Design and Implementation of Distributed Ledger Interoperability Adapters

Marc Iseli (16622289329)

CAS in Blockchain

Faculty of Business, Economics and Informatics

01.12.2019

ABSTRACT

This paper focuses on providing three additional adapters for the Bifröst API. Bifröst enables interoperability between different distributed ledger technologies (DLT) by allowing users to store and retrieve arbitrary data. In a first step, three DLTs were selected: *(i)* Litecoin, *(ii)* Monero, and *(iii)* Ripple. The paper shows how the data will be stored in decentralized Peer-to-Peer (P2P) networks using distributed ledger Applications and one of the three different mechanisms to achieve interoperability. The implementation consists of an adapter, which can be added to the Bifröst project and includes the functions create, include data, sign, and broadcast raw transaction. Bifröst is implemented with Python and relies on SQLite as a database to store credentials such as addresses, private keys, username and password. Furthermore, a characterization of the selected DLTs with respect to the following parameters is presente: block size, block time, data size constraints, deployment type, consensus mechanisms, processing time, storage requirements and scalability.

1 INTRODUCTION

A distributed ledger is a decentralized database which exists on multiple locations. Since there is not only one centralized machine, it is much more difficult to attack such a system. One of this distributed ledger technologies is BC. A DLT is an append-only linked list, where each block contains a reference to the previous block (Zheng et al. 2017). The link to a previous block is a cryptographic hash of the block information, such as timestamp, transactions and associated data. Once the data is stored in a BC, it cannot be altered without going back to this block and recalculating the follows. Since the BC is decentralized and in most cases publicly writable and readable, the key question is how to determine the next reliable block. This problem was solved in 2008 by Satoshi Nakamoto (*Satoshi Nakamoto - Wikipedia* n.d.) with the introduction of Proof-of-Work (PoW) in Bitcoin. Since the release of Bitcoin, however, new BCs with different characteristics were developed and proposed. On the one hand, there is a public variety of BCs where everyone has the right to read and write. On the other hand there is the private assortment, which is permissioned and where the identities are known.

BCs and other DLTs have attracted a lot of media attention. More and more banks and other supply-chain industries are discovering that DLT could be a great advantage for their business and starting to build their own solutions tailored to their needs. However, each new BC or other DLT has its own implementation. Thus, not being able to communicate with other solutions.

1.1 Interoperability Technologies

Interoperability technologies are the solutions for the interaction between different DLTs. Mainly, they provide a platform with a selection of different DLTs that can interact with each other without involving a third party. There are three main approaches.

- **Notary scheme:**The notary is the third party that checks wheter an event occurs in one chain in order to verify in another chain. For example, Interledger varie checks if a cryptocurrency is transferred to the connectors (Nodes with multiple accounts in different cryptocurrencies) and sends afterward from the connector the amount other cryptocurrency to the receiver (*Interledger Architecture — Interledger* n.d.). Herdius is another example for notary scheme cross-chain technology (*Herdius* n.d.). The trust is asigned to the notary instead of other registrars.
- **Sidechains / relays:** A sidechain is an individual BC that works independently but is bound to a parent-BC (also called as mainchain). With the use of a sidechain, there are advantages such as communication between different BCs, sidechains could work as test network, if the mainchain is hacked the sidechain will still work or vice versa, sidechains can make their own rules and can be changed and the mainchain users do not see what is happening in the sidechain (anonymity). The main drawbacks of sidechains are the security issues and the maintenance of consensus with appropriate evidence (*Liegt die Zukunft der Blockchain in der Sidechain? — Blockchainwelt* n.d.).
- **Hash-locking:** If a transaction is perormed over two different BCs, the operation must also reach atomicity. With hash-locking it is possible to make atomic swaps between two or more parties without the need for an intermediary. Figure 1 shows a transaction between A from one BC to B in another one. First A generates a hash of a secret s and sends it to B. In the next step, A locks his/her assets and put them into a contract. As soon as B sees that A locked his/her assets, B also locks his/her assets and put them to the same contract with the same number. If A gives the secret before the time runs out, the assets go form A to B, otherwise they go back to A. If B gets the correct secret before the time runs out, the assets go to A, otherwise the assets return to B.

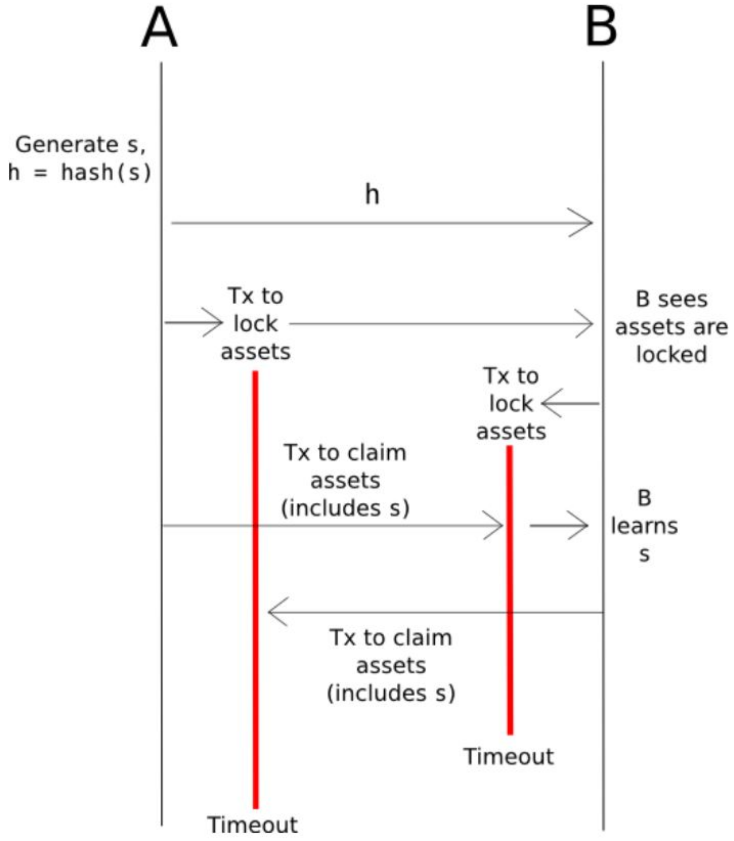


Fig. 1: Atomic Swap between two parties(*Chain Interoperability* n.d.) .

2 BIFRÖST

Bifröst is a tool that enables distributed ledger interoperability by relying on the notary scheme, providing an Application Programming Interface (API) to store and retrieve data from multiple DLTs (Hegnauer 2019). The API handles the different DLTs so that the users do not have to know exactly the API is doing. For each BC the same functions are implemented in order that users can switch between them. This means that the interactions with the DLTs are transparent to the user, abstracting their implementation differences. Figure 2 illustrates the architecture of the solution where the adapters contain the DLT specific functions.

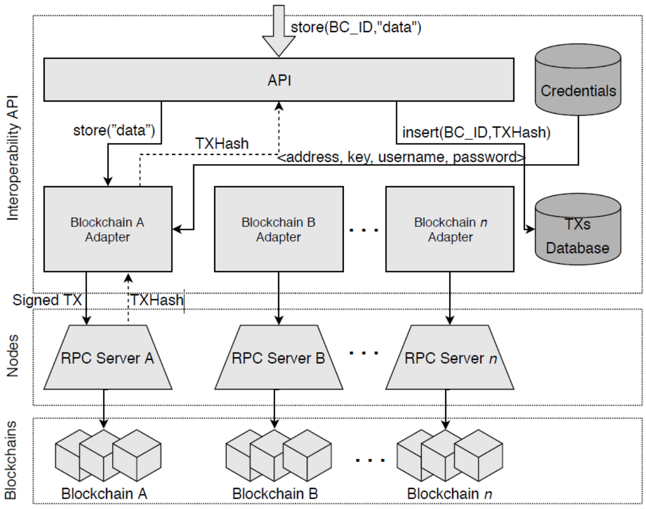


Fig. 2: Bifröst Concept (Hegnauer 2019).

In the API-area, there are public functions that a user can call. From this zone the transactions (tx) are stored in a SQL-Lite database. However, the entire tx is not saved, but only the hash with which the tx can be searched for and restored in the ledger structure. As mentioned above, the main functions for each DLT are stored in the adapters. These adapters also retrieves the access information from the local database. Private key, public key, user and password are stored there. Server name and port must always remain the same or be adjusted directly in the adapters.

3 SELECTED DISTRIBUTED LEDGERS

The following discusses and explains the properties and consensus of DLTs that should be added to the Bifröst API.

3.1 Litecoin (LTC)

This BC is a fork from Bitcoin and is still similar to it. Litecoin also works with the consensus Proof of Work (PoW) in a public, decentralized P2P-network. To verify legality and to avoid double spending, everyone can set up nodes, which calculate the hash problem. These nodes are called miner. New blocks are accepted once the miners have created a satisfactory block with associated hash. Litecoin has deployed a consensus that works on PCs because it should run on as many common machines as possible. For calculation, the consensus can run on Central Processing Units (CPU) and also on Graphical Processing Units (GPU). The aim of the LiteCoin BC is to make transactions faster and cheaper than Bitcoin. To achieve this, it creates a new Block every 2.5 min on average, which is four times faster than Bitcoin. There is a limit of 84 million Litecoins (*Lite Coin White Paper* n.d.) because of the halving the mining reward every four years. The BC will reach this limit around the year 2142.

3.1.1 Consensus To make transactions cheaper, Litecoin use scrypt as its PoW, which consumes less energy than other consensuses to calculate the same difficulty. Scrypt is a password-based key derivation function published by Colin Percival in 2010. To make attacks on the hash operations (parallelization of the attack) more difficult, Scrypt requests more memory with the function ROMix, which is under the oracle of a hash function sequential memory-hard, so that it needs a lot more energy for a brute-force attack than in other alogrithms (*scrypt.dvi* n.d.).

3.2 Monero (XMR)

Monero is a public BC whose size is 63.75 GB (25.11.2019). Each miner must download the entire blockchain or join a mining pool. Monero is focused on providing privacy and anonymity for their users. This is achieved by an obfuscated public ledger. Unlike other BCs, account balances, transactions and payments were hidden. Another special featur of Monero are the two public private key pairs. The first key pair “Private View Key ”has only the right to read the incoming transactions of the corresponding address. The second key pair ”Private Spend key ”allows reading and writing (*Private Keys in Monero — Monero Documentation* n.d.). Each new Block is based on the last 720 blocks and is generated every 2 min. The difficulty of mining changes every 6 months. Tor shall be integrated into Monero wallets to hide IP-Addresses.

3.2.1 Consensus Monero’s BC (*Monero – Wikipedia* n.d.) is dependent on the CryptoNight PoW algorithm, which in turn comes from the CryptoNote protocol. The consensus mechanism was developed to allow generic computers to become miners. This is achieved by implementing an algorithm that is inefficient when executed on Application Specific Circuits (ASIC) (*CryptoNight — Monero Documentation* n.d.). Each transaction is digitally signed with an one-time key and a key image. The digital signature is based on a public private key algorithm like ECDSA (*Elliptic Curve DSA – Wikipedia* n.d.). A key image is a cryptographically secure key that allows the Monero network to verify that no output transaction is spent multiple times. Cryptonight complicates memory access with three steps:

1. Initialize large area of memory with pseudo-random data. This memory is known as the scratchpad.
2. Perform numerous read/write operations at pseudo-random (but deterministic) addresses on the scratchpad.
3. Hash the entire scratchpad to produce the resulting value.

(*CryptoNight — Monero Documentation* n.d.)

Further the consensus use Ring Confidential Transactions (RingCT), which allow users to achieve unconditional unlikability (*whitepaper.pdf* n.d.).

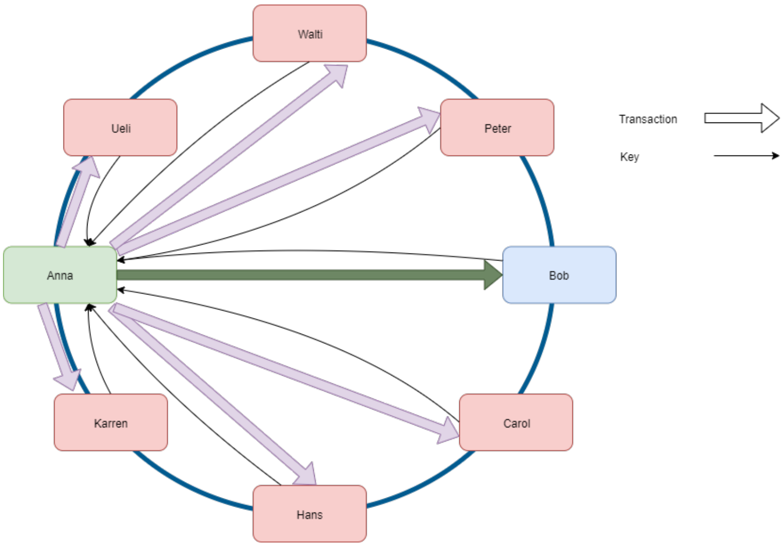


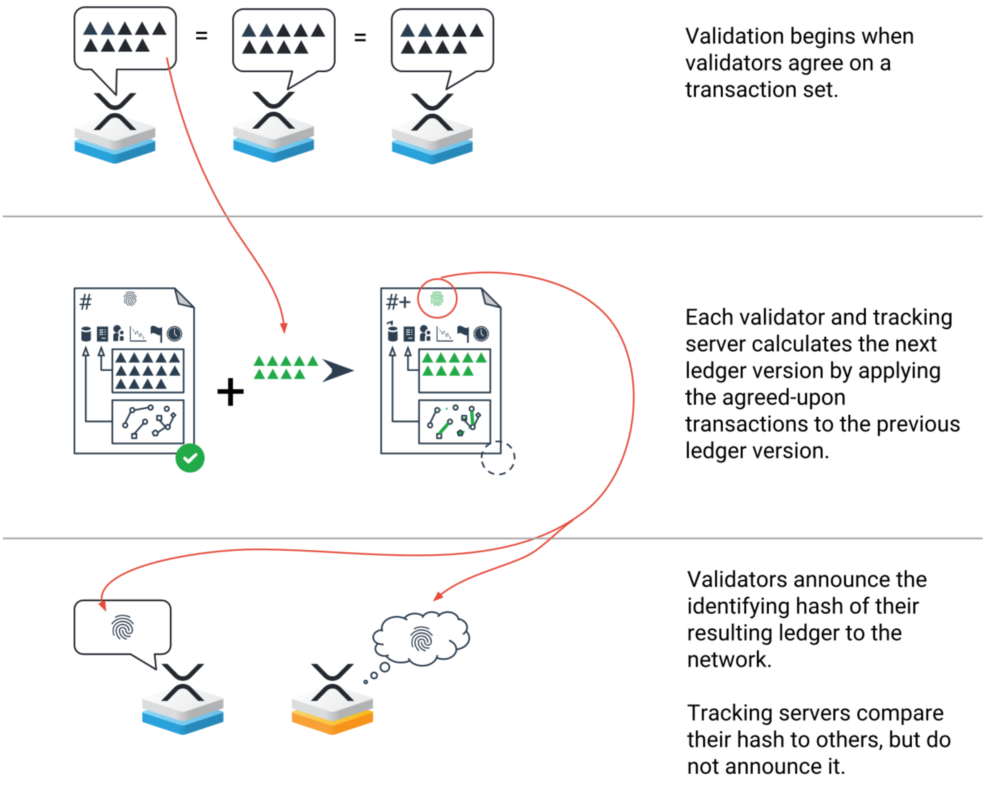
Fig. 3: RingCT Example.

It is not the conventional RingCt, but an improved version called “A Multi-layered Linkable Spontaneous Anonymous Group signature”. Figure 3 illustrates how a transaction is performed. In this example Anna would like to send Bob some XMR. First, Anna receives the public keys from all members in her with a triangular distribution method, specified ring and her account keys. The account keys contain all sent and received transactions. Next, she builds multiple outputs with this keys, so that only the receiver knows which output is the right one. He can verify this by using his private key and the key image and the public key from Anna. Since noboy knows which signers belongs to which account, for others it is impossible to trace tx (*moneropedia.entries.ringsignatures — Moneropedia — Monero - secure, private, untraceable* n.d.).

3.3 Ripple (XRP)

XRP is a digital asset designed to represent the money transferred over the Ripple Network (RippleNet). Ripple’s main goal is to connect the different currencies through a cheap and real-time connection. Ripple is not a BC, but a real-time gross settlement system(RTGS). XRapid is an on-demand-liquidity solution, that is used to convert the sender’s currency to XRP and at the receiver’s end to convert XRP back to the receiver’s currency after transmission (*Ripple Coin (XRP) einfach erklärt - Kryptowährungen* n.d.).

3.3.1 Consensus XRP and xRapid rely on the Ripple Protocol Consensus Algorithm (RPCA) used in the XRP ledger. In the XRP ledger, Unique Node List (UNL) is maintained by each individual node. These contain the trustworthy participants who check the tx of these nodes. The consensus needs neither mining nor network difficulty or hashrate, but uses trust-based validation. If 80% of the nodes agree with the transaction, it is verified. The tx are stored in the P2P XRP ledger network which is accessible to everyone. Every few seconds there is a new ledger version which is validated through the trusted nodes (*What is Ripple? — Bitcoin Magazine* n.d.). Figure 4 shows a validation cycle. Each ledger is recognized by two identifiers. The first is an ascending number (sequence number) and the second a hash of the content.



Validation begins when validators agree on a transaction set.

Each validator and tracking server calculates the next ledger version by applying the agreed-upon transactions to the previous ledger version.

Validators announce the identifying hash of their resulting ledger to the network.

Tracking servers compare their hash to others, but do not announce it.

Fig. 4: XPR Ledger Sequence (*Consensus - XRP Ledger Dev Portal* n.d.).

3.4 Comparisons

Each DLT has its own advantages and disadvantages and is the preferred option in its special field. Not always speed is required, but also security or large amounts of data. Table 1 shows some characteristics that illustrate the differences. While Litecoin is four times faster than Bitcoin, it is still beaten by Monero. The tx is already confirmed after 1200 seconds. Another intersting point is that the distributed ledger variant, which uses trust-validation, has the worst througput but the confirmation is the fastest.

Blockchain	Type	Consensus	Finality	Blocktime (s)	Confirmation After	Tps
Litecoin	Public	PoW	No	150	12 blocks	1800
Monero	Public	PoW	No	120	10 blocks	1700
Ripple	Private	RPCA	No	-	4-5 s	1500

Table 1. Blockchain characteristics

The next table 2 shows how much data can be transferred per tx. Litecoin has the same size like Bitcoin. Monero has two options, while the option with 8 Bytes is not covered in this document.

Blockchain	Maximum String Size
Litecoin	40 Bytes (*Developers Battle Over Bitcoin Block Chain* n.d.)
Monero	64 Bit or 8 Bytes (*moneropedia.entries.ringsignatures — Moneropedia — Monero - secure, private, untraceable* n.d.)
Ripple	1 kByte (*Transaction Common Fields - XRP Ledger Dev Portal* n.d.)

Table 2. Data size

4 IMPLEMENTATIONS

The implementation for each of the selected DLTs described in table 3 is different because there does not exist a standard. Figure 3 illustrates the differences between the three DLTs used. There are two main different for node type: Full node means that the Node that wants to access the specific network has to download the complete tx history. A second type are lightweight nodes which only download the header of all blocks or nothing. Lightweight nodes verify transactions using a method called Simplified Payment Verification (SPV).

Blockchain	Library Name	Node Type	Connection	Network
Litecoin	python-bitcoinrpc	Full Node	RPC	Public Testnet
Monero	Monero-python	Full Node	RPC	Public Testnet
Ripple	python-ripple-lib	Lightweight Node	RPC	Public Testnet

Table 3. Blockchain Adapter Implementation

Each adapter is connected to the API and must have certain functions. The following implementations can be called using the API:

- **store:** Listing 1 shows the store call in the api.py file. The Input parameters are the text to be stored and the chosen DLT. Immediately afterwards, Listing 2 illustrates how the adapter is called and which functions were required for the corresponding call.

```
def store(text, blockchain):
  """Store a text in a specific Blockchain:
  Args:
  Blockchain to use, e.g. Ethereum.
  Returns:
  string: The transaction hash.
  """
  adapter = Adapter[blockchain]
  transaction_hash = adapter.store(text)
  print(transaction_hash)
  return transaction_hash
```

Listing 1: Function “store” in api.py

```
@classmethod
def store(cls, text):
  # start = int(round(time.time() * 1000))  # Milliseconds
  transaction = cls.create_transaction(text)
  signed_transaction = cls.sign_transaction(transaction)
  transaction_hash = cls.send_raw_transaction(signed_transaction)
  if(WAIT_FOR_CONFIRMATION):
    if(cls.confirmation_check(transaction_hash)):
      cls.add_transaction_to_database(transaction_hash)
      return transaction_hash
    else:
      raise LookupError(
        'Transaction not confirmed and not added to DB')
  else:
    cls.add_transaction_to_database(transaction_hash)
    # cls.save_measurement(int(round(time.time() * 1000)) - start)
    return transaction_hash
```

Listing 2: Function “store” in adapter.py

- **retrieve:** The next Listing 3 illustrate how to perform a retrieve request. The only parameter it needs is the tx hash. With this hash, the function calls the other retrieve function in the adapter.py file (Listing 4) .

```
def retrieve(transaction_hash):
  """Get the text stored on the Blockchain:
  Args:
  Transaction hash
  Returns:
  string: The text belonging to the transactiont.
  """
  blockchain = database.find_blockchain(transaction_hash)
  adapter = Adapter[blockchain]
  text = adapter.retrieve(transaction_hash)
  print(text)
  return text
```

Listing 3: Function “retrieve” in api.py

```
@classmethod
def retrieve(cls, transaction_hash):
  """Get the transaction data from a tx hash:
  Args:
  param1 (str): The transaction hash.
  Returns:
  string: The transaction data as text.
  """
  transaction = cls.get_transaction(transaction_hash)
  data = cls.extract_data(transaction)
  return cls.to_text(data)
```

Listing 4: Function “retrieve” in adapter.py

- **migrate:** The migrate function in Listing 5 simply retrieves the stored data first and saves it in the asked Blockchain. It does not need another function in the adapter.py file.

```
def migrate(transaction_hash, blockchain):
  """Copy a value from a transaction to another Blockchain:
  Args:
  Transaction hash
  Returns:
  string: The transaction hash from the new transaction.
  """
  value = retrieve(transaction_hash)
  new_hash = store(value, blockchain)
  return new_hash
```

Listing 5: Function "migrate" in api.py

Each of the following adapters has to implement all the functions called from de adapter.py file.

4.1 Litecoin

Since the two cryptocurrencies are very similar, the Litecoin adapter is the same as the Bitcoin adapter which is already implemented (*Eder John Scheid / bifrost · GitLab* n.d.). The difference lies in the node in the background. Listing 6 shows the imported framework used for the Litecoin adapter.

```
from binascii import hexlify, unhexlify
from bitcoinrpc.authproxy import AuthServiceProxy
from db.config import ENCODING
from blockchain import Blockchain
from adapters.adapter import Adapter
import db.database as database
import collections

class LTCAdapter(Adapter):
  chain = Blockchain.LITECOIN
  credentials = database.find_credentials(Blockchain.LITECOIN)
  address = credentials['address']
  key = credentials['key']
  rpcuser = credentials['user']
  rpcpassword = credentials['password']
  endpoint_uri = f"http://{rpcuser}:{rpcpassword}@localhost:18332/"
  client = AuthServiceProxy(endpoint_uri)
  ...
```

Listing 6: Litecoin imports and constructor in ltc_adapter.py

Listing 7 illustrates the functions which are used to store data in the Litecoin BC.

```
@classmethod
def create_transaction(cls, text):
  input_transaction_hash = database.find_latest_transaction(Blockchain.LITECOIN)
  inputs = [{'txid': input_transaction_hash, 'vout': 0}]
  data_hex = cls.to_hex(text)
  output = cls.create_transaction_output(data_hex, input_transaction_hash)
  # Necessary so that the address is the first output of the TX
  output = collections.OrderedDict(sorted(output.items()))
  transaction_hex = cls.client.createrawtransaction(inputs, output)
  return transaction_hex

@classmethod
def create_transaction_output(cls, data_hex, input_transaction_hash):
  balance = cls.extract_balance(input_transaction_hash)
  relay_fee = cls.client.getnetworkinfo()['relayfee']
  change = balance - relay_fee
  return {cls.address: change, 'data': data_hex}

@classmethod
def extract_balance(cls, transaction_hash):
  transaction = cls.get_transaction(transaction_hash)
  output = transaction['vout'][0]['value']
  return output

@staticmethod
def to_hex(text):
  data = bytes(text, ENCODING)
  data_hex = hexlify(data)
  return data_hex.decode(ENCODING)

@classmethod
def sign_transaction(cls, transaction_hex):
  parent_outputs = []
  signed = cls.client.signrawtransaction(
    transaction_hex,
    parent_outputs,
    [cls.key]
  )
  assert signed['complete']
  return signed['hex']

@classmethod
  def send_raw_transaction(cls, transaction_hex):
  transaction_hash = cls.client.sendrawtransaction(transaction_hex)
  return transaction_hash

@staticmethod
  def add_transaction_to_database(transaction_hash):
  database.add_transaction(transaction_hash, Blockchain.LITECOIN)
```

Listing 7: Litecoin store functions in ltc_adapter.py

The next section of code (Listing 8) contains the functions needed for to retrieve text from the BC.

```
@classmethod
  def get_transaction(cls, transaction_hash):
  transaction_hex = cls.client.getrawtransaction(transaction_hash)
  return cls.client.decoderawtransaction(transaction_hex)

@classmethod
  def extract_data(cls, transaction):
  output = transaction['vout'][1]
  asm = output['scriptPubKey']['asm']
  _, data = asm.split()
  return data

@staticmethod
  def to_text(data_hex):
  data = unhexlify(data_hex)
  return data.decode(ENCODING)
```

Listing 8: Litecoin retrieve functions in ltc_adapter.py

4.2 Monero (XMR)

Monero wallets can have multiple accounts and each account can have multiple addresses. For each tx a new signature key is created. During the installation (1.2) there is the option to download the test blockchain or the productive one.

```
import db.database as database
from adapters.adapter import Adapter
from blockchain import Blockchain
from decimal import Decimal
from monero.wallet import Wallet
from monero.address import Address
from monero.seed import Seed
from monero.numbers import PaymentID
from monero.backends.jsonrpc import JSONRPCWallet
from monero.daemon import Daemon
from monero.backends.jsonrpc import JSONRPCDaemon
```

```

class MoneroAdapter(Adapter):
  chain = Blockchain.MONERO
  credentials = database.find_credentials(Blockchain.MONERO)
  wallet = Wallet(JSONRPCWallet(protocol=”http”, host=”127.0.0.1”, port=28088))
  # wallet = Wallet(JSONRPCWallet(protocol=”http”, host=”127.0.0.1”, port=18088, path=”/json_rpc”, user=”,password=”, timeout=30, verify_ssl_certs=True))
  daemon = Daemon(JSONRPCDaemon(port=28081))
  address = credentials[’address’]
  key = credentials[’key’]
  ...
```

Listing 9: Monero imports and constructor in monero_adapter.py

Listing 10 represents the functions required to store data with the API call. The text can only be stored if it does not have more than 64Bits what is equal to 16 ASCII chars.

```
@classmethod
  def create_transaction(cls, text):
  p1 = PaymentID(cls.to_hex(text))
  if p1.is_short():
    sender = cls.wallet.addresses()[0]
    sender = sender.with_payment_id(p1)
    transaction = cls.wallet.transfer(sender, Decimal(’0.000000000001’))
    return transaction

@staticmethod
def to_hex(text):
  s = text.encode(’utf-8’)
  return s.hex()

@classmethod
def sign_transaction(cls, transaction):
  return transaction

@classmethod
def send_raw_transaction(cls, transaction):
  transaction_hash = daemon.send_transaction(transaction)
  return transaction_hash

@staticmethod
def add_transaction_to_database(transaction_hash):
  database.add_transaction(transaction_hash, Blockchain.MONERO)
```

Listing 10: Creat tx with minimal amount in monero_adapter.py

In the next Listing 11, the code iterates through the outgoing tx to get the correct one. It then returns this to the caller.

```
@classmethod
def get_transaction(cls, transaction_hash):
  for item in cls.wallet.outgoing():
    if transaction_hash in str(item):
      return item
```

Listing 11: Send the tx to BC in monero_adapter.py

Monero Python module Documentation n.d.

4.3 Ripple

Listing 12 represents the imports and the constructor of the Ripple adapter. For Ripple there exists a ready rpc framework. Since a Ripple user does not need to download a BC before starting to make requests, it is also very easy to change between the productive and the test network.

```
from blockchain import Blockchain
from binascii import hexlify, unhexlify
import db.database as database
from db.config import ENCODING
```

```
import json
from adapters.adapter import Adapter
from ripple_api import RippleRPCClient

class RippleAdapter(Adapter):
  chain = Blockchain.RIPPLE
  credentials = database.find_credentials(Blockchain.RIPPLE)
  address = credentials[’address’]
  key = credentials[’key’]
  #Test
  rpc = RippleRPCClient(’https://s.altnet.rippletest.net:51234/’)
  #Prod with Fully History
  #rpc = RippleRPCClient(’https://s2.ripple.com:51234/’)
  ...
```

Listing 12: Ripple imports and constructor in ripple_adapter.py

Listing 13 shows how the transaction is built, signed and sent.

```
@staticmethod
def create_transaction(cls, text):
  memoData = cls.to_hex(text)
  query = {
    ”TransactionType”: ”Payment”,
    ”Account”: cls.address,
    ”Destination”: cls.address,
    ”Amount”: 0,
    ”LastLedgerSequence”: None,
    ”Fee”: 12,
    ”Memos”: [
      {
        ”Memo”: {
        ”MemoType”: ”42696672c3b67374”,
        ”MemoData”: memoData
        }
      }
    ]
  }
  return query

@staticmethod
def sign_transaction(cls, transaction):
  tx_object = cls.rpc._call(’sign’, transaction)
  return tx_object

@classmethod
def send_raw_transaction(cls, transaction):
  tx = transaction[’results’][’tx_blob’]
  tx_hash = cls.rpc.submit(tx_blob=tx)
  return tx_hash

@staticmethod
def add_transaction_to_database(transaction_hash):
  database.add_transaction(transaction_hash, Blockchain.RIPPLE)

@staticmethod
def to_hex(text):
  data = bytes(json.dumps(text), ENCODING)
  data_hex = hexlify(data)
  return data_hex.decode(ENCODING)
```

Listing 13: Ripple store functions in ripple_adapter.py

In Listing 14 a function is called which should return the tx. The data is then sent back to the caller.

```
\begin{lstlisting}[language=python]
@classmethod
def get_transaction(cls, transaction_hash):
  tx = cls.rpc._call(’tx’,{”transaction”: transaction_hash,”binary”: False})
  return tx
```

```python

@staticmethod
def extract_data(transaction):
  # Not required in case of DB
  return transaction

@staticmethod
def to_text(data_hex):
  data = unhexlify(data_hex)
  return data.decode(ENCODING)
```

Listing 14: Ripple retrieve functions in ripple_adapter.py

python-ripple-lib · PyPI n.d.

5 CONCLUSION

Interacting with different DLT implementation is not a trivial task. This is because to the fact that every BC is implemented differently that the others and do not follow any standard. The fact that algorithms in BCs like Monero change in such a way that they are still secure does not help when searching in the world wide web (www) for information. There are many old code examples with functions that are not even usable anymore. Since each BC has a different implementation in different programming languages, it takes a lot of time to read through everything and get the right tools to implement test cases. For example Monero has a testnet which everyone can download. Due to the slow download speed, it takes a lot of time (two days for test db, eight days for productive db) till the test can be done. Moreover, the time required to generate these adapters is unpredictable. Finally, it must be said that the code is not fully tested and only a few functions worked from the beginning.

Acknowlegments. Thank you to Edger John Schneid for useful discussions and guidance.

REFERENCES

Chain Interoperability (n.d.) https://allquantor.at/blockchainbib/pdf/vitalik2016chain.pdf. (Accessed on 11/25/2019).

Consensus - XRP Ledger Dev Portal (n.d.) https://xrpl.org/consensus.html. (Accessed on 12/01/2019).

CryptoNight — Monero Documentation (n.d.) https://monerodocs.org/proof-of-work/cryptonight/. (Accessed on 11/27/2019).

Developers Battle Over Bitcoin Block Chain (n.d.) https://www.coindesk.com/developers-battle-bitcoin-block-chain. (Accessed on 12/01/2019).

Eder John Scheid / bifrost · GitLab (n.d.) https://gitlab.ifi.uzh.ch/scheid/bifrost. (Accessed on 12/01/2019).

Elliptic Curve DSA – Wikipedia (n.d.) https://de.wikipedia.org/wiki/Elliptic_Curve_DSA. (Accessed on 11/27/2019).

Hegnauer, Timo (Feb. 2019) "Design and Development of a Blockchain Interoperability API" MA thesis. Zürich, Switzerland: Universität Zürich.

Herdius (n.d.) https://herdius.com/. (Accessed on 11/27/2019).

Install on Ubuntu or Debian Linux - XRP Ledger Dev Portal (n.d.) https://xrpl.org/install-rippled-on-ubuntu.html. (Accessed on 12/01/2019).

Interledger Architecture — Interledger (n.d.) https://interledger.org/rfcs/0001-interledger-architecture/. (Accessed on 11/27/2019).

Liegt die Zukunft der Blockchain in der Sidechain? — Blockchainwelt (n.d.) https://blockchainwelt.de/sidechain-zukunft-blockchain/. (Accessed on 11/27/2019).

Lite Coin White Paper (n.d.) http://zioncoins.co.uk/wp-content/uploads/2015/06/Lite-Coin-Whitepaper.pdf. (Accessed on 11/27/2019).

Monero – Wikipedia (n.d.) https://de.wikipedia.org/wiki/Monero. (Accessed on 11/27/2019).

Monero Python module Documentation (n.d.) https://buildmedia.readthedocs.org/media/pdf/monero-python/stable/monero-python.pdf. (Accessed on 11/28/2019).

moneropedia.entries.ringsignatures — Moneropedia — Monero - secure, private, untraceable (n.d.) https://web.getmonero.org/resources/moneropedia/ringsignatures.html. (Accessed on 11/30/2019).

Private Keys in Monero — Monero Documentation (n.d.) https://monerodocs.org/cryptography/asymmetric/private-key/. (Accessed on 11/30/2019).

python-ripple-lib · PyPI (n.d.) https://pypi.org/project/python-ripple-lib/. (Accessed on 11/29/2019).

Ripple Coin (XRP) einfach erklärt - Kryptowährungen (n.d.) https://www.cryptolist.de/ripple. (Accessed on 12/01/2019).

Satoshi Nakamoto - Wikipedia (n.d.) https://en.wikipedia.org/wiki/Satoshi_Nakamoto. (Accessed on 11/26/2019).

scrypt.dvi (n.d.) https://www.tarsnap.com/scrypt/scrypt.pdf. (Accessed on 11/27/2019).

Transaction Common Fields - XRP Ledger Dev Portal (n.d.) https://xrpl.org/transaction-common-fields.html. (Accessed on 12/01/2019).

What is Ripple? — Bitcoin Magazine (n.d.) https://bitcoinmagazine.com/guides/what-ripple/. (Accessed on 11/30/2019).

whitepaper.pdf (n.d.) https://cryptonote.org/whitepaper.pdf. (Accessed on 11/30/2019).

Zheng, Z., S. Xie, H. Dai, X. Chen, and H. Wang (June 2017) "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends". In: *2017 IEEE International Congress on Big Data (BigData Congress)*, pp. 557–564. DOI: 10.1109/BigDataCongress.2017.85.

APPENDIX

The developed code is available under https://github.com/marciseli/bifrost . Every installation was made on Ubuntu 18.04.3-desktop-amd64 on a ESXI Server 6.5.

1 INSTALLATION GUIDELINES

1.1 Litecoin

The Litecoin installation is nearly the same like Bitcoin. There are also frameworks used that also use bitcoin and sometimes just renamed. In Listing 1 the full node installation is shown.

```
# Berkey DB installation an required libraries
  add-apt-repository ppa:bitcoin/bitcoin
  apt-get update
  apt-get install libdb4.8-dev libdb4.8++-dev
  apt-get install libboost-all-dev libzmq3-dev libminiupnpc-dev
  apt-get install curl git build-essential libtool autotools-dev
  apt-get install automake pkg-config bsdmainutils python3
  apt-get install software-properties-common libssl-dev libevent-dev

# Generate a directory to save the installation files
  mkdir /downloads
  cd /downloads
  git clone https://github.com/litecoin-project/litecoin.git

# Install the Litecoin node and generate an account
  cd litecoin
  ./autogen.sh
  ./configure
  make
  make install

# download BC database
  litecoind -daemon
```

Listing 1: Install depencies for Monero and Monero itself.

1.2 Monero (XMR)

Some dependencies are pre-installed for the monero implementation. Listing 1.2 illustrates the installation of the testnet. When using the productive BC, it is only necessary to remove “–testnet”in the last line.

```
# dependencies
  apt-get install libboost-all-dev

# get monero files
  mkdir monero && cd monero
  wget https://downloads.getmonero.org/linux64
  tar -jxvf linux64

# directory could also has another name
  cd monero-v0.15.../

# installation
  install -sv monerod monero-wallet-cli /usr/local/bin/
  install -sv monero-wallet-rpc monero-blockchain-export /usr/local/bin/
  install -sv monero-blockchain-import /usr/local/bin/

# download blockchain testnet. Without "testnet" it's the productive one.
# Needs a lot time.
  monerod --testnet
```

Listing 2: Install depencies for Monero and Monero itself.

In Listing 3 a Wallet will be generated called “testwallet”. The second line is used to start a RPC-Server which can be called over port 28088. The BC has to be downloaded before executing this commands.

```
# "testwallet" is the name of the new wallet. This name must be included into db.
   config
  monero-wallet-cli --testnet --generate-new-wallet testwallet

# this command starts a server on port 28088.
  monero-wallet-rpc --testnet --wallet-file testwallet --password "" --rpc-bind-
   port 28088 --disable-rpc-login
```

Listing 3: Wallet installation and start rpc server on it.

Monero recommends to start the RPC-Client in a virtual environment. The installation of this tool is not part of this document. Next we see in the lower part of Listing 4 the installation of the Monero framework for python.

```
# create virtualenv directory
  virtualenv .venv
# start virtualenv
  source .venv/bin/activate

# install dependencies
  pip install -r requirements.txt
  pip install monero-python
```

Listing 4: Monero wallet installation and start rpc server on it.

To get some coins on the Monero test wallet, there is a faucet available on this site: https://community.xmr.to/faucet/testnet/

1.3 Ripple

The Ripple installation starts with the preparation of a virtual environment. As illustrated in Listing 5, the framework for the python library is also installed.

```
# create virtualenv directory
  virtualenv .venv
# start virtualenv
  source .venv/bin/activate

  pip install python-ripple-lib
```

Listing 5: Set up the virtual environment and python framework (*Install on Ubuntu or Debian Linux - XRP Ledger Dev Portal* n.d.) .

Listing 6 shows how the Ripple package signing GPG key is installed and afterwards the other packages needed for Ripple. At least the config file has to be adapted to make signing available. Ripples installation is a service which can be automatic started when the node is booted.

```
  sudo apt -y install apt-transport-https ca-certificates wget gnupg
  wget -q -O - "https://repos.ripple.com/repos/api/gpg/key/public" | \
  sudo apt-key add -

  echo "deb https://repos.ripple.com/repos/rippled-deb bionic stable" | \
  sudo tee -a /etc/apt/sources.list.d/ripple.list
  sudo apt -y update
  sudo apt -y install rippled
  systemctl status rippled.service

  nano /etc/opt/rippled.cfg
  # change file:
   [signing_support]
   true
```

Listing 6: Wallet installation and start rpc server on it (*Install on Ubuntu or Debian Linux - XRP Ledger Dev Portal* n.d.) .