Deep dive into Interledger: Understanding the Interledger ecosystem – Part 3 –

Lucian Trestioreanu, Cyril Cassagnes, and Radu State

Ripple UBRI @ Interdisciplinary Centre for Security, Reliability and Trust,
University of Luxembourg
29, Avenue JF Kennedy, 1855 Luxembourg, Luxembourg

Abstract. At the technical level, the goal of Interledger is to provide an architecture and a minimal set of protocols to enable interoperability for any value transfer system. The Interledger protocol is literally a protocol for Interledger payments. To understand how is it possible to achieve this goal, several aspects of the technology require a deeper analysis. For this reason, in our journey to become knowledgeable and active contributor we decided to create our own test-bed on our premises. By doing so, we notice that some aspects are well documented but we found that others might need more attention and clarification. Despite a large community effort, the task to keep information on a fast evolving software ecosystem is tedious and not always the priority for such a project. Therefore, the purpose of this series of documents is to guide, through several hands-on activities, community members who want to engage at different levels. The series of documents consolidate all the relevant information from generating a simple payment to ultimately create a test-bed with the Interledger protocol suite between Ripple and other distributed ledger technology.

Contents

List of Figures

List of Tables

1 What this document covers

In *Part 3*, we are going to discuss two other components of the Interledger protocol suite and of the infrastructure suite, namely the *Interledger protocol (ILP)* which is the core of the whole system and, respectively, the *Ledgers*. We are also going to discuss some examples along the way. For easier orientation, we kept the general chapter structure unmodified.

2 Who this document is for

No prerequisites regarding the Interledger ecosystem are expected from the reader. However, developers, computer science students or people used to deal with computer programming challenges should be able to reproduce our setup without struggle.

3 The Interledger ecosystem

3.1 The Interledger protocol suite

3.1.1 The Interledger Protocol

The *Interledger Protocol (ILP)*, currently at version 4, is the main protocol facilitating the Interledger money transfers. It provides a solution to route payments across disconnected ledgers while minimizing the sender and receiver's risk of losing funds. What makes it different from previous versions is that it is optimized for sending many low value packets:

"We talked about the idea of streaming payments, where if you make payments so efficient that you could pay for like a milliliter of beer or a second of video. That's the way we think about efficiency of payments."[1]

It is made for payment channels, which means faster and cheaper payments, while also accommodating any type of ledger.

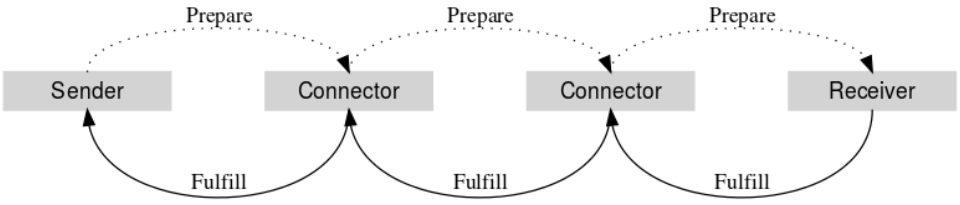

Fig. 1: ILP packet flow. [2]

ILPv4 involves *Hashed Time Lock Agreements (HTLA)* [3], and makes use of three packet types:

- *Prepare*, corresponding to request, with the following fields:
  - *destination* - ILP address,
  - *amount* - UInt64,
  - *condition* - UInt256,
  - *expiration* - timestamp,
  - *end-to-end (sender-receiver) data* - OCTET STRING.

  Example of an ilp-prepare packet:

```
{
amount= 69368000,
executionCondition= fHII9adb3JY3D5drSNSoquLTIUJJhNLMeiiADnW4li0=,
expiresAt= 2019-06-19T11:04:18.149Z,
destination= g.conn1.ilsp_clients.mduni.local.viby9ZjztwCVMtptFjaueqsdlIxWSUba
       y7Jo3BxJyGc.elrqFEKZEc8BMcZ4PDUiPEAF,
data= t6lmRiiFZecXhltYNsnyPYSgPld+Itmn+NefM5ytnFJiFDuMieyF9b2vB
       o2HPiNm34GpCBlU/HoGaCAsOQ==
}
```

- *Fulfill,* corresponding to response, and carrying the following fields:
  - *execution condition fulfillment* - UInt256,
    This is the proof that the receiver has been paid, so the fulfill packets are relayed back by the connectors from the receiver to the sender. It consists of a simple pre-image of a hash, and only the receiver can know this information.
  - *end-to-end (sender-receiver) data* - OCTET STRING.

  The components of the prepare and fulfill packets concerning HTLA are:
  - *amount, time* (expiration), and *condition* for Prepare, and
  - the *execution condition fulfillment (the hash)* for the Fulfill packet, which must be received before *expiration*. This implies that the machines involved in the process should be time-synchronized. This is not an absolute enforcement, but any time offsets will packet rejection chances.

- *Reject,* corresponding generally to error messages. They can be returned either by the receiver or the connectors in specific conditions and consist of:
  - *a standardized error code*,
  - *triggered by:* - ILP address;
    is the identifier of the participant that originally generated the error,
  - *user-readable error message* - UTF8String,
  - *machine readable error data* - OCTET STRING.

The connectors forward the *prepare* packets from the *sender* to the *receiver*, and relay back the response or the reject from the *receiver* to the *sender,* as shown in Figure 1. As such, ILP v4 uses a chaining of HTLAs to achieve an end-to-end transfer [3]. In ILP v4, HTLAs are mainly supported over Simple Payment Channels. Simple Payment Channels are generally supported by today's major blockchains like BTC, ETH, XRP,.. [4, 5].

Concerning Figure 1 and ILP v4: even if the original ILP packet is prepared by the *Sender* and addressed for the *Receiver* (end-to-end), the transfer from the *Sender* to the *Receiver* will be in fact a chaining of transfers between the directly connected (and trusted) peers. Each pair of directly connected peers generally uses a dedicated, separate Payment Channel to settle their obligations [3, 4, 5]. Other means are possible [5], but not really used or supported [4].

Example 1. We will further expand on *Part 1 - Example 1* and *Part 2 - Example 1*, using the Figure 2.

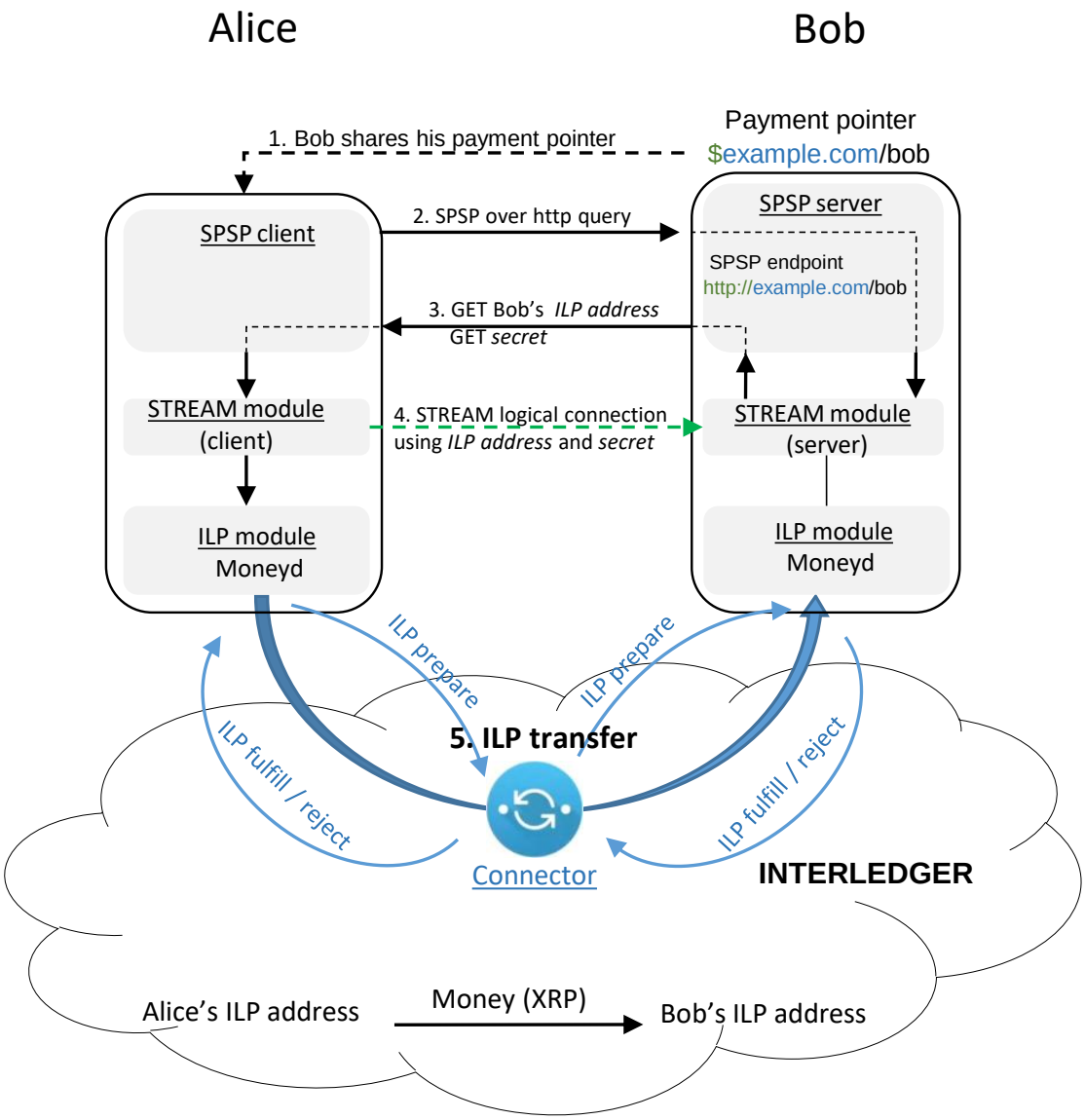

Fig. 2: Example 1: ILP.

- Alice's SPSP client:
  - resolves the *payment pointer* "*$example.com/bob*" to *https://example.com/bob*
  - connects over HTTP to Bob's SPSP server at address *https://example.com/bob* (2)
  - queries the SPSP server for Bob's *ILP address* and a *unique secret* (2). The SPSP server forwards the request to the STREAM server module and fetches the answer
- Bob's SPSP server sends Bob's *ILP address* and the *secret* to Alice's SPSP client (3)
- Alice's SPSP client passes the credentials to the STREAM client module which initiates a logical STREAM connection over ILP, using the ILP module, in our case, Moneyd (4)

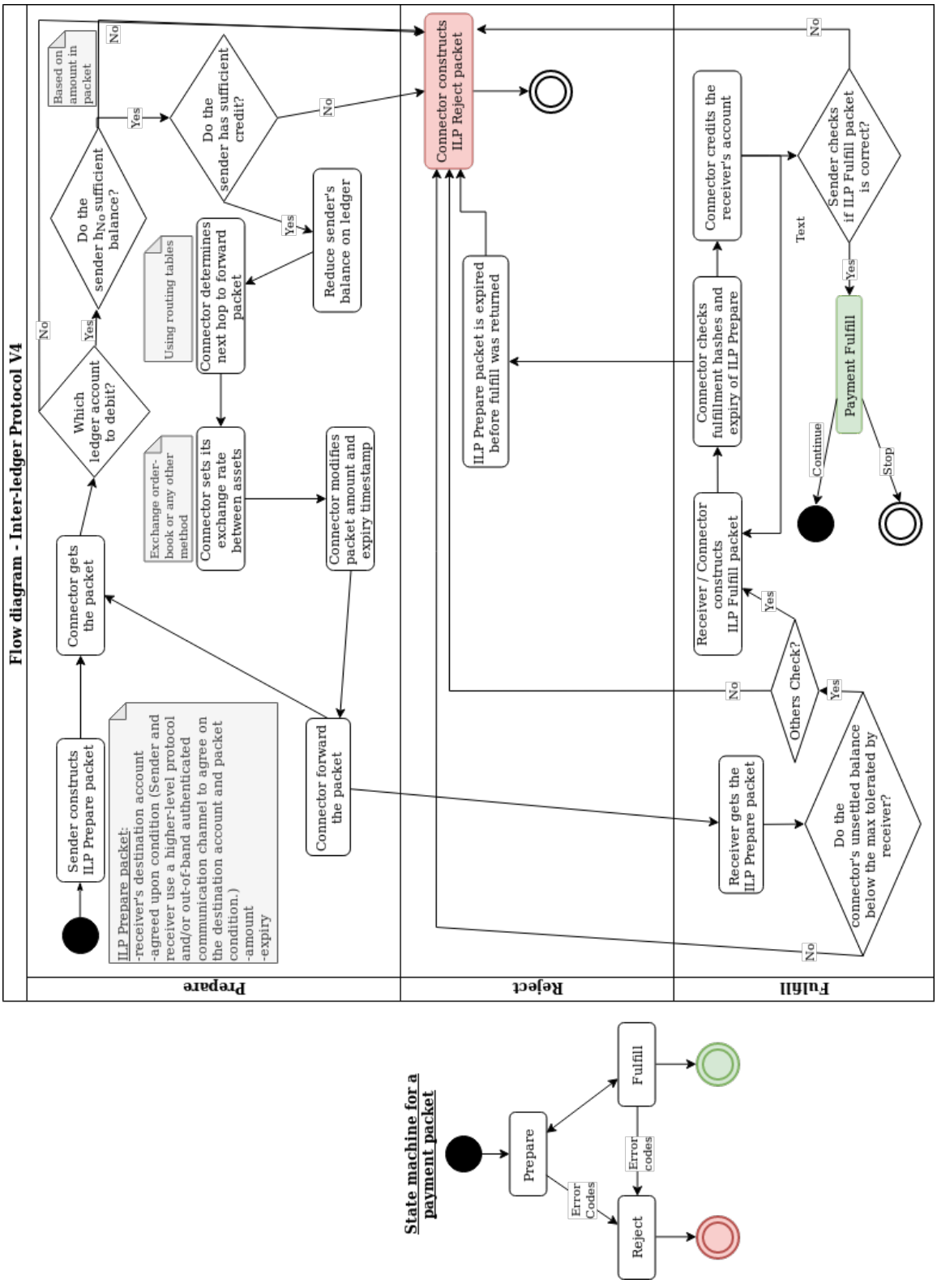

Fig. 3: ILPv4 flow diagram.

- The ILP module, Moneyd, sends the ILP packets corresponding to the STREAM virtual connection towards its upstream parent connector, which further routes them to its child, Bob's Moneyd module (5).

The STREAM module is able to break the payment into multiple packets, which would be sent over ILP using *prepare-fulfill-error* packets. The STREAM module at the *receiver's* end will finally reassemble the payment.

Wrapping-up ILP, Figure 3 presents the finite state machine diagram of an ILP packet and the protocol flow chart.

4 The ledgers

4.1 The Ripple ledger

The Rippled XRP ledger is made up of two types of servers: *"trackers"* (or *stock servers*) and *"validators"*. They run the same piece of software [6], just with a different configuration. Although they can answer user queries, validators should ideally just process the transactions they receive from the trackers.

'Ideally, validating nodes are clustered with at least two stock nodes, to prevent DoS attacks and to preserve availability while updating the stock nodes. This configuration enables the validating node to be cut off from the internet, except for messages to/from other trusted nodes in the cluster and SSH connections via a LAN connection. Using two stock nodes provides redundant communication to the validating node, which is useful in case one of the stock nodes crashes or goes offline. However, this means a validating node has 3x the cost, 3x the monitoring, and 3x the time commitment of a stock node. Production validating nodes should have at least 32 GB of memory as well as a 50 GB+ solid state drive. I encourage operator to refrain from making API calls (monitoring excepted) on validating nodes'. [7]

Validators participate in the consensus process and vote on fees and amendments. Trackers are meant to be placed in-between validators and the rest of the network, pick-up traffic and forward it to the validators. They work as relays, protecting the validators. They also can hold the full history of the ledger and answer queries about old ledgers. On the other hand, the validators can work with minimal stored history.

Below is the procedure to build and cold-start an independent local validators cluster. One aspect to keep in mind is that *'there is no rippled setting that defines which network it uses. Instead, it uses the consensus of validators it trusts to know which ledger to accept as the truth. When different consensus groups of rippled instances only trust other members of the same group, each group continues as a parallel network. Even if malicious or misbehaving computers connect to both networks, the consensus process overrides the confusion as long as the members of each network are not configured to trust members of another network in excess of their quorum settings.'* [8]

4.1.1 Preparation

To build a parallel Rippled servers (validators, trackers) network, which, in its entirety is also called the *"Rippled Ledger"*, the minimal required hardware resources[1] need to be planned in

[1] https://developers.ripple.com/system-requirements.html, accessed June 2019

advance. Depending on the available resources, each Rippled server can be deployed on a different physical machine or not. However, for high traffic use-cases, in order to streamline I/O, each server could have its own physical SSD.

To install the pre-packaged Rippled server, the instructions on the Ripple developer portal should be followed[2]. Then, another guide is disseminated on the developer portal to install Rippled from the source code[3]. After that, the following steps must be followed:

- Build the validators keys and tokens, using the published documentation[4] and code[5].
  - Create keys:

```
~/validator-keys-tool/build/gcc.debug$ ./validator-keys create_keys
```

  - Create tokens:

```
~/validator-keys-tool/build/gcc.debug$ ./validator-keys create_token --keyfile
    /home/user/.ripple/validator-keys.json
```

- Add generated [*validator_token*] to 'rippled.cfg'.
- Add generated [*validators*] public keys to 'validators.txt'. Comment the rest of 'validators.txt'.
- In 'rippled.cfg', add the peer validators' IPs in the field [ips_fixed] in the form of IP:port (51235).
- Check that 'validator.txt' file name is the same with the name referenced by 'rippled.cfg'.
- Configure clustering as per the Ripple documentation[6], using the validation_create[7] method:

```
~/rippled/ccabuild$ ./rippled --conf /home/user/rippled/cfg/rippled-example.cfg
    validation_create
```

4.1.2 Start up

In the case when Docker images have been used, after creating the Docker image of the Rippled server, this can be loaded and started on each physical/virtual server machine with the following:

```
- 'sudo docker load -i /path/to/your_docker_image.tar'
- 'sudo docker images' - to check the image name
- 'sudo docker run -ti -u root --network host --name <container_name> <image_name>'
- 'sudo docker exec -ti -u root <container_name> bash'. To open a second terminal
    to the container, just run the command again into a fresh terminal window.
```

With the docker images loaded on each Rippled server machine, the actual Rippled validators servers network can be cold-started as follows:

- Start the first Rippled server with 'quorum 1' and wait a few minutes for it to stabilize:

[2] https://developers.ripple.com/install-rippled.html, accessed June 2019
[3] https://developers.ripple.com/build-run-rippled-ubuntu.html, accessed June 2019
[4] https://developers.ripple.com/run-rippled-as-a-validator.html#enable-validation-on-your-rippled-server, accessed June 2019
[5] https://github.com/ripple/validator-keys-tool, accessed June 2019
[6] https://developers.ripple.com/cluster-rippled-servers.html, accessed June 2019
[7] https://developers.ripple.com/validation_create.html, accessed June 2019

```
./rippled --conf /home/user/rippled/cfg/rippled-example.cfg --quorum 1
```

- Start the remaining servers with the same command, waiting for each to stabilize, first.
- Restart the servers in the same order, waiting a few minutes for each to stabilize before starting the next, with 'quorum 2':

```
./rippled --conf /home/user/rippled/cfg/rippled-example.cfg --quorum 2
```

  In a new terminal window handling the Docker container, use the "stop" command in Table 1 to gracefully stop the servers before restart.

In this minimal set-up though, if any of the servers is restarted, it will lose previously kept ledger history - even with full history enabled. This won't stop it working after restart, as validators do not need full history to work properly. To be able to access previous ledger history, tracking servers should be also set up. Data API[8] is a useful history tool which could also be set-up if desired, although setting it up on a private network seems not too obvious.

'Ripple API'[9] provides the means to interact with the server. For example, in Ripple, all the money are created in the beginning, and stored in an account with a hard-coded address, called the "Genesis account'. One can check the 'Genesis account' with:

```
./rippled --conf /home/user/rippled/cfg/rippled-example.cfg account_info
    rHb9CJAWyB4rj91VRWn96DkukG4bwdtyTh
```

Some other useful server commands are provided in Table 1. These can be entered from a separate terminal window handling the docker container.

Table 1: Useful Rippled server commands.

command	effect
wallet_propose	create a new wallet with random seed credentials (inactive until funded)
stop	gracefully stop the server
restart	restart the server
server_info	various easy-to-read info about the server
server_state	almost same info as above, but easier-to-process instead of easy-to-read
peers	info on peer validators: connected? ledger sequences available? ...

Immediately after creating and starting the validators cluster network (which form the XRP ledger), one can open a few accounts with the 'wallet_propose' command above, and fund them using for example the following simple procedure. Regarding the wallets, they can be sometimes classified as 'hot' or 'cold' wallets. The difference is that 'hot' wallets are connected to the internet, while 'cold' wallets are not. 'Hot' wallets provide the advantage of quick access but lower security, like anything connected to the internet. 'Cold' wallets are slower to access (need to connect) but more secure due to generally not being online. It is generally recommended to hold only the amounts necessary for daily operation in the 'hot' wallet, while the bulk of the money would be kept offline.

[8] https://developers.ripple.com/data-api.html, accessed June 2019
[9] https://developers.ripple.com/rippleapi-reference.html, accessed June 2019

- Install Ripple-API for javascript[10].
- Place the two example scripts in the app folder: *'/home/user/ripple_api/get-account-info.js'*.
- Run them with *'./node_modules/.bin/babel-node get-account-info.js'*. The code should run on one of the Ripple servers.

Example script - get account info[11]:

```
//GET ACCOUNT INFO
'use strict';
const RippleAPI = require('ripple-lib').RippleAPI;

const api = new RippleAPI({
  server: 'ws://localhost:6006'
});
api.connect().then(() => {
  /* begin custom code ------------------------------------ */
  const testAddress = 'rHb9CJAWyB4rj91VRWn96DkukG4bwdtyTh';

  console.log('getting account info for', testAddress);
  return api.getAccountInfo(testAddress);

}).then(info => {
  console.log(info);
  console.log('getAccountInfo done');

  /* end custom code -------------------------------------- */
}).then(() => {
  return api.disconnect();
}).then(() => {
  console.log('done and disconnected.');
}).catch(console.error);
```

Example script - fund an account[11]:

```
 //Account funding
 const RippleAPI = require('ripple-lib').RippleAPI

// SENDER - ADDRESS 1
const ADDRESS_1 = "rHb9CJAWyB4rj91VRWn96DkukG4bwdtyTh"
const SECRET_1 = "snoPBrXtMeMyMHUVTgbuqAfg1SUTb"

// RECEIVER - ADDRESS 2
const ADDRESS_2 = "rMqUT7uGs6Sz1m9vFr7o85XJ3WDAvgzWmj"

const instructions = {maxLedgerVersionOffset: 5}
const currency = 'XRP'
const amount = '20000000' // this is not 'drops' but XRP

const payment = {
  source: {
    address: ADDRESS_1,
    maxAmount: {
```

[10] https://developers.ripple.com/get-started-with-rippleapi-for-javascript.html, accessed June 2019
[11] https://xrpl.org/rippleapi-reference.html, accessed July 2019

```
      value: amount,
      currency: currency
    }
  },
  destination: {
    address: ADDRESS_2,
    amount: {
      value: amount,
      currency: currency
    }
  }
}

const api = new RippleAPI({
  //server: 'wss://s1.ripple.com'                 //MAINNET
  //server: 'wss://s.altnet.rippletest.net:51233' // TESTNET
  server: 'ws://localhost:6006'                   // Localhost
})

api.connect().then(() => {
  console.log('Connected...')
  api.preparePayment(ADDRESS_1, payment, instructions).then(prepared => {
    const {signedTransaction, id} = api.sign(prepared.txJSON, SECRET_1)
    console.log(id)
    api.submit(signedTransaction).then(result => {
      console.log(JSON.stringify(result, null, 2))
      api.disconnect()
    })
  })
}).catch(console.error)
```

The known amendments seem not to be automatically enabled after cold starting a private network. In order to force them, we added the *[features]* stanza in each validator's config file. Otherwise, the validators would apparently work, but, when trying to open for example a paychan, would throw the error "logic not enabled' - because the paychan amendment is not enabled. According to documentation[12], for an amendment to become enabled, it needs the support of 80% of validators' votes for two weeks. If it loses this support, the amendment is temporarily disabled, and it can be re-enabled after it re-gains this support.

[features]
PayChan
Escrow
CryptoConditions
fix1528
.....

Below is an example config file for a private network cluster of 3 validators. The file is located in *'home/user/rippled/cfg'*. We used a docker container with a compiled version of Rippled.

[12] https://developers.ripple.com/amendments.html, accessed June 2019

```
[server]
port_rpc_admin_local
port_peer
port_ws_admin_local
port_ws_public
port_public

[port_rpc_admin_local]
port = 5005
ip = 127.0.0.1
admin = 127.0.0.1
protocol = http

[port_peer] //talk to other validators
port = 51235
ip = 0.0.0.0
protocol = peer

[port_ws_admin_local]
port = 6006
ip = 127.0.0.1
admin = 127.0.0.1
protocol = ws

[port_ws_public]
port = 6005
ip = 127.0.0.1
protocol = wss

[port_public] //connectors, moneyd, switch API will connect here
ip = 0.0.0.0
port = 51233
protocol = ws

[node_size] //required for full history
huge

# This is primary persistent datastore for Rippled. This includes transaction
# metadata, account states, and ledger headers. Helpful information can be
# found here: https://ripple.com/wiki/NodeBackEnd
# delete old ledgers while maintaining at least 2000. Do not require an
# external administrative command to initiate deletion.
[node_db]  //NuDB type required for full history
type=NuDB
path=/var/lib/rippled/db/nudb
#open_files=2000 //these are not needed for NuDB
#filter_bits=12
#cache_mb=256
#file_size_mb=8
#file_size_mult=2
#online_delete=2000
#advisory_delete=0

[ledger_history] //although enabled, full history seems not to work
```

```
                //correctly for validators, will need trackers for this.
full

[database_path]
/var/lib/rippled/db

# This needs to be an absolute directory reference, not a relative one.
# Modify this value as required.
[debug_logfile]
/var/log/rippled/debug.log

[sntp_servers] // servers for time sync
time.windows.com
time.apple.com
time.nist.gov
pool.ntp.org

# File containing trusted validator keys or validator list publishers.
# Unless an absolute path is specified, it will be considered relative to the
# folder in which the rippled.cfg file is located.
[validators_file]
validators-example.txt

# Turn down default logging to save disk space in the long run.
# Valid values here are trace, debug, info, warning, error, and fatal
[rpc_startup]
{ "command": "log_level", "severity": "trace" } //verbose logging

# If ssl_verify is 1, certificates will be validated.
# To allow the use of self-signed certificates for development or internal use,
# set to ssl_verify to 0.
[ssl_verify]
0

[ips_fixed]
192.168.1.97 51235 //IPs and ports of the other 2 peer validators
192.168.1.132 51235

[peer_private]
1

[node_seed]
shEm9dGAs2aq6MMe9XsXYXKrPmqft

[cluster_nodes]
n9LPJFoTLxVbTtdWADZzPpCwACwC3aLAYGhFcNNR61fD9DTc2w5L ripdbg1
n9KUMms9ZrDgHU7rN9pRTRGMKEWy5Ghk3qj53aCPAbJRur2sTqwp ripdbg3

[validator-token]
eyJtYW5pZmVzdCI6IkpBQUFBQUZ4SWUwYkVlUVp1bGNsKzRadk44cGhXUWJNNWhlV3RKY0hN
YUVKcUpadWVRWm9jWE1oQXYvVWY3MmlaQ0VQZndPZTd0TjNaY0V1UnFDd2Q3U2JkU3hPTnJq
TXlsNWlka2N3UlFJaEFKc3IzL3g2U0RiRGprOHc0Mks2eU91M1FPbW4vNjVIeTM4bkxjbnJa
c1ROQWlBSnRlRTRpdjVqSjRJMytvS0VseEFjTmFUL3VoQnRlSVFyK29RdmVoemJESEFTUU53
RnpLN21kV3lUaTZoTWY4SUJTRUxmZHI1cjhuMFdIeE5BSGNHSXJURDV1N09BK3FKZWZLMzkw

```

```
Smx3aE5ydGVLL09LWS8rQldDUHo0ejQ4VXptaHd3PSIsInZhbGlkYXRpb25fc2VjcmV0X2tl
eSI6IkJGMTcyRjJBMzNGQTZDOTdBQ0JBODhBNTA0NThGQzZFRURENzBCNjEwMzdEMjcwNjgz
RTQ3MzRBNUY2OURGRkMifQ==

[features]
PayChan
Escrow
CryptoConditions
fix1528
DepositPreauth
FeeEscalation
fix1373
MultiSign
TickSize
fix1623
fix1515
TrustSetAuth
fix1513
fix1512
fix1571
Flow
fix1201
fix1523
fix1543
SortedDirectories
EnforceInvariants
fix1368
DepositAuth
fix1578
```

4.2 The ETH ledger. Connecting the XRP and ETH ledgers through 'Machinomy'

For the scope of this work, we will assimilate the ETH network to a black-box holding the ETH wallet accounts, executing commands and providing immediate response. For testing purposes, such a friendly 'black-box' can be 'Ganache'[13], previously called 'TestRPC'. After download, Ganache can be started directly:

```
cd /Downloads
./ganache-1.3.1-x86_64.AppImage
```

'Machinomy'[14] is used to connect the XRP and ETH ledgers. It achieves this by deploying a specific contract on the ETH ledger. One contract manages all the channels for Ether micropayments (all the sender-receiver pairs). Thus, Machinomy creates the settlement capability when ILP payment interacts with the ETH ledger.

'Machinomy is a micropayments SDK for Ethereum platform. State channels is a design pattern for instant blockchain transactions. It moves most of the transactions off-chain. As transactions do not touch the blockchain, fees and waiting times are eliminated, in a secure way.' [9]

[13] https://truffleframework.com/docs/ganache/quickstart, accessed June 2019, accessed June 2019
[14] https://machinomy.com/, accessed June 2019

Machinomy should be installed[15] on the same machine with the ETH provider, in this case, Ganache. After installing Machinomy, a contract can be deployed on the ETH network using the following:

```
cd machinomy/node_modules/@machinomy/contracts
yarn truffle migrate --reset
```

Checking back in Ganache after Machinomy contract deployment, you will notice that a small amount of ETH has been subtracted from the first account, and in the *Transactions* tab, the contract has been deployed.

After a Ganache restart, the *'–reset'* option has to be used because Ganache is not persistent. The contract will be deployed on the first Ganache account. The other accounts can be used by ETH client wallets.

After deploying the Machinomy contract on the ETH network, apps like Switch API can be used to exchange XRP and ETH back and forth. The plugins should be set to access Ganache using *http://ganache_IP:ganache_port*. A detailed explanation on Switch API will be provided in *Part 4*.

In *Part 4* we will discuss the Bilateral Transfer Protocol (BTP) which is a link protocol and a carrier for ILP, a trading app named Switch, and we are going to see the architecture of an entire, functional, private Ripple network which is currently the basis of our test-bed.

Acknowledgements

This work was done in the framework of the Ripple UBRI initiative.

[15] https://github.com/machinomy/machinomy, accessed June 2019

Acronyms

API Abstract Programming Interface. 7, 9, 15, 16

ILP Interledger Protocol. 3–5, 7, 14

SPSP Simple Payment Setup Protocol. 5

Glossary

Moneyd An ILP provider, allowing all applications on an end-user computer to use funds on the live ILP network. 5, 7

Switch API A SDK for cross-chain trading between BTC, ETH, DAI and XRP with Interledger Streaming. 15

XRP Ripple's digital payment asset which is used for Interledger payments. 4, 9, 14, 15

References

[1] Ripple. *Ripple InterLedger Protocol's role in realizing the Internet of Value [IoV]*, [Online] Accessed: June 11, 2019. https://bcfocus.com/news/ripple-interledger-protocols-role-in-realizing-the-internet-of-value-iov/19033/.

[2] Ripple. *Install Rippled*, [Online] Accessed: June 6, 2019. https://developers.ripple.com/install-rippled.html.

[3] D. Appelt, A. Hope-Bailie, M. de Jong, E. Schwartz, B. Sharafian, S. Thomas, and B. Way. *ILP v4: Version 4 of the Interledger protocol, April 2018*, [Online] Accessed: June 13, 2019. https://github.com/interledger/rfcs/blob/2052575084cdeeb94c0e9bd2e3c37960b732fa2d/whitepaper/interledger.pdf.

[4] Evan Schwartz. *Trustlines Explanation*, [Online] Accessed: June 13, 2019. https://forum.interledger.org/t/trustlines-explanation/358.

[5] Ripple. *Hashed-Timelock Agreements (HTLAs)*, [Online] Accessed: June 11, 2019. https://interledger.org/rfcs/0022-hashed-timelock-agreements/#simple-payment-channels.

[6] Ripple. *Install Rippled*, [Online] Accessed: April 10, 2019. https://developers.ripple.com/install-rippled.html.

[7] Rabbit. *The Interledger Protocol*, July. 2018. [Online] Accessed: April 10, 2019. https://xrpcommunity.blog/rippled/.

[8] Ripple. *Parallel Networks and Consensus*, [Online] Accessed: April 10, 2019. https://mduo13.github.io/ripple-dev-portal/tutorial-rippled-setup.html.

[9] Sergey Ukustov, Andrei Riaskov, Alexander Burtovoy, and Matthew Slipper. *Machinomy*, [Online] Accessed: April 10, 2019. https://machinomy.com/.