ILP v4: Version 4 of the Interledger Protocol.

D. Appelt A. Hope-Bailie M. de Jong E. Schwartz
B. Sharafian S. Thomas B. Way

April 2018

Abstract

In 2015, Evan Schwartz and Stefan Thomas published version 1 of the Interledger protocol (ILP). After three years of research, we are proud to present its evolution, ILP version 4. ILPv4 uses a chain of Hash Time Lock Agreements (HTLAs) between network neighbors, to form multi-transfer Interledger payments that remotely interconnect non-neighbors.

1 Conditional Transfers between Neighbors

Consider a network of agents, where each agent has accounting relationships with at least one other agent in the network. An accounting relationship between two agents is defined by a unit of value and a log of HTLAs whose amounts are expressed in this unit of value, so that they can be netted against each other (total amount in one direction, minus total amount in the other direction), to define what one agent owes the other.

1.1 Hash Time Lock Agreements

Suppose Alice and Bob are two agents who are neighbors in an Interledger network. A Hash Time Lock Agreement (HTLA) between them is then defined by a `hash`, a `time`, and an agreed `amount`. If Alice sends such a HTLA to Bob, and Bob rejects the agreement, the state of their accounting relationship does not change. However, if Bob presents the SHA256 preimage of the `hash` before `time`, then the HTLA is added to the relationship's log, to say Alice now owes an additional `amount` units of value to Bob.

1.2 Interledger Transfers

An Interledger Transfer is a specific kind of Hash Time Lock Agreement, designed to form multi-transfer chains of HTLAs in the network. To this end, it specifies a `destination`, which is an ILP address: a hierarchical identifier for an agent in the network who may be able to produce the SHA256 preimage of the HTLA's hash. Apart from this destination field, the packets that describe an Interledger transfer also have `data` fields, which should be transported end-to-end over a multi-transfer chain (see below).

2 Multi-transfer Hashlocks

Suppose Alice sends an Interledger Transfer to Bob, whose `destination` points at one of Bob's other network neighbors, Charlie. Bob can then create a second transfer, from him to Charlie, with the same `hash`, and with a slightly earlier `time` deadline. If Charlie fulfills this transfer from Bob on time, then Bob will be able to use the preimage presented by Charlie, to fulfill the incoming transfer from Alice before its deadline. We call this a two-transfer payment (from Alice, via Bob, to Charlie) and the same chaining method can be repeated to create payments with more than two transfers.

3 ILP Addresses

As stated earlier, an ILP address is a hierarchical identifier for a specific agent in the network. An ILP address is a string, consisting of one or more segments, separated by dots (`.`). Each segment may contain alphanumeric characters (upper or lower case, case-sensitive), underscores (`_`), tildes (`~`), and hyphens (`-`).

3.1 Static Routing

In Interledger networks where the set of agents participating in the network and the connections between them don't change, an easy way to make sure that all agents in the network know how to route multi-transfer payments towards their destination, would be to assign an ILP address to each agent, and statically configure the forwarding rules for each destination in each agent. In any case, the first two agents to form a new network can choose their own ILP addresses that way. But when a new agent joins, one of two things can happen: they may become a child

of the agent through which they get joined to the network, or become a first-class citizen.

3.2 Sub-Addresses

If the new agent becomes a child, then it takes the address of the parent through which it joins the network, and adds one or more segments at the end. By using its parent's address as a prefix, other agents in the network will always be able to route payments to the new agent, provided they know how to route a payment to the new agent's parent. This is the hierarchical way to generate new ILP addresses.

3.3 Route Broadcasts

If the new agent wants to become a first-class citizen of the network, they can pick their ILP address at will, and send their neighbor a message to broadcast the address they picked. If that address already exists in the network, the neighbor should respond with an error message. Otherwise, they can forward the broadcast to inform all existing agents of the agent that newly joined the network. Just like a child node may only be reached through its parent, it may also be set up to only send payments through its parent. In that case, it does not need to take route broadcasts into account, and can instead rely on its parent to take care of routing each of its outgoing payments in the right direction.

4 Interledger Packets

An Interledger Packet is a standardized message, which two neighbors can use for communicating about Interledger transfers. There are three Interledger Packet types, `Prepare` (for proposing a transfer), `Fulfill` (for accepting a transfer from a neighbor), and `Reject` (for explicitly rejecting it before the deadline).

4.1 End-to-end Data

As multiple transfers are chained into an Interledger payment, the initiator of the first transfer in the chain (the 'sender') may want to send some data to the beneficiary of the last transfer in the chain (the 'receiver'). For this, Prepare packets have a `data` field. Each agent who participates in a chain of Interledger transfers is expected to copy this data unchanged from their incoming transfer's Prepare packet to their outgoing transfer's Prepare packet. Likewise, just like one `Prepare` packet

triggers the next, and its data is carried over, each `Fulfill` packet will generally trigger another `Fulfill` packet, and each `Reject` packet will trigger another `Reject` packet, on the return path. For these, the `data` field should also be copied over, so that the receiver can relay data back to the sender. This transportation of end-to-end data is not guaranteed by the chain of hashlock conditions, but can still be useful, for instance if sender and receiver sign the data payload cryptographically.

4.2 Prepare

A Prepare packet is an OER sequence containing five fields: `amount` (from the HTLA, as a `UInt64`), `expiresAt` (the HTLA deadline, as a `PrintableString (size 17)`), `executionCondition` (the HTLA hash, as a `UInt256`), `destination` (the ILP address of the payment's receiver, who may be able to present the SHA256 preimage of the `executionCondition`, as an `IA5String (size 1..1023)`), and `data` (to be copied to the next transfer in the payment chain, as an `OCTET STRING (size 0..32767)`), in that order (see appendix A).

4.3 Fulfill

A Fulfill packet is an OER sequence of two fields: `fulfillment`, a `UInt256` for the SHA256 preimage of the `executionCondition` from the transfer's Prepare packet, and `data` (to be passed back unaltered from receiver to sender), an `OCTET STRING (size 0..32767)`, in that order.

4.4 Reject

A Reject packet is an OER sequence of four fields: `code` (an `IA5String (size 3)` from the list of Interledger Error Codes, see appendix B), `triggeredBy` (`IA5String (size 1..1023)`, the ILP address of the agent who triggered this rejection), `message` (`Ia5String (size 0..8191)`, a human-readable error message), and `data` (to be passed back unaltered from receiver to sender, an `OCTET STRING (size 0..32767)`, in that order.

4.5 Timing Disputes

If Bob sends the Fulfill message before the `expiresAt` timestamp from the corresponding Prepare message, but Alice only receives that message after the timestamp, they would likely disagree on whether that HTLA was fulfilled on time. In case of such timing disputes, the party sending the Fulfill packet is always given the benefit of the doubt. Note

that this allows them to be owed the amount of the transfer, but they still rely on the other party's cooperation to actually net this debt against a transfer in the opposite direction, so it will not be a lucrative way for one agent to steal from a neighboring one.

4.6 Idempotency

In the communication channel between two neighbors, a unique identifier should be assigned to each transfer. Each transfer starts with one participant (the Preparer) sending a Prepare packet.

4.6.1 Using Message Acknowledgement

If the communication channel supports delivery acknowledgement for messages, then the Preparer will keep repeating the Prepare packet, with its unique transfer identifier, until an acknowledgement is received. If the other participant can present the fulfillment on time, they will send a Fulfill packet, and keep repeating it with that same transfer identifier, until an acknowledgement is received. If the other participant cannot present the fulfillment on time, they will send a Reject packet as soon as possible, and keep repeating it with the transfer identifier, until an acknowledgement is received.

4.6.2 Without Message Acknowledgment

If the communication channel between two neighbors does not support delivery acknowledgment, then the Preparer should repeat the Prepare packet until either a Fulfill or Reject packet for the same unique transfer identifier is received. When the Preparer stops repeating the Prepare packet, the Fulfiller or Rejecter can conclude that their response packet was delivered successfully.

5 Appendix A: asn1 definitions

```
InterledgerProtocol
DEFINITIONS
AUTOMATIC TAGS ::=
BEGIN

UInt8 ::= INTEGER (0..255)
UInt64 ::= INTEGER (0..18446744073709551615)
UInt256 ::= OCTET STRING (SIZE(32))
```

```
VarBytes ::= OCTET STRING

-- Readable names for special characters that may appear in ILP addresses
hyphen IA5String ::= "-"
period IA5String ::= "."
underscore IA5String ::= "_"
tilde IA5String ::= "~"

-- A standard ILP address
Address ::= IA5String
    (FROM
        ( hyphen
        | period
        | "0".."9"
        | "A".."Z"
        | underscore
        | "a".."z"
        | tilde )
    )
    (SIZE (1..1023))

-- ---------------------------------------------------------------------------

-- We are using ISO 8601 and not POSIX time, because ISO 8601 increases
-- monotonically and never "travels back in time" which could cause issues
-- with transfer expiries. It is also one of the most widely supported and mo
-- well-defined date formats as of 2017.
--
-- Our actual wire format leaves out any fixed/redundant characters, such as
-- hyphens, colons, the "T" separator, the decimal period and the "Z" timezon
-- indicator.
--
-- The wire format is four digits for the year, two digits for the month,
-- two digits for the day, two digits for the hour, two digits for the minute
-- two digits for the seconds and three digits for the milliseconds.
--
-- I.e. the wire format is: 'YYYYMMDDHHmmSSfff'
--
-- All times MUST be expressed in UTC time.

Timestamp ::= PrintableString (SIZE(17))
```

```
InterledgerPrepare ::= SEQUENCE {
    -- Local amount (changes at each hop)
    amount UInt64,

    -- Expiry date
    expiresAt Timestamp,

    -- Execution condition
    executionCondition UInt256,

    -- Destination ILP Address
    destination Address,

    -- Information for recipient (transport layer information)
    data OCTET STRING (SIZE (0..32767))
}

InterledgerFulfill ::= SEQUENCE {
    -- Execution condition fulfillment
    fulfillment UInt256,

    -- Information for sender (transport layer information)
    data OCTET STRING (SIZE (0..32767))
}

InterledgerReject ::= SEQUENCE {
    -- Standardized error code
    code IA5String (SIZE (3)),

    -- Participant that originally emitted the error
    triggeredBy Address,

    -- User-readable error message
    message UTF8String (SIZE (0..8191)),

    -- Machine-readable error data, dependent on code
    data OCTET STRING (SIZE (0..32767))
}

PACKET ::= CLASS {
```

```
    &typeId UInt8 UNIQUE,
    &Type
} WITH SYNTAX {&typeId &Type}

PacketSet PACKET ::= {
    {12 InterledgerPrepare} |
    {13 InterledgerFulfill} |
    {14 InterledgerReject}
}

InterledgerPacket ::= SEQUENCE {
    -- One byte type ID
    type PACKET.&typeId ({PacketSet}),
    -- Length-prefixed header
    data PACKET.&Type ({PacketSet}{@type})
}

-- F08: Amount Too Large
AmountTooLargeErrorData ::= SEQUENCE {
  -- Local amount received by the connector
  receivedAmount UInt64,

  -- Maximum amount (inclusive, denominated in same units as the receivedAmou
  maximumAmount UInt64
}
END
```

6 Appendix B: Interledger Error Codes

6.1 F__ - Final Error

Final errors indicate that the payment is invalid and should not be retried unless the details are changed.

6.1.1 F00: Bad Request

Generic sender error.

6.1.2 F01: Invalid Packet

The ILP packet was syntactically invalid.

6.1.3 F02: Unreachable

There was no way to forward the payment, because the destination ILP address was wrong or the connector does not have a route to the destination.

6.1.4 F03: Invalid Amount

The amount is invalid, for example it contains more digits of precision than are available on the destination ledger or the amount is greater than the total amount of the given asset in existence.

6.1.5 F04: Insufficient Destination Amount

The receiver deemed the amount insufficient, for example you tried to pay a $100 invoice with 10$.

6.1.6 F05: Wrong Condition

The receiver generated a different condition and cannot fulfill the payment.

6.1.7 F06: Unexpected Payment

The receiver was not expecting a payment like this (the data and destination address don't make sense in that combination, for example if the receiver does not understand the transport protocol used).

6.1.8 F07: Cannot Receive

The receiver (beneficiary) is unable to accept this payment due to a constraint. For example, the payment would put the receiver above its maximum account balance.

6.1.9 F08: Amount Too Large

The packet amount is higher than the maximum a connector is willing to forward. Senders MAY send another pakcet with a lower amount. Connectors that produce this error SHOULD encode the amount they received and their maximum in the data to help senders determine how much lower the packet amount should be. See `AmountTooLargeErrorData` in appendix A.

6.1.10 F99: Application Error

Reserved for application layer protocols. Applications MAY use names other than Application Error.

6.2 T__ - Temporary Error

Temporary errors indicate a failure on the part of the receiver or an intermediary system that is unexpected or likely to be resolved soon. Senders SHOULD retry the same payment again, possibly after a short delay.

6.2.1 T00: Internal Error

A generic unexpected exception. This usually indicates a bug or unhandled error case.

6.2.2 T01: Peer Unreachable

The connector has a route or partial route to the destination but was unable to reach the next connector. Try again later.

6.2.3 T02: Peer Busy

The next connector is rejecting requests due to overloading. If a connector gets this error, they SHOULD retry the payment through a different route or respond to the sender with a T03: Connector Busy error.

6.2.4 T03: Connector Busy

The connector is rejecting requests due to overloading. Try again later.

6.2.5 T04: Insufficient Liquidity

The connector would like to fulfill your request, but either the sender or a connector does not currently have sufficient balance or bandwidth. Try again later.

6.2.6 T05: Rate Limited

The sender is sending too many payments and is being rate-limited by a ledger or connector. If a connector gets this error because they are being rate-limited, they SHOULD retry the payment through a different route or respond to the sender with a T03: Connector Busy error.

6.2.7 T99: Application Error

Reserved for application layer protocols. Applications MAY use names other than Application Error.

6.3 R__ - Relative Error

Relative errors indicate that the payment did not have enough of a margin in terms of money or time. However, it is impossible to tell whether the sender did not provide enough error margin or the path suddenly became too slow or illiquid. The sender MAY retry the payment with a larger safety margin.

6.3.1 R00: Transfer Timed Out

The transfer timed out, meaning the next party in the chain did not respond. This could be because you set your timeout too low or because something look longer than it should. The sender MAY try again with a higher expiry, but they SHOULD NOT do this indefinitely or a malicious connector could cause them to tie up their money for an unreasonably long time.

6.3.2 R01: Insufficient Source Amount

The amount received by a connector in the path was too little to forward (zero or less). Either the sender did not send enough money or the exchange rate changed. The sender MAY try again with a higher amount, but they SHOULD NOT do this indefinitely or a malicious connector could steal money from them.

6.3.3 R02: Insufficient Timeout

The connector could not forward the payment, because the timeout was too low to subtract its safety margin. The sender MAY try again with a higher expiry, but they SHOULD NOT do this indefinitely or a malicious connector could cause them to tie up their money for an unreasonably long time.

6.3.4 R99: Application Error

Reserved for application layer protocols. Applications MAY use names other than Application Error.