University of Zurich[UZH]

MASTER BASIS MODULE – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a Blockchain Interoperability API

Patrick Widmer
Zürich, Switzerland
Student ID: 13-786-009

Supervisor: Eder John Scheid, Bruno Bastos Rodrigues
Date of Submission: August 6, 2018

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Basis Module
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Design and Implementation of a Blockchain Interoperability API

Patrick Widmer

December 11, 2018

1 Introduction and Motivation

Since the release of Bitcoin in 2009 [4], multiple cryptocurrencies have arisen, such as Ethereum [1], and Ripple [5]. However, the infrastructure supporting these cryptocurrencies can be used for a number of other applications rather than financial ones, resulting in a boom of purpose-specific blockchains. In essence, a blockchain is a distributed append-only immutable ledger that relies on cryptography to remove the need for Trusted Third Parties (TTP) to verify new transactions. Most blockchain implementations provide incentive mechanisms to perform this verification of transactions, issuing new coins to particular users, also known as miners. Also, with the popularization and speculation about the value of such coins, *i.e.* cryptocurrencies, the public interest, from technical and non-technical individuals, in such technology has grown as well. Thus, blockchain is now being applied in a myriad of fields, from agriculture [8] to pharmaceutical goods [7].

Although all blockchain implementations share the same goal, which is to provide an immutable and trusted ledger, these do not necessarily rely on the same underlying parameters as they were designed for a specific application. For example, in Bitcoin new blocks are created every 10 minutes with a fixed size of 1 MB. This limits the number of transactions that can be included in a new block resulting in a transaction rate of approx. 7 transactions per second. However, Ethereum has an average block size of 20 KB and targets the creation of new blocks by miners every 14 seconds, which results in approx. 20 transactions per second. Both blockchains deploy Proof-of-Work (PoW) algorithms to perform the verification of newly included blocks. However, since Ethereum and Bitcoin are developed for different usage scenarios, advantages and disadvantages may appear considering their deployment and usage in different application areas and use cases.

This work considers the supply-chain area where goods of all over the world are produced and distributed across a vast network of producers, retailers, distributors, transporters, and vendors in a complex arrangement of processes. Each one of these stakeholders has different requirements, and thus, may use different blockchains. Therefore, the system must take this aspect of multiple blockchain usage into consideration and allow the interoperability between blockchains by stakeholders.

This report is structured as follows. Section 2 presents the goals of this project. Next, Section 3 describes the advantages of providing interoperability between blockchains and

presents the design and implementation of the system. Challenges encountered during the development of such a system are described in Section 4. Finally, Section 5 concludes the report and presents future work directions.

2 Goal

The goal of this work is to determine the current technical blockchain interoperability challenges and to implement a system able to provide this interoperability with different blockchains. For example, the system must provide functions to select the desired blockchain, store and retrieve data from this blockchain, *e.g.* Ethereum or Bitcoin. To achieve that, the system must contain connectors to each blockchain implementation and databases to support the selection. Further, the system must automatically create the correct transaction template based on the input data and submit this transaction to the correct blockchain connector. It is expected that the system is extensible to include new blockchain transaction templates and connectors.

3 Approach

This section presents a unified interface to dynamically store data on various blockchains. It hides differences in their implementation details and provides an additional layer of abstraction. An advantage of such a system is robustness through independence of a particular blockchain technology. Another advantage is to gain flexibility to dynamically choose the optimal, *i.e.* best-suited, blockchain based on some evaluation criteria, *e.g.* the cheapest or the most-performant, for a particular application or use case.

3.1 Design and Implementation

The system will be incorporated into a framework in the future. Thus, a modular and extensible approach is crucial to guarantee a seamless integration process. Due to the limited scope of this project, the goal is a prototype, *i.e.* Proof-of-Concept (PoC), rather than a finished product. More advanced concepts such as security and testing are disregarded in the prototype. Initially, we focus on supporting a selection of blockchains. More precisely, the initial version of the prototype integrates three blockchains: Ethereum, Bitcoin and MultiChain [3]. To interact with blockchains, a client is required. In this project, we use geth to connect to Ethereum, bitcoind and bitcoin-cli interact with Bitcoin and multichaind and multichain-cli for MultiChain.

We choose Python as a programming language to implement the system. Python is widely supported and allows a fast development process. The systems entry point is a python module which exposes two functions, `store` and `retrieve`. The former function receives a text message and stores it on the specified blockchain. It returns a (hexadecimal) transaction hash, uniquely identifying the transaction containing the data on the blockchain. This transaction hash can be used to retrieve the text message

from the blockchain. Using a blockchain rather than a traditional database guarantees non-repudiation and immutability of the data.

The class diagram of the system is shown in Figure 1. The `store` and `retrieve` functions of the main module delegate the request to the corresponding adapter based on the chosen blockchain. Thus, each blockchain requires an adapter (or connector) to be implemented. We define an abstract base class `Adapter`. Each concrete adapter extends this class and implements the abstract properties and methods. The core of the `Adapter` base class are the `store` and `retrieve` methods. The former, shown in Listing 1, first creates a transaction encoding the given text message. It signs this transaction with the private key (stored in the database) and sends the signed transaction as a raw transaction by broadcasting it to various other nodes in the blockchain. Finally, it stores the transaction in the database before returning the transaction hash. The latter, shown in Listing 2, first retrieves the transaction corresponding to the given transaction hash from the blockchain. Finally, it extracts the data from the transaction and returns the restored text message. The described procedures are the same for all the blockchain adapters. However, the methods called, such as `create_transaction` are implemented in the concrete adapter classes and vary in their implementation details.

Listing 1: Store method of abstract Adapter class

```
1 def store(text):
2         tx = create_transaction(text)
3         signed_tx = sign_transaction(tx)
4         transaction_hash = send_raw_transaction(signed_tx)
5         add_transaction_to_database(tx_hash)
6         return tx_hash
```

Listing 2: Retrieve method of abstract Adapter class

```
1 def retrieve(tx_hash):
2         tx = get_transaction(tx_hash)
3         data = extract_data(tx)
4         return to_text(data)
```

Transactions are not unified and vary for different blockchains. In Ethereum, transactions require a number used once (nonce), a gas price and a gas limit. A template for a transaction in Ethereum is shown in Listing 3. Even though MultiChain is based on Bitcoin, the MultiChain transactions are not interchangeable with Bitcoin transactions. In early versions of Bitcoin there was no data field at all. Storing arbitrary data on the Bitcoin blockchain was not officially supported. However, it was still possible to store data using a workaround. Basically, data, *e.g.* an image or a text message, is converted into their hexadecimal representation then used as an output address in a transaction. To prevent this misuse of the system, data fields for transactions were introduced to Bitcoin Core in version 0.9.0. Since then, in Bitcoin, an optional data field is part of a transaction output. In contrast, in MultiChain, the data is independent of the transaction outputs

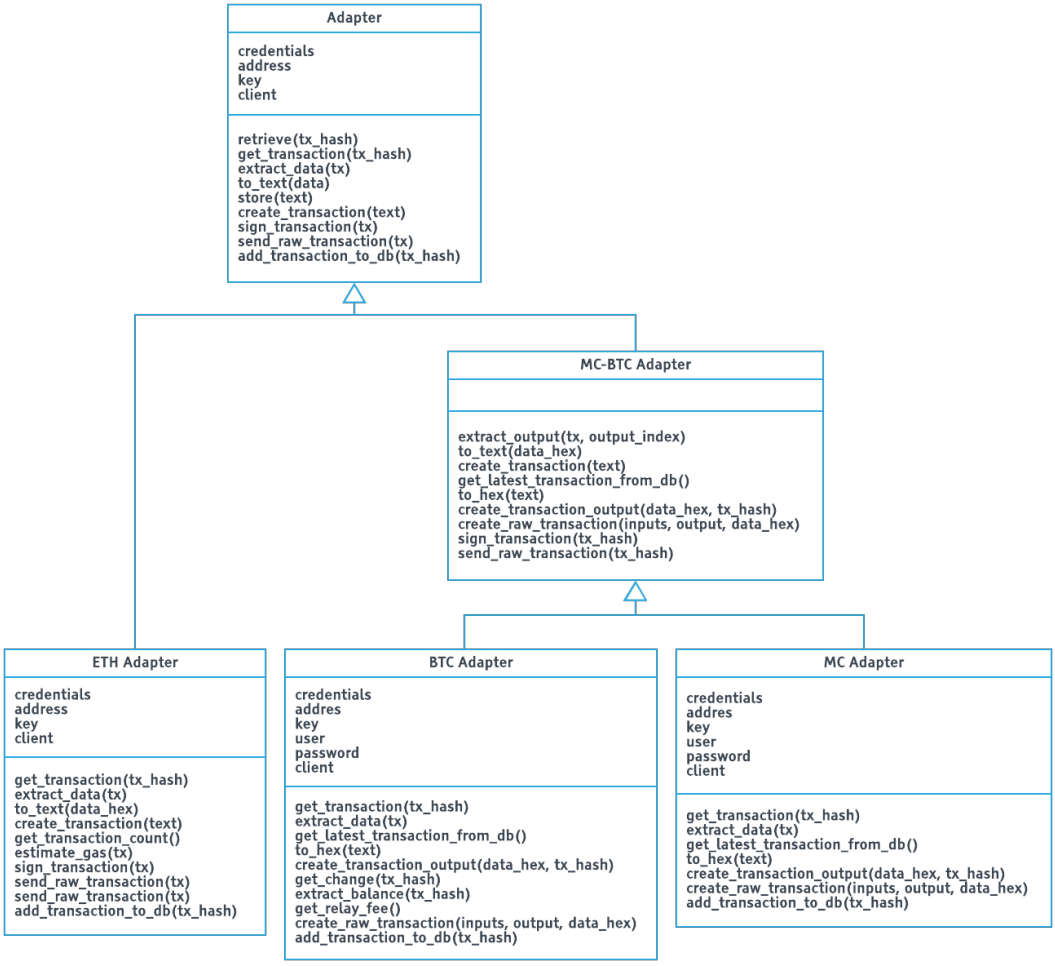

Figure 1: UML class diagram of the system.

(at least during creation of a transaction). Also, MultiChain supports a list of data fields compared to Ethereum and Bitcoin, which accept only a single data field. Another difference is the encoding of the data which varies for all the considered blockchains. Templates for transactions in Bitcoin and MultiChain are shown in Listings 4 and 5.

Listing 3: Template for transaction in Ethereum

```
tx_eth = {
        "from": from_address,
        "to": to_address,
        "gasPrice": gas_price,
        "gas": gas_limit,
        "value": amount,
        "data": data,
        "nonce", nonce
}
```

Listing 4: Template for transaction in Bitcoin

```
tx_btc = {
        "tx_in": [{
                "txid": tx_hash,
                "vout": output_index
        }],
        "tx_out": [{
                to_address: amount,
                "data": data
        }]
}
```

Listing 5: Template for transaction in MultiChain

```
tx_mc = {
        "tx_in": [{
                "txid": tx_hash,
                "vout": output_index
        }],
        "tx_out": [{
                to_address: amount
        }],
        "data": [
                data
        ]
}
```

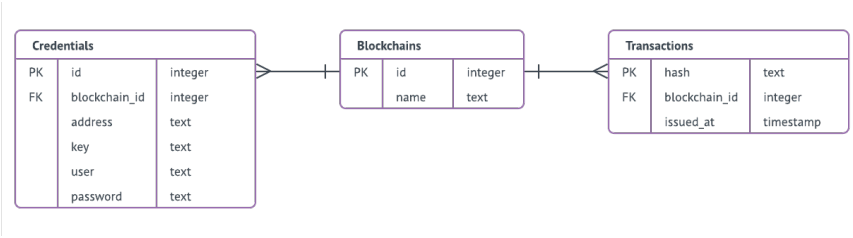

Figure 2: Database entity-relationship (ER) model of the system.

MultiChain is based on Bitcoin. Thus, these two blockchains share more similarities than other blockchains. The shared implementations are extracted into a common adapter class, `MCBTCAdapter`. This class implements methods such as `create_transaction` for both adapters. The implementation of this method is shown in Listing 6.

Listing 6: Create_transaction method of MC/BTC Adapter

```
1  def create_transaction(text):
2          input_tx_hash = \
3                  get_latest_transaction_from_database()
4          inputs = [{
5                  "txid": input_tx_hash,
6                  "vout": 0
7          }]
8          data_hex = to_hex(text)
9          output = create_transaction_output(
10                 data_hex,
11                 input_tx_hash
12         )
13         tx_hex = create_raw_transaction(
14                 inputs,
15                 output,
16                 data_hex
17         )
18         return tx_hex
```

This project uses a database based on SQLite. SQLite is a database management system. It is a light-weight solution and well-supported by Python. The credentials required to connect to the different blockchain clients are stored in the database. Also, every time data is stored on a blockchain, a new transaction reference is added to the database. This way, the system keeps track of all the stored transactions and knows, to which blockchain a transaction with a specific transaction hash belongs. The database model consists of three tables and is shown in Figure 2.

4 Challenges

One of the first issues encountered during this project was related to Ganache [2]. Ganache, formerly TestRPC, is a tool provided by the Truffle Suite. It is a local development blockchain server for Ethereum. It focuses on simplifying the development of smart contracts. However, when trying to send transactions with data to a non-contract address, the system crashes. As it turns out, this is caused by a bug in the implementation of Ganache [6], as it currently only allows data inside a transaction when the recipient address is a contract. Therefore, Ganache was substituted with a different Ethereum client, in this case, geth. With such a substitution the transactions were submitted to the blockchain without any problems.

Another issue we had to solve is related to offline signing. In offline signing, keys (especially private keys) are managed externally, *i.e.* outside of a wallet. The idea behind offline signing is to store keys on hardware devices or "cold storage" to improve security. However, certain operations such as `listunspent`, to list all the unspent transaction outputs corresponding to an address, are only available if a wallet is used. Offline signing requires a raw transaction and a private key to create a signed transaction that can be sent, *i.e.* broadcast to the network. Raw transactions (in Bitcoin and MultiChain) require at least one unspent transaction output, to be used as input for this new transaction, *e.g.* to account for the transaction fee. To find one such unspent transaction output, without importing the private key in a wallet, we find the latest transaction from the database. However, initially when the database is first setup, there is no transaction known to the system. As a workaround, we define a seed transaction for each blockchain (at least for Bitcoin and MultiChain) in the configuration file and store them in the database during the setup.

5 Conclusion and Future Work

In this report, an approach towards blockchain interoperability was presented. The approach consists of the development of a system that exposes an API to store and retrieve data in different blockchains. The system is implemented in a modular architecture to guarantee extensibility. It considers the differences in the implementation of various blockchains, and provides a level of abstraction. The system dynamically selects the adapter corresponding to the choice of blockchain and creates a valid transaction from a template for this blockchain. Credentials to access the blockchains and transactions are stored in a database.

The outcome of this project is a prototype. Therefore, some aspects were disregarded during the development. Currently, all the credentials, including private keys are stored in plain text, *i.e.* unencrypted in the database. Before releasing a final version of this system, a secure way of storing and accessing credentials has to be implemented.

Also, there are currently no test cases written for this module. Tests ensure a system works as intended and follows the specifications. Therefore, it is recommended to implement unit and integration tests in an upcoming development cycle.

As a possible extension, support for optional custom transactions could be implemented. Currently, only the data of a transaction is defined by the user. In a more flexible system, custom sender, recipient or amounts to transfer could be specified by the user as well.

Related to the last point, currently the system uses a single address for all transactions in a blockchain. In Ethereum, this is not problematic. In fact, it is intended, because Ethereum uses an account-based approach. However, other blockchains such as Bitcoin and MultiChain use an address-based approach. In these blockchains, it is still possible to reuse the same address in multiple transactions, *i.e.* use the sender address as the recipient address for the change amount. However, it is discouraged to do so, because it harms privacy [9].

Finally, the system could be extended to support more blockchains. This would include the implementation of new blockchain adapters. Because of the design of the system, *e.g.* defining an abstract base class for adapters, this should be well-supported.

References

[1] Ethereum. Ethereum White Paper. *Available at https://github.com/ethereum/wiki/wiki/White-Paper*, Accessed Jun, 2018.

[2] Ganache. One click blockchain. *Available at https://truffleframework.com/ganache*, Accessed Jul, 2018.

[3] MultiChain. MultiChain Private Blockchain – White Paper. *Available at https://www.multichain.com/download/MultiChain-White-Paper.pdf*, Accessed Jul, 2018.

[4] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Available at http://bitcoin.org/bitcoin.pdf*, Accessed Jun, 2018.

[5] Ripple. The Ripple Consensus Protocol. White Paper. *Available at https://ripple.com/files/ripple_consensus_whitepaper.pdf*, Accessed Jun, 2018.

[6] Truffle Suite. Ganache shouldn't assume that every transaction with data is a contract call. *Available at https://github.com/trufflesuite/ganache-core/issues/117*, Accessed Jul, 2018.

[7] T. Strasser T. Bocek, B. B. Rodrigues and B. Stiller. Blockchains Everywhere - a Use-Case of Blockchains in the Pharma Supply-Chain. *in 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, May 2017.

[8] F. Tian. An Agri-Food Supply Chain Traceability System for China based on RFID Blockchain Technology. *in 2016 13th International Conference on Service Systems and Service Management (ICSSSM)*, pages 1–6, June 2016.

[9] Bitcoin Wiki. Address reuse. *Available at* https://en.bitcoin.it/wiki/Address_reuse, Accessed Jul, 2018.