NDU...",
  expires="2026-08-21T10:32:00Z"
```

Decoding `request`:

```
{
  "amount": "100000",
  "channelId":
    "2D398F9458B0CF96284E3602E57A83E787C1A01658512F972CEDDB1819607E89",
  "methodDetails": {
    "cumulativeAmount": "200000",
    "network": "testnet",
    "reference": "a55d88b1-0174-4542-9c63-83aa7ac0db2f"
  },
  "recipient": "rhewi79quXUDwcqjkpj4bXuw3cuHYC9fwv"
}
```

`amount` is the increment for this request and `cumulativeAmount` the total already accepted, so the credential below signs their sum. The line breaks are for presentation.

A.2. Voucher Credential

Third request in a session, each charging 100,000 drops. The cumulative total is 300,000; the increment is 100,000.

```
{
  "action": "voucher",
  "channelId":
    "2D398F9458B0CF96284E3602E57A83E787C1A01658512F972CEDDB1819607E89",
  "amount": "300000",
  "signature": "3044022057..."
}
```

A.3. Redemption Transaction

After five such requests the server holds a claim for 500,000 drops and submits:

```
{
  "TransactionType": "PaymentChannelClaim",
  "Channel":
    "2D398F9458B0CF96284E3602E57A83E787C1A01658512F972CEDDB1819607E89",
  "Balance": "500000",
  "Amount": "500000",
  "Signature": "3044022057...",
  "PublicKey": "ED690468FD78177F3158..."
}
```

Five payments, two on-chain transactions.

Author's Address

Maxime Dienger
Ripple
Email: maximed@ripple.com