A Centralized Blockchain Implementation using RPC

CS403/534 - Distributed Systems
Programming Assignment (PA) #2, Fall 2021

Deadline: November 8, 2021 (23:55 UTC+3:00)

In this PA, you are asked to implement a blockchain class of which operations could be called remotely using RPC mechanisms we have seen before. Contrary to the nature and purpose of blockchains, your implementation will be centralized. Every blockchain instance will be run by a particular server of which opearations could be called remotely using proxies by clients (or other servers if necessary).

The implementation should be done in Python 3 using Pyro4. You can find installation instructions here: `https://pyro4.readthedocs.io/en/stable/install.html`.

1 `MyBlockChain` Class

The core of the blockchain implementation is `MyBlockChain` class. Its main state is basically a linked-list. Each node of the linked-list is a block. So, you need to first implement a class named `MyBlock` to be used by your `MyBlockChain` class.

The state of the `MyBlock` consists of a transaction and a pointer to the next block. The transaction attribute is a tuple of the form: $(txType, args)$ where *txType* is a string identifying the transaction type and *args* is a tuple denoting the parameters of the transaction. For each transaction type, the number of argumnets (size of the tuple) and their interpretation is different. Transaction types and their arguments will be explained in detail later. Of course, the next block pointer should point to an object of type `MyBlock`. In order to adhere to the spirit of blockchains, the transaction and the next pointer attributes must be set during the construction of the block and they should never change afterwards. So, `MyBlock` class might have getter functions for its attributes but it must not have any setter function.

The state of the `MyBlockChain` class contains at least two attributes: a `MyBlock` pointer called `head` that points to the latest block in the chain and a string called `chainName` that can be used for registering a `MyBlockChain` object to an RPC daemon or a name server. When a `MyBlockChain` instance is generated using the constructor you have written, `head` must point to `None` and `chainName` must be taken as an argument from the caller. `MyBlockChain`

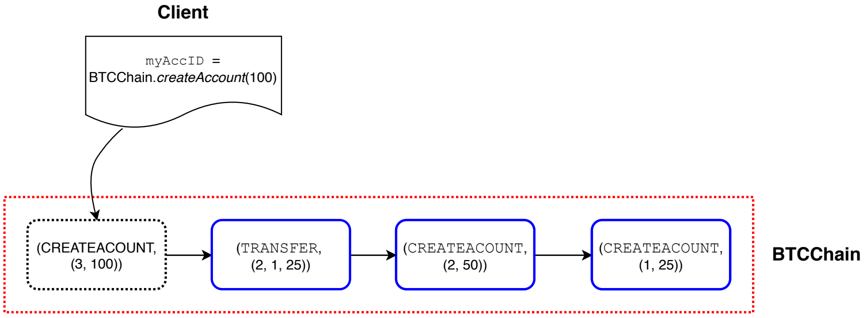


Figure 1: *createAccount* operation. A client adds a new account to the *BTC-Chain* with 100 initial balance. New account's account number becomes 3.

state might contain other attributes that you might find necessary, but above mentioned attributes are compulsory.

`MyBlockChain` class has different operations for different transaction types and these operations can be called remotely. Basically, each transaction operation takes arguments of the transaction as input, performs some validity checks for the transaction and if checks pass, it forms a new block containing this transaction and appends it to the head of the chain (see the figures).

To understand the details of possible transaction types and their validity checks, it is better from now on to consider a particular usage of blockchains called cryptocurrencies. Basically, transactions we consider keep track of accounts and money transfers among them.

Below are the descriptions of different transaction operations you need to implement for RPC.

- *createAccount*: For this operation, its input is an integer *amount* where *amount* is the initial amount of money deposited into the new account.

  This operation traverses the chain, checks all the account numbers seen in the transactions and finds the maximum of them. Let us call it *max*. Then, the account number for the new account becomes $max + 1$.

  Next, this operation generates a new block with the transaction value (`CREATEACCOUNT`, $(max+1, amount)$), appends it to the chain and returns the new account number to the caller. You might see a pictorial description of this operation in Figure 1.

- *transfer*: This operation's input parameters are $from$, *to* and *amount* where $from$ and *to* are two account numbers and *amount* is the transfer money amount.

  If *amount* is positive, it is deduced from the $from$ account and added to the *to* account. However, it is also possible that *amount* is negative. In this case, it should be interpreted as deducing $|amount|$ from *to* account and adding $|amount|$ to $from$ account.

  For this transaction type, the operation first checks existence of $from$ and *to* accounts by traversing the chain and checking if there is any transaction mentioning these accounts. Of course, just checking `CREATEACCOUNT` transactions are sufficient for this purpose. If one of these accounts is not created before, it should return -1.

Next, it should check whether the sending account has enough money. To do this, first you need to determine which account is sending and then traverse the chain for this account, consider its initial deposit amount, transfers it has performed and exchanges it has done with other chains (see the next transaction type) and calculate the balance of the account during the traversal. If the sending account does not have enough balance, again this operation returns -1.

If checks pass, `insertTx` generates a new block with transaction value (`TRANSFER`, $(from, to, amount)$), appends it to the chain and returns 1 denoting its success.

- *exchange*: This operation's input is of the form $from, to, toChain, amount$ where $from$ is an account in the current chain, *to* is an account in the other chain *toChain* and *amount* is the transferred amount. For the sake of simplicity, we assume that the exchange rate is 1, meaning that one unit of host chain currency corresponds to one unit of foreign chain currency. Similar to `TRANSFER` transactions, *amount* can be negative meaning that $|amount|$ must be reduced from *to* account in *toChain* and $|amount|$ is added to the $from$ account in the host chain.

  Validity checks of `EXCHANGE` is similar to `TRANSFER` as well. For this transaction type, the first check is if $from$ account exists in the host chain and *to* account exists in *toChain*. Next, it is checked whether the account that will lose money has enough balance. If *amount* is negative, this check is performed on *toChain* for *to* account. Of course, checks on *toChain* are performed by calling appropriate functions remotely on *toChain*. If any of the validity checks fail, *exchange* operation returns -1.

  If all the validity checks pass, *exchange* generates a transaction of the form (`EXCHANGE`, $(from, to, toChain, amount)$) and appends a block containing this transaction to the current chain. Lastly, it generates another transaction (`EXCHANGE`, $(to, from, currentChain, -amount)$), puts it in a block, appends this block to the *toChain* and returns 1. This behaviour is described in Figure 2.

  Note that the change performed in *toChain* must be reflected on the original object. **So, *toChain* can not be passed by value.**

- *calculateBalance*: This operation is not for generating a transaction but it is a helper function for transaction validity checks and it should be provided to client as a service. This operation takes an account number as input and calculates the balance of this account by considering the initial money deposited, transfers and exchange transactions from and to this account.

  printChain: Prints the transactions of all blocks in order to the console.

Since we interpreted blockchain as a cryptocurrency, it includes concepts like account and balance. You might keep them as attributes of `MyBlockChain` class to simplify some of the validity checks that otherwise requires traversal over the

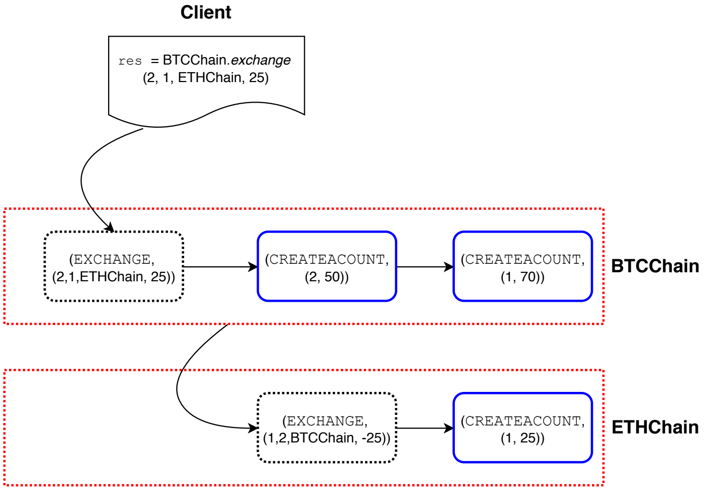


Figure 2: *exchange* operation. This operation is called for one chain. This single call performs validity checks on both chains and if they pass it adds two blocks to the chains as depicted.

chain. Although keeping these extra attributes are allowed, it is preferred that you do not keep them explicitly to have a better block-chain simulation.

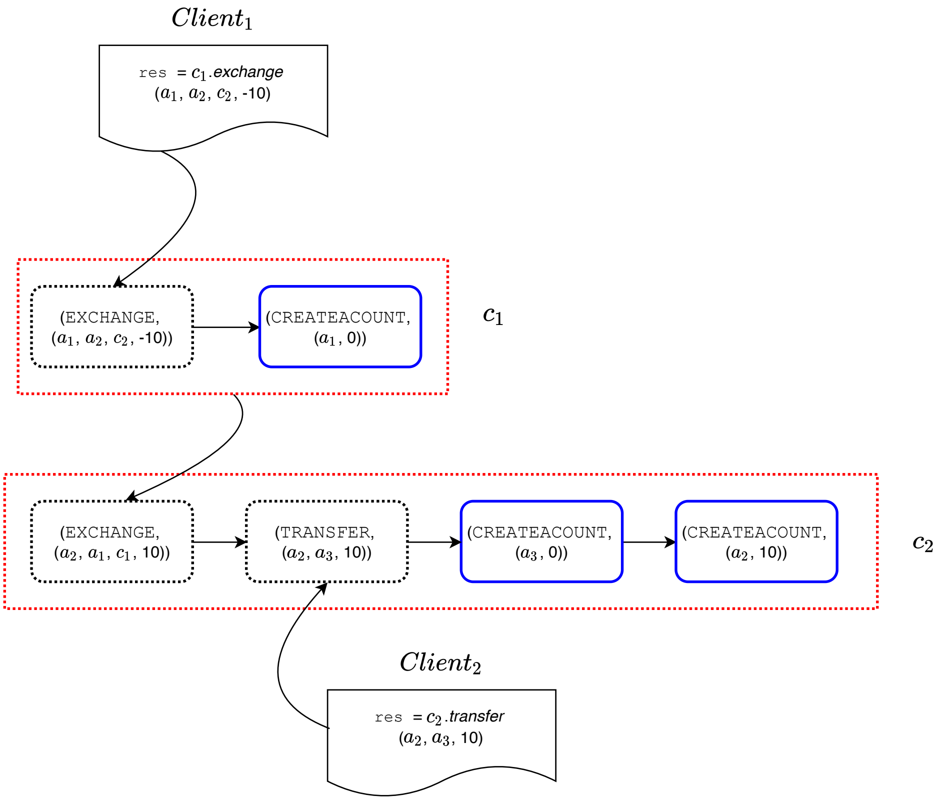
Apart from these methods, you might implement helper methods in the `MyBlockChain` class as you see necessary. For instance, you might write a method to check whether an account exists in the chain or for directly adding a block with transaction type `EXCHANGE` to the chain to help host chain to insert a block to the remote chain while processing an `EXCHANGE` transaction.

Registering chains to Pyro4. In order to do the block-chain simulation correctly, each chain instance must be registered to a Pyro4 daemon and/or name server with the string name given to the constructor. This registration can be made in multiple places like in the server code after creating the chain object or in the constructor of the chain. The place of registration is practically not important but you need to make sure that when a client asks for the proxy of a chain or gives the proxy as a parameter to a remote procedure, this object must have been already registered to the system. As it is stated in the submission guidelines section, our sample server files assume that the registration is done in the constructor but you are free to modify them and provide your own server and client files.

1.1 Bonus: Making Transactions Atomic

Transaction is a well-known term in Computer Science literature. A transaction consists of a sequence of operations that appear to be executed atomically (in sequence without interruption). If we ignore `EXCHANGE` transactions, `MyBlockChain` methods are already atomic since servers handle client requests sequentially.

However, `EXCHANGE` operations violate atomicity. Consider the case in Figure 3 with two chains c_1 and c_2. Assume that initially there is an account a_1 of



$Client_1$
res $= c_1$.exchange (a_1, a_2, c_2, -10)
(EXCHANGE, (a_1, a_2, c_2, -10))
(CREATEACOUNT, (a_1, 0))
c_1
(EXCHANGE, (a_2, a_1, c_1, 10))
(TRANSFER, (a_2, a_3, 10))
(CREATEACOUNT, (a_3, 0))
(CREATEACOUNT, (a_2, 10))
c_2
$Client_2$
res $= c_2$.transfer (a_2, a_3, 10)

Figure 3: A non-atomic execution case for *exchange*. When it is executed concurrently with *transfer*, it might generate negative balance. In this example, balance of a_2 becomes negative after transactions.

c_1 with 0 balance and another account a_2 of c_2 with balance 10. Suppose that a client $Client_1$ wants to perform an *exchange* with parameters $(a_1, a_2, c_2, -10)$ on c_1 and concurrently another client $Client_2$ performs a *transfer* with the input parameters $(a_2, a_3, 10)$ on c_2 where a_3 is another existing account number in c_2.

Now, consider the following scenario: Assume that $Client_1$ remotely calls *exchange* on c_1. The server of c_1 performs the validity checks and sees that a_2 has enough balance. Then, `EXCHANGE` transaction block is appended on c_1. Then, assume that $Client_2$ starts its execution and performs the remote call *transfer*. The server of c_2 receives this request, executes and appends the block with `TRANSFER` to its chain since validity checks also pass for this transaction. At the end, server of c_1 remotely asks server of c_2 to append the `EXCHANGE` transaction block to its chain which must be realized by the c_2 server since the corresponding block has already been appended to c_1. However, balance of a_2 becomes negative now although both transactions checked that a_2 had enough balance before the transaction.

This problem occurred because `EXCHANGE` transaction did not execute atomically. One needed to ensure that while this transaction was being processed by c_1 server, c_2 server must not let `TRANSFER` transaction to be processed. This might be realized by using shared-memory synchronization mechanisms like locks. So, if you want to attempt this bonus part after completing the PA, you might add synchronization attributes to the `MyBlockChain` class and modify transaction operations using these synchronization primitives. Of course, while using synchronization, you have to be careful about deadlocks, especially the ones similar to deadlocks occurring in dining-philosophers problem.

2 Submission Guidelines

In the PA2 materials, you are provided with two client and two server PYTHON files. You might use these files to test your `MyBlockChain` class implementation. Server test files just create instances of `MyBlockChain` and do not interact with PYRO4. This is the case because in our reference implementation, PYRO4 daemon is generated, the object is registered to the daemon and the name server with *chainName* attribute of `MyBlockChain` and the daemon starts to serve in the constructor of `MyBlockChain` (`__init__` function). This is a design choice. You might not do these steps in the constructor but explicitly perform in the server file. If you decide to do so, please add the modified test files to your submission. Otherwise you can just submit the ones provided without modification.

You need to submit following files:

- `MyBlockChain.py` contains `MyBlockChain` and `MyBlock` class implementations.
- `BTCServer.py`, `ETHServer.py`, `firstClient.py` and `secondClient.py` are test server and client implementations.

Required files explained above should be put in a single zip file named as `CS_403-534_PA02_name_surname.zip`. Submission will be open until November 8, 2021 23:55 Turkish time.

3 Grading

- PYRO4 usage (30 pts): To get full points, you need to correctly register chain objects to daemons and/or name servers and proxies must be registered before they are used.
- *calculateBalance* procedure (10 pts)
- *createAccount* procedure (15 pts)
- *transfer* procedure (15 pts)
- *exchange* procedure (30 pts)
- BONUS (20 pts): correct synchronization (10 pts), no deadlock (10 pts)