Final Year Undergraduate Project

ENABLING CROSS-BORDER VALUE TRANSFER USING BLOCKCHAIN TECHNOLOGY

Author: Muhammad Hamza

Supervisor: Professor Nikos Tzevelekos

June 25th, 2020

School of Electronic Engineering and Computer Science

Final Year Undergraduate Project 2019/20

Declaration

This report is the result of my own research and includes nothing which is the outcome of work done in collaboration.

This report does not exceed the maximum word limit of 11,000 words, excluding appendices and bibliography.

Abstract

In recent years, the blockchain technology has become a hot topic of discussion whilst having serious impact on traditional transaction process within businesses which required third party involvement for verification and centralised architectures. In particular, the financial industry has taken advantage of this technology's newly introduced paradigm, known as the decentralisation of financial system, which serves as an immutable distributed ledger coupled with cryptographically secured transactions. This project aims to solve the issue of cross-border high value transfers using blockchain technology. In other words, removal of customary intermediaries, which eliminates several shortcomings that are associated with it. This is accomplished mainly using Ethereum's decentralised platform to derive with fully functioning smart contracts with set requirements. With each milestone achieved, new design measures were taken to solve each corresponding problem in steps.

This project began with the intention of implementing smart contracts which would have enabled peer-to-peer fiat transfers outside the current jurisdiction of the smart contract caller. The transaction was planned to involve series of processes that includes Fiat-Ether conversion, peer-to-peer Ether transfers, subsequently converting the received Ether into receiver's fiat currency. This decentralized application makes the use of Ethereum blockchain.

However, upon further researching of the several ways this project would be implemented and having undergone several trial-and-errors. Several drawbacks of smart contracts were discovered—for each apropos features that were being implemented. The major issue with the project was discovering the fact that smart contracts cannot directly interact with external data outside the blockchain. This reemphasized how immutable smart contracts really are.

As a result, external adapters were implemented that would operate on Chainlink's node operators. Each node job is verified by series of decentralized oracles provided by Chainlink. These oracles provide the same security as a smart contract.

Overall, this project began as a software implementation project, however it is now a semi-research and semi-implementation, as it is currently both a proof-of-concept and a Minimum Viable Product to showcase what the supposed application would have resulted into—containing only working part of the planned-implementation. This concept is widely discussed, but no one has yet found a working solution.

Acknowledgments

First and foremost, Alhamdulillah—"praise be to Allah (SWT)”.

I thank my parents for their unparalleled support and sacrifices.

Especially, I thank you Professor Nikos for being an accommodating and understanding supervisor.

Contents

Chapter 1

Introduction

1.1 Motivation

One of the main motivational agents behind this project is to use the blockchain technology to derive with such a system that is able to perform a transaction in a peer-to-peer ecosystem whilst having no involvement of third parties and having enforced laws and policies in a decentralised and algorithmic manner. This trust is further solidified as there are no unforeseen legal ramifications.

1.2 Objective

This project aims to implement a system that utilises blockchain technology using Ethereum smart contracts to perform cross-border payments. The scope of this project is to facilitate the technology by taking advantage of its many features: immutability, transparency, and consistency. Consequently, it enables a transaction to occur with exclusion of any middlemen such as banks. This makes the transaction cost-effective by cutting down fees that are charged by correspondent banks: processing transaction and other activities such as collating and receiving payment messages.

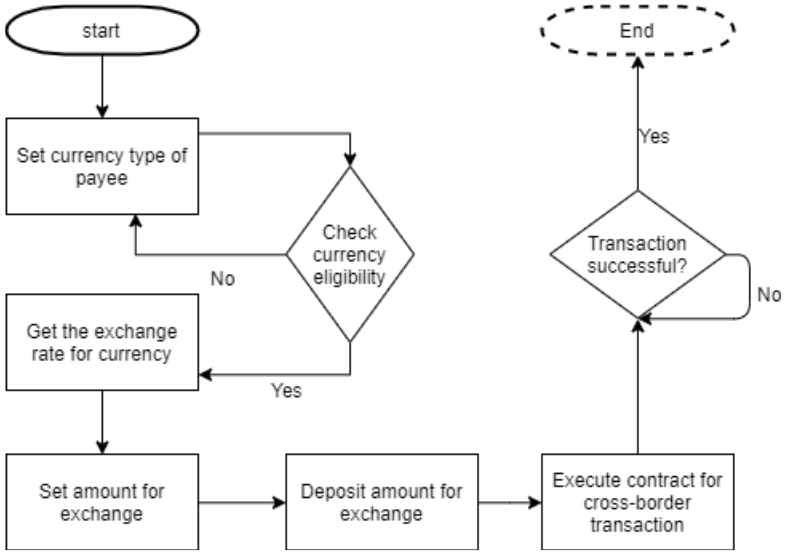


Figure 1.1: a flow chart diagram depicting a cross-border transaction.

1.3 Contributions

In order to implement such a system, external adapters are implemented which utilise API and enable oracle smart contracts to interact with the external data send to or received from the API. Any number of fiat payment networks can be supported. Each must provide a ChainLink external adapter implementation to handle paying out and checking payments in.

A pool of Senders provides fiat-currency and cryptocurrency liquidity to the system. They receive a fee for each trade. The fee is the same for all transaction callers.

The receiver “buys” Ether for a fiat currency or a fiat-currency for Ether. They get the current market price minus a fixed the sender’s fee.

Differences from local Ethereum network :

- Fiat-currency payments for sending are executed by Chainlink’s decentralized oracles
- Fiat-currency payments for receiving are executed outside the system but are checked and validated by the decentralized oracles before crypto is released
- Trades are immediate exchanges with fixed rates (there is no order book)

Differences from traditional centralized gateways:

- Transparency—all orders are initiated and executed on-chain
- Peer-to-peer transactions- registered senders to any receivers
- Trusted Chainlink’s decentralized Oracles executing payment on-chain network transactions

1.4 Features

This project implements a web application using blockchain technology with ReactJS as front end. The following are the main features:

- User can perform transaction within same jurisdiction of payee. This will not require the use of currency conversion smart contract.
- Users can store their fund into wallet that is integrated with the website.
- Users can perform transaction cross-border by selecting option of destination contract. Some currency may not be available for cross-border transaction.
- Users can purchase tokens or transfer tokens to other users.
- Users must create an account on the application before performing transaction.
- Encrypted receiver fiat payment destination
- Encrypted sender API credentials
- Fiat payments enabled via the external adapter, Paypal
- Ether to Fiat currency, or Fiat currency to Ether

1.5 Findings

The decision of choosing Ethereum as the development platform for smart contracts was straightforward because it offers more room for experimentation and customisation of smart contracts.

On the other hand, while there is an option of implementing a personal blockchain and cryptocurrency, it does not guarantee that it will solve the problems all cryptocurrencies have especially in the current time constraints. The problems are high price volatility, no chargebacks, and still in development.

Chapter 2

Background

2.1 Blockchain

A blockchain is a distributed ledger of secure, trusted digital transactions that are processed through realising the notion of *trustlessness*—the amount of trust required is minimised by 'distributing trust' among the actors at play in the system. The blockchain technology thus does not need a trusted intermediary as each peer of such a peer-to-peer network own copies of the data: increasing transparency and eliminating single point of failure. This also entails that two parties don't need to trust or know each other in order to perform a financial transaction. (Tapscott, 2016).

Furthermore, for a system to be blockchain-based, several key properties must be present. Blockchain uses encryption or "public key cryptography", which is a crucial piece of invention used in blockchain. Moreover, blockchain makes use of heavy mathematical and algorithmic functions, also known as *hash functions*. These functions are designed to make deconstruction of hashed data very difficult.

2.1.1 History

In the early nineties, Stuart Haber and W. Scott Stornetta described a "cryptographically secured chain of blocks". Following, computer scientists improved upon the invention: introducing a decentralised digital currency, 'bit gold', in 1998; theory of cryptographic secured chain was published in 2000; and a white paper was released establishing the "model for a blockchain". After the launching the initial code in 2009, developer(s), alternatively known under the pseudonym Satoshi Nakamoto, created the first cryptocurrency implementing blockchain, called Bitcoin. (Nakamoto, 2008)

2.1.2 Volatility

Cryptocurrencies face a major issue with high volatility. This is evident seeing Bitcoin become one of the most popular cryptocurrencies in 2017 (Bitcoin Wiki, 2020). However, as sudden it ascended, it came crashing down as well. Then, in 2018, Bitcoin saw another huge fall of as much as 65 percent. This also resulted in one of the biggest upset as the cryptocurrency market suffered loss of almost $342 billion (Wikipedia, 2018).

Figure 2.1: Bitcoin's volatility period highlights by time .

The reason for cryptocurrencies' volatility is the fact that they have no intrinsic value. As nothing sold is tangible, and profits aren't invested back into the cryptocurrency market as regularly; these reasons make it extremely hard to place a stable value on many of these currencies online. Furthermore, financial institutions such as hedge funds and banks have yet to give cryptocurrency a chance as a viable investment, which means that cryptocurrency faces a lack of institutional capital (Iinuma, 2018).

The reason why cryptocurrency is so volatile is because, at its source, a cryptocurrency is just another form of currency and a currency. The value of a cryptocurrency, in almost every case, is determined by the cost people are willing to pay for it. For the cryptocurrency market to see reduction in their volatility, they require a defined value, investors, and more regulations; however, it seems this is unlikely to happen anytime soon.

2.1.3 Decentralized ledger

Blockchain being a decentralized ledger provides the advantage to cryptocurrency transactions by removing the need for a centralized third party in order to validate peer-to-peer digital trade from anywhere. Moreover, these trades take place in the form of an electronic coin which is defined as a chain of digital signatures (Nakamoto, 2008). After each transaction of an electronic coin has executed, receiver can get verification of the sender‘s digital signature which is validated by the receiver accordingly to the proof-of-work i.e. to continue the chain of ownership

In hindsight, in almost any programming language a blockchain can be structured as a linked list or even an array. However, in doing so, one must piece together the essential ingredients which are the true mark-up of a blockchain. The key attributes that are essential are as follows:

- The hash of the current block
- The timestamp of when the current block was created

- The hash of the previous block
- The nonce, a number added to the hash block. This number increases incrementally per addition of each block starting from '01' from the very first block

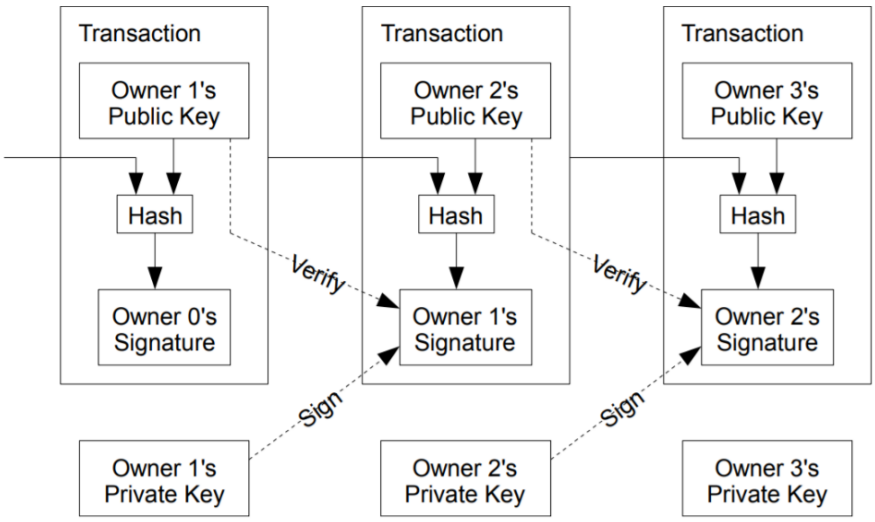


Figure 2.2: The structure of a blockchain: when a transaction is executed, a digital signature (hash) is signed of the previous transaction which also generates the public for the next signer. These chain of transactions form an immutable blockchain.

Blockchain provides the assurance that a coin cannot be spent multiple times, thus preventing double spending (Rubasinghe, 2018). This is because every transaction has a timestamp and hash of each block specific to its respective transaction. The timestamp strengthens the order of blocks and verifies the data that was present in order to create the hash for the transaction that is to occur subsequently. When a new block is created for the transactions data to be added, the nonce value is incremented continuously until it finds a valid hash for that block (Nakamoto, 2008).

Furthermore, there are two different types of blockchain, each with a different implementation method. Therefore, varying kind of platforms that are blockchain-based are already present in the market. The types of blockchain that currently exists are: Hyperledger, where blocks need to be validated before they can even enter the network; the other are open public blockchains, which are the tycoons of the cryptocurrency industry, Ethereum and Bitcoin (Hyperledger, 2020).

Ethereum has become one of the most powerful and useful blockchains. Ethereum has platform that allows implementation of smart contracts which enables us to create currencies and decentralized applications. This is all possible through the use of its scripting language called *Solidity.* These smart contracts are very useful in terms of reflecting any kind of transactions, which can be easily access of the Ethereum platform. The smart contracts work through the use of Ethereum's personal cryptocurrency called Ether. As previously mentioned, the Ether cryptocurrency is the essential token to facilitate transactions of all decentralized applications implemented and deployed on Ethereum network.

Ethereum's test network is one other added benefit when using Ethereum platform for smart contract implementation. Ethereum has the following test networks that are extremely popular for

application that are under development: Ropsten, Rinkeby, Kovan, and Goerli. Moreover, these test networks are particularly useful as developers can develop under testing environment, developing their applications and make transactions without the need of spending any real currency such as Bitcoin or Ether, before they can finally deploy their application on to the main network. This is very useful because there is no risk in losing currency. For this project, the Ropsten and Rinkeby were the primary test networks used for developing smart contract under testing environment.

2.2 Smart contracts

A smart contract is a computerised transaction protocol that executes its terms automatically once set requirements are met (Szabo, 1994). The general purpose of a smart contract is to minimise the need for trusted intermediaries, reducing transaction costs and potential fraud.

A smart contract can only be recorded on a blockchain, which indicates that they inherit its properties, namely immutability and high security. Furthermore, a smart contract controls the transfer of assets on a blockchain itself. Using a blockchain to execute smart contract makes them not prone to manipulation, thus making them very secure. There are no intermediaries involved in transaction settlement in a smart contract, which reduces associated costs. (Schulpen, 2018)

2.2.1 Gas system

On the Ethereum platform, smart contracts are run on machines by miners. the creators of smart contracts and its users pay compensation to the miners because of their computing resources that they are contribute; the amount is paid is specified in Ethers. The Ethers that are paid to the miners are calculated by multiplying the gas price with the set gas cost. The ‘gas price’ is specified by the transaction initiators, whereas the ‘gas cost’ depends entirely on how much the transaction cost is in terms of used computational resources.

The base unit used to quantify gas cost in a transaction under the Ethereum network is Wei. The minimum gas cost is one Ether which is roughly equivalent to 1018 Wei. If the gas price is specified to be too low, the transactions may not be executed. This is because the miners are able to choose which smart contract can be executed and subsequently broadcasted to the other Ethereum nodes on the blockchain (Wood, 2014). Therefore, in order to limit the ‘gas cost’, when a user transfers Ethers to invoke a smart contract, there will be a ‘gas limit’ which is due to the gas cost. The execution is ended when the ‘gas limit’ is exceeded by the gas cost—this raises an out-of-gas error exception.

2.2.2 Data location

In smart contracts, data can be stored in *memory*, *storage*, or *call-data* (Solidity — Solidity 0.6.8 documentation, 2020). Storage is a constant memory area to store data. The Ethereum Virtual Machine (EVM) assigns a storage slot Id for identification of each storage variable. Reading and writing on a *storage variable* is an operation that is considered to be the most expensive in

comparison with just reading from the two different locations. The second memory area is named *memory*. When the life cycle ends, the data of the memory variables are released. Hence, writing and reading to memory is cheaper than storage. However, Call-data is only allowed for external smart contract functions. Reading data from the Call-data in comparison with memory and storage, is much cheaper.

2.2.3 Contracts standard

Contracts can be created following strictly according specified protocol standard issued and implemented using Solidity. The most popular protocol standard in the Ethereum blockchain, is the ERC-20 tokens, which is the most significant token and is used for all smart contracts, are a shared set of rules for Ethereum tokens to use. As of June 2020, there are more than two hundred thousand ERC-20 compatible tokens on the main Ethereum network (Ledger, 2020).

The tokens that following the ERC-20 protocol standard have the follow must-include functions (ERC20 Token Standard, 2020):

- *transferFrom(address from, address to, uint256 value),* which is essential for Ether transfers
- *balanceOf(address owner_of_token),* this shows the balance of token of the holder, the owner
- *totalSupply(),* this is used to access the total number of supply of the token, which is ERC20, and is estimated to be 200,000 as of now

After implementing a smart contract, the following step is to deploy on the Ethereum network or any of its affiliate test networks, one of which is called Remix. Remix is an extremely popular IDE for smart contract implementation, especially for beginners, it runs on a local Ethereum node which enables the smart contract to interact with its blockchain and then compile for deployment for any application.

Once the smart contract is deployed, it is assigned a hexadecimal address. If the smart contract is a transaction for issuing tokens, then it can be written to the blockchain by specifying the address of the contract of the token for the application to be used or to be sent. These are the crucial measures needed to implement a basic cryptocurrency that is able for peer-to-peer transfers.

2.2.4 Storage

The method by which users are able to store their cryptocurrency assets is a wallet. A typical holder of a wallet will have access to a private key and a public key. The private key is used by users to sign each transaction, to make sure further security. Whereas, the public key is hashed, and the end result makes up a hexadecimal address where cryptocurrency can be sent to (Houben, 2018). One example of a wallet which is frequently used and is very popular for cryptocurrencies on the main Ethereum network and its test networks is *MyEtherWallet*.

2.3 Cross-border payments

Remittance payment is a transaction where funds move from one account to another; this can take place within the same bank or any other. However, when considering cross-border transfers, there are additional complications. Cross-border payments is transaction involves transfer of funds into a different jurisdiction. Thus, typically, the payer and payee don't have access to the same ledger: transactions involve foreign exchange as both parties hold different currency pertaining to their locality. The transaction is executed through series of logical steps: the fund of local currency is traded for foreign currency—of the payee. This is where the counterparty, foreign exchange trader, gets the fund for conversion and sends them back to the payer, which is then ready for transfer to the payee. Such use of intermediaries is at most likely cause of settlement risk—where the currency is delivered without its conversion. Such cause of risk can be hindered via removal of intermediaries. (Achanta, 2018)

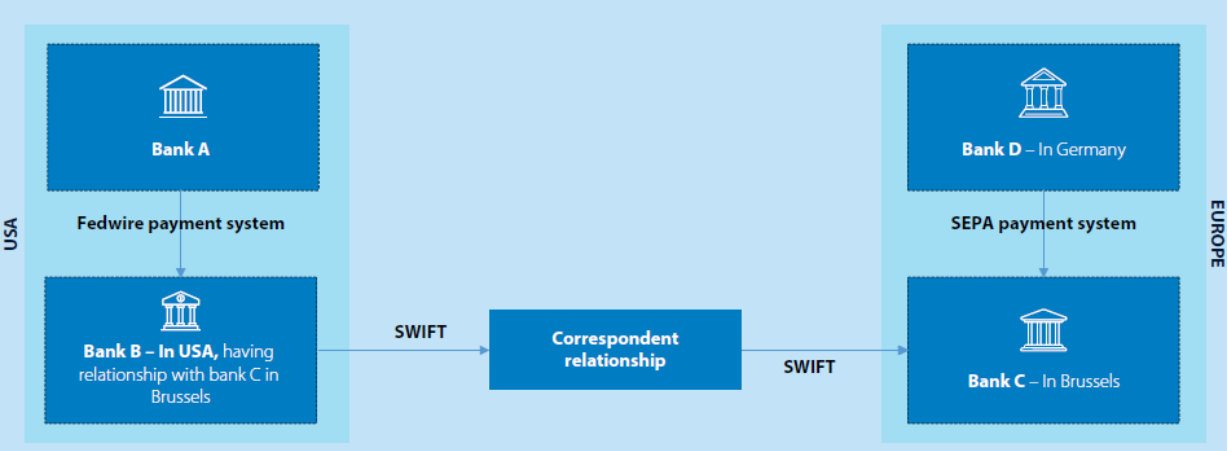


Figure 2.5: A payment flow-chart — Bank A sending euro amount to a euro bank account in bank D in Germany (Achanta, 2018).

Given the disadvantages of the current method of cross-border payments, blockchain technology and the concept of the distributed ledger has been widely talked about within the financial and banking sector. As a universal ledger in a distributed network, blockchain is available to everyone; a complete copy of the entire ledger is on each node in the network. For a ledger or entire database to be valid, it requires a consensus of nodes to validate for an agreement upon the state of the ledger, proving immutability. Thus, allows full public verifiability, as in any modifications in the ledger is then verified by other nodes. (Anon., n.d.)

Consequently, this proposes the understanding that money transfers can occur without concerns of manipulation, because blockchain technology eliminates involvement of any correspondent banks or intermediaries. Moreover, these exclusion of third parties reduce cost of processing payment and turnaround time for settlement (TAT), also eliminating the risk of discrepancies in record keeping. This is because the decentralised ledger holds irreversible and distributed record of every transaction available for all peers to see—enabling real time security. (MeikleJohn, 2018)

2.4 Solution 1: Using Stablecoins as means of fiat conversion

As indicated by the 2020 State of Stablecoins report, there are 134 stablecoins, of which thirty percent are live (**Anzalone, 2020)**. Stablecoins correspond to a characteristic answer for the extreme volatility most cryptocurrencies exude, this is because their values are backed by some form of asset. The advantage here is that stablecoins, without any obstacle, can be easily traded on the available platforms as the other cryptocurrencies. As previously mentioned, Ethereum is the most popular platform as of 2020. (**Anzalone, 2020)**.

2.4.1 Collateralisation:

Stablecoins are very flexible in terms of the asset their values are pegged to. For example, stablecoins can be backed by algorithms or collateral or which ascertain their true worth. Stablecoins that are collateral-backed can be valued by conventional collateral or an of the ubiquitous cryptocurrency. There are three types of stablcoins: fiat-collateralized, crypto-collateralized; and non-collateralized. Of the many conventional collaterals, they can range anywhere from fold to any fiat currency in the world.

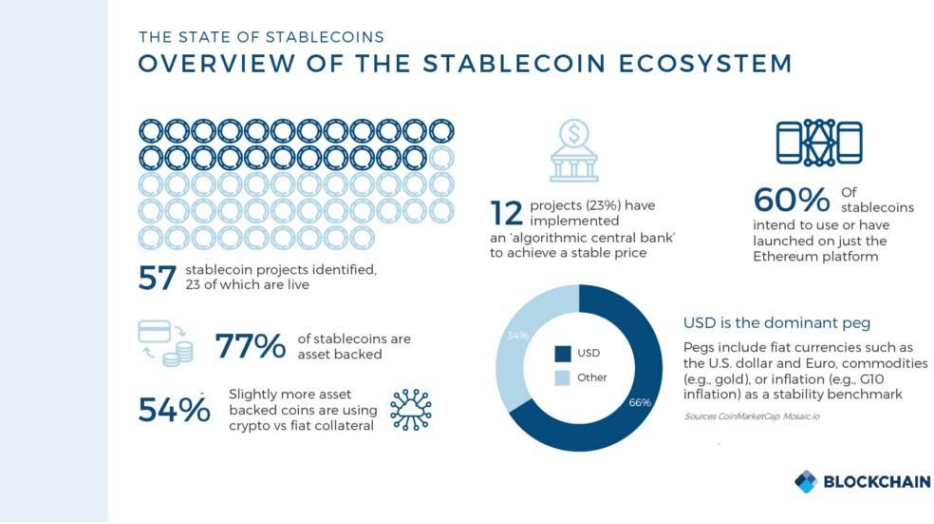


Figure 2.6: Overview of the stablecoin system

This project is concerned with fiat conversions from and into Ether, so no value is lost midst a transaction. Stablecoins have gained immense popularity among masses seeking such technology that will allow such transactions to occur. Such a technology's use case is to process these cross-border transaction without the need of an intermediary. However, stablecoins requires trust from an entity, thus requiring a third party.

Furthermore, investors and traders have intentions of receiving high returns and, consequently, would resort to other means of other financial gains. Moreover, because a cross-border system involves various types of fiat currencies, stablecoins would be unsuitable for such a system as they have undefined and poor regulations. Additionally, these regulations further places distrust and the

lack of a framework structure causes disconcertion amongst potential users further reducing its compatibility with the system. (Everything you need to know about Stablecoins and how they work, 2020)

However, despite the many positives of a stablecoin there is a major disadvantage that goes against choosing stablecoin as the means of fiat-conversion mediator . The reason is because stablecoins requires a third-party which breaks the main rule of a decentralized application. Moreover, external audits are required, in order to ensure the assets are accounted for. Moreover, as stablecoins are bind to traditional asset, they become depended on them, and on the traditional financial structure, which results in risks not found in conventional cryptocurrencies. (Bit.news, 2018)

2.5 Solution 2: Smart contracts

2.5.1 Unchecked External Calls:

In order to make transaction of Ethers or calling other smart contracts, there are external call functions that are provided by *Solidity*. The following methods: *address.call(), address.send(),* and *address.delegatecall(),* are likely to fail due to 'out-of-gas' errors or unidentified network (Solidity — Solidity 0.6.8 documentation, 2020). For example, the limit of gas for a fallback function is 2300, mentioned in Section 2.2.1. A Boolean value of *False* will be returned, whenever one of the functions is executed producing an error instead of throwing an exception. The code logic is prone to be incorrect when the return value is not checked by the method caller beforehand.

Furthermore, in a smart contract which involves transfer of Ether, functions *transfer* and *send* are called. These functions limit the gas of the fallback function to 2300, which is not a sufficient amount for the smart contract to write to *storage.* Moreover, an attacker can use the API function *selfdestruct(address_of_victim)* forcibly which does not result in a fallback, hence, the Ethers transferred cannot be rejected. Therefore, the equal balance logic fails due to the unexpected transfer.

2.5.2 Unable to interact with external data

Smart cannot directly interact with external data because the Ethereum blockchain is a consensus-based system, which means that for it to work properly every node has to reach an identical state whenever a transaction and a block is processed. Moreover, all the processes that takes place on a blockchain must be completely inevitable, with there being no possible way for any differences between subsequent blocks. Because a difference in the blockchain means the entire blockchain is invalid or broken—which, considering, its immutable state is an impossible feat. In other words, if two nodes are to contradict each other in regard to the chain's state, the entire blockchain becomes worthless.

As smart contracts are executed independent from on another by all nodes on a chain. Therefore, if

a smart contract receives data from an external source, this retrieval is performed repetitively and independently by every node. But because this is from an external source which falls outside of the blockchain, there is no assurance that each node will be receive the same data.

Therefore, the solution to the smart contract not functioning with external data is an ‘oracle smart contract’. This type of smart contract authenticates data from the external source and then stores it for future use. This is required as a smart contract cannot interact with the outside sources or in other words, pull data or receive or send request through the use of API as this prevents users from validation blockchain. (CoinDesk, 2016)

Chapter 3

Design and implementation

Provided the time constraint and its intensity being a final year project, below are the planned components that were meant to deliver a Minimum Viable Product for the Cross-Border Money Transfer Dapp (Decentralized Application), otherwise named as ‘XBorder’—which is a play on the “Cross-Border” phrase. XBorder aims to facilitate cross-border transactions, enabling fiat-currency-conversion when transferring. The main idea here is to use a cryptocurrency as the means for the value to transfer across the border into a different jurisdiction, being a different currency but keeping value. Thus, cryptocurrency, being decentralized, was the perfect solution and, unironically, chosen as the intermediary for these type of transactions.

3.1 Components

As mentioned above, the MVP ‘XBorder’ involves three main components:

- Smart contracts that is concerned with peer-to-peer Ether transfer from one wallet of the user to another.
- A frontend web application to facilitate the transaction in an easy-to-user environment
- External adapters implemented to be bridged at the node operator

3.2 Smart contract

3.2.1 Fiat payment method and trade pair

The Gateway smart contract keeps a registry of each market. A market is the combination of fiat payment method (eg. Paypal, SEPA, etc.) and a fiat / crypto pair (eg. ETH/USD).
The contract administrator adds new markets by calling addFiatPaymentMethod on the contract. After this senders may register liquidity with the market.

3.2.2 Sender registration

Anyone can sign-up as a sender/payer to provide liquidity for the given fiat-conversion function. To do so the sender will have to:

- Encrypt their API credentials for the fiat network and store them on InterPlanetary File System (IPFS)
- Encrypt their payment destination address for the fiat network and store it on IPFS
- Execute registerSender on the Gateway contract transferring:
    - Ether to be locked up for exchange
    - InterPlanetary File System hash of their encrypted fiat currency API credentials stored in an environment file which will not pushed on public service such as GitHub for security reasons. The Gateway smart contract keeps a registry of each number transfers of Ether to a Fiat currency
    - IPFS hash of their destination address

The smart contract stores a record of the sender in the smart contract then it transfers the details to the Paypal external adapter implemented and then bridged on the node operator of Chainlink. Then it is called by deploying on AWS Lambda so that it can be called via a node-job operated by chainlink's decentralized oracles.

3.2.3 Eth-to-Fiat conversion

Transfer of there is a minimal step process, as in there is only one, process for the receiver. They execute the function to call sellCryptoOrder on the Gateway contract transferring:

- Fiat-currency to purchase
- Ether to transfer which is held in the Escrow smart contract
- IPFS hash of encrypted destination of the fiat-currency at the end of the payment

sellCryptoOrder then call the Chainlink's oracle with the retrieved information. The external adapter then chooses a sender and transfers the Fiat-currency payment. On receiving of the Fiat-currency payment fulfillSellCryptoOrder is called on the Gateway smart contract with ID of the registered seller whose liquidity was used to make the transfer. The Ether is then transferred to that sender's Ethereum address.

3.2.4 Fiat-to-Eth conversion

Receiving is, on the other hand, a two-step process for the Receiver.

First, they make a call to buyCryptoOrderCreate to register to purchase Ether for Fiat-currency. They need to transfer:

- The Ethers to trade
- Number of Ether
- IPFS hash of encrypted receivers destination

buyCryptoOrderCreate will execute the Chainlink's oracle with the information. The external adapter will choose a sender and return the senders fiat-currency account number.

The Receiver executes the payment on the on-chain payment network and then executes buyCryptoOrderPayed to finalise the order. Again, Chainlink's Oracle will be called which is made to the paypal or binance external adapter. It will perform checks on the payment which was received and return a Boolean value for validation. The contract is releases the Ethers to the receiver from the Senders reserves.

3.3 Actors in the system

3.3.1 Sender

Senders are liquidity providers.

They must sign-up their fiat exchange network using credentials with the Chainlink's Oracle (for Ether selling orders) and stake their Ethers in the Gateway contract (for Ether conversion orders) in order for the FiatGateway system to execute orders on their behalf.

The Sender earns a fee from each order executed using their liquidity.

In this version Senders are not competitive as:

- sender selection
- fiat exchange prices are not set but taken from API requests

3.3.2 Receiver
Receivers are those who are receiving Ethers. This can be anyone.

3.3.3 Administrator
Is the owner of the Gateway smart contract.

Administrators adds FiatPayment network and tradeable pairs to the system by:

- launch an External adapter that handles the Fiat exchange transfer network
- connecting the External adapter to a Chainlink node operator
- call addFiatPayment on Gateway to register the payment method

3.4 Decentralized transfers

3.4.1 Prices

The current aggregated market price is used for all trades and a Senders fee (percentage) is take from each order.

3.4.2 Matching

A simple system is used to match a Receiver with a Sender.

This could later be replaced with a competitive Sender selection system with reputation and variable fees.

3.4.3 Trust
The gateway attempts to be as trustless as possible. When ChainLink supports TEE and system actors have a guarantee that a specific version of the Fiat payment adapter code is running it will be virtually trustless. Until then actors must trust that the external adapter (currently running as a serverless function on AWS) is running without modification.

3.4.4 Privacy

Fiat payment network account details are not revealed to anyone but the external adapter. This is achieved by encrypting the account destination with the external adapters public key. It is stored on IPFS and the IPFS hash is sent with the order so the adapter can decrypt it before filling the order

Chapter 4

Conclusion and future work

4.1 Critical overview of this project

Since the commencement of this project, several changes of varying severity levels were made. The initial project plan was to implement a blockchain system and a personal cryptocurrency. The aim was to remove high-profile drawbacks that exist within current existing cryptocurrencies (discussed in chapter 2). However, it became clear that the implementation process was expected to take longer than the time provided for this final year project.

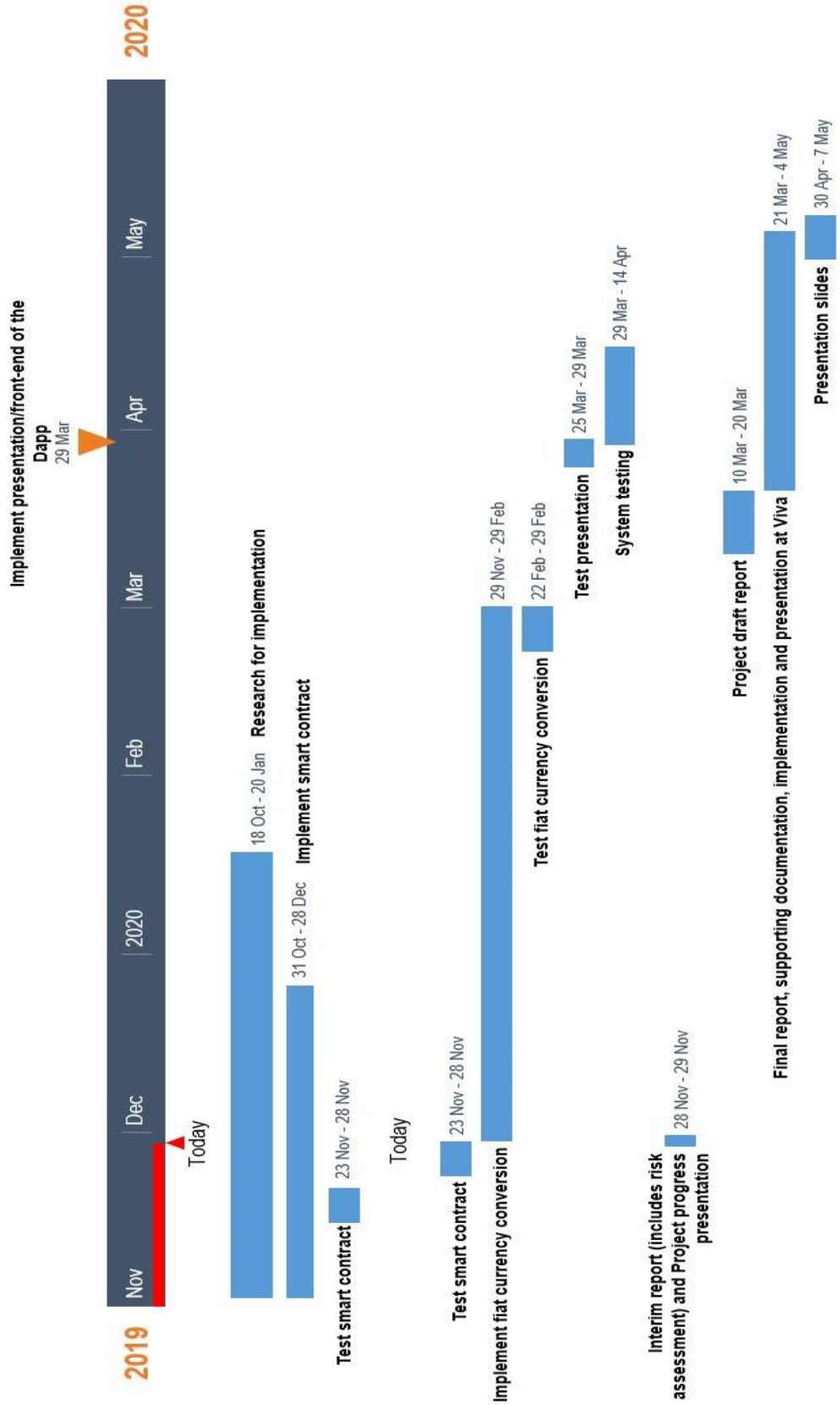
Due to these realisations, more time was taken to readjust and rederiving with a new plan. Further time was taken to learn about Ethereum smart contracts and the solidity programming language, which is used to write those smart contracts. However, the most critical and crucial part of this project is the conversion of fiat currency to, and from, cryptocurrency—which is decidedly ether. This part of the project is still in-research.

Current progress indicates that cryptocurrency transaction can be performed successfully, under testing environments only. However, these smart contracts are only executed using the accounts provided by Ganache-cli, which is part of the truffle suite of Ethereum development tools. (Truffle, 2019)

The final plan was to implement smart contracts and external adapters to connect to Chainlink so they can be bridged via Chainlink's node operators. However, midst this implementation several setbacks occurred, and due to the realisation of smart contracts true nature when interacting with external data, it became clear that smart contracts wouldn't work with the current determined plan.

Considering the time that was spent on research and learning these tools, the project's difficulty level is the actual problem. Since only one year is given to research and implement, more time was needed for proper learning and execution. Furthermore, having spoken to Chainlink's Vice President Andrew Thurmn it became clear that the vision of this project is correct, however the duration of the implementation was the main concern.

5. Gantt chart



Appendix A: Escrow smart contract test

Test code	Result and output (Pass √ || Fail X)

describe('Escrow Factory', () => {
  it('deploys an escrow contract', () => {
  assert.ok(escrow_factory.contract.options.address);
  assert.ok(escrow_contract.contract.options.address);
  });
 	
  it('emits an event when the escrow contract is deployed', () => {
  escrow_factory.getPastEvents('escrow_activated', (err, events) => {
  assert.notEqual(0, events.length, "No escrow_activated events were found");
  });
  });
 	
  it('sets the caller of open_escrow to be the payer on the escrow contract', **async** () => {
  let participants = **await** escrow_contract.get_participants();
  assert.equal(participants[0], accounts[0], "The payer was not properly set");
  });
 	
  it('creates a mapping entry for the newly created escrow under a session id',
  async () => {
  let session_id_hash = "0xdc345837d24517858368a28c2022936404ae5a64e78
dcac16331108d53eeca9c";
  let result = **await** escrow_factory.get_escrow(session_id_hash);
  assert.equal(result, escrow_contract.contract.options.address);
  });
  });
 	
	Contract: EscrowFactory
	  Escrow Factory
	  √ deploys an escrow contract
	  √ emits an event when the escrow contract is deployed
	  √ sets the caller of open_escrow to be the payer on the esc
	  √ creates a mapping entry for the newly created escrow un

```
describe("Escrow", () => {
    it('should allow a payee to deposit deposit_amount', async () => {
      let no_transfer_hash = "0xdc345837d24517858368a28c2022936404ae5a64e78dcac16331108d53eeca9c";
      const escrow_contract = await Escrow.deployed();
      let tx = await escrow_contract.deposit(no_transfer_hash, {
        from: accounts[1],
        value: web3.utils.toWei('5', 'ether')
      });
      console.log(tx.logs[0].event);
      assert.equal("payee_deposit_complete", tx.logs[0].event);
      let participants = await escrow_contract.get_participants();
      assert.equal(participants[1], accounts[1]);
    });
    it('should prevent the payer from depositing deposit_amount', async () => {
      try {
        let no_transfer_hash = "0xdc345837d24517858368a28c2022936404ae5a64e78dcac16331108d53eeca9c";
        let result = await escrow_contract.deposit(no_transfer_hash, {
          from: accounts[0],
          value: web3.utils.toWei('10', 'ether')
        });
        assert.fail('should have thrown before');
      }
      catch (err) {
        assert.include(err.message, "revert", "The error msg should contain 'revert'");
      }
    });
    it('should prevent the payee from depositing twice', async () => {
      let no_transfer_hash = "0xdc345837d24517858368a28c2022936404ae5a64e78dcac16331108d53eeca9c";
      let tx1 = await escrow_contract.deposit(no_transfer_hash, {
        from: accounts[1],
        value: web3.utils.toWei('5', 'ether')
      });
      try {
        let tx2 = await escrow_contract.deposit(no_transfer_hash, {
          from: accounts[1],
          value: web3.utils.toWei('7', 'ether')
        });
        assert.fail('should have thrown before');
      }
      catch (err) {
        assert.include(err.message,
          "Sorry the escrow is now locked.", "There error msg doesn't contain the require() error message");    }
    });
    it('should emit an event when a no transfer is requested', async () => {
      let tx = await escrow_contract.request_no_transfer({
        from: accounts[0]
      });
      assert.equal("no_transfer_requested", tx.logs[0].event);
    });
    it('should prevent anyone but the payer from requesting a no transfer',
      async () => {
        try {
          let tx = await escrow_contract.request_no_transfer({
            from: accounts[1]
          });}
        catch (err) {
          assert.include(err.message, "Only the payer can request a no transfer.",
            "The error msg doesn't contain the require() error message");
        }
    });
  });
```

```
  Escrow
    1) should allow a payee to deposit deposit_amount
    Events emitted during test:
    ---------------------------
    EscrowFactory.escrow_activated(
      session_id_hash: 0xdc345837d24517858368a28c2022936404ae5a64
    e78dcac16331108d53eeca9c (type: bytes32),
      escrow_contract: 0x77499376d6Be02A7762e99d3E
    3FAb9C77F27Dd82 (type: address)
    )
    ---------------------------
    √ should prevent the payer from depositing deposit_amount (106ms)
    2) should prevent the payee from depositing twice
    Events emitted during test:
    ---------------------------
    EscrowFactory.escrow_activated(
      session_id_hash: 0xdc345837d24517858368a28c2022936404ae5a64
    e78dcac16331108d53eeca9c (type: bytes32),
      escrow_contract: 0xfB35E324FCB2Dd58E72a0a
    41338908353d94B138 (type: address)
    )
    ---------------------------
    √ should emit an event when a no transfer is requested (70ms)
    √ should prevent anyone but the payer from requesting
    a no transfer (108ms)
```

```
describe("check_hold Deposit Factory", () => {

    it("deploys a check_hold deposit contract", async () => {
      assert.ok(hd_factory.contract.options.address);
      assert.ok(hd.contract.options.address);
    });
    it('emits an event when the check_hold contract is deployed', () => {
      hd_factory.getPastEvents('check_hold_deposit_created', (err, events) => {
        assert.notEqual(0, events.length, "No check_hold_deposit_created events were found");
      });
    });
    it('sets the caller of open_check_hold deposit to be the payer on the check_hold contract', async () => {
      let participants = await hd.get_participants();
      assert.equal(participants[0], accounts[0], "The payer was not properly set");
    });
    it('creates a mapping entry for the newly created check_hold contract under a session id',
      async () => {
      let session_id_hash = "0xdc345837d24517858368a28c2022936404ae5a64e78dcac16331108d53eeca9c";
      let result = await hd_factory.get_check_hold_deposit_contract(session_id_hash);
      assert.equal(result, hd.contract.options.address);
    });
  });
```

```
Contract: HoldingDepositFactory
  check_hold Deposit Factory
    √ deploys a check_hold deposit contract
    √ emits an event when the check_hold
contract is deployed
    √ sets the caller of open_check_hold deposit
to be the payer on the check_hold contract (39ms)
    √ creates a mapping entry for the newly created
check_hold contract under a session id
```

```
describe("check_hold Deposit", () => {
    describe("Constructor", () => {
      it("Sets the payer, status and opens the check_hold", async () => {
        let participants = await hd.get_participants();
        let payer = participants[0];
        assert.equal(payer, accounts[0]);
        let payer_curr_status = await hd.get_payer_curr_status();
        assert.equal(payer_curr_status, true);
        let open = await hd.is_open();
        assert.equal(open, true);
      });
    });
    describe("Depositing deposit_amount", () => {
      it("sets up the payee, their deposit and status in the check_hold contract",
        async () => {
        let tx = await hd.deposit_amount({
          from: accounts[1],
          value: web3.utils.toWei('1', 'ether')
        });
        let participants = await hd.get_participants();
        assert.equal(participants[1], accounts[1]);
        let payee_curr_status = await hd.get_payee_curr_status();
        assert.equal(payee_curr_status, true);
        let deposit = await hd.get_deposit_amount();
        assert.equal(Number(deposit), Number(web3.utils.toWei('1', 'ether')));
      });
      it("prevents the payer from depositing into their own check_hold", async () =>
        try {
          await hd.deposit_amount({
            from: accounts[0],
            value: 10
          });
        }
        catch (err) {
          assert.include(err.message, "revert");
        }
      });
      it("emits an event when the deposit is complete", async () => {
        let tx = await hd.deposit_amount({
          from: accounts[1],
          value: 10
        });
        assert.equal("deposit_amount_deposited", tx.logs[0].event);
      });
    });
```

```
check_hold Deposit
    Constructor
     √ Sets the payer, status and opens the
check_hold (105ms)
    Depositing deposit_amount
     3) sets up the payee, their deposit
and status in the check_hold contract
  Events emitted during test:
  ---------------------------
  HoldingDepositFactory.check_hold_deposit_created(
    session_id_hash: 0xdc345837d24517858368a28c20
22936404ae5a64e78dcac16331108d53eeca9c
(type: bytes32),
    check_hold_deposit_contract: 0x158ef9cF5B
7FCd2d703bD5570ac063a95Cc6540A (type: address))
√ prevents the payer from depositing into their
own check_hold (81ms)
     4) emits an event when the deposit is complete
  Events emitted during test:
---------------------------
  HoldingDepositFactory.check_hold_deposit_created(
    session_id_hash: 0xdc345837d24517858368a28c
2022936404ae5a64
e78dcac16331108d53eeca9c (type: bytes32),
    check_hold_deposit_contract: 0x9eeF92748A
D2CAF462a7285F527F164C2A0b509B (type: address))
```

```
it("lets the payee chec_curr their status, refunding them the deposit", async () => {
        let tx = await hd.deposit_amount({
          from: accounts[1],
          value: web3.utils.toWei('5', 'ether')
        });

        let pre_balance = Number(await web3.eth.getBalance(accounts[1]));

        let tx2 = await hd.check_curr_status(false, {
          from: accounts[1]
        });

        let post_balance = Number(await web3.eth.getBalance(accounts[1]));

        let deposit = Number(await hd.get_deposit_amount());
        assert.equal(deposit, 0);
        assert.isAbove(post_balance, pre_balance);
      });
it("emits an event when the status is deactivated", async () => {
        let tx = await hd.check_curr_status(false, {
          from: accounts[0]
        });
        assert.equal("curr_status_checkd", tx.logs[0].event);
      });
```

```
Updating status
    5) lets the payee check their current status,
refunding them the deposit
  Events emitted during test:
  ---------------------------
  HoldingDepositFactory.check_hold_deposit_created(
    session_id_hash: 0xdc345837d24517858368a28c20229364
64e78dcac16331108d53eeca9c (type: bytes32),
    check_hold_deposit_contract: 0x2dcb82d6
692A005016847EcAA44e57609782F410 (type: address))
  √ emits an event when the status is deactivated(84ms)
```

```
it("forfeits deposit", async () => {
        // payee deposits deposit_amount
        let tx1 = await hd.deposit_amount({
          from: accounts[1],
          value: web3.utils.toWei('5', 'ether')
        });

        let deposit_refund_checker_1 = await hd.is_refund_checker();
        assert.equal(true, deposit_refund_checker_1);

        // payer chec_currs deposit status
        let tx3 = await hd.chec_curr_deposit_curr_status(false, {
          from: accounts[0]
        });

        let deposit_refund_checker_2 = await hd.is_refund_checker();
        assert.equal(false, deposit_refund_checker_2);
        assert.equal("deposit_curr_status_checkd", tx3.logs[0].event);

        let pre_payer_balance = Number(await web3.eth.getBalance(accounts[0]));

        // payee chec_currs their status
        let tx4 = await hd.check_curr_status(false, {
          from: accounts[1]
        });

        let post_payer_balance = Number(await web3.eth.getBalance(accounts[0]));
        assert.isAbove(post_payer_balance, pre_payer_balance);

        let deposit = await hd.get_deposit_amount();
        assert.equal(0, deposit);
      });
```

```
    Handles a Complete check_hold Deposit from start to finish
      6) forfeits deposit
  Events emitted during test:
  ---------------------------
  HoldingDepositFactory.check_hold_deposit_created(
    session_id_hash: 0xdc345837d24517858368a28c20229364
4ae5a64e78dcac16331108d53eeca9c (type: bytes32),
    check_hold_deposit_contract: 0x89b4F6Db6
86F9721e9cb288111E33249f00b619b (type: address))
```

```
it("redeposit_amount deposit", async () => {
        // payee deposits deposit_amount
        let tx1 = await hd.deposit_amount({
          from: accounts[1],
          value: web3.utils.toWei('5', 'ether')
        });
        let pre_payee_balance = Number(await web3.eth.getBalance(accounts[1]));
        // payee chec_currs their status
        let tx4 = await hd.check_curr_status(false, {
          from: accounts[1]
        });
        let post_payee_balance = Number(await web3.eth.getBalance(accounts[1]));
        assert.isAbove(post_payee_balance, pre_payee_balance);
        let deposit = await hd.get_deposit_amount();
        assert.equal(0, deposit);
      });
```

```
7) redeposit_amount deposit
  Events emitted during test:
  ---------------------------
  HoldingDepositFactory.check_hold_deposit_created(
    session_id_hash:
0xdc34b837d24517858368a28c2022936404ae5a64e78dcac16
    check_hold_deposit_contract: 0x494A7D854f523CcA21b0d
address))
```

Appendix B: Chainlink smart contract tests

Test code	Result and output (Pass √ || Fail X)

describe('#createRequest', () => {
  context('without LINK', () => {
  it('reverts', **async** () => {
  await h.assertActionThrows(**async** () => {
  await cc.createRequestTo(
  oc.address,
  jobId,
  payment,
  url,
  path,
  times,
  { from: consumer },)})})})
 	
  context('with LINK', () => {
  let request
 	
  beforeEach(**async** () => {
  await link.transfer(cc.address, web3.utils.toWei('1', 'ether'))
  })
  context('sending a request to a specific oracle contract address', () => {
  it('triggers a log event in the new Oracle contract', **async** () => {
  const tx = **await** cc.createRequestTo(
  oc.address,
  jobId,
  payment,
  url,
  path,
  times,
  { from: consumer },
)
  request = h.decodeRunRequest(tx.receipt.rawLogs[3])
  assert.equal(oc.address, tx.receipt.rawLogs[3].address)
  assert.equal(
  request.topic,
  web3.utils.keccak256(
  'OracleRequest(bytes32,address,bytes32,uint256,
address,bytes4,uint256,uint256,bytes)',
),)})})}) })
	#createRequest
	  without LINK
	  √ reverts (148ms)
	  with LINK
	  sending a request to a specific oracle contrac
	  √ triggers a log event in the new Oracle cont

```
  describe('#fulfill', () => {
    const expected = 50000
    const response = web3.utils.toHex(expected)
    let request

    beforeEach(async () => {
      await link.transfer(cc.address, web3.utils.toWei('1', 'ether'))
      const tx = await cc.createRequestTo(
        oc.address,
        jobId,
        payment,
        url,
        path,
        times,
        { from: consumer },
      )
      request = h.decodeRunRequest(tx.receipt.rawLogs[3])
      await h.fulfillOracleRequest(oc, request, response, {
from: oracleNode })
    })

    it('records the data given to it by the oracle', async () => {
      const currPrice = await cc.data.call()
      assert.equal(
        web3.utils.toHex(currPrice),
        web3.utils.padRight(expected, 64),
      ) })

    context('when my contract does not recognize the request ID', () => {
      const otherId = web3.utils.toHex('otherId')
      beforeEach(async () => {
        request.id = otherId
      })

      it('does not accept the data provided', async () => {
        await h.assertActionThrows(async () => {
          await h.fulfillOracleRequest(oc, request, response, {
            from: oracleNode,})})})})
context('when called by anyone other than the oracle contract', () => {
      it('does not accept the data provided', async () => {
        await h.assertActionThrows(async () => {
          await cc.fulfill(request.id, response, { from: stranger })
})})})})
```

```
#fulfill
    √ records the data given to it by the oracle
    when my contract does not recognize the requ
      √ does not accept the data provided (84ms)
    when called by anyone other than the oracle c
      √ does not accept the data provided (77ms)
```

```
describe('#cancelRequest', () => {
  let request

  beforeEach(async () => {
   await link.transfer(cc.address, web3.utils.toWei('1', 'ether'))
   const tx = await cc.createRequestTo(
    oc.address,
    jobId,
    payment,
    url,
    path,
    times,
    { from: consumer },
   )
   request = h.decodeRunRequest(tx.receipt.rawLogs[3])
  })

  context('before the expiration time', () => {
   it('cannot cancel a request', async () => {
    await h.assertActionThrows(async () => {
     await cc.cancelRequest(
      request.id,
      request.payment,
      request.callbackFunc,
      request.expiration,
      { from: consumer },)})})})
  context('after the expiration time', () => {
   beforeEach(async () => {
    await h.increaseTime5Minutes()})
   context('when called by a non-owner', () => {
    it('cannot cancel a request', async () => {
     await h.assertActionThrows(async () => {
      await cc.cancelRequest(
       request.id,
       request.payment,
       request.callbackFunc,
       request.expiration,
       { from: stranger },)})}) })
   context('when called by an owner', () => {
    it('can cancel a request', async () => {
     await cc.cancelRequest(
      request.id,
      request.payment,
      request.callbackFunc,
      request.expiration,
      { from: consumer }, )})})})})
```

```
#cancelRequest
    before the expiration time
     √ cannot cancel a request (92ms)
    after the expiration time
     when called by a non-owner
      √ cannot cancel a request (85ms)
     when called by an owner
      √ can cancel a request (114ms)
```

```
describe('#withdrawLink', () => {
  beforeEach(async () => {
    await link.transfer(cc.address, web3.utils.toWei('1', 'ether'))
  })
  context('when called by a non-owner', () => {
    it('cannot withdraw', async () => {
      await h.assertActionThrows(async () => {
        await cc.withdrawLink({ from: stranger })
      })})})

  context('when called by the owner', () => {
    it('transfers LINK to the owner', async () => {
      const beforeBalance = await link.balanceOf(consumer)
      assert.equal(beforeBalance, '0')
      await cc.withdrawLink({ from: consumer })
      const afterBalance = await link.balanceOf(consumer)
      assert.equal(afterBalance, web3.utils.toWei('1', 'ether'))
    })})})})
```

```
#withdrawLink
    when called by a non-owner
      √ cannot withdraw (70ms)
    when called by the owner
      √ transfers LINK to the owner (155ms)
```

```
contract("Verifier", accounts => {
  let verifier_instance;

  beforeEach(async () => {
    verifier_instance = await Verifier.new();
  });

  it("should emit an event when verification is requested", async () => {
    let property_uid = "0xdc345837d24517858368a28c2
022936404ae5a64e78dcac16331108d53eeca9c";

    let result = await verifier_instance.verify(property_uid);
    assert.equal(result.logs[0].event, "verification_requested");
  });

  it("should emit the property_uid as part of the event", async () => {
    let property_uid = "0xdc345837d24517858368a28c20229
36404ae5a64e78dcac16331108d53eeca9c";
    let result = await verifier_instance.verify(property_uid);
    assert.equal(result.logs[0].args.property_uid, property_uid);
  });});
```

```
Contract: Verifier
  √ should emit an event when verification is requ
  √ should emit the property_uid as part of the ev
```

Appendix C: Paypal external adapter tests

Test code	Result and output (Pass √ || Fail X)

it('should fail on invalid method', (done) => {
// Notice method not set.
requestWrapper(req).then((response) => {
assert.equal(response.statusCode, 400, "status code");
assert.equal(response.jobRunID, jobID, "job id");
assert.isUndefined(response.data, "response data");
done();
}).catch((error)=> {
assert.isNotOk(error,'Promise error');
});
});
	√ should fail on invalid method

it('should send payment/payout', (done) => {
req.data = <SendRequest>{
method: "sendPayout",
amount: process.env.TEST_AMOUNT || 10,
currency: process.env.TEST_CURRENCY || "GBP",
receiver: process.env.TEST_RECEIVER ||
"your-buyer@example.com"
};
requestWrapper(req).then((response) => {
assert.equal(response.statusCode, 201, "status code");
assert.equal(response.jobRunID, jobID, "job id");
assert.isNotEmpty(response.data, "response data");
assert.isNotEmpty(response.data.result, "payout id");
payoutId = response.data.batch_header.payout_batch_id;
done();
}).catch((error)=> {
assert.isNotOk(error,'Promise error');
});
}).timeout(timeout);
	√ should send payment/payout (1483ms)

it('should get payout details', (done) => {
req.data = <GetRequest>{
method: "getPayout",
payout_id: process.env.TEST_PAYOUT_ID || payoutId
};
requestWrapper(req).then((response) => {
assert.equal(response.statusCode, 200, "status code");
assert.equal(response.jobRunID, jobID, "job id");
assert.isNotEmpty(response.data, "response data");
assert.isNotEmpty(response.data.result, "payout id");
done();
}).catch((error)=> {
assert.isNotOk(error,'Promise error');
});
}).timeout(timeout);
	√ should get payout details (517ms)

it('should get payout details using ENV variable', (done) => {
process.env.API_METHOD = "getPayout";
req.data = <Request>{
method: "sendPayout",
payout_id: process.env.TEST_PAYOUT_ID || payoutId
};
requestWrapper(req).then((response) => {
assert.equal(response.statusCode, 200, "status code");
assert.equal(response.jobRunID, jobID, "job id");
assert.isNotEmpty(response.data, "response data");
assert.isNotEmpty(response.data.result, "payout id");
done();
}).catch((error)=> {
assert.isNotOk(error,'Promise error');
});
}).timeout(timeout);
	√ should get payout details using ENV variable (521ms)

it('should fail sendPayout with missing amount', (done) => {
req.data = <SendRequest>{
method: "sendPayout",
receiver: "sb-taiki1615866@personal.example.com"
};
requestWrapper(req).then((response) => {
assert.equal(response.statusCode, 400, "status code");
assert.equal(response.jobRunID, jobID, "job id");
assert.isUndefined(response.data, "response data");
done();
}).catch((error)=> {
assert.isNotOk(error,'Promise error');
});
}).timeout(timeout);
	√ should fail sendPayout with missing amount

it('should fail sendPayout with missing receiver', (done) => {
req.data = <Request>{
method: "sendPayout",
amount: 10
};
requestWrapper(req).then((response) => {
assert.equal(response.statusCode, 400, "status code");
assert.equal(response.jobRunID, jobID, "job id");
assert.isUndefined(response.data, "response data");
done();
}).catch((error)=> {
assert.isNotOk(error,'Promise error');
});
}).timeout(timeout);
	√ should fail sendPayout with missing receiver

it('should fail getPayout with missing payout id', (done) => {
req.data = <GetRequest>{
method: "getPayout"
};
requestWrapper(req).then((response) => {
assert.equal(response.statusCode, 400, "status code");
assert.equal(response.jobRunID, jobID, "job id");
assert.isUndefined(response.data, "response data");
done();
}).catch((error)=> {
assert.isNotOk(error,'Promise error');
});
}).timeout(timeout);
	√ should fail getPayout with missing payout id

Appendix D: Blockchain implementation example

```
# creating a blockchain
class Blockchain:
    def __init__(self):
        self.chain = [] #contains blocks
        self.transactions = [] # crypto
        self.create_block(proof=1, previous_hash='0')
        self.node = set()

    def create_block(self, proof, previous_hash):
        block = {'index': len(self.chain) + 1,
                 'timestamp': str(datetime.datetime.now()),
                 'proof': proof,
                 'previous_hash': previous_hash,
                 'transactions': self.transactions}
        self.transactions = []
        self.chain.append(block)
        return block

    def get_prev_block(self):
        return self.chain[-1]

    def proof_of_work(self, previous_proof):
        new_proof = 1
        check_proof = False
        while check_proof is False:
            hash_operation = hashlib.sha256(str(new_proof**2-previous_proof**2).encode()).hexdigest()
            if hash_operation[:4] == '0000':
                check_proof = True
            else:
                new_proof += 1
        return new_proof

    def hash(self, block):
        encoded_block = json.dumps(block, sort_keys=True).encode()
        return hashlib.sha256(encoded_block).hexdigest()

    def is_chain_valid(self, chain):
        previous_block = chain[0]
        block_index = 1
        while block_index < len(chain):
            block = chain[block_index]
            if block['previous_hash'] != self.hash(previous_block):
                return False
            previous_proof = previous_block['proof'];
            proof = block['proof']
            hash_operation = hashlib.sha256(str(proof ** 2 - previous_proof ** 2).encode()).hexdigest()
            if hash_operation[:4] != '0000':
                return False
            previous_block = block
            block_index += 1
        return True

    def add_transactions(self, sender, receiver, amount):
        self.transactions.append({'sender': sender,
                                  'receiver': receiver,
                                  'amount': amount})
        previous_block = self.get_previous_block()
        return previous_block['index'] + 1
```

Bibliography

Bitcoin Wiki. *Bitcoin History* and *Genesis Block.* Web. Accessed 5 Feb 2019. pages 5, 6

MeikleJohn, S., 2018. *The-Future-is-Decentralised,* s.l.: United Nations Development Programme.

Nakamoto, S., 2008. *Bitcoin: A Peer-to-Peer Electronic Cash System,* s.l.: Bitcoin.

Schulpen, R., 2018. *Smart contracts in the Netherlands,* Tilburgh: Tilburg University. Szabo, N., 1994. *Smart Contracts,* Amsterdam: Phonetic Sciences.

Tapscott, D., 2016. *Blockchain Revolutiom.* s.l.:Penguin Random House.

Truffle, 2019. Truffle suite. [Online]Available at: https://www.trufflesuite.com/ganache [Accessed November–— 2019].

Iinuma, Arthur, 2018. Why Is the Cryptocurrency Market So Volatile: Expert Take. [Accessed 1 March 2020]. pages 6

Wikipedia. 2018 Cryptocurrency Crash. Web. Accessed 21 Jan 2020. pages 5

Achanta, R., 2018. *Cross-border money transfer - enabled by Big Data,* s.l.: InfoSys.

Anon., n.d. *The essential guide to blockchain,* s.l.: ICAEW.

Solidity.readthedocs.io. 2020. *Solidity — Solidity 0.6.8 Documentation*. [online] Available at: <https://solidity.readthedocs.io/en/v0.6.8/> [Accessed 24 April 2020].

G. Wood, 2014. “Ethereum: A secure decentralised generalised transaction ledger,”, Ethereum Project Yellow Paper. Available at: https://gavwood.com/paper.pdf (Accessed: 13 April 2020).

Jiachi Chen, 2020. “Defining Smart Contract Defects on Ethereum”, Transactions on Software Engineering. Available at: https://arxiv.org/pdf/1905.01467.pdf (Accessed: 15 May 2020).

Flexiple Blog. 2020. *Everything You Need To Know About Stablecoins And How They Work*. [online] Available at: <https://blog.flexiple.com/what-are-stablecoins-and-how-do-they-work/> [Accessed 24 May 2020].

PayPal Engineering – Medium *(2020). Available at: https://medium.com/paypal-engineering (Accessed: 24 May 2020).*

The Coinbase Blog *(2020). Available at: https://blog.coinbase.com/ (Accessed: 21 May 2020).*

MEW (MyEtherWallet) Works With Cloudflare to Keep Users Safe | Cloudflare *(2020). Available at: https://www.cloudflare.com/case-studies/mew-myetherwallet/ (Accessed: 24 May 2020).*

Still Among the First: Why Do Investors Buy Ethereum? | FinSMEs *(2019). Available at: http://www.finsmes.com/2019/03/still-among-the-first-why-do-investors-buy-ethereum.html (Accessed: 24 May 2020).*

Coinbase Pro to Enable Chainlink Trading - CoinDesk *(2019). Available at: https://www.coindesk.com/coinbase-pro-to-enable-chainlink-trading (Accessed: 24 May 2020).*

Metamask Wallet for Chrome, Firefox, Brave Browsers – BitcoinWiki *(2020). Available at: https://en.bitcoinwiki.org/wiki/Metamask (Accessed: 23 May 2020).*

InterPlanetary File System *(2020). Available at: https://en.wikipedia.org/wiki/InterPlanetary_File_System (Accessed: 24 May 2020).*

Infura, a ConsenSys Company, Launches Infura+ for Improved Ethereum Infrastructure Support | ConsenSys *(2020). Available at: https://consensys.net/blog/press-release/press-release-infura-a-consensys-company-launches-infura-for-improved-ethereum-infrastructure-support/ (Accessed: 24 May 2020).*

Setting up the Go Ethereum (geth) environment in Ubuntu Linux *(2018). Available at: https://medium.com/@priyalwalpita/setting-up-the-go-ethereum-geth-environment-in-ubuntu-linux-67c09706b42 (Accessed: 24 May 2020).*

Features *(2020). Available at: https://chain.link/features/ (Accessed: 24 May 2020).*

Rubasinghe, Iresha. (2017). Transaction Verification Model over Double Spending for Peer-to-Peer Digital Currency Transactions based on Blockchain Architecture. International Journal of Computer Applications. 163. 24-31. 10.5120/ijca2017913531.

Hyperledger – Open Source Blockchain Technologies *(2020). Available at: https://www.hyperledger.org/ (Accessed: 25 May 2020).*

Ledger *(2020). Available at: https://www.ledger.com/academy/crypto/what-are-erc20-tokens (Accessed: 25 May 2020).*

ERC20 Token Standard *(2020). Available at: https://en.bitcoinwiki.org/wiki/ERC20 (Accessed: 25 May 2020).*

Houben, Robby, et. al. (2018) Europarl.europa.eu. Available at: https://www.europarl.europa.eu/cmsdata/150761/TAX3%20Study%20on%20cryptocurrencies%20and%20blockchain.pdf (Accessed: 25 May 2020).

Anzalone, R. (2020) Stablecoins Are Legion, And More Are Coming. Are We Creating A New Crisis?, Forbes. Available at: https://www.forbes.com/sites/robertanzalone/2020/01/30/stablecoins-are-legion-and-more-are-coming-are-we-creating-a-new-crisis/#46576bf5541d (Accessed: 25 May 2020).

Bit.news. (2018) Stablecoin: Advantages And Disadvantages. Available at: https://en.bit.news/stablecoin-advantages-and-disadvantages/ (Accessed: 25 May 2020).

Architecture for Blockchain Applications (2020). Available at: https://books.google.co.uk/books?id=2dyLDwAAQBAJ&pg=PA154&lpg=PA154&dq=smart+contract+cannot+interact+external+api&source=bl&ots=Is0JktDQcD&sig=ACfU3U2M7NsI89w6UlkMZm9pjMBayKyWVA&hl=en&sa=X&ved=2ahUKEwjX4-Wy0M7pAhVnSxUIHbOsDL4Q6AEwBHoECAoQAQ#v=onepage&q=smart%20contract%20cannot%20interact%20external%20api&f=false (Accessed: 25 May 2020).

CoinDesk (2016), Why Many Smart Contract Use Cases Are Simply Impossible. Available at: https://www.coindesk.com/three-smart-contract-misconceptions (Accessed: 25 May 2020).