Bitcoin Zero-Confirmation Transactions

NSSL spring 2019
Instructor: Itay Tsabary
Barak Gahtan
Lior David

Table of Contents:

Abstract

Blockchain-based systems, among them is Bitcoin, facilitate transactions among their users. To assure secure execution these systems enforce non-negligible transaction confirmation times. That hinders their usability – as it requires transacting parties either to accept unconfirmed transactions, or to transact at the slow confirmation rate.
In this project we design and implement a system that overcome these issues by using a third-party mediator. We utilize Bitcoin’s transaction scripts in a way ensuring that a transaction can be deemed confirmed even without waiting for an actual confirmation. We also ensure that transactions carry out, as intended, even if one of the involved parties act maliciously.

Introduction

A system based on blockchain is a decentralized trustless system. The system is organized as a sequence of append only blocks. The blocks are readable and tamperproof. Bitcoin is a blockchain based system.

Every transaction in Bitcoin consists of several inputs and outputs. Each transaction input can be seen as money. Each output states its destination. As in the real world where money cannot be used more than once, inputs act the same. Since money is a one-time use, a buyer can mislead a seller by sending him used money.

A system based on blockchain introduces a new problem called double spending. A new transaction that is introduced into the blockchain needs to be confirmed by at least several blocks. When making a bitcoin transaction, it enters into a pool of unconfirmed transactions. Bitcoin users then select the transaction and place it into a block of transactions. A user solves a special mathematical puzzle called a proof of work. After that, the Bitcoin network confirms the block and adds it to the blockchain. Each new block added to the blockchain is another confirmation for the transaction.

All of the described scenario takes an amount of time that can cause many problematic scenarios. Such a scenario consists of:

1. A coffee place - "B".
2. A buyer - "A".

"A" wants to buy a cup of coffee. "B" wants to sell the cup of coffee. In order for "B" to sell "A" the coffee, "B" has to wait for a confirmation. This waiting time is problematic when buying a cup coffee, since it should be an immediate on demand purchase. In order to deal with such a problem, we introduce a system that enables zero confirmation time of transactions.

Our purposed system will include a mediator - "M" that allows us to make sure every party in the transaction will fulfill its part. Our system's initial state would include a deposit from "A" to a joint address with "M", that can be used under certain circumstances. If for some reason, "A" would not pay "B" the due money, the deposited money would be transferred to "B".

In order to use the deposited money, "A" and "M" would set up a joint address. The address would require the approval of both "A" and "M" in order to be used.

“A” can choose if to mislead “B” or not. If “A” would not act maliciously towards “B”, the deposited money would return to “A” from the joint address. Otherwise “A” deceives “B”, the deposited money would be transferred to “B” from “M”. After the confirmation period has passed, the money from the joint address would return to “A”.

Our implementation is based on the Bitcoin Core software. We implemented wrappers using Python scripting. We used RPC to communicate with the Bitcoin core from our local machine. We manipulated some of the wrapper, using the Bitcoin core software, in order to fulfill our needs.

The system is implemented on a clean environment without prior history of transactions. The parties introduced are the only entities in the system. We use the “regtest” mode of the Bitcoin Core client.

Background

Address

Address is an identifier of 26-35 alphanumeric characters. The address begins with the number "1", "3" or "bc1" that represents a possible destination for a payment.

Private key

Private key is a secret number that allows to encrypt and decrypt mini programs. The private keys are mathematically related to the address associated with the user. It can be seen as a password to an address.

Public key

Public key is a unique number mathematically generated from a private key. A public key is used to receive bitcoins.

Hash functions

A hash function is a mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size. It is a one-way function which practically is infeasible to invert.

Transaction

A transaction is a transfer of values between Bitcoin addresses. Each transaction has its own id (TXID) and contains the following:

1. Inputs - a transaction id and an output number (vout).
2. Outputs – it is the hash to which the inputs were sent to.
3. Amount – it is the amount of Bitcoin that was sent.

Each transaction can contain multiple inputs and outputs.

Blocks

Blocks are files where data pertaining to the network are permanently recorded. A block records some or all of the most recent transactions that have not yet entered any prior blocks. A block represents the 'present' and contains information about its past and future.

Miners

The role of miners is to secure the network and to process every transaction. Miners achieve this by solving a computational problem which allows them to chain together blocks of transactions.

Mining
It is the process of solving the cryptographic puzzle, created by the selection of transactions to fit in a block. The primary purpose is to set the history of transactions in a way that is computationally impractical to modify by any one entity.

Transaction fee
By solving mathematical puzzles, miners confirm new transactions to the blockchain, which will credit them with fees.

Blockchain
A blockchain is a database that is shared across a network of computers. Once a record has been added to the chain (a block has been added) it is very difficult to change. To ensure all the copies of the database are the same, the network makes constant checks.

Bitcoin
Bitcoin is a digital and global money system currency based on blockchain. It allows people to send or receive money across the internet, even to someone they don't know or don't trust. Money can be exchanged without being linked to a real identity. It consists three modes:

1. "Mainnet" - The main mode contains the real-world Bitcoin transactions.
2. "Testnet" – It is primary for developers, using unreal money.
3. "Regtest" - Is for custom private blockchain. Our project would be based on "Regtest".

Time
Time is measured in a blockchain based system by the number of blocks that have been approved, since the transaction submitted.

Mempool
The Mempool is a waiting area for Bitcoin's unconfirmed transactions. After a transaction is confirmed it is picked by a miner and inserted into a block.

Base 58
Base58 is an encoding of binary data in a sequence of printable characters. These encodings are necessary for transmission of data when the channel does not allow binary data. It is used to represent large integers as alphanumeric text.

Digital signature
A digital signature is a value you can use to show that you know the private key connected to a public key, without having to reveal the actual private key.

Multisig

Multisig refers to requiring more than one key to authorize a Bitcoin transaction. For a transaction to happen, it requires multiple signatures, possibly by different private keys.

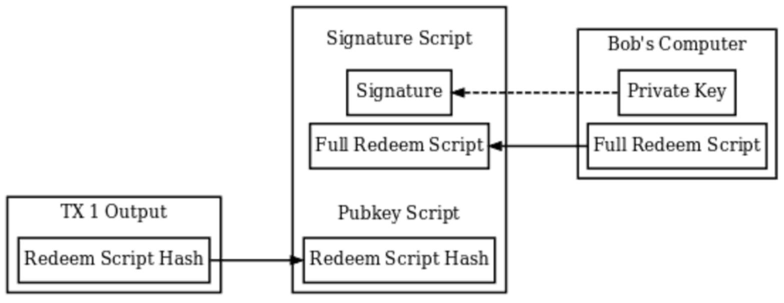
Scripts

A script is essentially a list of instructions recorded with each transaction. Those instructions describe how the next person wanting to spend the Bitcoins being transferred, can gain access to them. The script for a typical Bitcoin transfer to a destination address simply encumbers future spending of the bitcoins with two things the spender must provide:

1. A public key. When the public key is hashed, it yields a destination address embedded in the script.
2. A signature. In order to prove ownership of the private key corresponding to the public key, one has to present a signature.

Scripting provides the flexibility to change the parameters of what's needed to spend Bitcoins. For example, the scripting system could be used to require two private keys, or a combination of several keys, or even no keys at all. In our implementation we are using two different scripts:

1. Pay to public key hash (P2PKH)
   It is a type of an output created with a public key. To commit such an output as an input in a transaction, one should provide a digital signature. The digital signature is proved by matching private key of the public key used in the output creation.
   In order to unlock an output, one needs to provide signatures using their private keys for spending Bitcoins on their public address. This address is called a P2PKH address which actually holds bitcoins using a script. If the script conditions are met, the coins are unlocked. In case of a mismatch, the coins remain locked.
2. Pay to script hash (P2SH)
   P2SH allows transactions to be sent to a script hash instead of a public key hash. When the recipient wants to unlock the coins on this P2SH, they need to produce the whole script, the script hash and the requirements to unlock it on the blockchain.



Spending A P2SH Output

Opcodes

Operation codes from the Bitcoin script language. The opcodes push data or perform functions within a public key script.

Redeem-Script

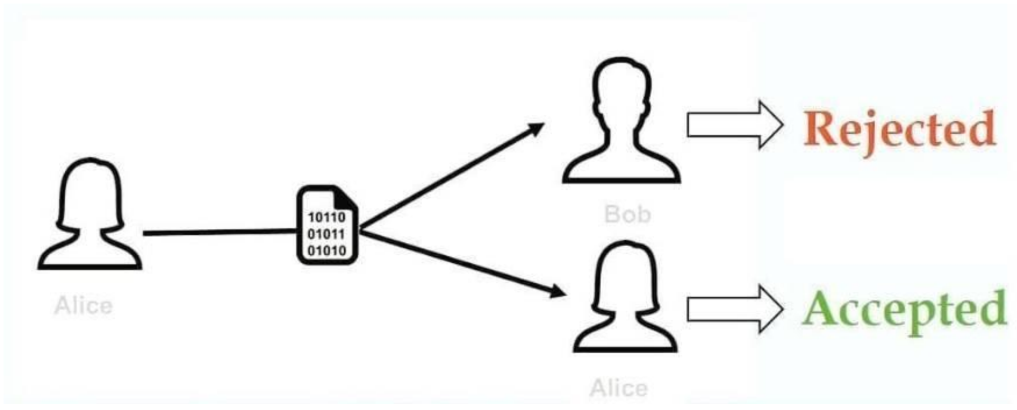
A script included in outputs that sets the conditions that must be fulfilled for Bitcoins to be spent. The hash of the redeem script has the same properties as a public key hash. Consequently, it can be transformed into the standard Bitcoin address format, in particular a P2SH address.

The problem

Buying coffee should be an immediate on demand purchase. Such a purchase in a blockchain based system is rather problematic. Each new transaction committed into the blockchain remains in the Mempool until confirmed. The waiting time can cause many problems, one of them is called double spending. This is unique to digital currencies because digital information is something that can be reproduced rather easily.

We illustrate the problem with the following scenario:

1. Alice creates a P2PKH between her and Bob.
2. Alice creates another one to herself, using the same inputs.
3. Alice's second P2PKH is confirmed.
4. Bob delivers the goods to Alice.
5. The transaction between Alice and Bob is canceled. As explained earlier, the inputs cannot be used more than once, hence the transaction is not accepted.
6. Therefore, Bob doesn't get the payment.

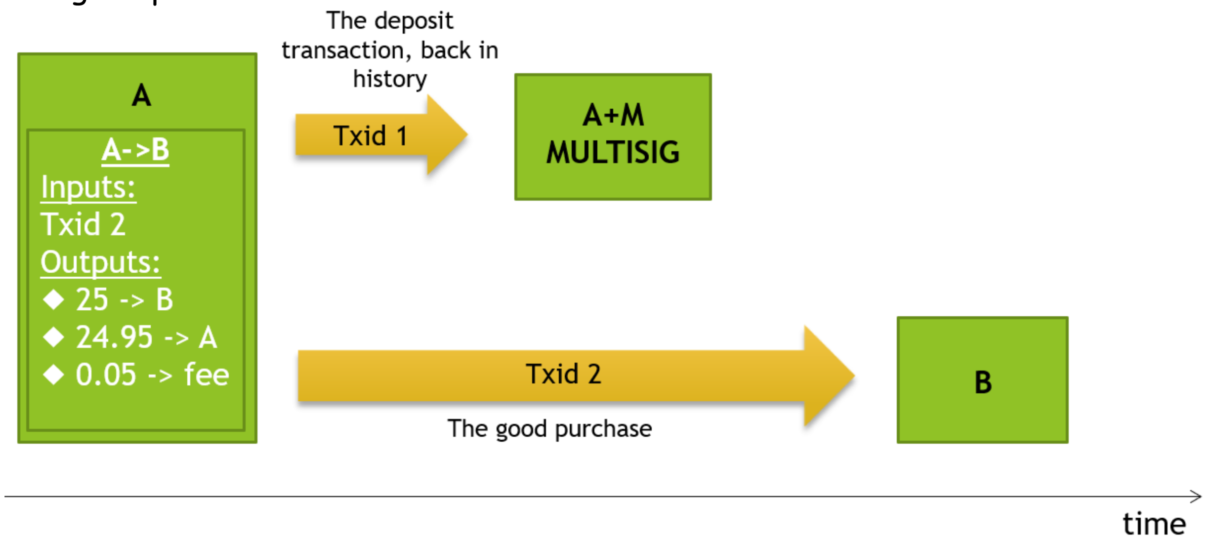


The solution

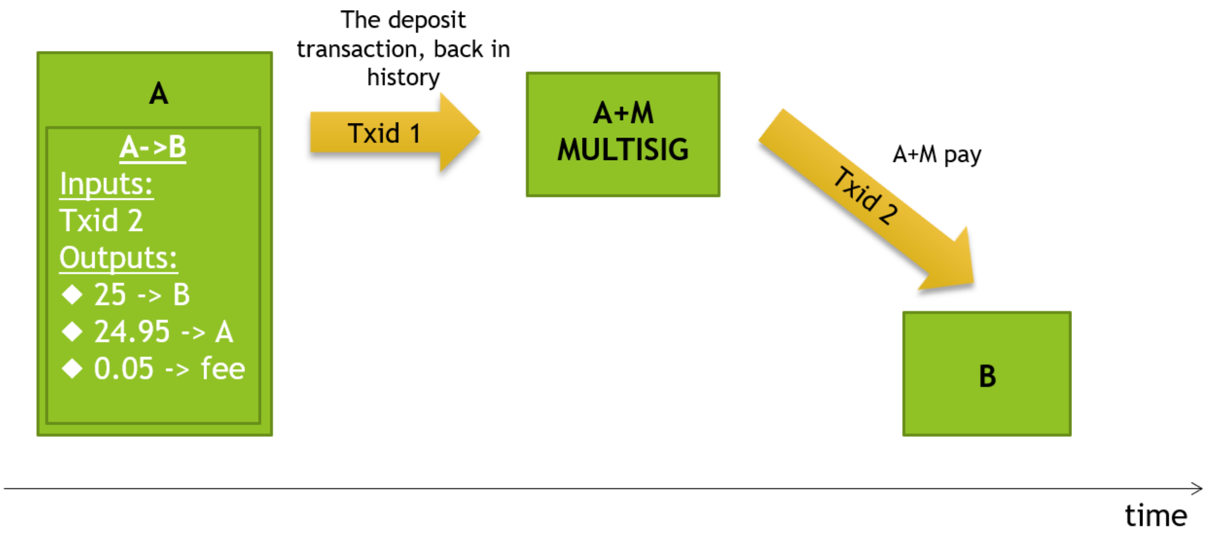
A Bitcoin based system that enables zero-confirmation of transactions. Our purposed system includes a mediator - "M". The system's initial state includes a deposit from "A" to "M" that can be used under certain conditions. If for some reason, "A" would not pay "B" the due money, the deposited money would be transferred to "B". If "A" would try to mislead "B", "M" would transfer the money to "B".

Our purposed mechanism:

The good purchase:



The bad purchase:



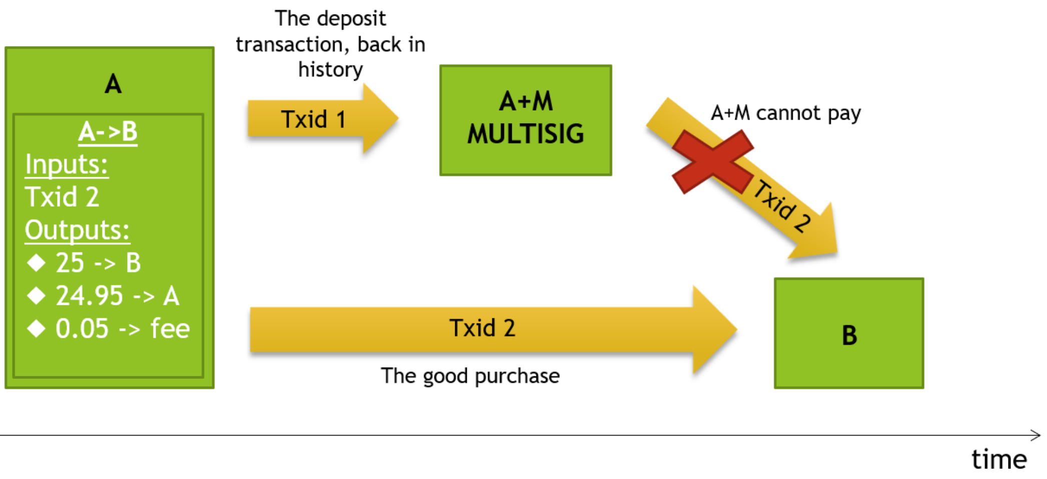
Mechanism explanation

Our mechanism relies mainly on the properties of the Bitcoin language scripts. The stored outputs of the transaction between "A" and "M" would be the inputs to the transaction between "A" and "B".

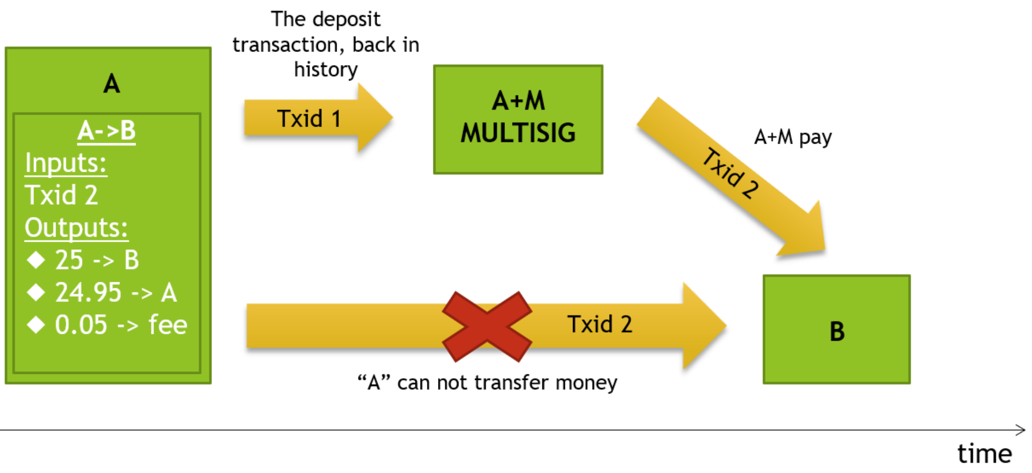
If for some reason "A" would not pay "B" the money it is supposed to pay, "M" would send the money to "B", after a timeout. We set the timeout to be 500 blocks. We chose that amount, because it guarantees that "A" acted maliciously. "M" would send using a multisig address that in order for it to send, needs to have at least two private signatures, one from "A" and one from "M" itself, only then it would be able to send to "B".

If the transaction from "A" to "B" was approved, the transaction between the joint address and "B" cannot happen. This is because it uses the same outputs as inputs.

1. If the good purchased occurred, the transaction between "M" and "B" should not be able to occur.



2. If the bad purchase occurred, the transaction between "B" and "M" would occur, after a timeout.



Detailed Explanation

Setting the environment

- We created an empty environment, without any history, by using the "regtest" mode.
- Each run of the system, all the prior information from the last run is deleted.
- We generated the environment by communicating with the Bitcoin core software using RPC methods.

Creating the users

- We created a private key for each of the users, using our method "key_generator_func".
- We generated a matching public key that is derived from the private key.
- We generated the address of the user, that is derived from the public key.

Generating funds to "A"

- We generated a block for "A" and transferred to the block 50 Bitcoins. The transfer allows "A" to have possession of coins.
- We saved the transaction's id.
- We generated 100 blocks to the environment, so the block of "A" would be approved. The generation of those blocks, allows a transaction to exit the Mempool and be inserted as a confirmed transaction to a block.

Creating the multisig address

- We created a multisig address of "A" and "M", by the following steps:
    1. Appending the public keys of "A" and "M" to an array.
    2. Adding the address to the blockchain.
- Once we created the multisig address, we added it to the environment.
- Derived from the multisig address, we received the redeem script for the address. This will aid us to extract the funds from the address later.
- When the address was created, the signature of "A" was added to the address. If "M" would want to spend, it will only require the signature of "M".

Transfer between "A" to "M" - initial state

- By using "A"'s private key, address and the transaction id, that was saved earlier, we sent a transaction from "A" to "M" using the method "transaction_Single_To".
- By communicating with the Bitcoin Core using RPC, we extracted the new transaction id for both, "A" to "B" or "A" to "M", depending on the behavior of "A".
- Each transaction has an id and outputs, we would use the outputs of the transaction between "A" and "M", as inputs to the transaction between "A" and "B".

The Good Case

"A" does not mislead "B" by transferring the money within the time limit.

- Using the method "transaction_Single_To_Single", "A" transferees money to "B".
- The method uses the address of "A", the destination – "B"'s address, the transaction id and the redeem script.
- We wait until the transaction is approved.
- If "M" would try to send money to "B" after a timeout, the transaction will fail, because it is using the same inputs.

The Bad Case

"A" misleads "B" by not transferring the money within the time limit.

- After a timeout, it is understood that "A" stole from "B".
- Using the method "transaction_Multi_To_Single", "M" transfers money to "B".
- The method uses the address of the multisig, the destination – "B"'s address, "M"'s private keys, the transaction id and the redeem script.
- We wait until the transaction is approved.
- If "A" would try to send money to himself, the transaction will fail, because it is using the same inputs.

Library key functions

We implemented our module using python in addition to JSON-RPC. We used the reg-test network in order to check the running of our cases.

Function – "generate"

General outline:

Generates blocks to the blockchain.

Key arguments:

1. nblocks (numeric) How many blocks are generated immediately.

Output:

[blockhashes] (array) hashes of blocks generated.

Function – "generatetoaddress"

General outline:

Mine blocks immediately to a specified address. (before the RPC call returns)

Key arguments:

1. nblocks (numeric, required) How many blocks are generated immediately.
2. address (string, required) The address to send the newly generated bitcoin to.

Output:

[blockhashes] (array) hashes of blocks generated.

Function – "createrawtransaction"

General outline:

Create a transaction spending the given inputs and creating new outputs.
Outputs can be addresses or data.
Returns hex-encoded raw transaction.

Key arguments:

1. "inputs" (array, required) A json array of json objects

```
[{
   "txid": "id", (string) The transaction id.
   "vout": n, (numeric) The output number.
   "sequence": n, (numeric) The sequence number.
  }]
```

2. "outputs" (array, required) a json array with outputs (key-value pairs)[
   {"address": x.xxx,(obj, optional) A key-value pair. The key (string) is the bitcoin address, the value (float or string) is the amount in BTC}{
   "data": "hex" (obj, optional) A key-value pair. The key must be "data", the value is hex encoded data.}]

Output:

transaction" (string) hex string of the transaction.

Function – "decoderawtransaction"

General outline:

Return a JSON object representing the serialized, hex-encoded transaction.

Key arguments:

1. "hexstring" (string) The transaction hex string.

Output:

hex-encoded transaction.

Function – "getrawtransaction"

General outline:

Get a transaction by id from the blockchain.

Key arguments:

1. "txid" (string, required) The transaction id.
2. verbose (bool, optional, default=false) If false, return a string, otherwise return a json object.

Output:

A transaction id.

Function – "signrawtransaction"

General outline:

Sign inputs for raw transaction (serialized, hex-encoded).

Key arguments:

1. "hexstring"(string, required) The transaction hex string.
2. "prevtxs" (string, optional) An json array of previous dependent transaction outputs.

 [(json array of json objects, or 'null' if none provided) {
"txid":"id", (string, required) The transaction id.
"vout":n, (numeric, required) The output number.
"scriptPubKey":"hex",(string, required) script key.
"redeemScript":"hex", (string, required) redeem script.
"amount": value (numeric, required) The amount spent}]

Output:

The hex-encoded raw transaction with signature.

Function – “createmultisig”

General outline:

Creates a multi-signature address with n signature of m keys required. It returns a json object with the address and redeem Script.

Key arguments:

1. nrequired (numeric, required) The number of required signatures out of the n keys.
2. "keys" (string, required) A json array of hex-encoded public keys. ["key" (string) The hex-encoded public key]

Output:

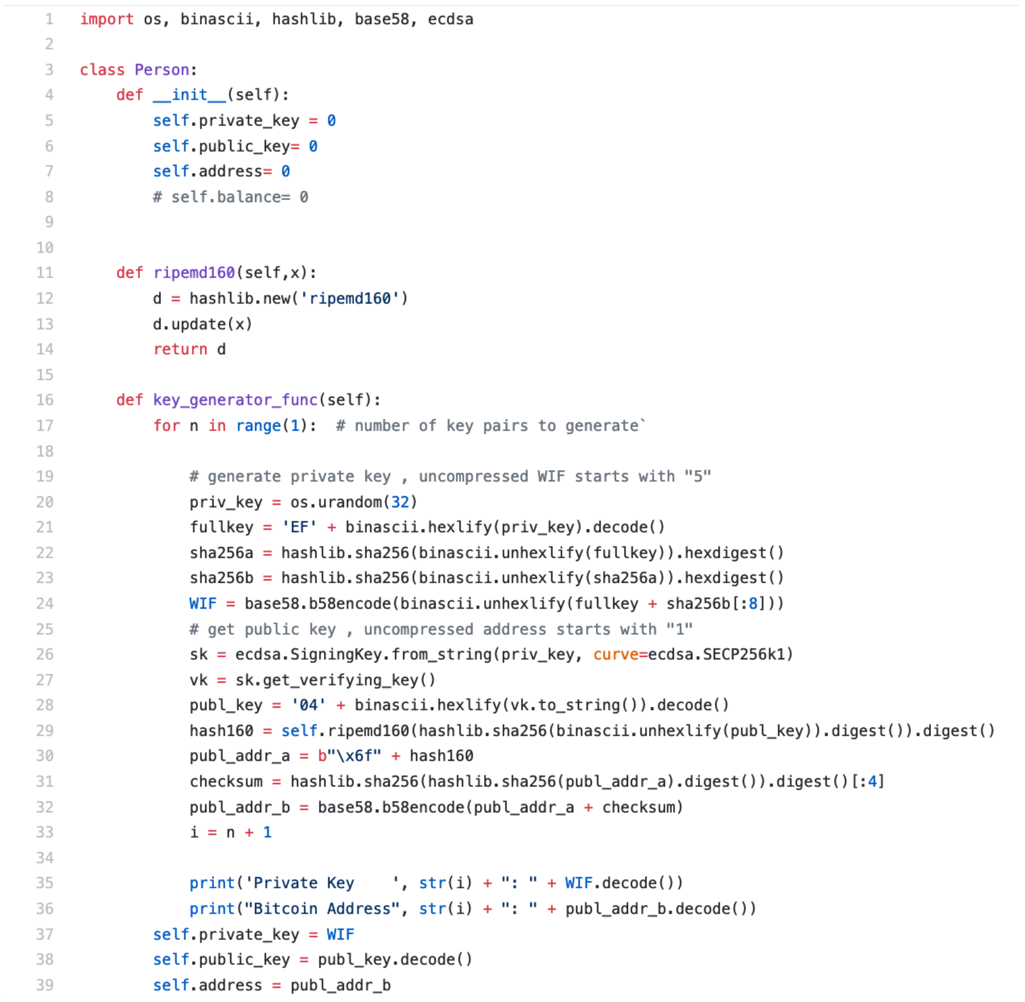
True/false.

Code Section

We used GitHub to sync our project with Itay and ourselves.
The entire project and its libraries can be found on the following link:
https://github.com/BarakGahtan/bitcoin_project_s19

We now present several key functions which we implemented.
Each function will have the written code, a summary and explanation for input and output.

Function: "Private Key generator"



```python
import os, binascii, hashlib, base58, ecdsa

class Person:
    def __init__(self):
        self.private_key = 0
        self.public_key= 0
        self.address= 0
        # self.balance= 0


    def ripemd160(self,x):
        d = hashlib.new('ripemd160')
        d.update(x)
        return d

    def key_generator_func(self):
        for n in range(1):  # number of key pairs to generate`

            # generate private key , uncompressed WIF starts with "5"
            priv_key = os.urandom(32)
            fullkey = 'EF' + binascii.hexlify(priv_key).decode()
            sha256a = hashlib.sha256(binascii.unhexlify(fullkey)).hexdigest()
            sha256b = hashlib.sha256(binascii.unhexlify(sha256a)).hexdigest()
            WIF = base58.b58encode(binascii.unhexlify(fullkey + sha256b[:8]))
            # get public key , uncompressed address starts with "1"
            sk = ecdsa.SigningKey.from_string(priv_key, curve=ecdsa.SECP256k1)
            vk = sk.get_verifying_key()
            publ_key = '04' + binascii.hexlify(vk.to_string()).decode()
            hash160 = self.ripemd160(hashlib.sha256(binascii.unhexlify(publ_key)).digest()).digest()
            publ_addr_a = b"\x6f" + hash160
            checksum = hashlib.sha256(hashlib.sha256(publ_addr_a).digest()).digest()[:4]
            publ_addr_b = base58.b58encode(publ_addr_a + checksum)
            i = n + 1

            print('Private Key    ', str(i) + ": " + WIF.decode())
            print("Bitcoin Address", str(i) + ": " + publ_addr_b.decode())
        self.private_key = WIF
        self.public_key = publ_key.decode()
        self.address = publ_addr_b
```

First, we identified if the address was intended for the Bitcoin mainnet, testnet or regtest.
In our case, the address was created for the Regtest.
There are three configurations

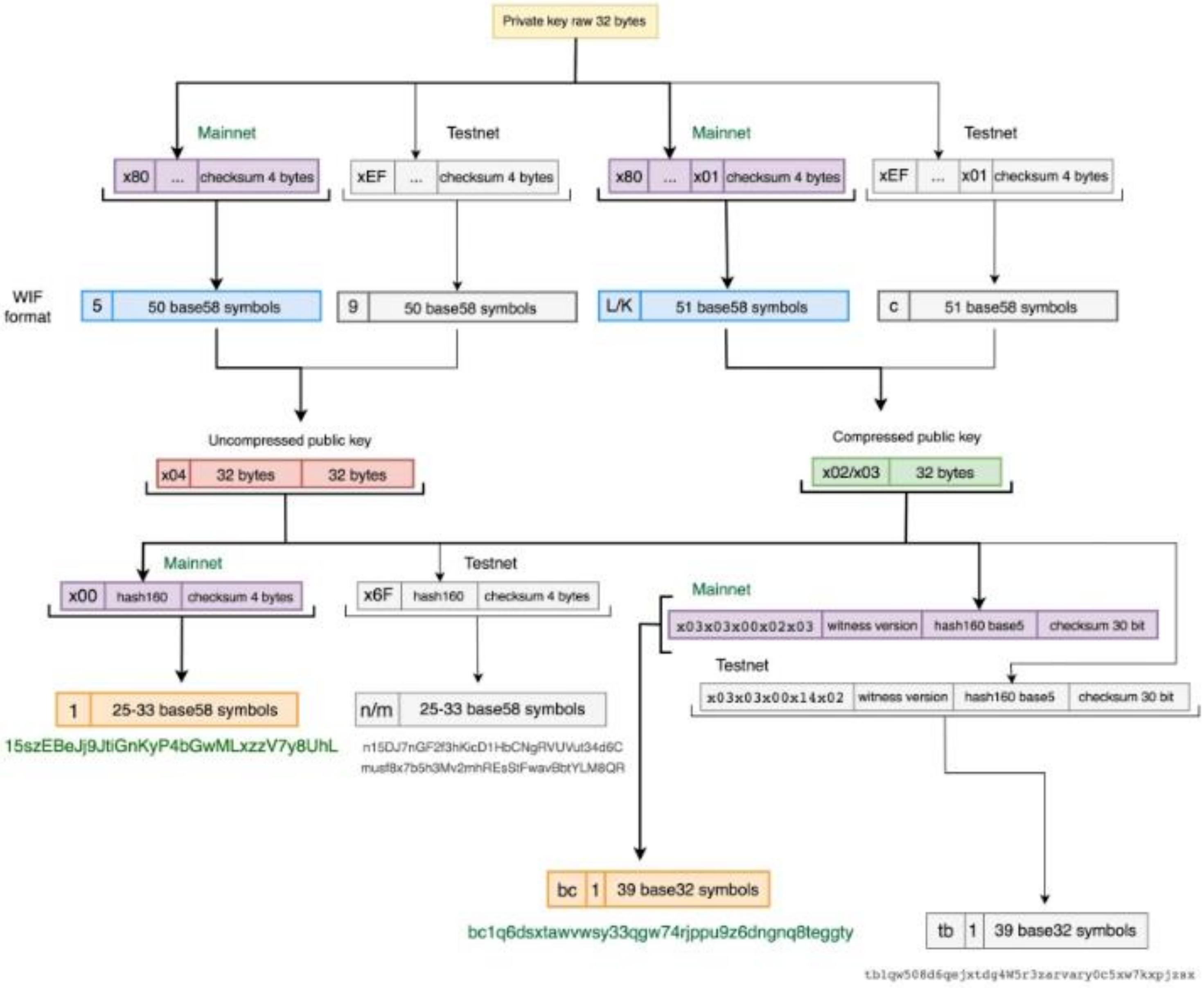
1. Mainnet's address should start with "0x00".
2. Testnet's address should start with "0x6F".
3. Regtest's address should start like Testnet.

A serialized bitcoin address consists of:

1. 1-Byte Version prefix.
2. 20 byte hash digest – double hashed public key.
3. 4 byte checksum.

The serialized format is encoded in Base58. This information is later required to recreate a Bitcoin address from a given private key.

In addition, both the private and public key should start with 0x04 on both the "mainnet" and the test net. Therefore, as shown above in line 28 the prefix of "04" is added to the key.
As shown above in line number 30, the prefix of the "Testnet". The addition of "x6f".
Below you can see an illustration of the configuration for each to the private keys for each mode of Bitcoin.
As can be seen, the public key is derived from the private key.



Private key raw 32 bytes
Mainnet
Testnet
Mainnet
Testnet
x80 | ... | checksum 4 bytes
xEF | ... | checksum 4 bytes
x80 | ... | x01 | checksum 4 bytes
xEF | ... | x01 | checksum 4 bytes
WIF format
5 | 50 base58 symbols
9 | 50 base58 symbols
L/K | 51 base58 symbols
c | 51 base58 symbols
Uncompressed public key
x04 | 32 bytes | 32 bytes
Compressed public key
x02/x03 | 32 bytes
Mainnet
Testnet
x00 | hash160 | checksum 4 bytes
x6F | hash160 | checksum 4 bytes
Mainnet
x03x03x00x02x03 | witness version | hash160 base5 | checksum 30 bit
Testnet
x03x03x00x14x02 | witness version | hash160 base5 | checksum 30 bit
1 | 25-33 base58 symbols
15szEBeJj9JtiGnKyP4bGwMLxzzV7y8UhL
n/m | 25-33 base58 symbols
n15DJ7nGF2f3hKicD1HbCNgRVUVut34d6C
musf8x7b5h3Mv2mhREsStFwavBbtYLM8QR
bc | 1 | 39 base32 symbols
bc1q6dsxtawvwsy33qgw74rjppu9z6dngnq8teggty
tb | 1 | 39 base32 symbols
tb1qw508d6qejxtdg4W5r3zarvary0c5xw7kxpjzsx

Function: “transaction Single To”

From a generated block to another address

A function that transfers bitcoins from a generated block to another address using Bitcoin's library functions.
There are three main steps:

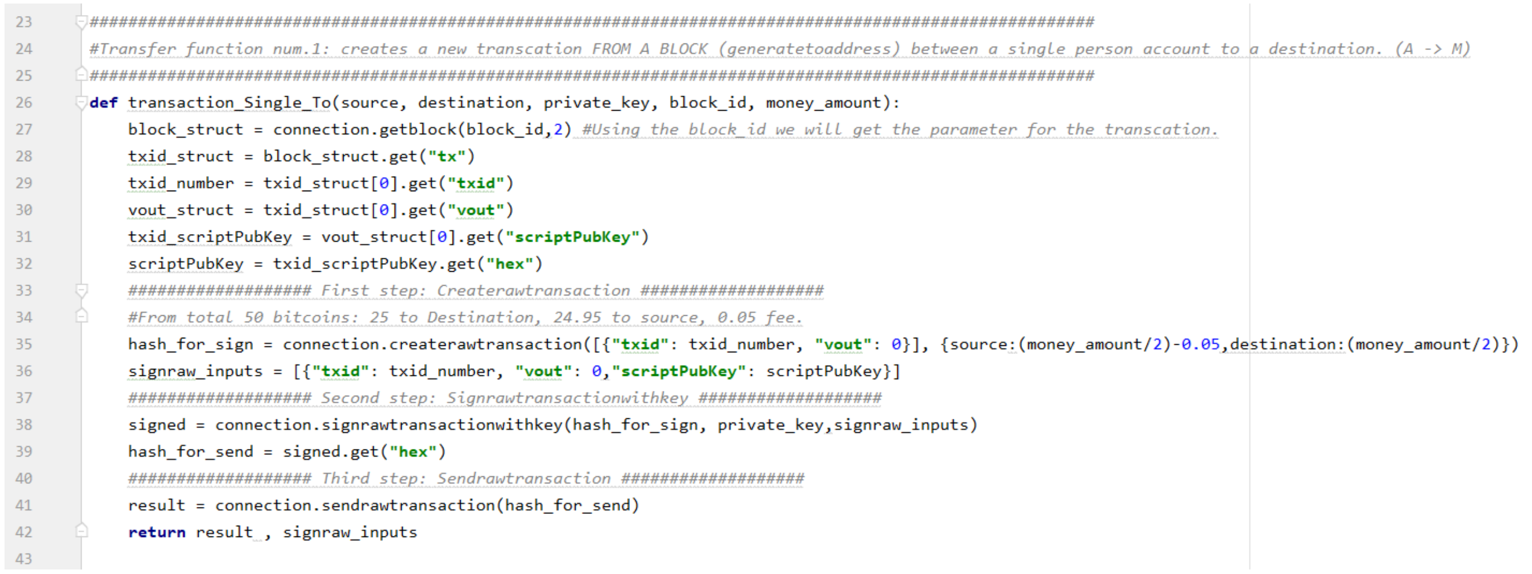
- We extract the transaction’s id (txid) from the “block_id” input. Using the txid and coins amount we can use the method “createrawtransaction”.
- we extract the scriptPubKey and txid number of the block from block_id input. In addition, we save the hash which is returned from “createrawtransaction”. After extracting those inputs, we would call the method “signrawtransactionwithkey”.
- The return hash from “signrawtransactionwithkey” will be used as input to “sendrawtransaction” method.

Key arguments:

1. "source"(string). Address of the sending address.
2. "destination"(string). Address of the receiving address.
3. "private_key" (String). The private key of the source address.
4. "block_id" (Array of blocks). Output from last transaction – the bitcoins for current transaction.
5. "money_amount" (double). The amount of money we have – will split between the sender, receiver. The spare will be a fee for the miner.

Outputs:

1. "result" (bool). A bool parameter indicates if the whole transaction went successfully or not.
2. "signraw_inputs" (Array of strings). Array the has inputs for “signrawtransactionwithkey” function, such as “txid_number”, “vout” and “scriptPubKey”.



```python
####################################################################################################
#Transfer function num.1: creates a new transcation FROM A BLOCK (generatetoaddress) between a single person account to a destination. (A -> M)
####################################################################################################
def transaction_Single_To(source, destination, private_key, block_id, money_amount):
    block_struct = connection.getblock(block_id,2) #Using the block_id we will get the parameter for the transcation.
    txid_struct = block_struct.get("tx")
    txid_number = txid_struct[0].get("txid")
    vout_struct = txid_struct[0].get("vout")
    txid_scriptPubKey = vout_struct[0].get("scriptPubKey")
    scriptPubKey = txid_scriptPubKey.get("hex")
    ################## First step: Createrawtransaction ##################
    #From total 50 bitcoins: 25 to Destination, 24.95 to source, 0.05 fee.
    hash_for_sign = connection.createrawtransaction([{"txid": txid_number, "vout": 0}], {source:(money_amount/2)-0.05,destination:(money_amount/2)})
    signraw_inputs = [{"txid": txid_number, "vout": 0,"scriptPubKey": scriptPubKey}]
    ################## Second step: Signrawtransactionwithkey ##################
    signed = connection.signrawtransactionwithkey(hash_for_sign, private_key,signraw_inputs)
    hash_for_send = signed.get("hex")
    ################## Third step: Sendrawtransaction ##################
    result = connection.sendrawtransaction(hash_for_send)
    return result , signraw_inputs
```

Function: "transaction Single To Single"

From a single address to another single address

A function that transfers bitcoins from a single address to another address using Bitcoin's library functions.
There are three main steps:

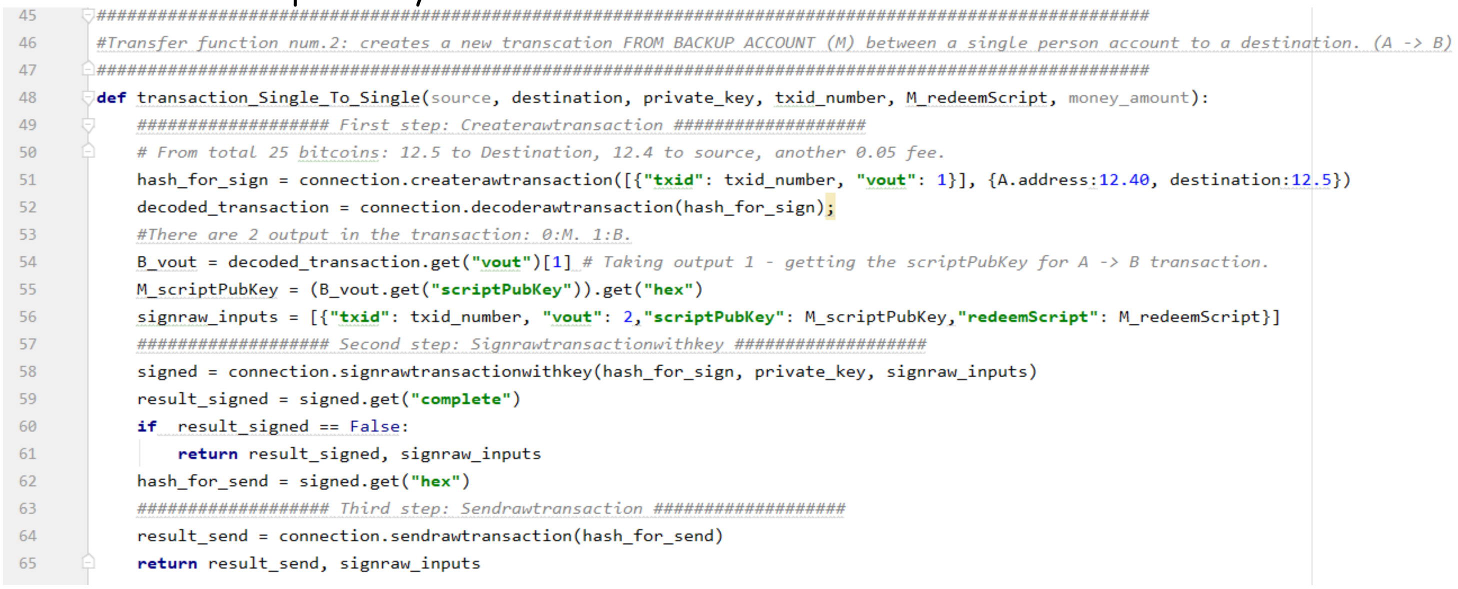
- We extract the transaction's id (txid) from the "block_id"'s input. Using the txid and money amount we can use the method "createrawtransaction".
- The returned argument from the first step is the hash for "signrawtransactionwithkey" method. From there we can extract the "scriptPubKey".
  Now, using "txid_number", "scriptPubKey" and redeem script we can use the method "signrawtransactionwithkey".
- The returned hash from "signrawtransactionwithkey", is the input for the method "sendrawtransaction".

Key arguments:

1. "source"(string). Address of the sending address.
2. "destination"(string). Address of the receiving address.
3. "private_key" (String). The private key of the source's address.
4. "txid_number" (int). txid of the block. From that block we get the inputs – the bitcoins for current transaction.
5. "M_redeemScript" (string). The redeem Script of our multisig address.
6. "money_amount" (double). The amount of money we have – will split between the sender, receiver. The spare will be a fee for the miner.

Outputs:

1. "result" (bool). A bool parameter indicates if the whole transaction went successfully or not.
2. "signraw_inputs" (Array of strings). Array the has inputs for "signrawtransactionwithkey" function, such as_"txid_number", "vout" and "scriptPubKey".



```python
######################################################################################
#Transfer function num.2: creates a new transcation FROM BACKUP ACCOUNT (M) between a single person account to a destination. (A -> B)
######################################################################################
def transaction_Single_To_Single(source, destination, private_key, txid_number, M_redeemScript, money_amount):
    ################### First step: Createrawtransaction ###################
    # From total 25 bitcoins: 12.5 to Destination, 12.4 to source, another 0.05 fee.
    hash_for_sign = connection.createrawtransaction([{"txid": txid_number, "vout": 1}], {A.address:12.40, destination:12.5})
    decoded_transaction = connection.decoderawtransaction(hash_for_sign);
    #There are 2 output in the transaction: 0:M. 1:B.
    B_vout = decoded_transaction.get("vout")[1] # Taking output 1 - getting the scriptPubKey for A -> B transaction.
    M_scriptPubKey = (B_vout.get("scriptPubKey")).get("hex")
    signraw_inputs = [{"txid": txid_number, "vout": 2,"scriptPubKey": M_scriptPubKey,"redeemScript": M_redeemScript}]
    ################### Second step: Signrawtransactionwithkey ###################
    signed = connection.signrawtransactionwithkey(hash_for_sign, private_key, signraw_inputs)
    result_signed = signed.get("complete")
    if  result_signed == False:
        return result_signed, signraw_inputs
    hash_for_send = signed.get("hex")
    ################### Third step: Sendrawtransaction ###################
    result_send = connection.sendrawtransaction(hash_for_send)
    return result_send, signraw_inputs
```

Function: "transaction Multi To Single"

From a multisig address to another single address

A function that transfers bitcoins from a multisig address to another single address using Bitcoin's library functions.
There are three main steps:

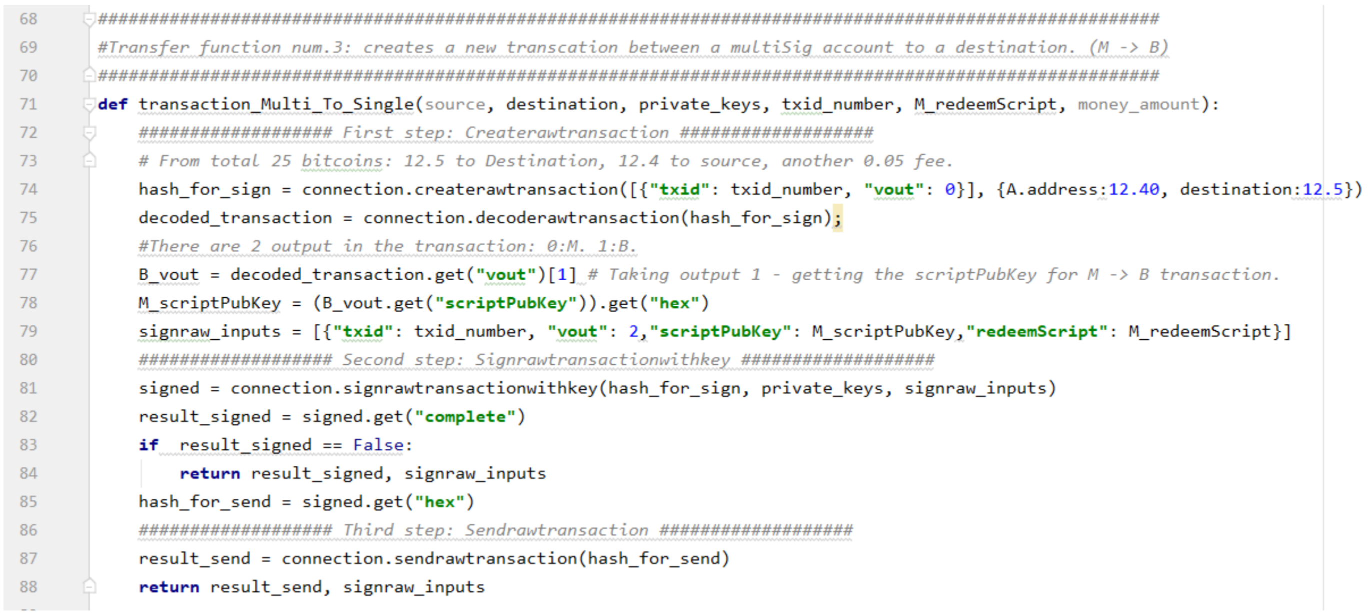
- We extract the transaction's id (txid) from the "block_id"'s input. Using the txid and money amount we can use the method "createrawtransaction".
- The returned argument from the first step is the hash for "signrawtransactionwithkey" method. From there we can extract the "scriptPubKey".
  Now, using "txid_number", "scriptPubKey" and redeem script we can use the method "signrawtransactionwithkey".
- The returned hash from "signrawtransactionwithkey", is the input for the method "sendrawtransaction".

Key arguments:

1. "source"(string). Address of the sender.
2. "destination"(string). Address of destination.
3. "private_keys" (String). The privates key of the source address (the multisig address).
4. ""txid_number" (int). txid of the block. From that block we get the inputs – the bitcoins for current transaction.
5. "M_redeemScript" (string). The redeem Script of the multisig address.
6. "money_amount" (double). The amount of money we have – will split between the sender, receiver. The spare will be a fee for the miner.

Outputs:

1. "result" (bool). A bool parameter indicates if the transaction was successfull.
2. "signraw_inputs" (Array of strings). Inputs for "signrawtransactionwithkey" function, such as_"txid_number", "vout" and "scriptPubKey".



```python
#########################################################################################
#Transfer function num.3: creates a new transcation between a multiSig account to a destination. (M -> B)
#########################################################################################
def transaction_Multi_To_Single(source, destination, private_keys, txid_number, M_redeemScript, money_amount):
    ################### First step: Createrawtransaction ###################
    # From total 25 bitcoins: 12.5 to Destination, 12.4 to source, another 0.05 fee.
    hash_for_sign = connection.createrawtransaction([{"txid": txid_number, "vout": 0}], {A.address:12.40, destination:12.5})
    decoded_transaction = connection.decoderawtransaction(hash_for_sign);
    #There are 2 output in the transaction: 0:M. 1:B.
    B_vout = decoded_transaction.get("vout")[1] # Taking output 1 - getting the scriptPubKey for M -> B transaction.
    M_scriptPubKey = (B_vout.get("scriptPubKey")).get("hex")
    signraw_inputs = [{"txid": txid_number, "vout": 2,"scriptPubKey": M_scriptPubKey,"redeemScript": M_redeemScript}]
    ################### Second step: Signrawtransactionwithkey ###################
    signed = connection.signrawtransactionwithkey(hash_for_sign, private_keys, signraw_inputs)
    result_signed = signed.get("complete")
    if  result_signed == False:
        return result_signed, signraw_inputs
    hash_for_send = signed.get("hex")
    ################### Third step: Sendrawtransaction ###################
    result_send = connection.sendrawtransaction(hash_for_send)
    return result_send, signraw_inputs
```

Conclusion

We started the project without knowing anything about Bitcoin nor blockchain.
Throughout the project, we learnt on the fly many aspects of blockchain. We found it to be a fascinating subject to study.
For the first time we got to work with RPC protocols. RPC protocols communicate with remote servers and commit to them tasks to be computed. We used RPC to communicate with the Bitcoin core software.
We built a system which that enables zero confirmation time of transactions. The system allows an immediate, on demand purchase between two parties to occur, while dealing with double spending problem. The system ensures that both parties will receive their due.
We managed our project on the GitHub website, and worked on it remotely together. We found it to be a very easy and friendly way to implement and share methods.

Bibliography

https://bitcoincore.org/en/doc/0.17.0/rpc/
https://en.wikipedia.org/wiki/Bitcoin#Blockchain
https://en.wikipedia.org/wiki/Blockchain
https://www.investopedia.com/terms/b/bitcoin-mining.asp
https://github.com/libbitcoin/libbitcoin-system/wiki/Addresses-and-HD-Wallets