Archiving Financial Data on Ethereum Blockchain

Bachelor's Thesis

CMPE492 SPRING 2018 - FINAL REPORT

4 JUNE 2018

FURKAN ERDEM PERŞEMBE, CEYHUN UZUNOĞLU

Ph.D. Professor Can Özturan

Contents

1. Introduction and Motivation

Decentralized systems have always been a question and also a chance for humanity. As a decentralized system, blockchain had created an enormous interest in people at the present time. There are many reasons behind this interest, i.e. high price increase in crypto currencies. However from the point of computer science, blockchain is highly valuable software architecture.

In our capstone project, we are developing a smart contract that holds currency data from TCMB(The Central Bank of Turkey) and ECB(European Central Bank) in Ethereum blockchain network. Our smart contract keeps all past currency data from these two banks, so people may use these currency data by consuming our smart contract.

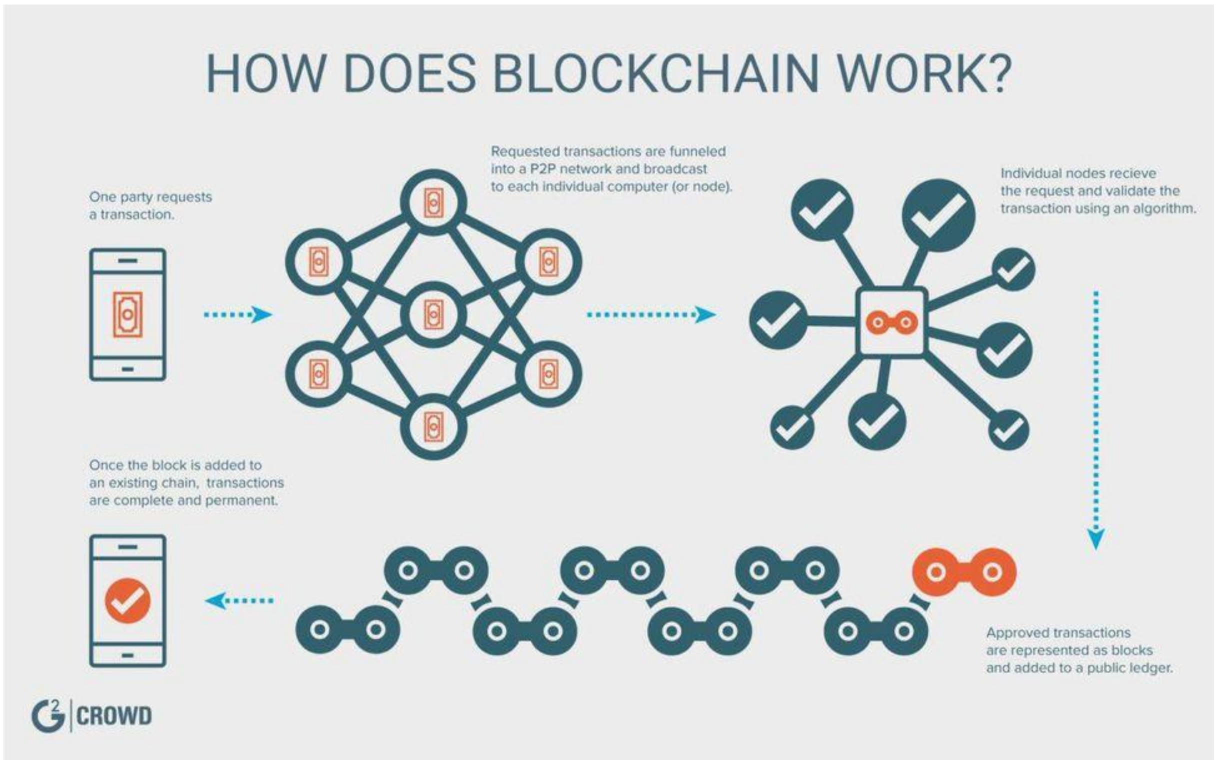
It will be mentioned later, but with the aim of clear understanding, smart contracts cannot retrieve data from outside of the blockchain network. Smart contracts can only get data from other smart contracts or its creator.

Blockchain has its own currencies and people should convert their own country money to blockchain currency vise versa. Almost all transaction and operation in blockchain network related about money, hence they are all related with currency values.

Because of these reasons, in this project we want to offer trustable currency data for all people in Ethereum blockchain network with minimum fees. People can retrieve any data from 1999 to until today, can convert currencies in any date between 1999 and today. The reason of the date 1999, TCMB and ECB gives currency data till that date. Our initial focus is Turkish users, however all people in around the world can use our smart contract trustfully.

2. State of the Art

2.1 What is Blockchain ?


HOW DOES BLOCKCHAIN WORK?
One party requests a transaction.
Requested transactions are funneled into a P2P network and broadcast to each individual computer (or node).
Individual nodes recieve the request and validate the transaction using an algorithm.
Approved transactions are represented as blocks and added to a public ledger.
Once the block is added to an existing chain, transactions are complete and permanent.
G2 CROWD

A blockchain is a continuously growing list of records, called blocks, which are linked and secured using cryptography. Each block typically contains a cryptographic hash of the previous block, a timestamp and transaction data. By design, a blockchain is inherently resistant to modification of the data. It is "an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way". For use as a distributed ledger, a blockchain is typically managed by a peer-to-peer network collectively adhering to a protocol for validating new blocks. Once recorded, the data in any given block cannot be altered retroactively without the alteration of all subsequent blocks, which requires collusion of the network majority.

Blockchains are secure by design and exemplify a distributed computing system with high Byzantine fault tolerance. Decentralized consensus has therefore been achieved with a blockchain. This makes blockchains potentially suitable for the recording of events, medical records, and other records management activities, such as identity management, transaction processing, documenting provenance, food traceability or voting.

Blockchain was invented by Satoshi Nakamoto in 2008 for use in the cryptocurrency bitcoin, as its public transaction ledger. The invention of the blockchain for bitcoin made it the first digital currency to solve the double spending problem without the need of a trusted authority or central server. The bitcoin design has been the inspiration for other applications.

2.2 Ethereum Blockchain and its Features

Ethereum is an open-source, public, blockchain-based distributed computing platform and operating system featuring smart contract (scripting) functionality. It supports a modified version of Nakamoto consensus transaction based state transitions. In popular discourse, the term Ethereum is often used interchangeably with Ether to refer to the cryptocurrency that is generated on the Ethereum platform.

Ether is a cryptocurrency whose blockchain is generated by the Ethereum platform. Ether can be transferred between accounts and used to compensate participant mining nodes for computations performed. Ethereum provides a decentralized Turing-complete virtual machine, the Ethereum Virtual Machine (EVM), which can execute scripts using an international network of public nodes. "Gas", an internal transaction pricing mechanism, is used to mitigate spamand allocate resources on the network.

Ethereum was proposed in late 2013 by Vitalik Buterin, a cryptocurrency researcher and programmer. Development was funded by an online crowdsale that took place between July and August 2014. The system went live on 30 July 2015, with 11.9 million coins "premined" for the crowdsale. This accounts for approximately 13 percent of the total circulating supply.

In 2016, as a result of the collapse of The DAO project, Ethereum was split into two separate blockchains – the new separate version became Ethereum (ETH), and the original continued as Ethereum Classic (ETC). The value of the Ethereum currency grew over 13,000 percent in 2017.

2.3 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is the runtime environment for smart contracts in Ethereum. It is a 256-bit register stack, designed to run the same code exactly as intended. It is the fundamental consensus mechanism for Ethereum. The formal definition of the EVM is specified in the Ethereum Yellow Paper. It is sandboxed and also completely isolated from the network, filesystem or other processes of the host computer system. Every Ethereum node in the network runs an EVM implementation and executes the same instructions. In February 1, 2018, there were 27,500 nodes in the main Ethereum network. Ethereum Virtual Machines have been implemented in C++, Go, Haskell, Java, JavaScript, Python, Ruby, Rust, and WebAssembly (currently under development). The Ethereum-flavoured WebAssembly (dubbed "e-WASM") is expected to become a major component of the "Web 3.0", a World Wide Web where users interact with smart contracts through a browser.

2.4 Smart contracts

Ethereum's smart contracts are based on different computer language, which developers use to program their own functionalities. Smart contracts are high-level programming abstractions that are compiled down to EVM bytecode and deployed to the Ethereum blockchain for execution. They can be written in Solidity (a language library with similarities to C and JavaScript), Serpent (similar to Python, but deprecated), LLL (a low-level Lisp-like language), and Mutan (Go-based, but deprecated). There is also a research-oriented language under development called Viper (a strongly-typed Python-derived decidable language).

There is ongoing research on how to use formal verification to express and prove non-trivial properties. A Microsoft Research report noted that writing solid smart contracts can be extremely difficult in practice, using The DAO hack to illustrate this problem. The report discussed tools that Microsoft had developed for verifying contracts, and noted that a large-scale analysis of published contracts is likely to uncover widespread vulnerabilities. The report also stated that it is possible to verify the equivalence of a Solidity program and the EVM code.

2.5 Applications

Ethereum blockchain applications are usually referred to as DApps (decentralized application), since they are based on the decentralized Ethereum Virtual Machine, and its smart contracts. Many uses have been proposed for Ethereum platform, including ones that are impossible or unfeasible.Use case proposals have included finance, the internet-of-things, farm-to-table produce, electricity sourcing and pricing, and sports betting. Ethereum is (as of 2017) the leading blockchain platform for initial coin offering projects, with over 50% market share.

As of January 2018, there are more than 250 live DApps, with hundreds more under development. Project applications listed in this section are not exhaustive and may be outdated.

- Digital signatures that ensure authenticity and proof of existence of documents: the Luxembourg Stock Exchange has developed such a system
- Slock.It is developing smart locks
- Digital tokens pegged to fiat currencies: Dai, stablecoin pegged to US dollar. Decentralized Capital. Spanish bank Santander is also involved in such a project.
- Digital tokens pegged to gold: Digix
- Improved digital rights management for music: Imogen Heap used the technology
- Platforms for prediction markets: Augur, Gnosis Stox
- Platforms for crowdfunding: the DAO
- Social media platforms with economic incentives: Backfeed, Akasha
- Decentralized marketplaces: FreeMyVunk, TransActive Grid
- Remittance: Everex
- Online gambling: CoinPoker, Etheroll
- Electric car charging management: RWE
- Secure identity systems for the Internet: uPort
- Labour economics: Blocklancer, Ethlance
- Video Games: Cryptokitties popularity in December 2017 caused the Ethereum network to slow down.

3. Methods

3.1 Currency data retrieving from trusted sources

As it is mentioned in introduction, in this project we are creating a smart contract that keeps currency data from trusted sources and serve these data to Ethereum network. Blockchain is a decentralized network, so there is no authority to serve all country currencies to users. If someone, like us, will serve these data to users, their source of data should be trustable and non-debatable. For the purpose of this, we retrieve data from TCMB(The Central Bank of Turkey) and ECB(European Central Bank).

TCMB has its XML data source for daily currency values according to TRY(Turkish Lira) and crossrate values for USD(American Dollar). TCMB publishes 18 currencies ratio according to TRY. These values are daily and TCMB publishes daily data at 15:30 (GMT +03, Istanbul, Turkey).

ECB is European Central Bank. ECB has its XML data source for daily currency values according to ratio of EUR(Euro). ECB publishes 32 currency value in daily manner. ECB publishes these values at 14:00 (CET, Central European Time) and updates the values at 16:00 (CET, Central European Time).

Our methodology in this part of the project is that retrieving currency values from both of TCMB and ECB separately into currency classes. Each class gets all the information in these two banks. For instance, TCMB has four kind of currency values: ForexBuying, ForexSelling, BanknoteBuying, BunknoteSelling. Each classes keeps timestamps and data in the TCMB and ECb sources.

Retrieving data can be cumbersome at some points due to the sources can change their way of publishing data. Because of this, data publishing of these sources should be observed and some general decisions should be taken.

3.2 Ethereum private network connection

In this project, we used eBloc private Ethereum network that was created by assistants and students of Prof. Can Özturan in Boğaziçi University. The reason that we are using private Ethereum network is that smart contract creation and data insertion in to that contract needs fees. Tests of smart contract can be so much expensive in public Ethereum network.

Ethereum smart contracts are generally compiled in Remix online compiler. Remix can installed to personal computer and it can compile smart contracts offline. Also there are other smart compiling ways in other languages that are provided in Ethereum github resources of language APIs.

Thanks to Can Özturan and his assistants, we are provided eBloc private Ethereum network and enough ether money are provided to us.

3.3 Smart Contract functionalities

Smart Contracts have not any database or database integrations. Smart Contracts live in EVM(Ethereum Virtual Machine). All data in smart contract are preserved in Solidity data types like arrays, maps, strings, etc. .

Because of there are no kind of database in smart contracts, currency data should be kept in Solidity data type of maps. Date of data is substantially important. Keys of maps should be dates of the retrieval date of the currency data.

a-) Maps as a data resource

TCMB has four kind of currency values as it is mentioned above. For ForexBuying, ForexSelling, BanknoteBuying, BanknoteSelling and CrossRate values there should be maps for each of them. Keys of maps should be dates of retrieval times of currency data.

ECB has one kind of data, so one map can be enough for ECB.

b-) Data insertion functions

In Ethereum Blockchain network, only the owner of the smart contract can insert data into the smart contract. Creator of smart contract should create Solidity functions for itself. Data insertion functions should guarantee that data is inserted only by the contract creator. It is so important for *trustworthiness*.

Solidity provide “msg” object at the creation of smart contract and also all transactions in blockchain network. This “msg” object has a “owner” field. Insertion functions should check the account that is inserting data into the smart contract is owner or not. If the account that is inserting data into the smart is not the owner, function should not be executed.

Data should be inserted into maps by giving matched date of data retrieval and currency codes. Each map should have its insertion functions.

c-) Data retrieval functions

Data should be served to users with the principle of ease of use. Each bank, TCMB and ECB should have its own data get functions. People may want to get only a value in a specific date or ay want to convert the currencies between each other in a specific date. There should be functions for all of these operations.

Mainly TCMB should have its currency get function for all four types and crossrateusd type, and also currency conversion function. Same as with TCMB, ECB should have its currency get function and also currency conversion function.

For all functions, date values are so important. Date values should be unix time or universal time conversion type.

3.4 Smart Contract value types in Solidity

Solidity is a restrictive programming language because of its necessaries. There is no for loops, and generic big data types. It was because of someone who is unconscious or unreliable can cause a bottleneck in blockchain network with that kind of language properties.

Solidity has so small value types and restrictive features as it is mentioned above. Blockchain network transactions fees are calculated according to how much data is used. Hence, it is crucial to use as much as less data in smart contract. Because data insertion/retrieval and keeping data in smart contract needs money. Data types in our project till midterm report are:

- Dates are keys of maps. Unix time requires uint64. Example of unix time is “1521375531”.
- Currency codes requires bytes3. Currency code has a universal standard and it should be three alphabetical words long. Example of currency codes are “TRY”, “EUR”, “USD”.

a-) Currency precision

One of the most significant and problematic situation in Solidity is that there aren’t any floating numbers in Solidity. Because of this reason, currency values should be multiplied by some big integer value before it is inserted into the smart contract. Precisions are important in currency values. Multiplying with a big integer value preserve the precision value. When it was retrieved from smart contract it should be divided to that specific big integer number for precision correction.

As solution of this problem, Currency values should be multiplied with 10^9 before inserting into the smart contract and should divided to 10^9 after retrieved from smart contract.

b-) Currency conversion

Each bank should have its currency conversion functions for consistency.

Currency conversion algorithms requires the division operation. Division operation can cause precision values if the result is floating point number. In solidity, division operations are all integer divisions, that means no precisions are provided.

There is a generic solution to this general problem. In order to get precision values from division operations in solidity, the dividend number should be multiplied with a 10^X number. X is the number of precision that you want.

In our smart contract we always give currency value as 10'9 multiplied. In order to preserve this decision, we multiplied the dividend number in divisions operations with 10^9. Hence, users that use our smart contract get function should divide all currency conversion values by 10^9.

3.5 ROPSTEN Test Network and Metamask

ROPSTEN is an Ethereum based Test Network developed by Etherscan.io. The network can used as a replacement of any EVM for testing purposes, because they remove transaction data after a while. Smart Contracts can be deployed to ROPSTEN and testing is very easy compared to real Ethereum networks.

Metamask is a Browser-Extension that allows users use an Ethereum account inside browser. Using Metamask everyone can make a Decentralized Web-Application using Node.js.

We used Metamask to provide an account and ROPSTEN to deploy our contract and combine them into a single d-App.

3.6 Decentralized Web Application

A Decentralized Web Application is a blockchain based web application that communicates with blockchain network. In our case , that network is Ethereum Network , and our d-app connects to our smart contract located inside ROPSTEN.

Our web app provides a front-end solution to users and eases testing for us. We used node.JS, HTML, CSS, Web3.js for achieving that solution.

3.7 DigitalOcean Server

We need a server to deploy our crawlers and put data into smart contracts on eBloc EVM and ROPSTEN. So, we tried to find cheapest and most stable server, and our decision was DigitalOcean. We generated a machine with 25GB SSD, 1Ghz CPU and 1 GB of RAM. Since we won't connect the real Ethereum network, we installed geth and eblocPoa inside server. That specs are enough for running geth with eblocPoa.

3.8 Data Retriever

Retrieving data is very easy, you just give the url and nearly all programming languages can gather data from that url. The main challenge we faced in that project was tidying up and sending the correct data with useful format.

Thankfully both TCMB and ECB have publicly open data , and they gave that data in beautiful XMLs. Maybe JSON was more useful but their XMLs are very beautiful so we used powerful python libraries to tidy up the data and parse all data into numeric data types.

Data Retrievers run continuously on our server, and gets data every day from both TCMB and ECB.

4. Results - Works Done So Far

4.1 Curency Data Retrieving from Trusted Sources

We created two parser classes in python language for both TCMB and ECB. These classes takes data from TCMB and ECB daily data resources in different format but in a XML type. Python libraries ElementTree, urllib and json are enough for XML data retrieval from resources of these two banks. observations BankoteBuying and BunkoteSelling can be empty in TCMB source.

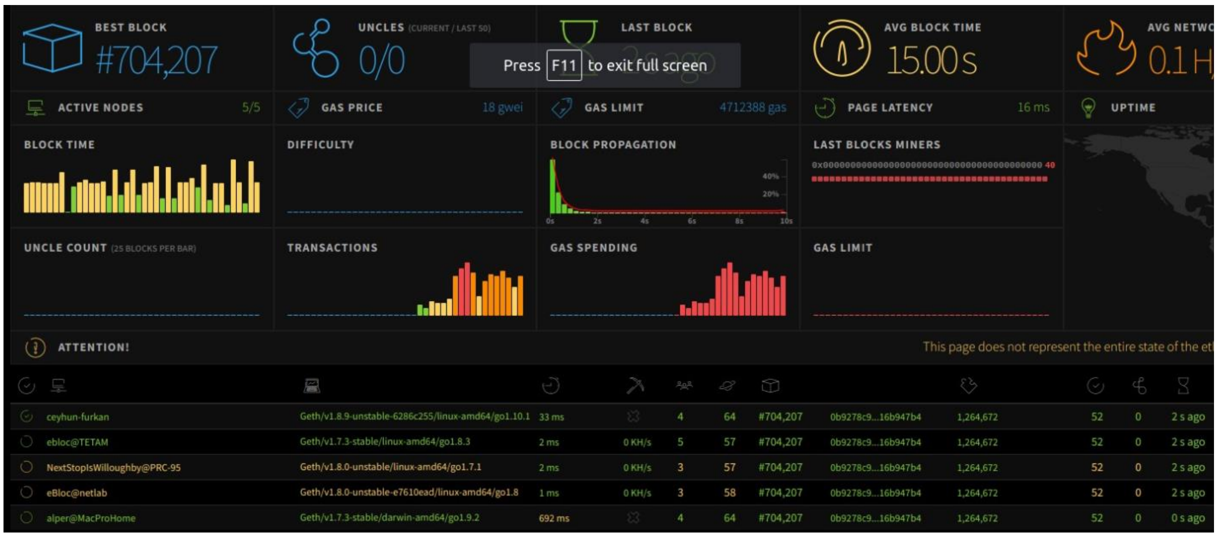
TCMB provides a XML source for daily currency data. For Turkish Lira(TRY) ratios there are four kinds of values: ForexBuying, ForexSelling, BanknoteBuying, BanknoteSelling. Also TCMB provides USD values for other than TRY currency in CrossRate column. Our TCMB parser class retrieves all these values and provides in a key-value manner.

ECB provides a simple XML source for daily currency values. Our ECB parser class gets all these values and provides in a key-value manner.

4.2 Ethereum private network connection

We developed our smart contract in eBloc Ethereum private network. Compilation of smart contracts are performed in Remix online ethereum compiler.

There are JavaScript bash client for Ethereum network that you can perform all necessary operations in the network. We used "web3.py" python library as a Ethereum blockchain network API. All data insertion, some smart code tests are executed via using "web3.py" library of python. Using "web3.py" we created a python language API for our smart contract.



ceyhun-furkan	Geth/v1.8.9-unstable-6286c255/linux-amd64/go1.10.1	33 ms		4	64	#704,207	0b9278c9...16b947b4	1,264,672	52	0	2 s ago
ebloc@TETAM	Geth/v1.7.3-stable/linux-amd64/go1.8.3	2 ms	0 KH/s	5	57	#704,207	0b9278c9...16b947b4	1,264,672	52	0	2 s ago
NextStopIsWilloughby@PRC-95	Geth/v1.8.0-unstable/linux-amd64/go1.7.1	2 ms	0 KH/s	3	57	#704,207	0b9278c9...16b947b4	1,264,672	52	0	2 s ago
eBloc@netlab	Geth/v1.8.0-unstable-e7610ead/linux-amd64/go1.8	1 ms	0 KH/s	3	58	#704,207	0b9278c9...16b947b4	1,264,672	52	0	2 s ago
alper@MacProHome	Geth/v1.7.3-stable/darwin-amd64/go1.9.2	692 ms		4	64	#704,207	0b9278c9...16b947b4	1,264,672	52	0	0 s ago

4.3 Smart Contract Functionalities

1. Maps as a data source

We used maps as a data source in our project. Because there are no way of keeping data other than maps and arrays. Our maps are nested. Our map architecture is:

mapping(uint256 => mapping(bytes3 => uint)) ECB;

This way of mapping implementation is primitive, however we will change this way of implementing in to unix time and universal time. For ease of tests, we used this kind of solution approach to the data keeping problem.

At the beginning of our project, we designed our structure in a more simple way. We created two struct types that holds all TCMB currencies and ECB currencies. However, inserting 18 currency for TCMB and 32 currency for ECB in one shot is impossible or infeasible in Solidity. Because, a function in Solidity can take 8 parameters at most.

Using date and currency codes as keys of maps is more simple than other solutions in our point of view. Disadvantage of our architecture is there are additional transactions in data insertion, however it is not so important. Important aspect of our smart contract is ease of data retrieval.

2. Data insertion functions

Only the owner of smart contract can insert data into smart contract. There is built-in “require” function in Solidity to check some condition in order to execute a smart contract function. In the creation of our smart contract, creator account is held in a “owner” variable. Each insert function in our contract checks the account of the callee of the function with “require” built-in function by comparing the owner with the calle account.

TCMB has five maps all for ForexBuying, ForexSelling, BanknoteBuying, BanknoteSelling and CrossRate. Each map has its own insertion function.

ECB has two maps for euro and dollar maps. Dollar map values are generated from euro values. Because of the reason dollar is used so much in transactions, we keeps its values in a seperated map.

Data insertion are performed by us. Hence, we developed a python API for this purpose. This API consumes TCMB and ECB parser classes and calls smart contract functions by using “web3.py” library.

TCMB and ECB parser classes provide floating point values. Insertion API multiplies these values with 10^9 and convert them into Solidity integer type. Python integer and Solidity integer are different. “Web3.py” library provides pleasant API for this kind of Solidity functions. As it is said before, all values in maps that holds values are the currency values that are multiplied with 10^9. Thus, we solved the precision problem.

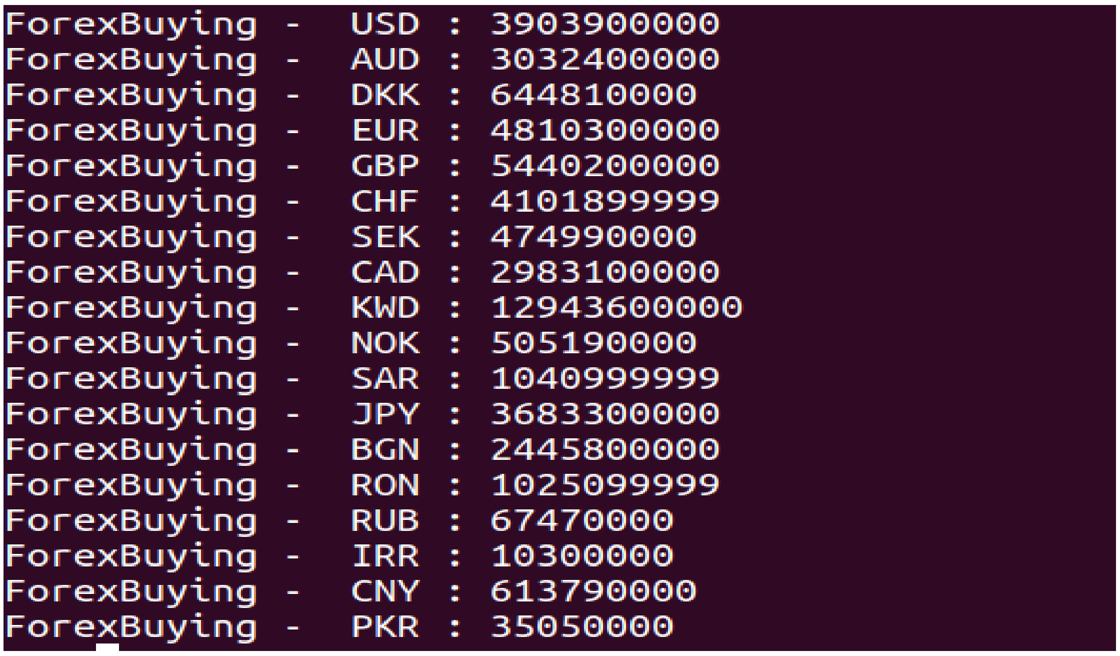
3. Check data availability

There are two check availability functions , one for today one for given date. They checks if there is the data for that day, and returns true or false. isTodayAvailable() makes that check for today. As we put data after 6 pm, because ECB and TCMB publishes daily data after 3 pm, that methods plays a very important role for our users.

4. Data retrieval and converter functions

There are two kind of data retrieval in our smart contract. The first one is simple get function type. It accepts a date parameter and a currency code parameter, and it returns a currency value that should be divided to 10^9.

Second kind of retrieval function in our smart contract is currency converter. Currency converter functions takes a date parameter and two currency code parameter which are converted between them. Currency converter function multiplies the first currency value with 10^9 and divide it by converted currency value. Result is the 10^9 times of actual value. In our smart contract all retrieved values should be divide to 10^9, and this rule is also prevailing in currency conversion functions.



```
ForexBuying -  USD : 3903900000
ForexBuying -  AUD : 3032400000
ForexBuying -  DKK : 644810000
ForexBuying -  EUR : 4810300000
ForexBuying -  GBP : 5440200000
ForexBuying -  CHF : 4101899999
ForexBuying -  SEK : 474990000
ForexBuying -  CAD : 2983100000
ForexBuying -  KWD : 12943600000
ForexBuying -  NOK : 505190000
ForexBuying -  SAR : 1040999999
ForexBuying -  JPY : 3683300000
ForexBuying -  BGN : 2445800000
ForexBuying -  RON : 1025099999
ForexBuying -  RUB : 67470000
ForexBuying -  IRR : 10300000
ForexBuying -  CNY : 613790000
ForexBuying -  PKR : 35050000
```

4.4 Decentralized Web Application

We must develop a Decentralized Web Application for testing purposes, in addition to that we can make it functional for users to try our smart contract.

Firstly, we must use Web3 provider to reach ROPSTEN and call our smart contract functions. Web3 contains libraries for several languages and Javascript is one of them. We also used Metamask to sign our transactions going into ROPSTEN. Web3 alone can’t sign transactions but Metamask provides that feature for all browsers. In that way we can successfully use our smart contract and get data.

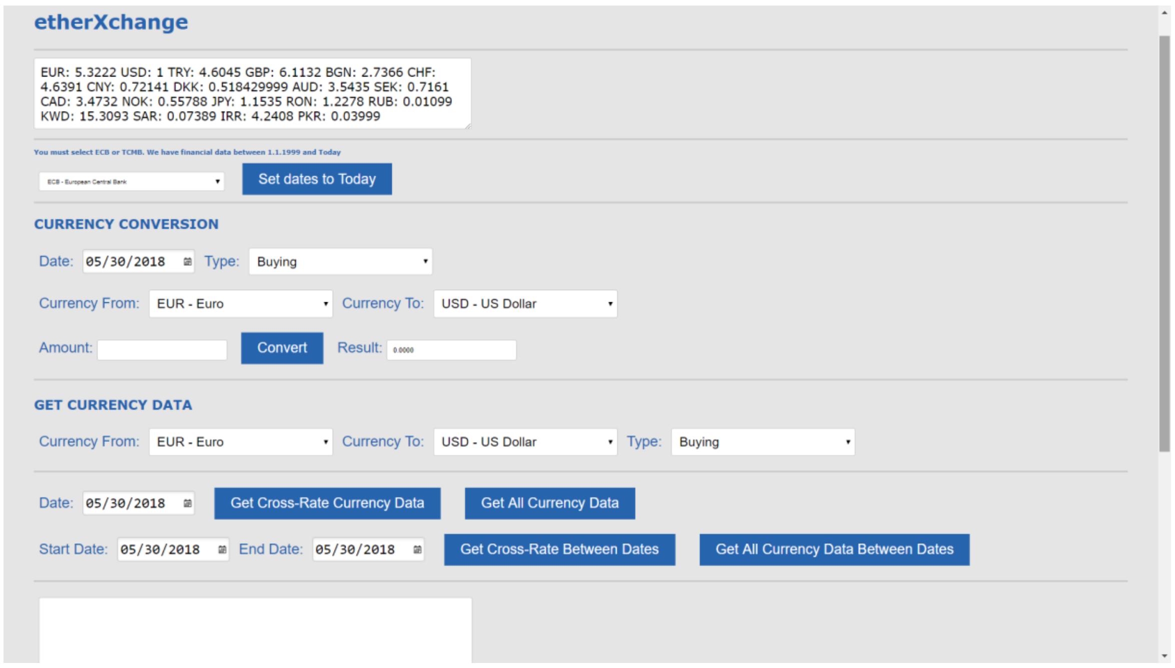
Then, we used Javascript to tidy up display objects and reach our Smart Contract. Here is a list of functions from **functions.js** and their duties.

- **converToHex():** Converts given string to Hex string for giving currency names to smart contract.
- **stringFromDate():** Returns string of given date object as “DD-MM-YYYY”
- **stringFromEpochTİme():** Returns string of given epoch time as “DD-MM-YYYY” It actually calls stringFromDate() after converting epochtime to date object.
- **onStart():** Called just before page loaded , and calls two functions sequentially.
- **changeCurrencies():** Called by onStart() or “bank” selector value changed. Re-arranges the currency and type select objects.

- **setToday():** Called by onStart() or “Set dates to Today” button is clicked. Checks data for today is available, and sets the dates to today or yesterday. For the first run, it calls getToday().
- **getToday():** Gets data for today from TCMB Forex Selling and prints it into top textview.
- **getCrossRate():** Button "Get Cross-Rate Currency Data" onClick action. Calls function that fits bank name.
- **getCrossRateBetweenDates():** Button "Get Cross-Rate Currency Data Between Dates" onClick action. Calls function that fits bank name.
- **getCrossRateECB():** Get variables from display objects and calls getCrossRateDataECB() between given dates.
- **getCrossRateDataECB():** Generates contract variable with given abi and address and then call convert_x_to_y_ecb() from contract. Prints result into bottom textview.
- **getCrossRateTCMBSelling ():** Get variables from display objects and calls getCrossRateTCMBDataSelling() between given dates.
- **getCrossRateTCMBDataSelling ():** Generates contract variable with given abi and address and then call convert_x_to_y_tcmb_forexselling () from contract. Prints result into bottom textview.
- **getCrossRateTCMBBuying ():** Get variables from display objects and calls getCrossRateTCMBDataBuying() between given dates.
- **getCrossRateTCMBDataBuying ():** Generates contract variable with given abi and address and then call convert_x_to_y_tcmb_forexbuying () from contract. Prints result into bottom textview.

- **getAllData():** Button "Get All Data" onClick action. Calls function that fits bank name.
- **getAllDataECB():** Get variables from display objects and calls getECB() for all currencies.
- **getECB():** Generates contract variable with given abi and address and then call get_ecb () from contract. Prints result into bottom textview.
- **getAllDataTCMBSelling ():** Get variables from display objects and calls getTCMBSelling () for all currencies.
- **getTCMBSelling ():** Generates contract variable with given abi and address and then call get_ecb () from contract. Prints result into bottom textview.
- **getAllDataTCMBBuying ():** Get variables from display objects and calls getTCMBBuying () for all currencies.

- **getTCMBBuying ():** Generates contract variable with given abi and address and then call get_ecb () from contract. Prints result into bottom textview.
- **convert():** Button "Convert" onClick action. Calls function that fits bank name.
- **convertECB():** Get variables from display objects and calls getCrossRateDataECB(). Then multiplies the result with “amount” and prints it into “result” textfield.
- **convertTCMBSelling ():** Get variables from display objects and calls getCrossRateTCMBDataSelling (). Then multiplies the result with “amount” and prints it into “result” textfield.
- **convertTCMBBuying ():** Get variables from display objects and calls getCrossRateTCMBDataBuying (). Then multiplies the result with “amount” and prints it into “result” textfield.


etherXchange

```
EUR: 5.3222 USD: 1 TRY: 4.6045 GBP: 6.1132 BGN: 2.7366 CHF:
4.6391 CNY: 0.72141 DKK: 0.518429999 AUD: 3.5435 SEK: 0.7161
CAD: 3.4732 NOK: 0.55788 JPY: 1.1535 RON: 1.2278 RUB: 0.01099
KWD: 15.3093 SAR: 0.07389 IRR: 4.2408 PKR: 0.03999
```

You must select ECB or TCMB. We have financial data between 1.1.1999 and Today

ECB - European Central Bank

CURRENCY CONVERSION

Date: 05/30/2018 Type: Buying

Currency From: EUR - Euro Currency To: USD - US Dollar

Amount: Result: 0.0000

GET CURRENCY DATA

Currency From: EUR - Euro Currency To: USD - US Dollar Type: Buying

Date: 05/30/2018

Start Date: 05/30/2018 End Date: 05/30/2018

Set dates to Today
Convert
Get Cross-Rate Currency Data
Get All Currency Data
Get Cross-Rate Between Dates
Get All Currency Data Between Dates

4.5 Setting Up Server

As we mentioned before we rented a server using DigitalOcean. We deployed all our github repository into our server. We installed geth, eBlocPoa and nodejs. After deploying the code, we wrote a crontab file that runs our python scripts that gathers data from banks and puts them into Smart Contract at eBloc on Bogazici University, and nodejs server, for our website.

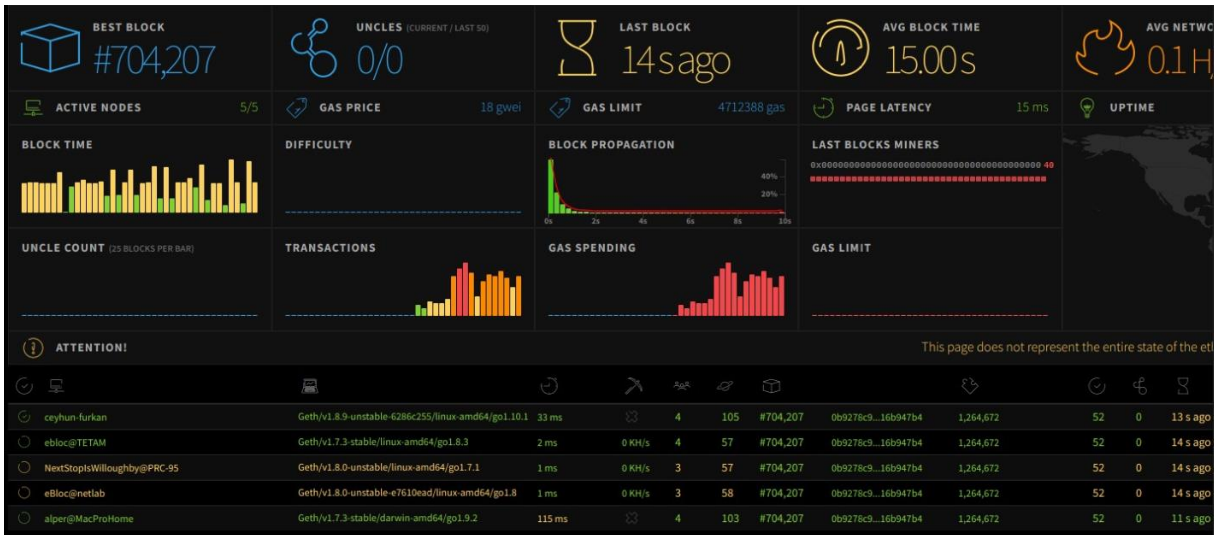
4.6 Crawlers (Data Providers)

We wrote crawlers for gathering data from European Central Bank and Türkiye Cumhuriyeti Merkez Bankası. A crawler runs continuously and checks data availability. If there are new data , checkout that data. Just after checking out our crawlers call other scripts that will tidy-up the new data and send it to smart contract.

We have 2 different providers after gathering data with crawlers:

- **ROPSTEN Data Provider**
  Main duty of this crawler is providing data for Smart Contract inside ROPSTEN. We used Infura Proider to communicate with ROPSTEN.
- **eBloc Data Provider**
  Main duty of this crawler is providing data for Smart Contract inside eBloc at Bogazici University. We used IPC Provider to communicate with eBloc EVM.

A cron job is a job that called again and again with some interval and runs in background. Our crawlers run as cron jobs such that works in a daily manner and transactions and its receipt logs produces logs to our server.



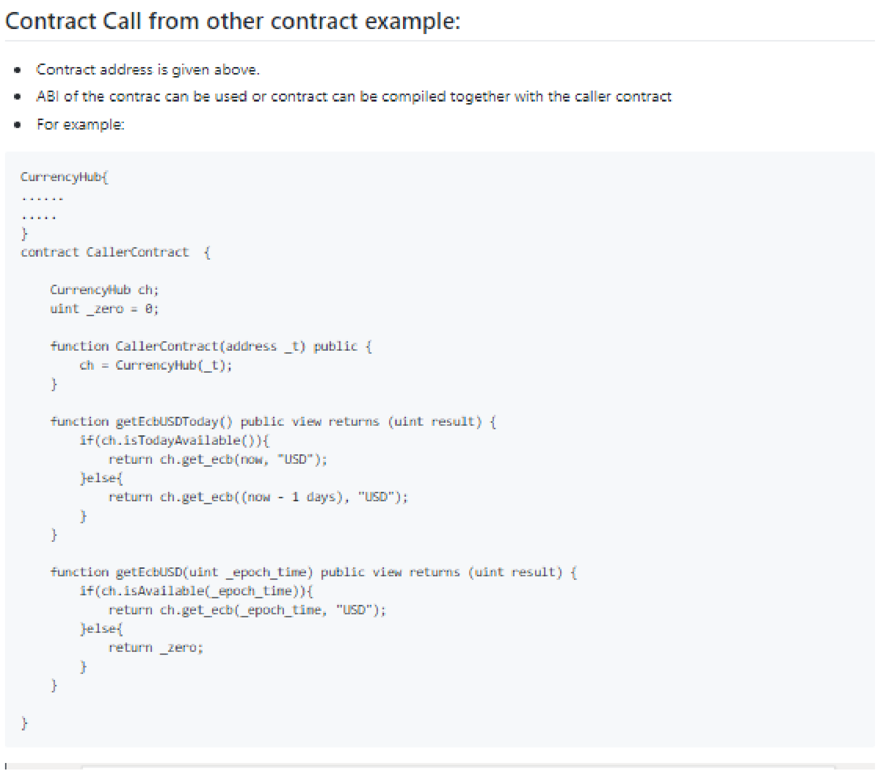
ATTENTION! This page does not represent the entire state of the et

ceyhun-furkan	Geth/v1.8.9-unstable-6286c255/linux-amd64/go1.10.1	33 ms		4	105	#704,207	0b9278c9...16b947b4	1,264,672	52	0	13 s ago
ebloc@TETAM	Geth/v1.7.3-stable/linux-amd64/go1.8.3	2 ms	0 KH/s	4	57	#704,207	0b9278c9...16b947b4	1,264,672	52	0	14 s ago
NextStopIsWilloughby@PRC-95	Geth/v1.8.0-unstable/linux-amd64/go1.7.1	1 ms	0 KH/s	3	57	#704,207	0b9278c9...16b947b4	1,264,672	52	0	14 s ago
eBloc@netlab	Geth/v1.8.0-unstable-e7610ead/linux-amd64/go1.8	1 ms	0 KH/s	3	58	#704,207	0b9278c9...16b947b4	1,264,672	52	0	14 s ago
alper@MacProHome	Geth/v1.7.3-stable/darwin-amd64/go1.9.2	115 ms		4	103	#704,207	0b9278c9...16b947b4	1,264,672	52	0	11 s ago

Figure 1 While transactions processing

4.7 Caller Contract

We designed a caller contract that consumes our main smart contract and we showed how to use our functions.



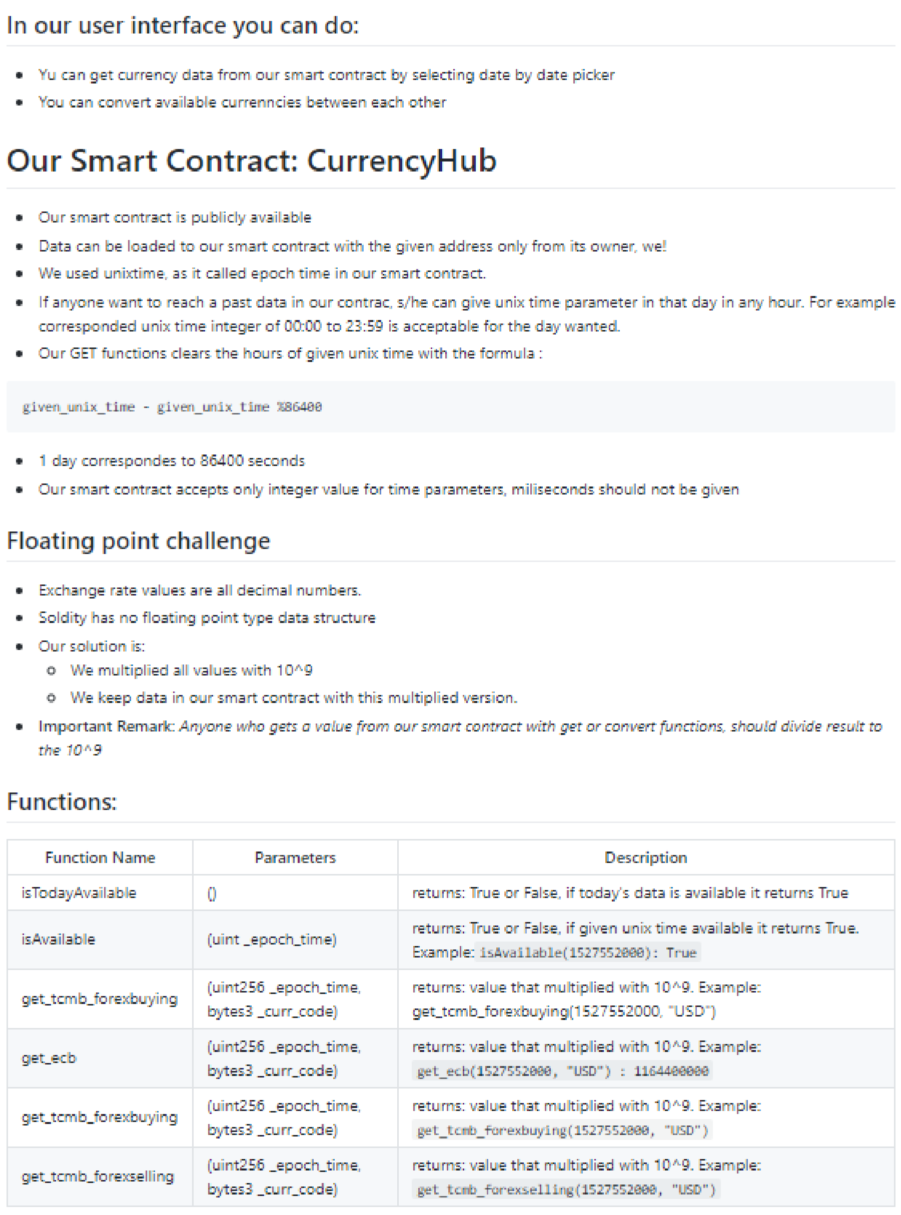
Contract Call from other contract example:

- Contract address is given above.
- ABI of the contrac can be used or contract can be compiled together with the caller contract
- For example:

```
CurrencyHub{
......
.....
}
contract CallerContract  {

    CurrencyHub ch;
    uint _zero = 0;

    function CallerContract(address _t) public {
        ch = CurrencyHub(_t);
    }

    function getEcbUSDToday() public view returns (uint result) {
        if(ch.isTodayAvailable()){
            return ch.get_ecb(now, "USD");
        }else{
            return ch.get_ecb((now - 1 days), "USD");
        }
    }

    function getEcbUSD(uint _epoch_time) public view returns (uint result) {
        if(ch.isAvailable(_epoch_time)){
            return ch.get_ecb(_epoch_time, "USD");
        }else{
            return _zero;
        }
    }

}
```

4.8 Info Page

We designed an Info Page for users. That info page lives inside github as Readme.md. It is written in Markdown and shows how to use our smart contract on ebloc at Bogazici University.



In our user interface you can do:

- Yu can get currency data from our smart contract by selecting date by date picker
- You can convert available currenncies between each other

Our Smart Contract: CurrencyHub

- Our smart contract is publicly available
- Data can be loaded to our smart contract with the given address only from its owner, we!
- We used unixtime, as it called epoch time in our smart contract.
- If anyone want to reach a past data in our contrac, s/he can give unix time parameter in that day in any hour. For example corresponded unix time integer of 00:00 to 23:59 is acceptable for the day wanted.
- Our GET functions clears the hours of given unix time with the formula :

```
given_unix_time - given_unix_time %86400
```

- 1 day correspondes to 86400 seconds
- Our smart contract accepts only integer value for time parameters, miliseconds should not be given

Floating point challenge

- Exchange rate values are all decimal numbers.
- Soldity has no floating point type data structure
- Our solution is:
  - We multiplied all values with 10^9
  - We keep data in our smart contract with this multiplied version.
- Important Remark: *Anyone who gets a value from our smart contract with get or convert functions, should divide result to the 10^9*

Functions:

Function Name	Parameters	Description
isTodayAvailable	()	returns: True or False, if today's data is available it returns True
isAvailable	(uint _epoch_time)	returns: True or False, if given unix time available it returns True. Example: `isAvailable(1527552000): True`
get_tcmb_forexbuying	(uint256 _epoch_time, bytes3 _curr_code)	returns: value that multiplied with 10^9. Example: get_tcmb_forexbuying(1527552000, "USD")
get_ecb	(uint256 _epoch_time, bytes3 _curr_code)	returns: value that multiplied with 10^9. Example: `get_ecb(1527552000, "USD") : 1164400000`
get_tcmb_forexbuying	(uint256 _epoch_time, bytes3 _curr_code)	returns: value that multiplied with 10^9. Example: `get_tcmb_forexbuying(1527552000, "USD")`
get_tcmb_forexselling	(uint256 _epoch_time, bytes3 _curr_code)	returns: value that multiplied with 10^9. Example: `get_tcmb_forexselling(1527552000, "USD")`

4.9 API

We designed an API that reaches Smart Contract and returns results. It is basically a python script that does the nearly same work with functions.js mentioned above. But it uses web3.py and calls smart contract functions using that. Different from JS functions, API reaches directly into eBloc at Bogazici University.

5. Conclusion and Discussion

Solidity is easy to use programming language. However, it has its necessary restrictions. They cause us some troubles. First of all there is no floating point numbers. There are some widely used solutions for that problem like creating a struct type with two integer field, one holds decimal part of the floating number and second one holds the precision values. In our solution to this problem is multiplying the floating point with 10^9. The reason for that solution is that there are so many mathematical operations in our smart contract like conversions. It is impractical to use other solutions for floating point problem like struct with two integer parts. Secondly, Solidity functions can take at most 8 parameters. It was a problem for our insertion functions, but we solved this problem also.

We faced so much challenges while implementing our project:

The most important challenges we faced was because of ebloc and geth IPC provider. Firstly, we used Web3.py version 4.0. The new version eases our work but it also had some issues and that issues are very rare we can barely find inside github issues. For example: we faced unsufficient funds error, while sending data and gasPrice is enough. Secondly, we figured out our transactions didn't pass, and the source of the error was unlockAccount. We must unlock account before sending any transaction, but this process was expiring after some time, and we get IPC based errors while connecting using web3.py.

Another challenge was caused by Turkish Republic Central Bank (TCMB). TCMB includes 5 data types, but some of the currencies have only 2 fields filled on data sheet. That situation causes errors while gathering data. We only used "Forex Buying" and "Forex Selling" types because of that. Both columns are filled.

There was a challenge about ROPSTEN too. While we tried to install geth for ROPSTEN to our server, our server ran out of RAM. After we generated a swap space for that , our Harddisk ran out this time. We must use Metamask account for sending transactions(data) to our contract inside ROPSTEN. We signed the transaction that we are adding data to smart contract, with private key of our Metamask account. This particular sitution is not possible web3py library version 3, we used cutting edge version 4 for this operation.

The last challenge we faced was caused by Node.JS. We must use async calls to reach our contract and gather data. But we had to use loops to iterate through currencies or days. Loops runs sequential, but async network calls responds unsequentially. That causes data to mix up. We solved this problem writing an additional function for any loop, with giving index of loop as parameter. Then we printed all results after all async network calls responded.

Conclusion:

Blockchain has a great philosophical background. Decentralized systems can be a revolution for the world if it is used for the sake of humanity. All centralized systems have their backups and disaster scenarios. However, in blockchain as a decentralized system, data lives forever in network if some conditions hold.

6. Future Work

6.1 Including Crypto-World

In the future outlook of the project crypto-currency rates and other financial data of crypto-world can be included. There are so much financial data inside crypto-world and that data is so valuable.

There are two big problems about cryptocurrency data. Firstly, the data is not consistent, the rates change nearly at every millisecond. It can be solved using only meaningful changes. Secondly, there isn't a authority we can trust like ECB or TCMB. That problem can be solved using top 5 biggest blockchain exchange rates and giving all of them.

6.2 Increase Data Reliability

Our smart contract should be trustable and convenient. We have one universal trusted source a.k.a. ECB and one locally trusted source a.k.a. TCMB. In future we can include another universal and locally trusted sources for another regions.

In addition to that, we can compare and analyze the data to provide more useful statistics to our users.

6.3 Predict Financial Data with an AI

After increasing data reliability, we got a huge amount of financial data. While doing analysis and providing useful statistics, we can also use Machine Learning methods to predict currency rates, especially for crypto-world.

6.4 Scope of Contract

There are rumours about eBloc to run into another universities. If the network widens , more people can use our smart contract. The scope of our contract increases and we must think about more user experience when that day comes. A better user guide and more useful command-line interface can be added.

6.5 Beyond our project horizon

It was beyond our project horizon, but it can be implemented that all financial data can be retrieved from central banks and some significant financial institutions. There are so many financial data about countries and companies in financial industry like stock exchange data. They are required in some operations.

Also these financial data can served with the needs of industry. Like currency converter, there are some financial calculations by consuming above financial data. In this smart contract, these kind of functions can be implemented.

7. References

- https://solidity.readthedocs.io/en/v0.4.21/
- http://remix-alpha.ethereum.org
- https://remix.ethereum.org/
- https://github.com/ethereum/web3.py
- https://web3py.readthedocs.io/en/stable/
- https://github.com/ethereum
- https://en.wikipedia.org/wiki/Ethereum
- https://theethereum.wiki/w/index.php/Main_Page
- http://web3py.readthedocs.io/en/stable/contracts.html
- https://ropsten.etherscan.io
- https://medium.com/@blockchain101/calling-the-function-of-another-contract-in-solidity-f9edfa921f4c
- http://web3py.readthedocs.io/en/stable/middleware.html#geth-style-proof-of-authority
- https://www.nbs.sk/en/statistics/exchange-rates/ecb-foreign-exchange-reference-rates
- https://github.com/ethereum/web3.py/issues/492
- https://medium.com/@jason.carver/whats-new-in-the-web3-py-v4-beta-453d17231758
- https://medium.com/@jacksonngtech/syncing-geth-to-the-ethereum-blockchain-9571666f3cfc