IMPLEMENTING A STABLECOIN ON AN EXISTING BLOCKCHAIN USING SMART CONTRACTS.

by

JAMIE MICHEL
URN: 6426374

A dissertation submitted in partial fulfilment of the
requirements for the award of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

May 2021

Department of Computer Science
University of Surrey
Guildford GU2 7XH

Supervised by: Helen Treharne

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

Jamie Michel
May 2021

Abstract

Stablecoins play a crucial role in the current push for Cryptocurrency adoption; the volatility of traditional Cryptocurrency assets such as Bitcoin hinders most businesses and individuals from using them as a means of payment. Indeed, coins such as Bitcoin, DOGE, and countless others have become vehicles for speculative investment rather than disrupting the global banking system as intended.

Whilst early success has been found in creating fiat-backed stablecoins such as Tether, the tides are turning as people search for a more trustless option. Projects such as Dai are leading this new wave of stablecoins and promise to deliver a stable asset pegged to 1 USD through purely on-chain, verifiable collateral.

This project introduces an initial design for an alternative Cryptocurrency-backed stablecoin, which is pegged to the value of 1 GBP using concepts inspired by successes within the industry, such as Dai. There is a need for such stablecoin; UK businesses and individuals are much more likely to adopt a stablecoin representative of the native currency. Before introducing the novel approach, an initial implementation will be presented that could be used for the simplest possible on-chain representation of 1 GBP.

The result of this project is a dual-coin stablecoin system that attempts to maintain a market value of 1 GBP through the holding of Ether as collateral and the sale of a secondary coin that de-risks the stablecoin. It is entirely transparent as all transactions, code, and variables can be verified on the blockchain at any given moment. Moreover, a responsive web application has been implemented to interface between the on-chain system and a users Cryptocurrency wallet. Due to its implementation on an Ethereum network, the stablecoin benefits from all the strengths of the underlying blockchain protocol and is universally accepted by popular Cryptocurrency wallets and decentralised exchanges.

Acknowledgements

The completion of this project would not have been possible without the help and guidance from my project supervisor Helen Treharne, I am extremely grateful for the many hours spent on Microsoft Teams discussing the direction and scope of the project. Helen truly exceeded her duties and went the extra mile to provide support during the difficult work environment of 2020-2021.

Additional gratitude is expressed to Luke Riley, whose invaluable suggestions and ideas molded the project from the beginning. Furthermore, the learning resources he provided based on his industry experience were vital for developing robust technologies upon the Ethereum blockchain.

Finally, I am thankful for my friends and family's continuous support - in particular the proof reading, second opinions, and nourishment.

Contents

List of Figures

List of Tables

Abbreviations

BTC	Bitcoin
BCS	Chartered Institute for IT
PoW	Proof of Work
PoS	Proof of Stake
USDT	Tether
DAI	Dai
XAUT	TetherGold
ESD	EmptySetDollar
QTM	Quantity Theory of Money
USDC	USD Coin
PAXG	PAX Gold
CDP	Collateralized Debt Position
EVM	Ethereum Virtual Machine
PoA	Proof of Authority
DeFi	Decentralized Finance
Dapp	Decentralized App
ABI	Application Binary Interface

Chapter 1

Introduction

1.1 Motivation

In the midst of the global financial crisis, on January 3rd, 2009, the first block of the Bitcoin [1] network created by the elusive Nakamoto was mined [2]. Bitcoin (BTC) was the very first decentralized digital currency which leverages blockchain technology to create an immutable distributed ledger of all transactions. The term "Cryptocurrency" was coined to refer to such systems. Bitcoin's stated goal is to be an "electronic payment system based on cryptographic proof instead of trust" which would remove the need to trust third parties such as traditional banks.

Initially, Bitcoin was seen as a novelty and rarely used for meaningful transactions; in fact, the first recorded use of Bitcoin to make a tangible purchase was over a year later in May of 2010 [3]. At the time, 10,000 Bitcoin seemed like a fair payment for two pizzas, as this was valued at 41 US Dollars. Ten years later, this same transaction would be valued at 153,993,000 US Dollars. This incredible growth in value indicates one of the key problems with Cryptocurrencies.

Currency, regardless of its form, must serve three primary functions within an economy: it must function as a store of value, a unit of account, and a medium of exchange [4]. Cryptocurrencies such as Bitcoin clearly achieve being a store of value. Yet, extreme volatility driven by their speculative nature means that the other two functions are not met. stablecoins attempt to bridge the gap and fulfill these functions by "pegging" their value to a more stable asset such as the value of gold or the US dollar. In this case, transactions can be made with the benefits of blockchain technology without volatility.

Most stablecoins can be grouped into the category of asset-backed stablecoins; the asset could be commodities, fiat money, or Cryptocurrency assets. Fiat money refers to a "paper currency which is decreed by governments as legal tender" [4]; therefore fiat backed coins are issued with a promise to store fiat in a bank account with a ratio of 1:1. An example would be Tether, which claims to store 1 US Dollar in a secure account for every Tether unit that is minted [5]. Crypto-backed coins employ a similar technique; the main difference lies in the collateral being another digital asset – for example, the Dai [6] from MakerDAO, which leverages the native currency of the Ethereum [7] network, alongside several other Cryptoassets as the collateral to peg the coin to the value of 1 US Dollar. The former is far easier to implement as one may imagine. However, it undermines Cryptocurrency's fundamental ideas as one must trust that the token's issuer does hold the correct amount of fiat – this can lead to controversy and speculation, especially when it is not possible to audit this balance [8]. Stabilising one digital asset through collateral of another volatile asset is no easy task; it is achieved through the application of economic game theory to create a set of algorithms which operate on the blockchain.

1.2 Aims and Objectives

The overall aim of this project is to implement a stablecoin which is backed by Cryptocurrency assets. However, there are several aims which must be achieved for such a piece of work to be delivered. This section details those aims, alongside specific objectives that are essential for setting the scope of the research and development.

1.2.1 Aims

1. To investigate the current stablecoins and gain an understanding of the underlying blockchain technology.
2. To create a fully functioning Cryptocurrency asset on a blockchain.
3. To create a system that aims to peg the value of the token to 1 GBP through the use of Cryptocurrency asset collateral and smart contract [9] algorithms.

1.2.2 Objectives

1. Examine protocols that allow blockchains to function at great depth.
2. Perform a literature survey on existing stablecoins
3. Research the feedback mechanisms employed by Dai and other stablecoins, which stabilize the Cryptocurrency; understand the economic principles which are applied.
4. Select a blockchain on which to develop a stablecoin on.
5. Learn how to write and deploy smart contracts on a blockchain.
6. Implement a stablecoin token on the selected network.
7. Develop a smart contract that will mint new tokens in response to the collateral being deposited and also burn tokens in response to the collateral being withdrawn.
8. Design and implement a stabilization strategy using smart contracts.
9. Test and evaluate the functionality and viability of the smart contract system.

1.3 Report Structure

The project has been divided up into several different chapters. Each chapter aims to fulfil one or more of the objectives detailed above. Chapter 1 introduces the motivation and need for the project alongside the aims, objectives, and the report's structure.

Chapter 2 contains a thorough literature survey focused on demystifying blockchain technology at large, exploring existing stablecoins, and researching the Ethereum blockchain ecosystem. Chapter 3.1 comprises two sections: problem analysis and design choices. Here the investigation into developing a stablecoin and the reasoning behind the dual-implementation approach is explained and reasoned. The various technologies employed are detailed, alongside how they apply to solve the problem and why they were selected over competing options. Furthermore, detail is given on how the final stablecoin system could be effectively and robustly tested. During the development phases of this project, some assumptions were made, which are justified in this chapter.

Chapters 4 and 5 contain the design and documentation for the two on-chain implementations outlined in chapter 3.1. Both chapters detail specific design ideas, how they were translated from idea to reality, and a technical evaluation. These evaluations expand on what the particular implementation achieves and suggests how the functionality built in the specific chapter can be tested. Chapter 6 contains the design and some minor implementation details about the frontend, the full implementation documentation is located in Appendix A.

User testing scenarios encompassing the final stablecoin system are covered in chapter 7. These scenarios serve to ensure that the design expressed in chapters 5 and 6 has been implemented correctly. Concrete and verifiable evidence is provided for each test case, alongside a justification of the selected test scenario. Chapter 8, is where the project's successes will be evaluated against the objectives. Additionally, this chapter will house a section on possible improvements and future works.

The penultimate chapter, 9, is responsible for providing a critical evaluation into the various legal, social, ethical, and professional aspects of the project, which are discussed both in terms of this project's scope and the industry at large. Moreover, this chapter explains the correct use of open source software under the MIT license. It demonstrates how the project was carried out according to the guidelines laid out by the Chartered Institute for IT (BCS).

Finally, chapter 10 contains the final conclusions for the project; reflecting on the projects aims, stating what experience has been gained, and how the approach could be improved.

Chapter 2

Literature Review

2.1 Introduction

This section contains the literature review for the project. Firstly, it will look into the components that make up a blockchain and how they are secured through a consensus protocol. Secondly, the existing stablecoins and the problems they face will be researched. They will be categorised according to their stability strategy, and the game theory behind each approach will be explored. Finally, the technology directly relevant to the implementations of the project will be researched: the Ethereum blockchain, how Cryptocurrency tokens can be created with smart contracts, how to develop and deploy smart contracts, and how to interface with the smart contracts on the blockchain.

2.2 Background

When Bitcoin was first brought to life, and Satoshi Nakamoto published its whitepaper, the stated goal was to facilitate "online payments to be sent directly from one party to another without going through a financial institution" [1]. However, evidence published by Glaser et al. [10] shows that Bitcoin is seen as a speculative investment rather than a medium of exchange. It is demonstrated that the majority of people who buy into Bitcoin do not intend to use the network for transactions. Rather they buy the currency on an exchange and wait for this to increase in price. Furthermore, it is noted that the innovative features such as anonymity and security are not of interest to these users; they are interested in the possibility of high

rewards from the massive volatility. Additionally, Pavel Ciaian et al. [11] showcased the significant "inverse relationship between altcoins and BitCoin". This leads to the conclusion that all Cryptocurrencies are plagued with the same volatility as Bitcoin.

As a result, there have been multiple attempts to create a Cryptocurrency asset which maintains a stable price. In November of 2020, the market share of stablecoins within the domain of all Cryptocurrencies was 5.21% with a market cap of 22.97 Billion USD [12]. This is a tremendous amount of capital and is divided among 38 notable stablecoins - Tether [5] being the largest with a market cap of 17.27 Billion USD. Whilst all of these coins attempt to bring something new to the market; commonality can be found in their approach to collateral, "stablecoins can be characterized as either a) 'asset-backed' and b) 'algorithmic'" [13]. Asset-backed stablecoins include Cryptocurrencies in which the value is stabilised through collateral of any kind; Cryptocurrency assets, fiat money, and commodities. Algorithmic stablecoins require no collateral and attempt to pragmatically mimic the functiion of central banking in a decentralised fashion.

2.3 Blockchains

The phrase "blockchain technology" is often thrown around in discussions about Cryptocurrencies such as Bitcoin or decentralised smart contract networks like Ethereum. To begin implementing a stablecoin, it is crucial that the underlying technologies, and protocols are understood. "Blockchains are tamper evident and tamper resistant digital ledgers implemented in a distributed fashion" [14]. Bitcoin is an example of the first and most basic level of implementation; a record of transactions which is stored within a shared ledger across a community of nodes. From a high level, it is possible to categorise blockchains based on their permission model. Blockchain networks such as Bitcoin and Ethereum are said to be "permissionless" because anyone can download the open-source software and attempt to publish new blocks to the network. On the other hand, blockchains such as Hyperledger Fabric [15] are "permissioned" because a central authority can control who is able to publish new blocks.

Whilst there are many different blockchain networks that boast unique features, the underlying model comprises a set of components; nodes running the software, cryptography (hash functions, nonces, and asymmetric-key cryptography), transactions, addresses, blocks, a method of chaining different blocks together, and a consensus mechanism.

2.3.1 Nodes

For any blockchain to exist, there needs to be a community of nodes that maintain the decentralised ledger through reaching a consensus together. Different protocols require different nodes, but there are two general types: full nodes and light nodes. “Full nodes act as a server in a decentralized network. Their main tasks include maintaining the consensus between other nodes and verification of transactions. They also store a copy of the blockchain, thus being more secure and enable custom functions such as instant send and private transactions.” [16]. Light nodes simply expose the functions which allow users to make transactions without storing the entire blockchain locally by broadcasting transactions to the network. Full nodes store a copy of the blockchain (archival nodes store the entire history, whilst pruned nodes keep an incomplete copy); this allows them to partake in consensus mechanisms (section 2.3.7), and contribute to the state of the blockchain directly. A node which proposes new blocks to the consensus can be labelled as a “publishing” node. In order for a transaction to be actioned by the network, it must be published to the blockchain by one of these nodes.

2.3.2 Cryptography

A cryptographic hash function can be defined as any function that can “compress an input of arbitrary length to a result with a fixed length” [17]. This result can also be referred to as an output or digest. Within the scope of blockchains, three properties must be fulfilled by the hash function: preimage resistance (it is computationally infeasible to reverse engineer the input value, given the output value), collision resistance (it is computationally infeasible to find any two inputs which produce the same output), and second preimage resistance (it is computationally infeasible to locate two inputs which produce the same output). A hash function meeting these requirements can be used to maintain the integrity of the blockchain. The application of hash functions in a blockchain network includes: "Address derivation, Creating unique identifiers, Securing the block data – a publishing node will hash the block data, creating a digest that will be stored within the block header, and Securing the block header – a publishing node will hash the block header" [14].

Asymmetric key encryption serves to create identities on a blockchain [18], and involves the production of two different keys. “The public key, available for anyone to use. The other key is known as the private key. Data encrypted with the public key can only be decrypted with

the private key, and data encrypted with the private key can only be decrypted with the public key" [19]. All transactions (explained in section 2.3.4) made on a blockchain are "digitally signed" with a private key. The freely available public key can then be used to verify that the transaction sender has access to the private key and has the right to make the transaction. Another application of cryptography within the blockchain is the use of a "cryptographic nonce".

A nonce is defined as a "time-varying value that has, at most, an acceptably small chance of repeating" [20]. Nonces are used in the proof of work consensus model, explained in section 2.3.7.

2.3.3 Addresses

"The public key is not only very long and unhandy – around 65 characters – but can also be subject to typing errors. Further, exposing it can also set you on risks if ECDSA [(the algorithm which is used to generate a private key's corresponding public key)] is ever broken, for example by quantum computing. This is why the bitcoin developers created a method to derive an address from the public key" [21]. A cryptographic hash function is used to create an alphanumeric string which is much shorter and easier to manage than the lengthy public key. Users of a blockchain network can use the address to send transactions to others on the same network. Additionally, an address generally includes a checksum at the end, which serves to prevent typing errors. Advanced methods of generating addresses exist, such as the "Pay to script hash" [22] implemented by Bitcoin, which allows one account to have multiple addresses.

2.3.4 Transactions

Transactions can be referred to as the "smallest building block of a blockchain system" [23], transactions represent interactions between parties on the blockchain network. The data within a transaction can differ significantly between blockchain networks. However, the data generally includes the senders' address, the senders public key, the digital signature and some other transactional inputs. Within a Cryptocurrency specific chain such as Bitcoin, the information usually provide provenance of the funds. However, on a smart contract capable chain such as Ethereum, the inputs may include a payload of data for computation. The outputs of a transaction include the address of the recipients for the transfer of digital assets. Of course, "determining the validity and authenticity of a transaction is important" [14]. A transaction can be deemed valid

if it meets the requirements of the blockchain protocol and satisfies any smart contract-specific requirements for data formatting. The authenticity of a transaction is essential; the transaction must be signed using the private key associated with the senders' public address. Once a transaction has been entered into the blockchain, it is transmitted to a network of peer-to-peer nodes running the blockchain software worldwide. These computers will ensure that the validity and authenticity of the transaction are upheld.

2.3.5 Blocks

As described above, transactions can be submitted to a network via software running on a host computer connected to the network. Once the transaction has been propagated through the various network nodes, it will eventually reach a publishing node. The transaction is only added to the blockchains ledger once a publishing node has published a block containing the transaction.

A block can be broken into two parts: a header and the data. The metadata for the block is contained within the header; this includes information such as the timestamp, version, the cryptographic nonce and the hash of the previous block. A common sight in the metadata for a block in several blockchains is the "Merkle root". "A Merkle root uses mathematical formulas to check if the data is not corrupted, hacked, or manipulated. For example, Suppose one block has 10 transactions, then to identify that block we need 10 transactions to combine and form one Hash Value, so it uses the concept of the binary tree to create the hash of the block and that value is called the Merkle root" [24].The block data contains a list of transactions. Once a correct block has been published to the network, the new block will be propagated through the network and adopted by other nodes; eventually, all of the ledgers across the network nodes will reflect the changes made by the transactions contained within the block. A blocks correctness is dictated by the consensus mechanism (section 2.3.7) used by the blockchain.

2.3.6 Chaining of blocks

The chaining of individual blocks to form a blockchain is essential in producing a tamper-evident ledger that all network participants can trust. Each block is linked to the previous block by including a hash of the preceding blocks header within its header, "Each additional block strengthens the verification of the previous block and hence the entire blockchain" [25]. Suppose

the data of some block in the chain was altered; this would result in the hash contained in the header of this block changing. All subsequent blocks would also have different hashes because they follow on in a chain. Because the alteration of hashes propagates down the entire chain, it is easy to detect an altered block and subsequently reject it.

2.3.7 Consensus

"In the applications of blockchain, we need to solve two problems- double spending and Byzantine Generals problem"[26]. The double-spending problem refers to instances where the same digital asset can be used for multiple transactions simultaneously; this can be devastating for cryptocurrencies because there is a high incentive for malicious parties to abuse for financial gain. To address this, once a transaction has been published to the blockchain in a block, any other transaction that aims to spend from the same currency will be deemed invalid.

The Byzantine Generals problem was introduced in 1982 [27] long before blockchains were proposed. The problem can be summarized into the following: "how do you ensure that a peer-to-peer, distributed network with no central authority can make correct decisions, even if some of the nodes in it turn rogue? Can we make a distributed system that is "trustless" and doesn't automatically assume that the participants are going to act ethically and work in the interest of the group?"[28]. The problem is addressed within the scope of blockchains through a consensus algorithm that allows nodes to detect manipulated or incorrect data, and maintain a state consistent with other nodes in the network.

2.3.7.1 Proof of Work (PoW)

The most widely recognised consensus mechanism for permissionless blockchains is Proof of Work (PoW) [29] employed by Bitcoin, Ethereum and countless other networks. "PoW assumes that each peer votes with his ***computing power*** by solving proof of work instances and constructing the appropriate blocks [30]. This proof of work instance is a computationally intensive cryptographic puzzle; once a publishing node has found the solution, they can publish the block alongside the solution, proving that they have performed the "work". Nodes that receive this newly published block can verify that the puzzle has been solved correctly; they will then add this to their blockchain and send this to their peers. An essential feature of the puzzle is that it must be challenging to solve but inexpensive to check the validity of a solution. Bitcoin im-

plements a PoW mechanism that involves producing a hash lower than a block-defined "target value". As stated in section 2.3.5, the hash is taken over the entire header of the current block; different values for the cryptographic nonce must be tried to adjust the hash to find a suitably low value. The nonce value which produces a low enough hash digest is referred to as the ***golden nonce***.

2.3.7.2 Proof of Stake (PoS)

Another popular consensus model is the Proof of Stake (PoS) [31] model. The thought behind PoS is that if a user has invested into a system, it follows that they are less likely to act against the interest of the system because they have a vested interest in the system succeeding. This investment occurs in the form of depositing a "stake".

There are many blockchains experimenting with different PoS implementations, however a common pattern is as follows: for each block that is produced, a "lottery" takes place among all users whom hold a valid stake, "each user has a way to locally compute (using a pseudorandom process) whether they won the lottery" [32]. The winner of said lottery is then granted the ability to publish a block alongside proof that they won the lottery, in return they will receive a reward usually in the form of the native Cryptocurrency.

2.4 Stablecoins

Cryptocurrencies built upon blockchains such as bitcoin are truly groundbreaking. "For the first time, ownership of virtual property is possible without the need for a central authority – a development with the potential to fundamentally change the current financial system and many areas in both the public and the private sectors" [33]. However, Bitcoin and other Cryptocurrencies are flawed when it comes to being a currency that can be used for a medium of exchange due to the extreme volatility. "All central banks in developed countries attempt to stabilise the value of their currencies. They do this by providing an 'elastic currency', in other words, they adjust the aggregate supply of money to a changing aggregate demand"[33]. The supply of Bitcoins is pragmatically set at 21 million units, once this supply has been reached no further coins will be produced, furthermore no central authority has the ability to manipulate the price via the currency supply. As a result the price is strictly determined by speculation and the finite supply.

Aside from the volatility, Cryptocurrencies have many advantages which make them a viable form of currency: "low transaction costs, international borderless transfer ability and convertibility, trustless ownership and exchange, pseudo-anonymity, real-time transparency, and immunity from legacy banking system problems" [5]. stablecoins such as Tether, Dai, Binance USD, among countless others have been proposed in a bid to address this volatility and achieve the stability of a strong fiat currency.

Stablecoin landscape (20/04/2021, data from CoinGecko.com)			
Stablecoin	**Blockchain**	**Pegging Mechanism**	**Market Cap (USD)**
Tether (USDT)	Bitcoin	1:1 USD fiat held in a reserve	48,927,998,083.00
Dai (DAI)	Ethereum	Cryptocurrency assets held as collateral	3,427,335,374.00
TetherGold (XAUT)	Ethereum	Physical gold held in reserve at ratio of 1 XAUT: 1 Ounce of gold	151,984,964.00
FRAX (FRAX)	Ethereum	Partially algorithmic and partially Cryptocurrency asset backed	114,745,832.00
EmptySetDollar (ESD)	Ethereum	Algorithmic	113,172,801.00

Table 2.1: Overview of the different Stablecoins on the market.

As discussed in section 2.2, there are two main categories of stablecoins: asset-backed and algorithmic. It is worth noting that these categories are not mutually exclusive, as evidenced by the stablecoin FRAX [34], which both holds Cryptocurrency collateral and applies algorithms to stabilise the value. Each of the stabilisation strategies has associated benefits and problems. Table 2.1 contains a sample of stablecoins on the market, covering various stabilisation mechanisms. A necessary clarification is that most stablecoins are built as a second layer atop an existing blockchain such as Ethereum or Bitcoin. As shown in the table above, Ethereum is a popular choice due to the high level of decentralisation coupled with its smart contract capabilities (section 2.5). All stablecoins, regardless of the pegging mechanism, apply the Quantity Theory of Money (QTM) [35] to maintain a stable value. "QTM [...] was originally formulated

by the economist Irving Fisher in 1911 as follows: $M * V = P * Q$ Where M is the money supply (demand side), V is the velocity of money (demand side), P is the price per unit (supply side), and Q is the quantity produced (supply side). This theory helps us understand how money moves and how to adjust the money supply" [36].

2.4.1 Algorithmic stablecoins

Algorithmic stablecoins employ a purely algorithmic approach to maintain a stable market value without the need for collateral of any form. There are several different implementations for algorithmic stablecoins. However, commonality can be found where they deploy a mechanism based on QTM. EmptySetDollar (ESD) [37], shown in figure 2.1, is a popular algorithmic stablecoin in which the "stability mechanism is an iteration on Basis". The original whitepaper for the Basis Protocol [38] states that "the core idea is that to maintain a peg in the long term, we just need to measure the price of Basis and adjust Basis supply accordingly". The core ideas on how to manage the supply behind Basis, alongside the majority of other algorithmic stablecoins, are variations of Sams Seignorage Shares [39]. Parallels can be drawn between the algorithm itself and a central bank; "crypto projects adopting the seigniorage shares §model are attempting to do what the Federal Reserve does in a decentralized, algorithmic way" [40].

An algorithmic stablecoin sounds fantastic on paper due to the inherent transparency and auditability, which addresses the critical flaw in Tether's model (section 2.4.2. However, in practice, they are challenging to implement correctly and analyze accurately. ESD is currently valued at 0.14 USD (22/04/2021), a far cry from the target of 1 USD. Furthermore, "Critics of this model argue that's it's difficult to create an algorithm that decides issuance and contracting rules that is resilient to manipulation. Consistently being able to contract the money supply while maintaining value is difficult and requires participants to believe that demand will increase in the future" [41].

2.4.2 Asset-backed stablecoins

"An asset-backed stablecoin design is one where some asset, most often US dollars but increasingly Cryptoassets like ether (ETH), is held in reserve with the aim of supporting the stablecoin's exchange rate" [42]. The three different backing asset types discussed in this literature review are fiat money, commodities, and Cryptocurrency assets.

2.4.2.1 fiat-backed

"Tokens which are backed by a national currency are called fiat-backed stablecoins" [43]. A prime example is Tether which currently accounts for over 50 per cent of the total market capitalization of all stablecoins (22 April 2021). The market value of these stablecoins is maintained to the peg through the principle of QTM. However, unlike algorithmic stablecoins, which directly alter the supply of coins, the users of this system enforce the price by taking advantage of price fluctuations.

Suppose such a stablecoin pegged to 1 USD is being traded at anything above the peg, for example, 1.02 USD. Users will be incentivised to purchase new coins directly from the central entity responsible for handling the collateral; by making a purchase directly they make a small profit by immediately selling the coin on an exchange. The influx of users selling on exchanges will naturally inflate the supply until the coin is again trading at 1 USD. In an inverse scenario where the same stablecoin is trading at 0.97 USD, users will be able to redeem the token directly through the central entity in return for the 1 USD being held as collateral per token, essentially buying 1 USD for 0.97 USD. As users flock to this action, the supply will naturally contract until redeeming is no longer profitable once the value has returned to the peg.

This mechanism is extremely attractive and makes sense at first glance; no responsibility for manipulating the supply and complicated game theory is not required. However, this design is the polar opposite of the "trustless" nature of blockchains. Immense trust must be placed in the central entity holding the collateral: "every unit of tokenised funds represents a claim on the issuer over the funds it received from users" [44]. Tether has been the recipient of criticism from investors and governments alike because they are not openly audited. [45]. If the reserves were revealed as inadequate, the stabilisation would undoubtedly fail in the case of a bank run, and investors would lose out.

USD Coin (USDC) [46] is an alternative stablecoin that is implemented on the Ethereum blockchain as an ERC-20 Token (section 2.7). The stablecoin is issued by Centre who promise that the coin is backed 1:1 in regulated institutions. "A significant feature claimed by Center about USDC is that it is audited by a well-known independent organization (Grant Thornton LLP) and regulated in the US" [47]. Additionally, USDC coins are mintable by a set of financial institutions that meet Anti Money Laundering requirements and are open to audits. These steps work towards alleviating the trust issues facing Tether. However, USDC is self-admittedly at

the mercy of US regulations, and there is still a high level of trust that must be placed in Centre to honour all redemptions.

2.4.2.2 Commodity-backed

A commodity is a basic good used in commerce that is interchangeable with other goods of the same type [48]. Commodity backed coins such as PAX Gold (PAXG) [49] maintain a stable value pegged to a unit of the relevant commodity. Each PAXG token is backed by a single fine troy ounce of gold stored in a physical vault under the custody of the Paxos Trust Company. This class of asset-backed stablecoin achieves stabilisation through the same mechanism outlined above; users of the token always can redeem their tokens directly with the central authority for either the physical asset or fiat money to the current market value of the asset. This class of tokens certainly achieves greater stability than traditional Cryptocurrencies. However, compared to fiat-backed stablecoins, it is important to remember that these commodities can, and are more likely to, fluctuate in price and therefore have the potential to lose value. [50]. Additionally, these coins suffer from the same downsides as their fiat-backed counterparts in regards to trust requirement.

2.4.2.3 Cryptocurrency-backed stablecoins

The final implementation of this project will involve creating a Cryptocurrency-backed stablecoin, as such this is the class of asset-backed stablecoins that the literature review will focus on most extensively. Stablecoins of this class take advantage of smart contracts(section 2.5.1) to create a fully decentralised system, which accepts collateral in the form of the selected blockchains native currency. "On-chain collateral has the benefit of decentralisation. Any pure Cryptoasset can be truly held by a smart contract and so can be as censorship-resistant as the consensus mechanism it's implemented on top of" [51]. The need for a central entity that can control the supply of coins is removed, thus addressing the troubles associated with off-chain backed coins. However, using Cryptocurrency assets as collateral is no silver bullet due to the high volatility.

One way of accounting for this volatility involves adding a "secondary coin which absorbs the volatility of the first" [52], this can be labelled as a ***dual coin*** system. Holders of the secondary token help prop up the price when the currency inflates and are rewarded during deflationary periods" [52]: When the trading price of the stablecoin falls below the peg, the secondary coin

attempts to increase the price through the contraction of supply. When the stablecoin is trading at a price above the peg, holders of the second asset are financially rewarded with some newly minted stablecoins. Unfortunately, a stablecoin implemented in this fashion could meet a legal definition of a financial security which can bring regulatory problems. Whilst regulatory issues can be navigated; the second issue can be more devastating. If investors do not believe that the stablecoin will appreciate above the peg, there is no incentive to invest in the secondary token. If there is a shortage of people holding the secondary coin, any adverse fluctuations in the collateral assets price will result in an inadequate reserve of collateral.

The Celo protocol [53] is an example of a dual-coin system that is actively trading on markets with a market capitalisation of 45,524,328 USD [54]. The protocol implements two tokens: The stablecoin Celo dollar and the variable value asset CELO. Figure 2.1 displays the price and market statistics for the stablecoin.

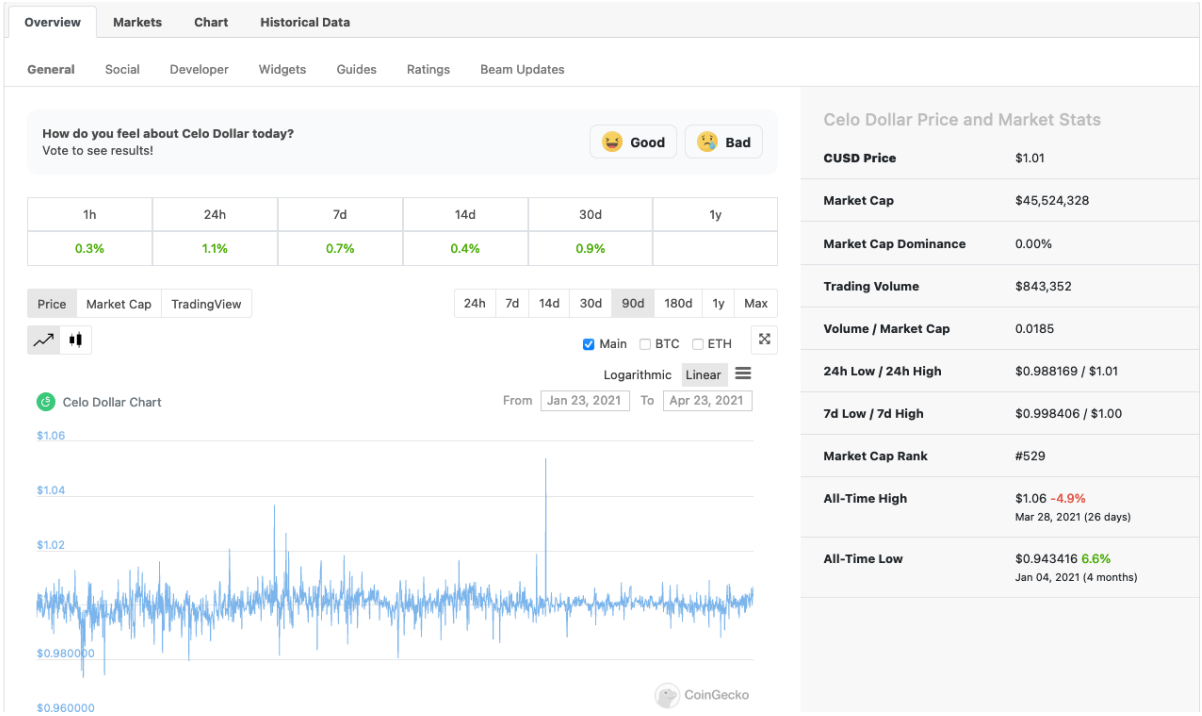


Figure 2.1: Market statistics for Celo dollar (CoinGecko, 23rd April 2021)

It can be seen that the all-time high of the coin was 1.06 USD, whilst the all-time low was 0.94 USD. Additionally, the graph shows that there is some fluctuation. However, it fluctuates around the 1 USD price point. At the time of writing, the markets are in a very turbulent state; the price maintaining at 1.01 USD shows some merit to the Celo system.

"To balance Celo Dollar demand excess or shortage, the protocol sets incentives for users to adjust Celo Dollar supply to match Celo Dollar demand at the price target." [55]. When the stablecoin is trading above 1 USD, the protocol incentivises holders of the secondary coin, CELO, to redeem their balance for some stablecoins. Users can redeem 1 USD worth of CELO (at market price) for 1 Celo dollar; this Celo dollar can then be sold on an exchange at market price for a profit. This action by users will increase the trading supply of the stablecoin and eventually return the asset to the peg value. When the stablecoin trades below the peg value, users can redeem Celo dollars for CELO at the value of 1 USD per stablecoin. This provides an incentive to immediately sell the CELO coins on an exchange for a profit. An interesting feature of the Celo protocol is that the collateral is held in a basket of different cryptocurrency assets: CELO, CeloGold, Bitcoin, and Ether.

Another type of Cryptocurrency-backed stablecoin are ***leveraged loans***; these stablecoins inherit from the dual-coin design yet combine aspects of algorithmic stablecoins and introduce over-collateralisation. The prime example of such a system is the Dai [6], which is pegged to the value of 1 USD. Dai is backed by Ether and several other ERC-20 tokens (including the controversial addition of USDC). "The Dai stablecoin system employs smart contracts on Ethereum that actively stabilise Dai's exchange rate through the use of Collateralized Debt Positions (CDPs) and autonomous feedback mechanisms. Appropriately incentivised external actors (e.g., market traders) also play a role in stabilising Dai" [13].

CDP's are smart contracts that allow users to mint Dai up to 2/3 of the value of cryptocurrency assets deposited into the contract. This newly minted Dai is said to be borrowed. The deposit is held by the contract until the debt and stabilisation fees (essentially an interest charge on the borrowed Dai) are paid back. If the value of deposited collateral in the CDP contract falls below 150% of the borrowed Dai, the debt position is liquidated. During a liquidation, the Maker platform itself buys out the collateral of the CDP and sells it immediately in an auction.

The secondary asset in this system is the MKR token; holders of the MKR token can vote on "internal governance variables" such as the stability fee or penalty fees charged to users interacting with the system. Funds accrued from the fees charged to Dai system users are distributed to MKR token holders. To raise funds for buying out the collateral during liquidation, a debt auction takes place. This involves "diluting the supply of the MKR token and selling it to bidders in an auction format [...] In parallel, the collateral of the CDP is sold in a Collateral

Auction" [6]. All of the proceeds from this auction up to the CDP debt (plus a liquidation penalty) are then used to buy MKR. This process counteracts the diluting of MKR, which was necessary to buy out the collateral. If some party bids enough Dai to fully cover the CDP debt and liquidation penalty, leftover collateral is returned to the original depositor.

When the price of Dai falls below the peg, users are incentivised to buy the coin at market price and then pay off their Debts, allowing for the collateral to be returned at a discount. An influx of users performing this action serves to reduce supply and increase the Dai market price back to peg. An additional mechanism to reduce supply is the increase of stability fees; users who have borrowed Dai will be incentivised to pay back their debt to avoid paying the fees. Inversely, when Dai is trading above the pegged value, anyone who holds a balance of Ether can deposit into a CDP to mint new tokens and sell these on an exchange. The funds from selling the Dai can then be used to pay back the Debt position; any excess is profit. Of course, there are stabilisation fees to consider, which can be reduced to encourage this kind of interaction.

Dai has been on the market since November of 2019; since its inception, it has been relatively stable. The all-time high was 1.22 USD (March 13th 2020), whilst the all-time low was 0.90 USD (November 25th 2019). The current market capitalisation of the coin is 3,442,110,857 USD, clearly indicating that many users trust the system's stability.

Despite the current success of Dai as a stable asset, critics have argued that Dai will fail to hold up under an extreme market crash. This theory was put to the test on March 12th 2020, when there was "an outright collapse of both the capital and Crypto markets over a 36-hour period" [56]. On this date, both Ether and Bitcoin lost over 50% of their total market capitalisation; the increase in activity as people panicked resulted in extreme delays, and increased gas prices, on Ethereum blockchain transactions. The peg of Dai to 1 USD was certainly stressed; the coin traded as high as 1.22 USD. Due to the sharp decline in collateral value, many CDP's were liquidated. Generally, there is a competition between actors that take part in the collateral auctions. However, the huge number of network delays allowed some devious parties to win auctions with "near zero-bids" [57]. Because the auctions were sold for such a small sum, there was no collateral returned to the CDP owners, resulting in a total loss of 8,320,000 USD. "To help re-establish the peg, the MakerDAO governance has voted to add USDC as a collateral type. This will allow for users to trade between USDC and Dai, thus bringing the price of Dai down closer to a dollar." [58]. The addition of USDC to the collateral portfolio brings Dai's

decentralisation into question. Nonetheless, the coin's popularity continues to grow, and the value has been relatively stable since.

2.4.2.4 Pricing Problem: Oracles

As explained in the sections above, the key to maintaining a stable price involves applying mechanisms that act on the available supply of the currency in response to changes in the stablecoins market value. Fetching up-to-date pricing information is trivial for implementations with a central authority (fiat-backed and commodity-backed). Still, it is a deceivingly tricky task for decentralised systems (algorithmic and Cryptocurrency backed). Furthermore, coins such as Tether rely on user actions to maintain the peg when prices on exchanges fluctuate, essentially removing any need for them to track the price whatsoever.

As explained above, to maintain stability, these types of decentralised stablecoins employ a set of smart contracts, which are hosted on a blockchain network. A feature of the blockchain is that it is self-contained and cannot fetch data from any external system; this serves to make the distributed ledger secure and reliable but comes at a cost. In many cases, with stablecoins, real-world data is necessary for a smart contract system to function correctly. "Bridging the connection between the blockchain (on-chain) and the outside world (off-chain) requires an additional and separate piece of infrastructure known as an oracle'" [59].

"The main challenge with designing oracles is that if the oracle is compromised, the smart contract relying on it is also compromised" [60]. The centralisation of an Oracle is somewhat unavoidable because there are a finite number of reliable sources, which poses a source of conflict to the decentralised nature of blockchains. There are several designs of Oracles that attempt to minimise this trust conflict and threat vector to the overall system.

The most common form is an external software oracle that interacts with sources of information (pricing information from an exchange API, for example) and subsequently updates this data on the blockchain through a transaction to the oracle smart contract. This kind of implementation is centralised because a single entity (program sending updates to the contract) is trusted with updating the data, presenting a single point of failure, which weakens the overall system. Some naive implementations of these Oracles use a single source of pricing data; this is not acceptable for obtaining reliable and correct data. A typical pattern to address unreliable sources involves querying multiple sources of information and then calculating a median value on-chain (Dai

employs such a method). However, calculating the median is time-consuming and can result in a data feed with more latency; for stablecoins that need to adjust in real-time, this could be devastating.

Another option for obtaining data is creating a ***crowd oracle***, also known as the SchellingCoin mechanism [61]. The previously discussed Celo protocol relies on a crowd oracle under the assumption that the voters will favour long-term growth rather than short-term profits. The SchellingCoin mechanism involves a set of independent sources who must report the truthful state of the data without colluding with one another. Such systems are not immune to manipulation and are vulnerable to problems such as collusion between parties.

Stablecoin systems which make use of the SchellingCoin system are at a heightened risk of malicious value reporting. This increased risk is because voters have an incentive to lie. When the price of a stablecoin is trading above the peg, mechanisms that act to increase the supply should help bring the coin's price back to peg. "Suppose the price of the currency is trading above $1. Participants could dutifully report the truth, and trigger the mechanism that dilutes the coin to reduce its price. But this would result in a net loss for them. Instead, they have an incentive to report a price that is lower than the truth, so there is less of the currency put into circulation. It is in the best interest of the participants to falsely claim that the price is still $1, or even lower. Such false reporting will lead to a short-term gain for the current holders, since no new coins will be printed, enabling the current holders to liquidate their coins above the targeted peg value" [62]. Of course, this incentive only exists if the users involved in the price reporting are stakeholders in the stablecoin.

Some implementations reward reports close to the median value to address untruthful reporting and slash (financially punish) participants who vote with a significant difference. At first glance, this seems like a reasonable solution. However, in reality, this incentivises participants to vote for the value they believe others will vote for – this concept is known as a Keynesian beauty contest [63]. This problem is further exacerbated when there are many untruthful voters because otherwise honest parties will be forced to vote dishonestly to avoid punishment.

A decentralised oracle network such as ChainLink [64] provides decentralised oracles that "do not rely on a single source of truth; thereby, such oracles increase the authenticity of the information provided to smart contracts" [65]. Chainlink allows certain third parties such as well known exchanges, to become a node that feeds specific data to the blockchain. Collateral must be

provided to create a node; this collateral will be deducted from and rewarded to users in the case of incorrect information. When another user requests data from a ChainLink node, a fee in LINK [66] is rewarded to the node. Using ChainLink as an Oracle provider "eliminates the reliability issues that might occur if using only a single centralised source" [67]. Selecting ChainLink or a similar network can abstract the need to build an oracle system that can be a costly endeavour and introduce an element of risk if not implemented correctly. However, because only selected entities, that have been approved by the ChainLink team are selected to be nodes, the source of information is somewhat centralised.

2.5 Ethereum blockchain

In 2013 the original Ethereum whitepaper [7] was published by V Buterin. The paper was titled "A Next-Generation Smart Contract and Decentralised Application Platform". It introduced the idea of a "blockchain with a built-in fully fledged Turing-complete programming language that can be used to create "contracts" that can be used to encode arbitrary state transition functions". A common misconception is that Ethereum is simply another Cryptocurrency like Bitcoin. They are both blockchains that employ a PoW consensus mechanism. However, the Bitcoin protocol only facilitates a "weak" form of scripting in a Turing-incomplete language. Within the Ethereum blockchain, transactions are more than a transfer of value; each transaction has an optional data field. This data field can be used to provide data to particular functions on a smart contracts interface.

Furthermore, Ethereum introduces the concept of gas (section 2.5.2). Each transaction must specify a gas limit and a gas price value. When a transaction is sent, there is a reserved quantity of Ether used to pay any gas costs.

2.5.1 Smart Contracts

To understand the Ethereum blockchain, one must first understand what a smart contract is. They were first introduced in the 1990s by Szabo as a computerised protocol to execute the terms of a contract on a decentralised ledger [68]. In current times, with the emergence of blockchain technologies, smart contracts have been defined as "a piece of code programmed onto a blockchain, which defines the terms of a particular transaction. Upon the receipt of a given

trigger or input, the smart contract will execute and perform its assigned tasks" [69]. The use case for smart contracts extends to any scenario where a trusted intermediary is required. Within the scope of stablecoins, smart contracts are responsible for handling all of the on-chain logic. For fiat-backed coins, it can be as simple as providing the ability to mint, burn, and track tokens, whilst decentralised implementations such as Dai would cease to exist without smart contracts.

"A smart contract has an account balance, a private storage and executable code. The contract's state comprises the storage and the balance of the contract. The state is stored on the blockchain, and it is updated each time the contract is invoked" [70]. The code of the contract cannot be changed once it has been deployed. Variables that are stored in the state, however, can be altered through interactions. Smart contracts on the Ethereum blockchain are written in the Solidity [71] programming language and are executed on the Ethereum Virtual Machine (EVM). Solidity is a contract-oriented programming language used to write deterministic smart contracts; determinism is crucial because the code will be executed by many different nodes and must consistently return the same value if consensus is to be reached.

2.5.2 EVM

The Ethereum Virtual Machine (EVM) is a powerful, sandboxed virtual stack embedded within each full Ethereum node, responsible for executing contract bytecode. Contracts are typically written in higher-level languages, like Solidity, then compiled to EVM bytecode [72]. Within the Ethereum whitepaper, the Ethereum blockchain is described as a "distributed state machine" rather than a traditional distributed ledger such as Bitcoin. Each node of the network stores a copy of the Ethereum networks state, which contains a list of balances for all addresses and a "machine state". This machine state can change from block to block via a "state transition function". This state transition occurs each time a variable-altering transaction is made to a smart contract; executing this transition requires computation by the network nodes – this computation comes with an associated cost known as "gas".

When a smart contract is deployed to the blockchain, it must first be compiled into EVM bytecode – a low-level language for machines, consisting of a set of opcodes. These opcodes "perform standard stack operations like XOR, AND, ADD, SUB, etc. The EVM also implements a number of blockchain-specific stack operations, such as ADDRESS, BALANCE, KECCAK256,

BLOCKHASH, etc" [73]. Each time a particular smart contract function is transacted with, the relevant opcodes to achieve the desired state change are executed on the EVM. "Every opcode has its own base gas cost. Furthermore, several complicated opcodes also charge a dynamic gas cost. [...] Computationally expensive instructions charge a higher gas fee than simple, straightforward instructions. On top of that, every transaction starts at 21000 gas" [74].

As the opcodes of a particular transaction interaction are being executed, a counter of the used gas increases accordingly. Once execution has been completed, the correct amount of Ether is debited from the estimated fees sent with the transaction. If at any point the transaction runs out of gas before the execution is complete, the transaction is aborted (minus the fees for used gas). As a security feature to prevent endless computation, there is a hard limit for the maximum amount of gas any transaction can spend; this is essential to avoid network congestion.

2.5.3 Ethereum Blockchains

The Ethereum protocol is open source, and any interested party can create a private network if they have the necessary infrastructure. Multiple networks can operate individually and have no interaction with one another. There are several public networks that anyone can become a node on and interact with using an Ethereum account. A singular "mainnet" exists; this is the production network "where actual-value transactions occur on the distributed ledger" [75]. This is arguably the most secure Ethereum blockchain because it has the highest degree of decentralisation. Aside from the mainnet, a few public "testnets" are designated to test smart contracts and protocols before launching on the mainnet. Testnets are important because they allow no-cost testing; Ether on the mainnet has a real-world value, which means that gas costs can become expensive quickly. The different testnets are shown in figure 2.2. Ropsten is the only PoW testnet; the rest of them use a Proof of Authority (PoA) consensus mechanism, in which a small set of Authoritative nodes are chosen to validate transactions and create the new blocks. It can be challenging to incentivise mining on a PoW testnet because the Ether holds no market value. As a result, the Ropsten network is relatively unstable and should never be used for high-value production releases. It is also possible to run local development networks that are not connected to a real blockchain; these local networks provide the least friction for testing out smart contracts because there is no need to acquire Ether, and there is no need to wait for transactions to mine.

Network	Consensus Mechanism
Ropsten	Proof of Work
Kovan	Proof of Authority
Rinkeby	Proof of Authority
Görli	Proof of Authority

Figure 2.2: Overview of the different Ethereum testnets

2.5.4 Limitations of the Ethereum blockchain

The Ethereum blockchain has made the likes of Decentralised Finance (DeFi) [76] possible by allowing numerous parties to interact in a trustless manner without the need for a central authority. The network is poised to have a real impact on the financial industry and has recently found favour with institutional investors [77]. The technology, however, is not quite up to the standard that would be needed to handle large-scale adoption [78]. The Ethereum protocol currently has a throughput of 15 transactions per second; this is not adequate for achieving the protocols long-term vision. The cracks can be seen when there is any substantial increase in network activity; gas fees increase exponentially such that the system becomes economically unviable, and transactions can take so long that decentralised systems that rely on them can fail. [79]. As shown in section 2.4.2.3, network congestion can have severe consequences for any systems that rely on oracle data to maintain a viable system.

To address the scalability problems associated with the current protocol and increase security whilst decreasing the networks carbon footprint, Ethereum will transition to a new protocol – the Casper protocol (Ethereum 2.0) [80]. This new protocol will switch from the current PoW consensus mechanism to a much less computationally intensive PoS mechanism. Furthermore, By a process called "sharding," the new Ethereum will break up consensus into various pieces and have them separately validated [78]. As a result of the Casper upgrades, the transaction throughput will be upgraded to two or three thousand transactions per second – a stark increase compared to the current 15.

2.6 Development on an Ethereum Blockchain

An application that combines a smart contract for the backend infrastructure, alongside some form of a frontend user interface, can be described as a decentralised application (dapp) [81]. Even if the frontend is hosted on a centralised server, the application is said to be decentralised because the logic is executed on the Ethereum blockchain. Smart contracts, as explained above in section 2.5.1 are written in the Solidity language and then compiled into EVM bytecode during deployment. Previously in this document, smart contracts have been explained from a high level; this section will explore how Solidity smart contracts are structured and the fixed-point arithmetic limitations. Furthermore, the logic of any Cryptocurrency collateralised stablecoin system lives on the blockchain in smart contracts. However, the communication from a user interface to the system is facilitated through the Web3 API [82].

2.6.1 Solidity smart contracts structure

"Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of accounts within the Ethereum state" [83]. A Solidity source code file contains an instruction for the compiler version, imports for contracts and libraries, and contract declaration(s). Within a contract declaration, the variables, constructor, and other desired functions of the contract are defined. Solidity is statically typed; therefore, each variable, and the return value of any function, must have the type and visibility specified. Furthermore, if a function wishes to accept incoming Ether in a transaction interaction, it must be defined as "payable" – the incoming Ether will be credited immediately to the contract's balance.

2.6.2 The problem of fixed point arithmetic

Solidity does not support fixed-point arithmetic, meaning that numbers with decimal places cannot be represented in the usual fashion. Many real-world applications of a smart contract system require decimal representation; stablecoins are such cases. In the case of a Cryptocurrency collateralised stablecoin, the asset's current market price will undoubtedly need to be used in calculations. It can be assumed that this number will likely not be a whole integer. The concept of a wad was introduced on the DappHub GitHub repository [84] as part of the

ds-math library. A wad is a decimal number with 18 digits of precision; the traditional integer value is multiplied by 10$\hat{1}$8 to achieve this: 1 would be represented as (1 * 10$\hat{1}$8). This allows for calculations that involve values with decimal places. This type of representation can become confusing and appear messy in code. However, it is a necessary sacrifice that must be paid to achieve the required functionality.

2.6.3 Interacting with a deployed contract – Web3

Once a smart contract has been successfully deployed to an Ethereum blockchain, there are two different types of high-level transactions that are relevant:

- Value transactions (sending Ether)
- Contract invocation transactions

Contract invocation transactions are necessary when any variable of the contracts state is changed through a function invocation. Reading data from the blockchain is free of charge and does not require any form of transaction; the only requirement is access to an Ethereum node on which the state can be queried. Suppose a web frontend wishes to interface with a particular smart contract on the blockchain; it would need to leverage the Web3 API alongside a connection to an Ethereum blockchain node to query. The Web3 API is a "collection of libraries which allow you to interact with a local or remote Ethereum node, using a HTTP or IPC connection. The web3 JavaScript library interacts with the Ethereum blockchain. It can retrieve user accounts, send transactions, interact with smart contracts, and more" [85].

Access to an Ethereum node can be achieved by running a node locally – such as a local geth [86] client or through a so-called provider. "A Provider is an abstraction of a connection to the Ethereum network, providing a concise, consistent interface to standard Ethereum node functionality" [87]. The Web3 API communicates with an Ethereum node via JSON RPC – Remote Procedure Calls [88]. The Ethereum address of the contract and its Application Binary Interface (ABI) are required to communicate with a specific smart contract on the blockchain. The ABI specifies how to encode arguments such that the EVM understands them during execution, and also, one may decode the operation's response. The Web3 libraries can pull data from the blockchain into a frontend application and interface with a dapp users wallet such as MetaMask [89].

2.7 ERC-20 Tokens

"The major difference between ERC-20 tokens and other cryptocurrencies is that ERC20 tokens are created and hosted on the Ethereum blockchain, whereas bitcoin and bitcoin cash are the native currencies of their respective blockchains" [90]. Because they are built on top of the Ethereum blockchain, they can be transferred between Ethereum network addresses in a similar fashion to the native currency Ether. The ERC-20 token standard is introduced and defined on the official Ethereum GitHub repository as "a standard interface for tokens" [91]. This standard describes six functions and two events that must be present on any smart contract which aims to be ERC-20 compliant:

- function totalSupply()
- function balanceOf(address _owner)
- function transfer(address _to, uint256 _value)
- function transferFrom(address _from, address _to, uint256 _value)
- function approve(address _spender, uint256 _value)
- function allowance(address _owner, address _spender)
- event Transfer(address indexed _from, address indexed _to, uint256 _value)
- event Approval(address indexed _owner, address indexed _spender, uint256 _value)

There are three additional optional functions:

- function name()
- function symbol()
- function decimals()

As long as the base functions are implemented, the token is considered compliant; this opens plenty of avenues for development because there are no limits on other functionalities to be added. Any smart contract that implements a specific ERC-20 token is responsible for creating the tokens, handling transactions, and keeping track of each users' balance.

Performing transactions with ERC-20 tokens requires the sending address to hold a small amount of Ether to pay the gas fees; the transfer is a request sent to the smart contract, telling it to assign the necessary balance of the token to another address. Adjusting this value on the blockchain requires computing, therefore creating the gas fee that must be paid. The creation of new tokens, which are added to the contracts totalSupply value, and a users balance, is said to be a "minting" operation [92]. The inverse operation of "permanently removing coins from circulation" [93] is referred to as "burning".

Using an ERC-20 token makes it possible to create an asset on the Ethereum blockchain, which can benefit from the security provided by the massively decentralized network. Furthermore, following the standard, in particular, allows wallets and other platforms such as exchanges to interface with the contract via a uniform language – "Before ERC-20 tokens, developers might use other terminology in the code - e.g. one token uses [totalAmount] while another uses [totalNumber]. Exchanges and wallets needed to build their platforms to accommodate for each one token's code" [94].

2.8 Ethereum Wallets

Ethereum wallets are applications that let you interact with your Ethereum account [95] (private and public key pair). Wallets are tools to help one manage their Ethereum account and provides a user interface that facilitates balance queries, transactions, and connections to third party applications. Of course, it is possible to make transactions on the Ethereum network without a wallet as long as one has possession of the private key. However, wallets are at the heart of user interactions on the blockchain because they abstract away all of the technical work behind the scenes (building transactions, communicating with an Ethereum node, transmitting transactions). Wallets are generally classified into two categories: cold storage and hot storage. Cold storage refers to a wallet that is not connected to the internet, whilst hot wallets are connected to the internet [96]. In general, cold storage is the most secure option, but this comes at the cost of usability and the up-front cost of purchasing a hardware wallet.

A popular open-source browser extension, MetaMask, serves as an Ethereum wallet and allows the storing of Ether alongside ERC-20 tokens across all of the public Ethereum blockchains. "Dapps automatically connect to MetaMask, simplifying the connection process. Within the Dapp, if payment is required, a pop-up window will appear asking to confirm the transaction

from the MetaMask account" [97]. Creating a MetaMask account is as simple as installing the browser extension, picking a password, and then noting down the "secret backup phrase". MetaMask can interface with several hardware wallets. Alternatively, it can provide a hot wallet within the extension free of charge.

2.9 Summary

The literature search has revealed a great deal of information regarding the currently available stablecoins and the underlying blockchain technology they are built upon. Understanding the protocols that allow blockchains to function exposes the ideals and motivations behind decentralised ledgers; these ideals will undoubtedly be considered for all design choices. The research into existing stablecoins and the different strategies employed to achieve stability is very significant for this project. In particular, the knowledge of the economic principles such as QTM and how they motivate the different mechanisms implemented by stablecoins such as Dai and Tether will be directly applicable to the strategy developed for stabilisation in this project.

The exploration of the oracle problem (section 2.4.2.4) will undoubtedly be relevant for this project; a significant component of any Cryptocurrency-backed stablecoin is to track the current market price, such that the system can enact informed decisions. Some form of oracle will be needed, and understanding the possible options to the table will be crucial in selecting an appropriate method of feeding real-world data into the system.

Whilst exploring successful stablecoins, a trend emerged in the blockchain they were built on top of, Ethereum. Ethereum is the number one choice for developing decentralised applications; these applications are made using smart contracts. The specific research on Ethereum granted an understanding of how the EVM works and the different testnets available for development. Of course, this project does not aim to realise a production-ready product, and as such, one of these test networks will undoubtedly be used for the project. Understanding how the EVM functions will be essential when building the system, and especially when making software design decisions; the gas cost of operations must be considered.

The final three areas of the literature review are directly applicable to the design and development of a Cryptocurrency-backed stablecoin on the Ethereum blockchain. ERC-20 Tokens present themselves as a fantastic option for creating a fungible token to stabilise, whilst the research in

section 2.6 aids in understanding how to develop and interact with a fully operational on-chain system. As explained in the Ethereum section, all on-chain state changes of a smart contract require a transaction; the research into the different Ethereum wallets will be helpful to create a Dapp that can interface with a users wallet to allow interactions with the system.

Chapter 3

Problem Analysis and Design Choices

3.1 Introduction

This section details how the problem was narrowed down and explains how the project will approach the problem in three phases. Additionally, each of the vital design decisions made throughout the project will be explored.

3.2 Problem Analysis

At the core of this project, the problem is to implement a stablecoin on top of an existing blockchain. The literature search has revealed that there are two types of stablecoin at a high level: Asset-backed (Section 2.4.2) and Algorithmic (section 2.4.1). The asset selected for backing the value of a stablecoin can be further classified into commodities, fiat money, and Cryptocurrency assets. The scope of this project shall be on creating a Cryptocurrency collateralised stablecoin.

This type of implementation has been selected because it presents an opportunity to explore the technologies applied in the Decentralised Finance movement [98] and because it is an area with lots of active development and room for creativity. A fiat-backed or commodity-backed implementation has not been selected because much of the work would be focused on setting up the banking and legal infrastructure. Whilst this would present some exciting areas for development, it would not result in the desired blockchain implementation challenges and experience. Furthermore, as discussed in the literature review (section 2.4.2.1), such an implementation

violates the trust-less and decentralised future that Cryptocurrencies promise to deliver. The second alternative of creating an algorithmic stablecoin is on the bleeding edge of technology and has an extremely high barrier to entry. Time limitations mean that such an implementation would be too ambitious due to the necessary understanding of financial markets and economic game theory.

Much of the current stablecoin development has been centred around creating an asset pegged to 1 USD; "while it might seem attractive to have a single global currency, the reality is that regional economies often benefit from having their own currencies that can respond independently to local demand shocks. This is because demand shocks can concentrate in a particular region, in isolation from the rest of the world" [38]. This project aims to take a novel approach at creating a stablecoin pegged to the value of 1 GBP using Cryptocurrency collateral.

3.3 Approach to the problem

The implementation for this project will be divided into three distinct phases:

1. Deploying a token on a live blockchain as an example of how a stablecoin backed by fiat money could be implemented. (Chapter 4)

2. Deploying a stablecoin on a live blockchain that attempts to achieve a stable price using Cryptocurrency assets as collateral. (Chapter 5)

3. Implementing a frontend that displays data needed to test that the system is working as intended. Here, the game theory doesn't need to be bulletproof as this is not an economic project. However, the completed system does need to abide by the intended design. (Chapter 6)

The ultimate goal for this project is to achieve a stablecoin that is implemented on a public blockchain and pegged to the value of 1 GBP through the collateralisation of Cryptocurrency assets. This is a technically challenging problem and has been divided into the three phases above to make it a manageable problem. Each of the phases will involve some development "on-chain" and "off-chain", on-chain development refers to any software or code that will be deployed and executed on a live blockchain. In contrast, off-chain development refers to the development of software that can be run on any server or client device.

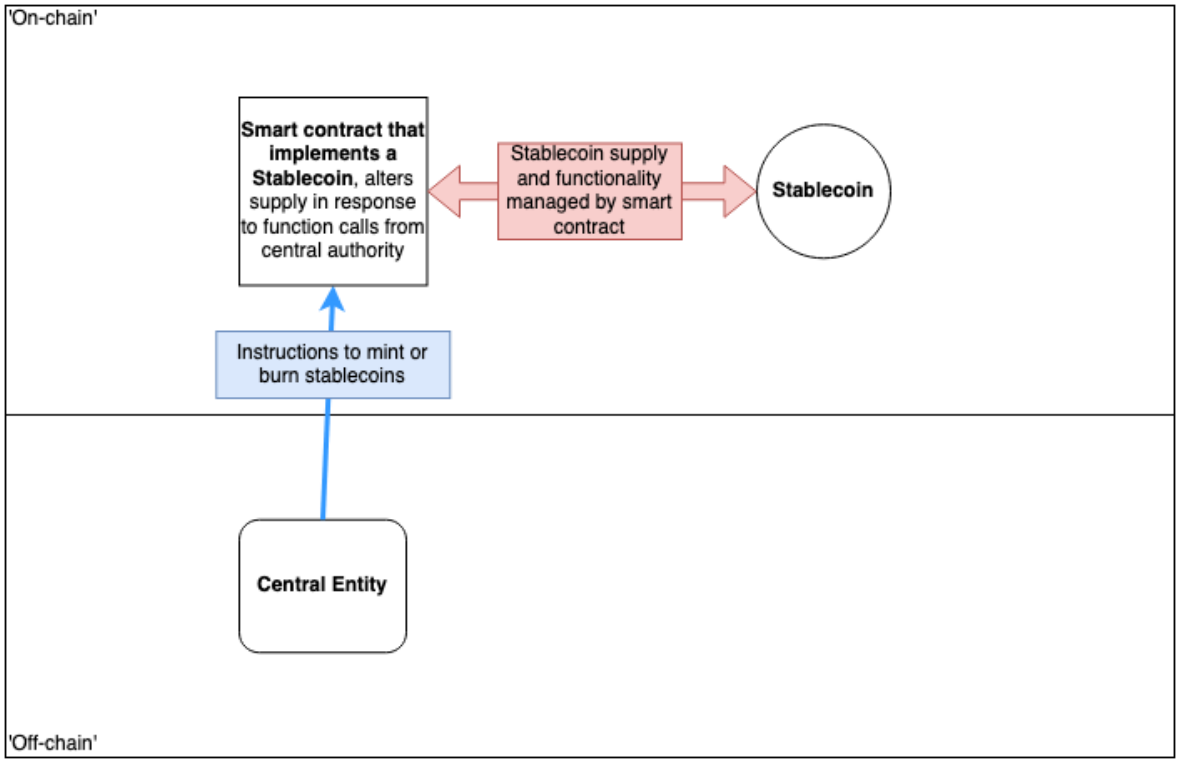


Figure 3.1: Proposed system design for a base fiat-backed stablecoin

The first phase is purely focused on the on-chain logic required to implement the most basic form of a fiat-backed stablecoin and how the Web3 API can facilitate interactions with the stablecoin. This phase is a necessary step towards the final Cryptocurrency backed implementation. It will allow a working understanding to be gained of working with blockchain technologies that can be carried forward. The proposed design for this phase is shown in figure 3.1. As mentioned above, much of the challenges in creating a fiat-backed stablecoin are focused on the banking and payment handling infrastructure; this is outside of the phases' scope because the focus is on learning about building and deploying smart contracts on a live blockchain. A simple frontend will also be developed such that the functionality of Web3 can be discovered and understood.

The technical skills and understanding gained from the completion of phase one can be carried on into phase two and three. The proposed design for these two phases can be seen in figure 3.2. The specific style of stablecoin has been referred to as a 'dual coin' design in literature (section 2.4.2.3. As the name suggests, two different Cryptocurrency assets need to be developed: one which will be pegged to 1 GBP (the stablecoin), and another which is responsible for providing stability to the system (the supporting asset). The system smart contract will handle the values

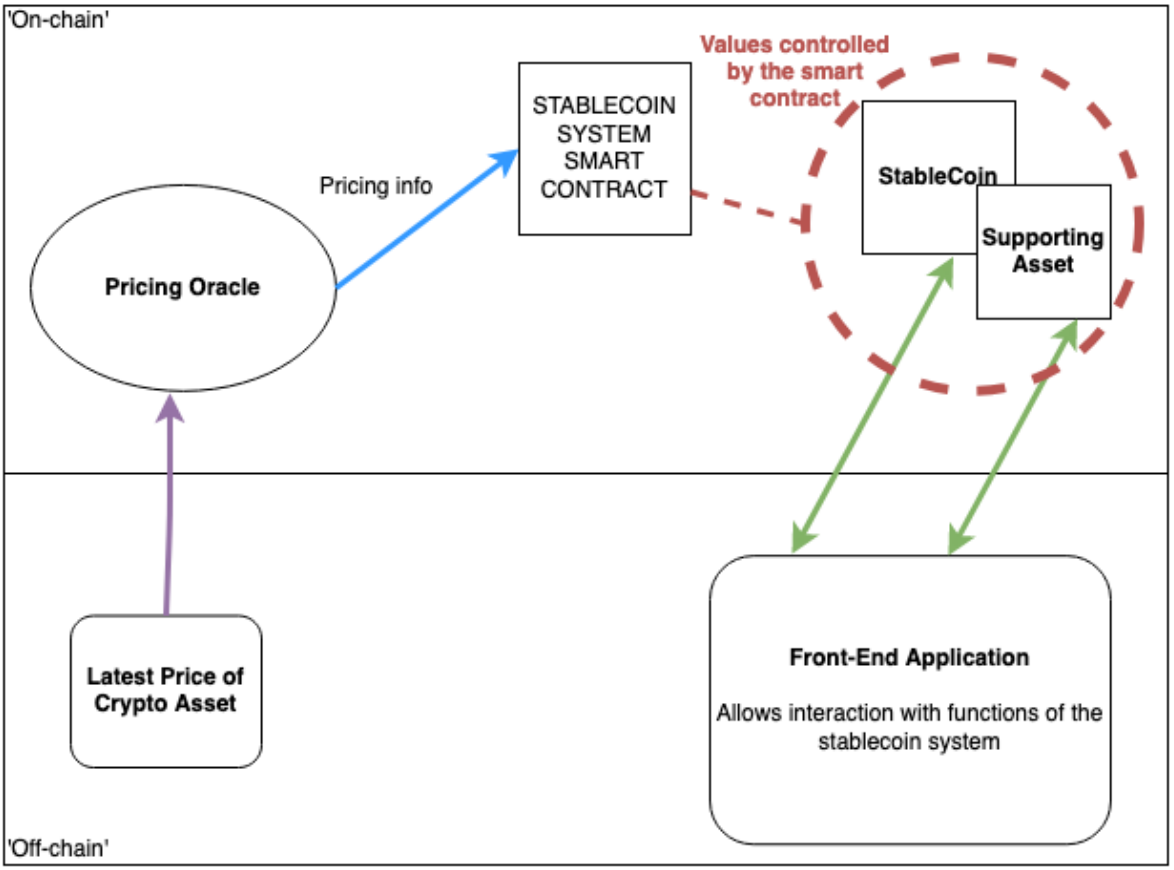


Figure 3.2: Proposed system design for implementing a stablecoin backed by Cryptocurrency assets

and all logic for interacting with the two assets. Of course, for the contract to target the value of 1 GBP as the peg, it will need access to the current market value of the collateral asset in terms of GBP. Through the deposit of a selected Cryptocurrency asset to the smart contract by an actor, the relevant quantity of either the stablecoin or supporting asset will be rewarded to the actor. An inverse of this procedure will occur in the case of a withdrawal.

Phase two will encompass the development of a working oracle system, two unique cryptocurrency tokens, and the smart contract responsible for achieving the stabilisation. Everything in figure 3.2, aside from the frontend will be implemented during this phase. Once phase two has been finalised, the fully-functioning stablecoin system will be live upon a blockchain and available for testing. This project also aims to achieve a working proof-of-concept that can be used to demonstrate and thoroughly test the system's functionality. This aim will be met by completing phase three; in this phase, a frontend will be implemented to interface with a users

wallet and the blockchain.

3.4 Design Choices

This section details each of the critical design choices for the completion of this project in accordance with the phases outlined above, alongside the reasoning behind that choice and detailed insight into its implications. The decisions detailed below were motivated by the state of the art standards and best practice towards blockchain development and influenced by some assumptions that were necessary to scope the project down to a proof-of-concept that can be delivered under the given time constraints. The majority of the design choices are related to phase two of development because this is where the most innovation was required.

3.4.1 Choice of blockchain: Ethereum

As shown in the system diagrams for both phase one and two, a crucial part of the designs are smart contracts that maintain the supply of the stablecoins. Ethereum is the most popular smart-contract capable blockchain with the most extensive development community and a range of specialised development tools. Furthermore, literature review sections 2.5, and 2.4 reveal that Ethereum has multiple test blockchains that are free to develop on. The test network used in this project is the Ropsten blockchain because it is a PoW network with the closest resemblance to the Ethereum Mainnet.

At the time of writing this report, the biggest competitor to Ethereum is the Binance Smart Chain[99]. However, this competitor runs the EVM, and its current success has been attributed to the high gas fees of Ethereum. This project will be developed on a test network; therefore, the gas prices do not matter due to the essentially free transactions. Additionally, this blockchain is not as decentralised and thus goes against the grain of the project.

3.4.2 Choice of the collateral asset: Ether

The natural choice for the collateral asset is Ether, the native currency of the Ethereum blockchain on which the system is being developed. Payable functions for smart contracts make it possible for a smart contract to handle Ether during contract operations and store this value of Ether in its balance. This lends itself perfectly to the use-case of handling deposits and withdrawals

of collateral in the system. Moreover, Ether is provided free of charge on all Ethereum test blockchains, so there will be no funding limitations during testing.

Other options for collateral are available due to the increasing popularity of cross-chain interoperability [100]. However, these options are still somewhat experimental and would add unnecessary complexity to the implementation.

3.4.3 Creating the stablecoin and supporting asset: ERC-20 tokens

ERC-20 tokens are industry standard for implementing fungible tokens on the Ethereum blockchain, the concept and advantages of ERC-20 tokens are explained in section 2.7. ERC-20 tokens are essentially the same as any other Cryptocurrency, except they are implemented on the Ethereum blockchain. By implementing the stablecoin and the supporting asset for this design as ERC-20 tokens, they will be supported automatically by decentralised exchanges and wallet providers. Deploying and customising an ERC-20 compliant token is a well-documented process with lots of development and troubleshooting documentation online, which is very valuable in such a progressive and fast-moving area.

3.4.4 Development environment: Remix

Remix [101] is an open-source IDE used in the development of Solidity smart contracts (section 2.6.1). It is the development environment of choice for this project because it facilitates the entire development process of Solidity contracts. This will be integral for developing the smart contracts for the ERC-20 tokens and other necessary smart contracts for the systems. Developers can access Remix online through any web browser that supports modern JavaScript, making it a very accessible option perfect for this project with minimal setup overhead. The IDE itself contains modules that facilitate the debugging, testing and deployment of smart contracts. A particularly useful module is the JavaScript VM, which allows immediate access to a simulated Ethereum blockchain locally without installing more complex software such as Ganache [102]. The IDE's UI has been designed exceptionally well and will enable complete focus on the contract development rather than configuring various tools locally.

3.4.5 Verifying smart contract code

All of the smart contracts implemented within phase one, and phase two, will have their code published and verified through a third-party service such as EtherScan [103]. The word "trustless" is often mentioned when describing the benefits of a blockchain or decentralised Cryptocurrency. This is true for smart contracts because they are on the blockchain and will consistently execute what is written in the code precisely. However, it does not mean much if parties interacting with the code cannot see what will be executed.

When deployed onto the public blockchain, the human-readable Solidity code is compiled into low-level bytecode that is difficult to understand. The Solidity source code will be published and verified with Etherscan to maintain transparency and earn the trust of all system participants. Once it has been verified, any interested party can locate the smart contract via its address and then inspect the Solidity code. Trust is undoubtedly invaluable for any financial product, especially in the Cryptocurrency space, where any viable product that wants to reap the trustless benefits of a blockchain implementation must release the verifiable source code in a manner.

3.4.6 Selecting an Oracle

Section 2.4.2.4 addresses what an Oracle is, why they are necessary for on-chain computation that requires real-world information, and the issues surround the topic. The collateral currency is Ether. The desired peg is the value of 1 GBP; therefore, an oracle that provides the currency pricing information of Ether in regards to GBP will be needed to ascribe a value to interactions with the system.

When selecting a pricing oracle, a decision must be made, balancing the centralisation and data source risks. Implementing a crowd oracle (explained in the literature review) would be too complicated and is unlikely to work without an active community of information reporters. The two remaining options are an external software oracle or an existing decentralised oracle network such as ChainLink. Unfortunately, the ChainLink oracle network is not available on the Ropsten test network that the development is taking place on; ChainLink would have been a good option for bootstrapping a working system with reliable data as quick as possible. The only viable option that remains is to build an external software oracle.

As discussed in the literature review, the use of such an oracle system presents two main prob-

lems: a single source of failure from the server updating the value, and the system is vulnerable to pricing manipulation or errors if a single source of data is used. The design proposed in figure 3.3 will mitigate the issues caused by a single source of data by calculating a median value from a set of trusted inputs. The risks associated with this type of oracle are explored in the literature review and are understood and accepted.

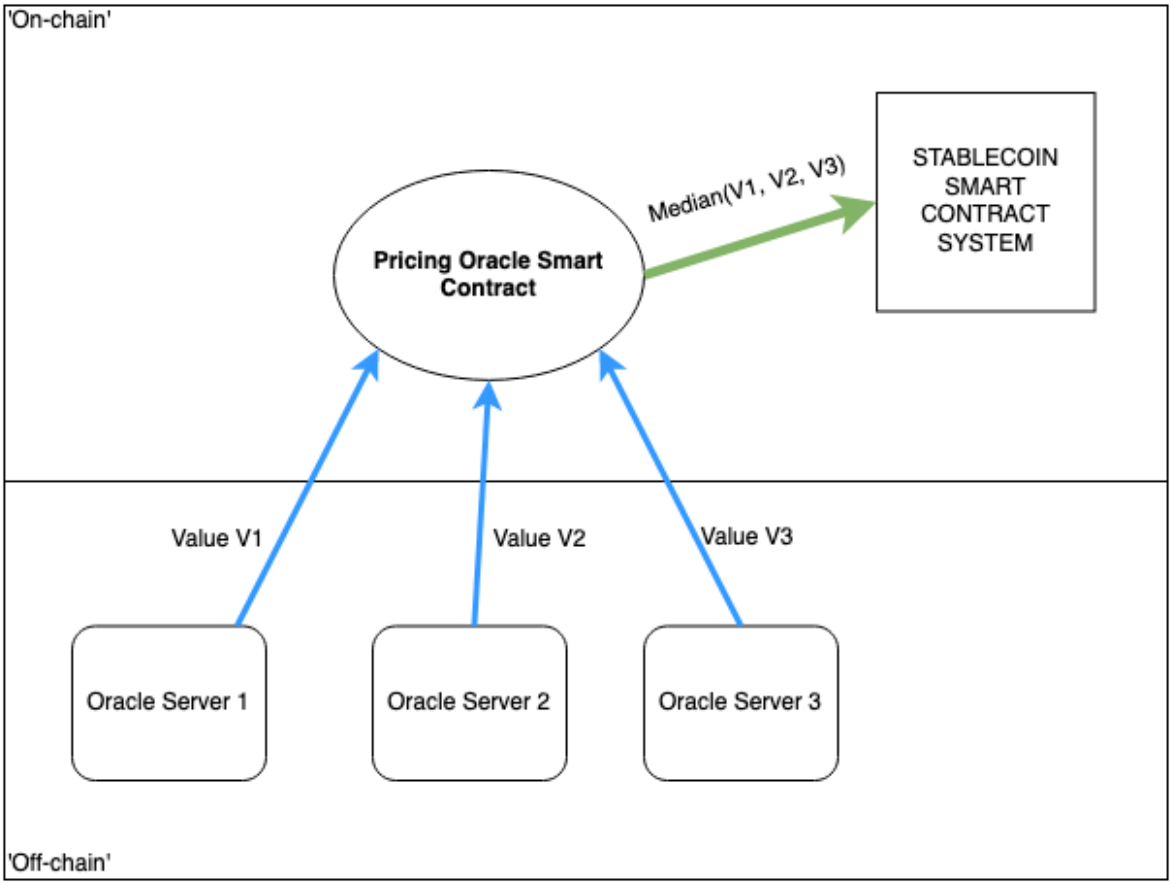


Figure 3.3: Proposed design for an external software oracle.

3.4.7 Off-chain servers and displaying a frontend

Where a supporting back-end infrastructure is necessary to interface with the deployed system or to create a web application, NodeJS shall be employed. NodeJS allows JavaScript to be run serverside and has a rich ecosystem of packages that support the development of Dapps (Dapps are explained in section 2.6). In particular, the Web3 (section 2.6.3) package for NodeJs will be invaluable as it provides an API for directly interacting with content on the blockchain. Where

other packages are needed, they shall be justified, explained, and referenced in the relevant implementation section.

Implementing the stablecoin token on the Ethereum blockchain is, of course, the main focus of this project, but a functional frontend will not be ignored because it will be necessary to allow the testing of the system. NodeJS is the selected server-side language for building the off-chain logic. Therefore libraries within the NodeJS ecosystem will be leveraged for rendering a frontend via a web application. A web-based application can be created and deployed quickly and has the benefit of being accessible on many devices. Furthermore, a web app will allow the frontend to use the MetaMask wallet browser extension for making transactions.

The two NodeJS packages of interest for the frontend are ReactJS [104] and EJS [105]. React is a library that enables the client-side rendering of a web application whilst EJS is rendered serverside and then served to the client as HTML files. Within the different implementation phase chapters, the reasons for using the particular package will be explained.

3.4.8 Using an external contracts library: OpenZeppellin

OpenZeppellin [106] is a library of modular and reusable smart contracts that have been audited heavily and can be trusted as a very secure base for implementation. During phase one, all of the contracts will be written manually. However, in phase two, the OpenZeppellin library will be leveraged for faster and more robust implementation of the base ERC-20 functionality, allowing greater focus on developing the stabilisation mechanisms.

3.4.9 Making security assumptions

During the design and implementation of the three phases, the assumption that the backend servers are completely confidential has been made such that a working proof of concept can be delivered under time constraints. When system interactions require transactions to be sent from a central authority (owner address on the Ethereum blockchain), the private key must be accessible to sign the transaction; in real-world examples, the private key would be held in an encrypted keystore [107]. However, the assumed complete confidentiality means that it is acceptable to store the private key in plaintext.

3.4.10 Stabilisation strategy

The stabilisation strategy for the Cryptocurrency backed stablecoin will involve the application of mechanisms designed to control the supply of the token, and maintain a substantial level of collateral to fully back the token. The strategy is explained and justified in section 5.2.1.

Chapter 4

Implementing the base on-chain logic for a Fiat backed stablecoin

4.1 Introduction

This chapter covers the base implementation for a stable coin on the Ethereum blockchain. First, the design shall be described and explained. Next, the implementation process is documented in detailed sections for each component. The final section of this chapter will contain the evaluation. This design for creating a tokenised British pound on an Ethereum blockchain involves extending a base ERC-20 smart contract's functionality to allow the minting and burning of the tokens (ERC-20 explained in section 2.7). The most straightforward, or base implementation for a token that is stabilised to the value of £1 involves holding British pounds as collateral in a bank account at a ratio of 1:1.

The critical components of this design are:

1. An ERC-20 token extended to enable the minting and burning of tokens.
2. A server equipped with the owners private key, able to mint and burn tokens.
3. A web frontend that can demonstrate these interactions.

The on-chain code can be written within a single smart-contract using the solidity programming language, and this code will need to comply with all the ERC-20 standards. By default, any smart contract that implements the ERC-20 interface will work with most digital wallets sup-

porting the Ether Cryptocurrency; this will allow holders of the token to transfer the currency on the Ethereum network with ease. Furthermore, ERC-20 tokens are compatible with many decentralised trading platforms such as Uniswap, making for an interesting use case.

The smart contract must restrict the ability to mint new tokens such that only the collateral holder has access. The design can achieve this by storing the address which deployed the smart contract, the owner, in a variable and only allowing transactions signed with this address to mint tokens in the future. In a fully functioning implementation, this function would be called by a backend server using the owner address when there is a successful deposit of collateral.

The power to burn tokens is another critical component of this base implementation. Suppose a token holder wishes to redeem their on-chain currency for the return of fiat currency held as collateral; they will need to essentially "delete" the tokens to maintain the 1:1 ratio, thus keeping the peg intact. There must be a function that makes it possible for the backend to burn tokens allocated for fiat redemption.

The final piece of the puzzle is a NodeJS server responsible for interacting with the contracts interface using the Web3 API. In the completed implementation, the server would use a payment processor such as RapidAPI to process fiat payments and withdrawals. Upon a successful transaction, the relevant function on the contract will be actioned. The backend can use the "owner" addresses private key to sign transactions being made to the contract, allowing it to mint tokens. Furthermore, the web app should be integrated with MetaMask to enable users safe interaction.

The web frontend will be hosted on the same NodeJS server mentioned above and demonstrates how the server interfaces with the contract to create a fully functional stable coin system. HTML is rendered server-side and then presented to users of the website. Through the user interface, it is possible to test each of the functions written into the smart contract. In order to demonstrate the functionality to allow the burning of tokens there are some JavaScript functions which will grant the frontend access to the MetaMask browswer extension for proposing transactions.

Processing fiat payments falls outside the scope of this exploratory implementation. It is a building block towards the skills and design process needed for the more advanced on-chain collateral handling and highlights some of the flaws in a fiat-backed stable coin.

4.2 Implementation

4.2.1 Smart contract implementation

The first step in developing any quality piece of software is selecting a development environment that suits your needs. Remix is designed with this in mind and is ready to work for creating smart contracts straight out of the box. The solidity file which contains the source code for the ERC-20 Token is stored within Remix and can be accessed through the "file explorer" within the IDE.

Buterin and Vogelsteller outline the interface for an ERC-20 smart contract in detail on the official Ethereum GitHub repository. One may find this interface on the EIPS branch under "eip-20.md"[reference]. The smart contract implements each of the mandatory functions such that they meet the universal standard; as mentioned in section 2.7. Furthermore, the contract contains optional methods responsible for returning the token symbol and the number of decimals. The constructor for an example token named SimpleStable, shown in figure 4.1 assigns the relevant values to the variables stored within the smart contract. This token is designed with tokenising fiat currency in mind; therefore, the number of decimals is set to 2. Initialising totalSupply as 0 is also crucial because when the token is initially deployed, the assumption can be made that there will be no collateral being held.

```
constructor() public {
totalSupply = 0;                    // Total supply set to 0
name = "SimpleStable";     // Set the name for display purposes
decimals = 2;                       // Amount of decimals for display purposes
symbol = "SIMP";               // Set the symbol for display purposes
owner = msg.sender;        // Set the contract owner as address deploying contract
}
```

Figure 4.1: Constructor for example ERC-20 token.

The mint and burn functionalities were more challenging because there is no pre-defined interface on the official GitHub repository. Minting tokens in this design is kept minimal; an address to

send the new tokens and the number of tokens to create are the inputs as shown in figure 4.2.

```
function mint(address _to, uint256 _value) public returns (bool success){
require(msg.sender == owner); //Only allow calls to function from owner address.
require(totalSupply + _value >= totalSupply); //Prevent overflow of totalSupply
totalSupply += _value; //Adding the value to totalSupply
balances[_to] += _value; //Adding the value to the destination addresses balance
emit Transfer(address(0),_to,_value);
return true;
}
```

Figure 4.2: Minting function for example ERC-20 token.

The code immediately aborts any transaction which does not originate from the owner address; this upholds the design by ensuring only the backend can mint tokens (assuming the secret key of the owner address remains confidential). Aside from checking the address, there is also a logical test to prevent the totalSupply value from overflowing. An overflow, in this case, would occur if the combination of totalSupply and _value was greater than $2^{256} - 1$. In an active stablecoin project, an overflow like this could result in lost funds to the magnitude of millions. If the transaction is safe to continue, totalSupply and the balance mapping to the supplied address are incremented by the input value. Following this, the server emits a Transfer event to notify any listeners that the tokens have been minted.

At first, the ability to burn tokens was achieved in a very similar manner; the critical difference was that it decreased the supplied address's totalSupply and balance. However, it became apparent that this was a poor implementation because the owner address would have complete control to not only mint tokens but also delete tokens at will. After some reflection, it became apparent that the standard "allowance" supplied by the ERC-20 interface could be leveraged. Suppose a token holder wishes to withdraw their collateral and burn their tokens; they can make a transaction that calls the "approve" function assigning an allowance to the owner address. The backend will then use the owner address to burn these tokens and subsequently issue the fiat collateral in return through a payment processor. Token holders are not allowed to call a burn

function directly; this ensures that the backend is aware of the transaction so that the collateral is returned, which maintains the 1:1 peg.

In both new functions, the Transfer event is emitted rather than a custom event. This is important because all the wallets that support ERC-20 tokens will understand these events natively as specified in the standard whereas they would not react to custom events, thus maintaining the benefits of such a design choice. The address provided to these invocations is address(0); this is a shorthand way to refer to the address 0x0000000000000000000000000000000000000000. Using this address is regarded as safe because the chances that a real user interacting with the contract has the private key for this address is meagre.

4.2.2 Deployment to local blockchain

The built-in JavaScript virtual machine within Remix was an invaluable asset whilst testing out the contract. It is possible to deploy the contract locally and interact with all the functions without waiting for block processing times or connecting to a live blockchain. Firstly, the contract is compiled to bytecode which runs on an Ethereum blockchain. Secondly, a local address can be selected to make the deployment transaction and set the transaction values. On this VM, all the addresses have 100 Ether by default. The transaction values (gas limit and value) are irrelevant here because it is being deployed locally. The constructor for this contract does not take any values as input. Therefore, there is no need to enter arguments for the transaction; one press of the "Deploy" button, and the contract becomes available on the VM. Once the code is live in this virtual environment, it is possible to make transactions with and subsequently call functions on the contract. A console that displays all transactions with their hash and status is visible within the IDE. Figure 4.3 illustrates the console alongside the contract interface. Following several re-iterations of the code, the example ERC-20 token, SimpleStable, with mint and burn functionality, was working flawlessly on the VM.

4.2.3 Deployment to live Ethereum test blockchain

Once the local VM deployment was successful, it was possible to realise a deployment on an Ethereum Blockchain. The network upon which SimpleStable was deployed is the Ropsten test blockchain. To create a new contract, a transaction with the bytecode as the payload must be made to the zero address. The address of this contract is unique and is derived from the original

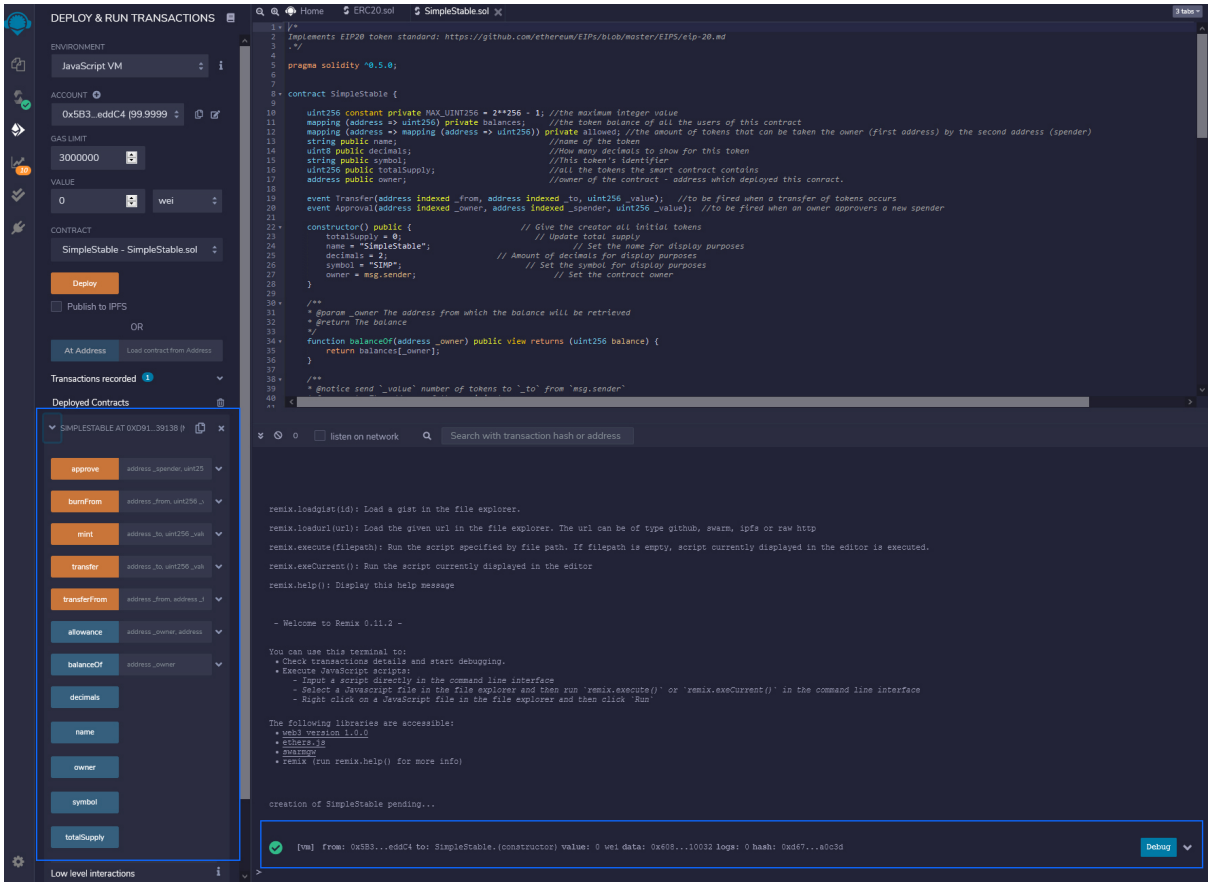


Figure 4.3: Remix user interface after deploying the contract on local VM.

deployment transaction. Any transaction on the blockchain requires a balance of Ether to pay towards gas. As previously discussed in 2.5, the gas pays for both the computation carried out to process the transaction and execute the bytecode on the EVM.

Ethereum addresses are global between the test networks and the main network, therefore to avoid risking any personal assets, a brand new address was created and subsequently imported into a MetaMask wallet. This was done by generating a unique public and private key pair using an open-source Ethereum account generator [108]. This is particularly important because the private key for the owner address will be stored in plaintext, accessible for the backend to sign transactions sent to the smart-contract. Keeping the private key in plaintext is an intentional decision for this design, and the risks of doing so are understood; this is a temporary measure for the development phase.

Fortunately, the Ropsten test network has multiple "faucets" that provide Ether free of charge to any address on the network. Generally, these faucets offer between one and five Ether and

typically have a cool-down timer to prevent abuse. The previously generated address was sufficiently funded to pay for the deployment. The MetaMask browser extension must be installed to allow Remix to interface with this address; The extension makes it possible for Remix to propose a transaction to the wallet owner. The user interface then presents the ability to set the gas price and finally confirm or reject the transaction, this is displayed in figure 4.4.

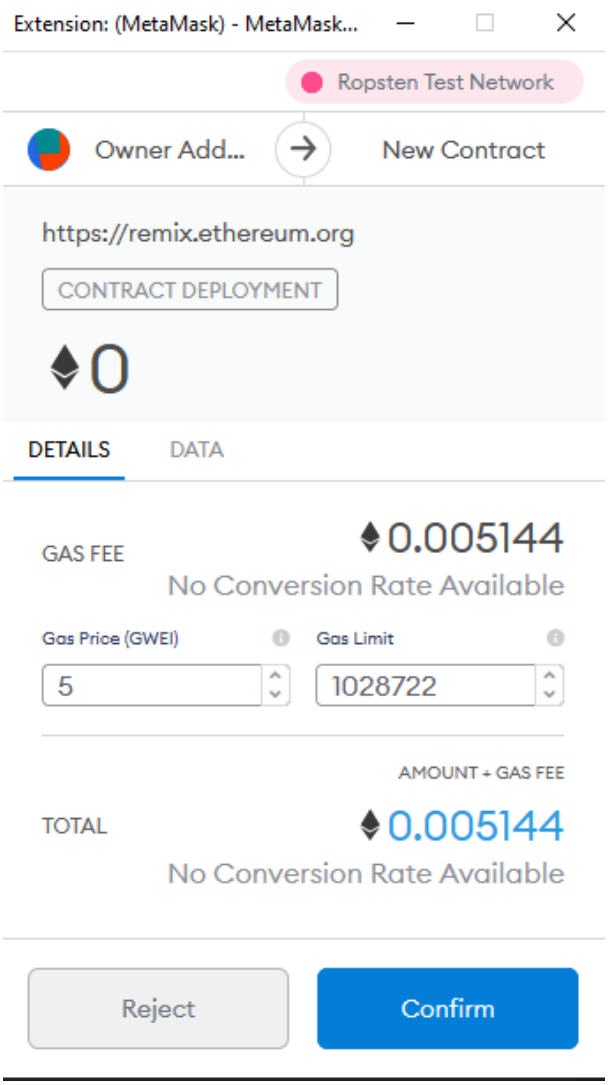


Figure 4.4: Deployment transaction proposal on MetaMask.

Within the IDE, the deployment environment was set to "Injected Web3"; this prompted the MetaMask extension to "connect" to Remix. Once everything connected, clicking the "Deploy" button triggered a new transaction proposal to MetaMask; approving said transaction displayed the transaction information in the console. Once a transaction has been created, it is sent to a mining pool and placed into a block. Once this block has been mined, the EVM will execute

the constructor, and the code will be stored on the blockchain permanently.

Remix stores a cache of "recorded transactions", which grants the ability to interact with the contract on Ropsten through the same interface as the local VM. Function calls are not as quick, however, as transactions must always be processed on the blockchain and require Ether to cover the gas cost.

It is best practice to verify a smart contract's source code on a block explorer such as Etherscan, as shown in figure 4.5. This is necessary for projects which want transparency, something that can be very valuable for a Stablecoin; this value is expanded on in section 3.4.5. Furthermore, it is also an excellent way for the developer to verify that the contract's code on the network is as expected. It is not easy to translate compiled bytecode back into the source code; this is partly down to the significant differences using different compiler versions has on the resulting bytecode. Etherscan allows one to upload a full copy of the source code and the compiler version used; it will then verify that this source code can be compiled to the same bytecode stored at the contracts address on the blockchain. Once this has been confirmed, it is possible to view the source code and read public values directly through Etherscan.

4.2.4 Building the server

Interacting with a contract on an Ethereum network using the Web3 API requires a provider to be specified; this provider allows the application to communicate with an Ethereum node by taking in JSON-RPC requests and returning the response of the function being called. The most decentralised option would be to set up a local Ethereum node. However, this is time-consuming and requires setting up a complicated infrastructure. Infura was chosen as the provider; they facilitate rapid and painless access to the Ropsten network. Connecting to Infura requires a "project ID", which can be obtained for free by creating an account on the website; any time the application makes requests to Infura, it must use this project ID. The final piece of information required for Web3 to function is the contract ABI (Application Binary Interface); this can be obtained in JSON format directly through Remix and is how contract calls are encoded into bytecode which the EVM can understand. Conversely, the ABI is also required to read the binary data returned from transactions. The two types of contract interaction are read and write. Reading data requires no on-chain computation and can be completed without paying any gas. The only requirement to read information is a connection to an Ethereum

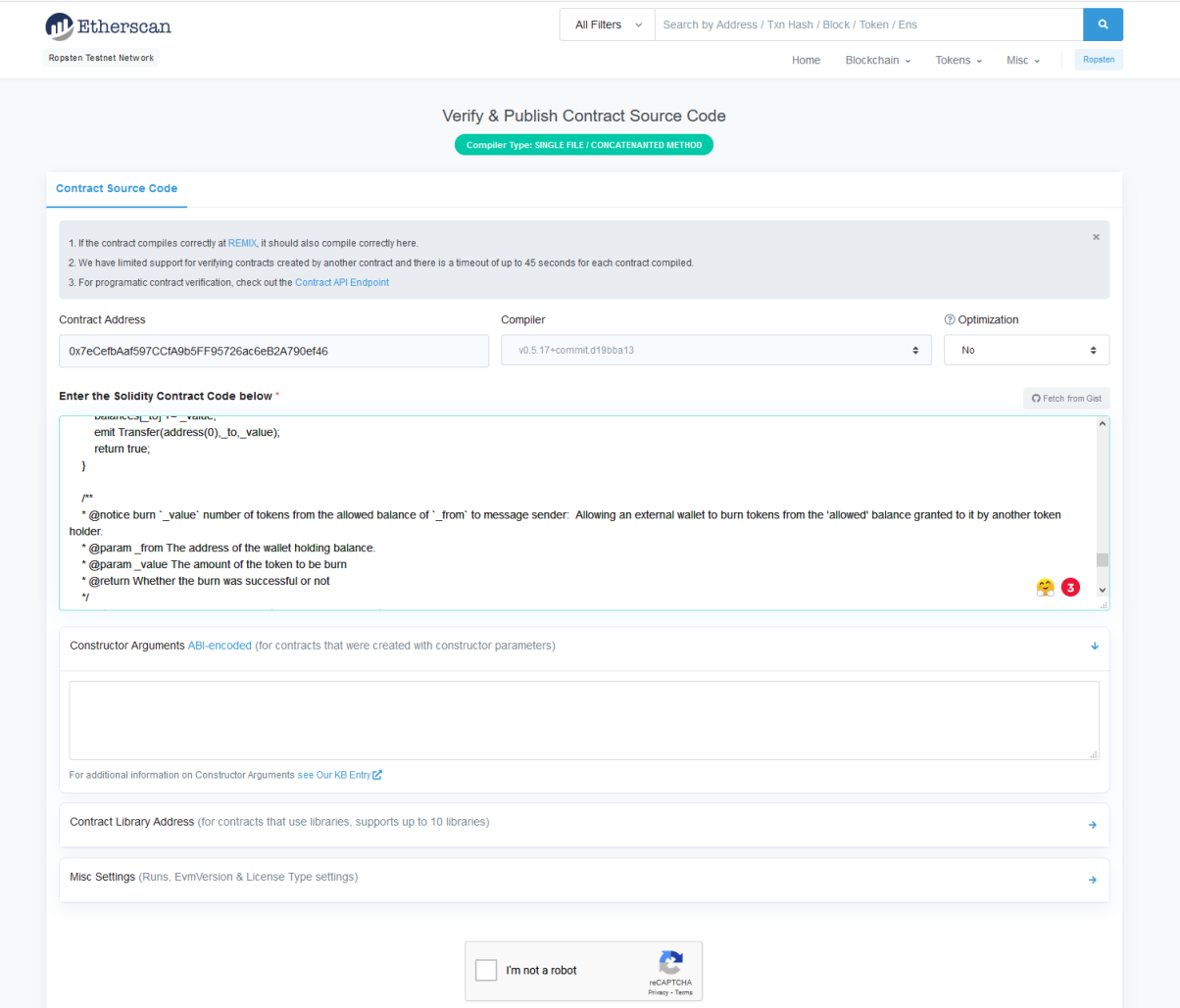


Figure 4.5: Verifying smart contract on Etherscan.

node (achieved with Infura) and possession of the contract's ABI. However, writing data to the blockchain has an inherent gas cost because computation must be carried out to action these changes to the blockchains state. To invoke these functions that modify data, transactions are made to the address of the contract and encoded with the relevant payload using the ABI.

The demonstrative backend shows how the SimpleStable contract implements a fully functional ERC-20 token that could be pegged to the value of £1 if the payment processing infrastructure were built. The backend needs to have the ability to mint new tokens and burn these tokens if the approval has been granted. ExpressJS is the web framework that implements the server API alongside NodeJS, allowing JavaScript to be executed server-side. Express makes it possible to implement a fully fleshed out RESTful API route with a few lines of code, as demonstrated in figure 4.6.

```
app.get('/', function (req, res) {
 console.log('Loading Index Page');
 res.render('index', {balanceIs: null, info: null});
})
```

Figure 4.6: Express route that renders the index page.

This code sample defines a route that handles a "GET" request made to the root of the domain; The server is hosted locally and listening to port 3000. Therefore, this route would apply to any GET request made to "http://localhost:3000/". In the event of this request being made, any data passed with the request can be accessed through the "req" object, whilst the "res" object represents the response sent back to the client.

Some of the contract interactions involve writing data to the blockchain from the owner account; ergo, the backend needs to send transactions; this means that it requires access to the owner's private key. The private and public keys are hardcoded as variables alongside the address for the contract. As mentioned previously in section 3.4.9, this is not the most secure design and would not be acceptable in a production-ready server. Equipped with the contract ABI and setup with Infura as the provider, Web3 can make all the necessary function calls to the SimpleStable contract on the blockchain.

On the server, there is a unique express route implemented to demonstrate each of the critical functionalities: minting tokens, burning tokens, transferring tokens, and checking the balance of an address. The most straightforward route on the server, shown in figure 4.7, is responsible for fetching an addresses balance; this is a read function and does not require any transaction to be made.

The remainder of the functions which require express routes involve writing data to the blockchain; therefore, transactions must be made. The ability to call the mint or burn function is limited only to the owner address, and this is where the hardcoded private key is crucial in signing the transactions. The server employs the lightwallet[reference to npm page] and etheremjs-tx [reference to npm page] packages for NodeJS to make transactions. As shown in figure 4.8, building and subsequently sending a transaction can be quite a lengthy process.

The first step in making the transaction is defining the transaction options within an object (tx-

```
app.post('/balance', function (req, res) {
console.log('Getting balance to display on webpage for address: ' + req.body.address);
if (typeof req.body.address != 'undefined'){
//Setting Web Provider
const web3 = new Web3(
     new Web3.providers.HttpProvider('https://ropsten.infura.io/v3/' + InfuraEndPointKey)
);

//Load contracts interface
const contract = new web3.eth.Contract(ContractInterface, ContractAddress);
contract.methods.balanceOf(req.body.address).call(function(err, result) {
if(err) {
//if there was an error when calling the function do the following:
console.log("error when calling the smart contract function: " + err);
//display error to the user
res.render('index', {balanceIs: null, info: err});
     } else {
//if the contract call succeeds
console.log("the contract function returned: " + result);
//display information to user
res.render('index', {balanceIs: 'The balance of address ' + req.body.address +
' is: ' + result, info: null});
     }
});
} else {
res.render('index', {balanceIs: null, info: 'toAddress or value is not undefined'});
}
})
```

Figure 4.7: Express route which returns the balance of a given address.

Options); these must be configured correctly for the transaction to be processed. The options contain the transaction's destination address (the SimpleStable contract address), gas price, gas limit, and the nonce. EVM computation requires gas to fund it; any contract calls must be sent with an adequate amount of gas. The nonce within txOptions refers to the number of transactions that have been made by the owner address and is essential in preventing double-spending and ordering the processing of transactions. Every transaction on an Ethereum blockchain requires the sender to supply their nonce value.

```
    let txCount = await web3.eth.getTransactionCount(ownerAddress);
console.log("txCount: " + txCount);
var txOptions = {
nonce: txCount,
gasPrice: 3000000000*1.4,
gasLimit: 8000000,
to: ContractAddress }
console.log("Your transaction options: " + JSON.stringify(txOptions));
var rawTx = txutils.functionTx(ContractInterface, "mint" ,


console.log('signing transaction');
var privateKey = new Buffer(ownerPrivateKey, 'hex');
var transaction = new tx(rawTx, {'chain':'ropsten'});
transaction.sign(privateKey);
//Sending transaction
console.log('sending transaction');
//before sending the transaction it needs to be serialised into hex format
var serializedTx = transaction.serialize().toString('hex');
web3.eth.sendSignedTransaction('0x' + serializedTx, function(err, result) {
...}
```

Figure 4.8: Express route which returns the balance of a given address.

Secondly, an RLP-encoded hex string, rawTx, which defines the transaction and the computation to be carried out on the blockchain is created. To obtain this string, the contract ABI (Contract-Interface), function name (mint), function arguments, and an object containing the transaction options are passed into functionTx from the txutils module within lightwallet. Thirdly, a signed transaction is created using the Transaction module from the etheremjs-tx package (imported as tx) and the owner's private key. The same module is used again to complete the transaction's RLP encoding, serializedTx, via the Transaction.serialise() method. Finally, Web3 is used to send the fully configured, signed, and processed transaction.

4.2.5 Building the user interface

Besides implementing the server API, Express also makes it possible to use static HTML template files, which are altered at runtime by the server prior to being served to the client; this functionality can be unlocked by selecting a view engine to be used. The view engine chosen for this project is EJS [105] (Embedded JavaScript) because it provides all the functionality required with minimal overhead. The Index.ejs file is the template that renders the entire user interface and can be visited by navigating to http://localhost:3000 whilst the server is running.

Within the rendered user interface, as illustrated in figure 4.9, five different sections contain a form: each responsible for demonstrating a function of the token. The various input fields for a given section must be completed before pressing the button. For this demonstration, the server can be considered the owner because it has full knowledge of the private key and uses it to make transactions on the blockchain. Minting, burning, and transferring tokens through this frontend are demonstrated using the owner's account; filling in the data and pressing the button will result in a post request being made to the appropriate Express route. The code within the route will then be executed, and the server will complete the transaction to carry out the desired function.

The demonstration for allowing the owner to burn tokens is the only section that does not have an Express route on the server; it is unnecessary because the allowance must be granted to the owner by a third party that holds a balance of SimpleStable tokens. Allowing tokens to the owner requires writing data to the blockchain. As a result, a transaction must be made to the smart contract with the relevant payload to make the correct change.

Within the web-application, there are two separate JavaScript files, connect.js and web3.min.js

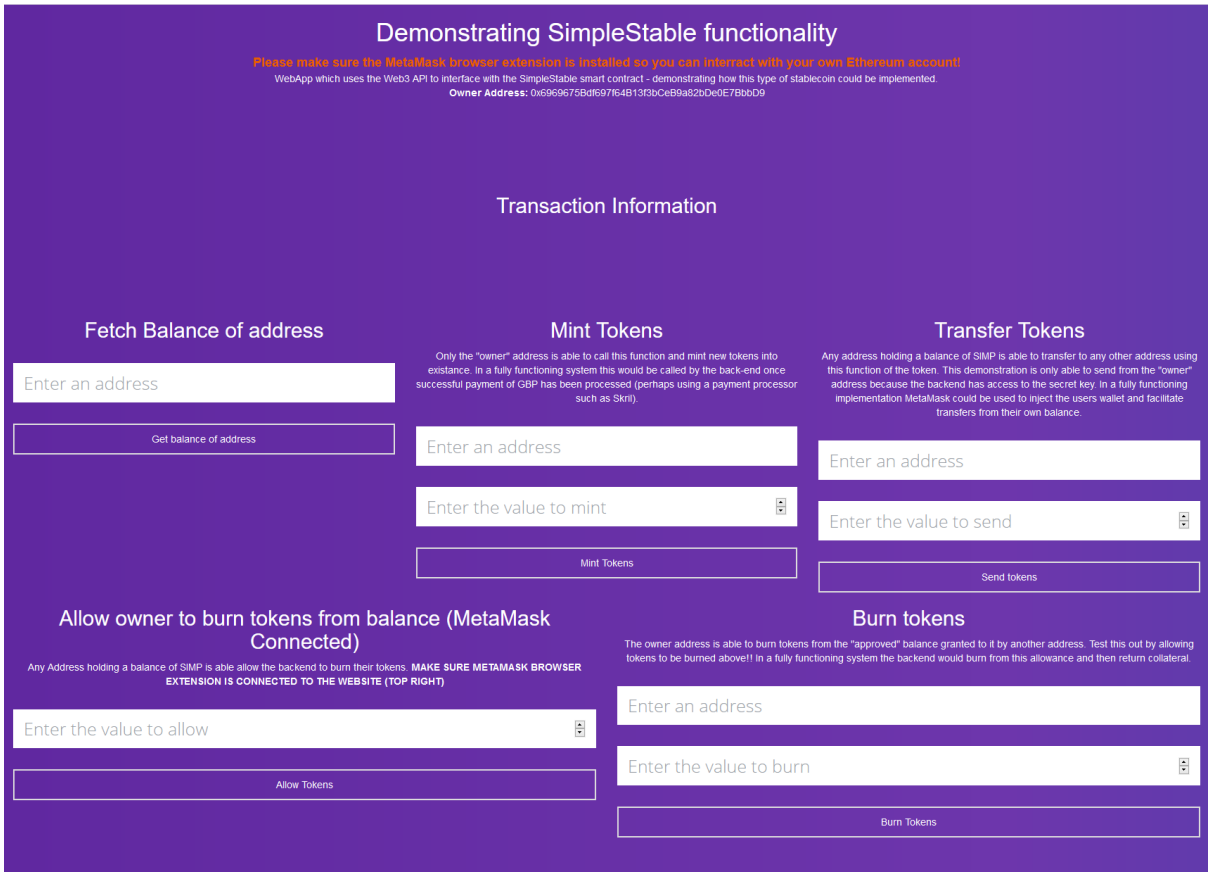


Figure 4.9: User Interface of the demonstration app.

, which are served to the client. Importing the JavaScript functions within the files makes it possible for the user interface to interact with the MetaMask browser extension. Upon first visiting the webpage, there is a prompt to allow a connection to MetaMask.

```
function allowTokens(to, ammt) {
  //contract instance
  contract = new web3.eth.Contract(abi, contractAddress);
  contract.methods.approve(to, ammt).send(option).then(console.log).catch(console.error
}
```

Figure 4.10: JavaScript function which allows the web app to propose a transaction to client via MetaMask.

Suppose a user possesses a balance of SimpleStable tokens and wishes for them to be burned;

they would enter the quantity to allow and press the button labelled "Allow Tokens". Assuming the user has the MetaMask extension installed, the application will then propose a transaction to the user, using the code from figure 4.10, which will make a call to the approve function on the smart contract with the correct data. The contracts ABI is available as a hardcoded JSON within the connect.js file to correctly encode this transaction's payload. An example of the MetaMask transaction proposal can be seen in figure 4.11

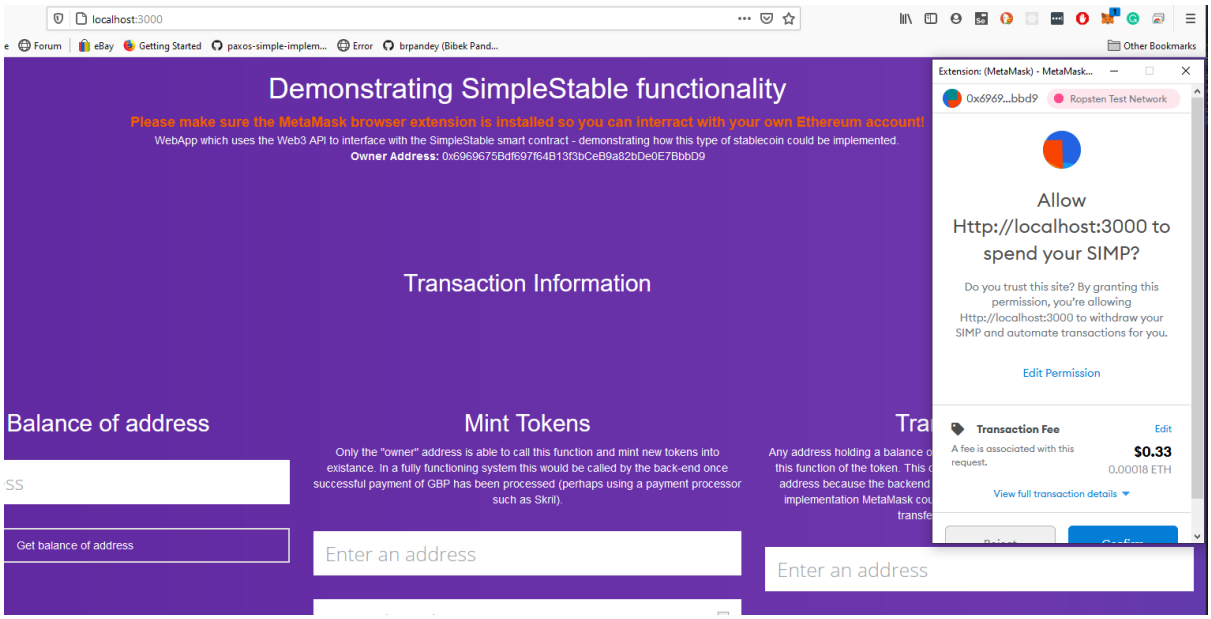


Figure 4.11: Transaction proposal for transaction which allows tokens to be burned.

4.3 Evaluation

The development and subsequent deployment of a fully functional ERC-20 Token, named SimpleStable, on an Ethereum blockchain has been achieved. A web-based demonstration has been built, making it possible to see how a central entity can mint and burn these tokens in return for the deposit or withdrawal of fiat collateral. Following the minting of tokens, it is possible to test other token functions, such as transferring the balance to another Ethereum account using the web application.

SimpleStable, SIMP, has been deployed publicly on the Ropsten blockchain. Additionally, the source code has been verified on Etherscan [103]. Therefore, it is possible to see evidence of

the transactions being made and how they impact the values stored on the blockchain. A test scenario that demonstrates all of the functions of the token is listed below.

1. Minting x quantity of tokens to the owners Ethereum account.
2. Checking the balance of the owner account.
3. Transferring some balance of SIMP to an account y, which is connected to the MetaMask browser extension.
4. Allowing the owner to burn some of y's balance
5. Burning this allowance from y.
6. Checking if the expected changes are made to totalSupply and the balances of the owner and y accounts.

This scenario has been tested using the web-app, and the transactions highlighted in figure 4.12 provide proof of the on-chain activity for steps 1, 3, 4 and 5. It is impossible to verify that the balance checks of step 2 and 6 occurred through transaction logs because they read functions. The Etherscan block explorer for the contract can be found at https://ropsten.etherscan.io/address/0x3af9ff085ca4b3

One of the most troublesome development areas arose during the testing of the live contract on the Ropsten network. As described throughout this paper, all transactions on the blockchain require gas to pay for the necessary computation; when a transaction is made without an adequate amount, it is aborted. Smart contracts and blockchain technology, in general, are very much in the development stage still, and as a result, the libraries needed are updated frequently. An update made to the txutils library meant that the gasPrice in the transaction options no longer required to be encoded in hex. Making the transactions whilst using hex encoding had a strange interaction where they would be made with a minuscule gas amount and automatically time out after 750 seconds.

The Ropsten network is a proof of work blockchain, which means that network nodes known as miners must provide computational effort to validate the next block. The gas supplied with the transaction is paid to the miners. Therefore they will generally carry out transactions with a higher fee first. Because the gas being sent with the transactions was extremely low, no miners selected them to be computed. The error logs would state that the gasPrice was set too low. Multiple increases were made, yet it displayed the same result. Eventually, after printing the

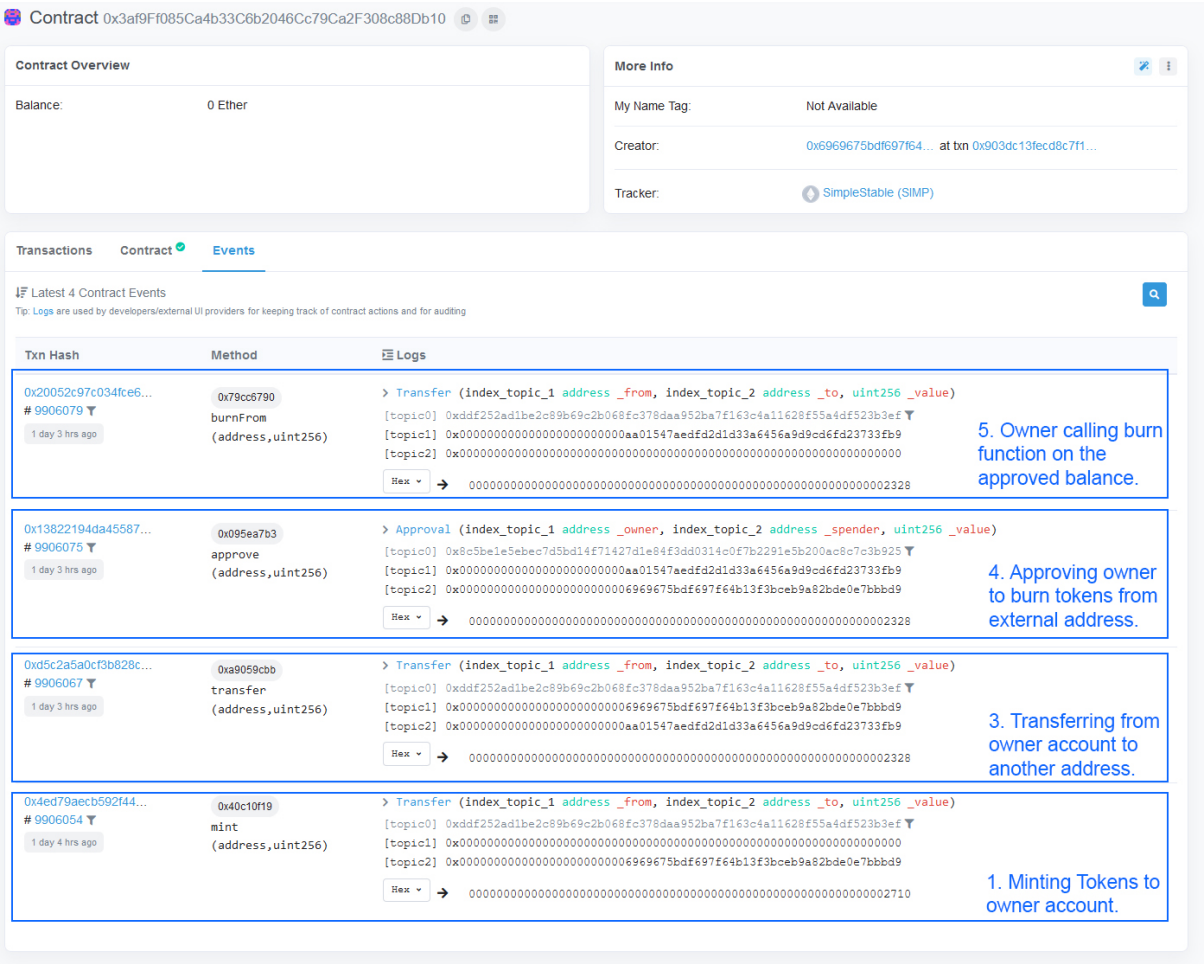


Figure 4.12: Transactions on the blockchain as a result of test scenario.

transaction information, it became clear that somehow the value was not being passed into the transaction correctly as the gasPrice value did not update according to the hex value. Simply removing the hex encoding and passing it in as a regular integer fixed this error.

Through the completion of the base implementation, some of the flaws in this fiat-backed stablecoin design become very apparent; if a single entity can mint tokens, the token's stability entirely relies on the secrecy of the owner's private key. Keeping the secret key in an encrypted form would be a good improvement. However, the token's stability would further rely on the encryption strength and password used during the encryption. A crypto-collateralised (Ethereum in specific) would remove the need altogether for an owner account to mint tokens; the contract itself would be able to react to on-chain deposits and mint the necessary quantity.

Creating robust and functional smart contracts will be crucial for a future implementation, which uses on-chain collateral to maintain a peg. The various NodeJS packages (lightwallet,

Web3 and EthereumJS-tx) will also be vital when implementing any off-chain logic on a server, or a frontend, which needs to interface with a smart contract or query data on the blockchain. In particular, stabilising the tokens value using cryptocurrency assets will require an Oracle that provides pricing data to the smart contract; this can be implemented using NodeJS to update the Oracle data periodically. It is worth noting that the various advantages of utilizing the ERC-20 standard for the base design also apply to the future implementation.

Chapter 5

Implementing a Cryptocurrency backed stablecoin

5.1 Introduction

This section covers phase two of the project, the final on-chain implementation of the project: a stable coin on the Ethereum blockchain pegged to the value of 1 GBP using purely on-chain assets. Firstly, the design shall be explored. Secondly, the implementation process is documented in several key steps, followed by a critical evaluation.

5.2 Design

This design has been inspired by the ideas introduced in the Dai stablecoin project (section 2.4.2.3) and attempts to replicate the success of Dai through a design that applies the fundamental economic principles of the Quantity Theory of Money (section 2.4). The stablecoin token introduced in this project, ST, attempts to maintain a stable value by applying various mechanisms that serve to contract or expand the token supply. The solution does not need to attract investors to buy into the supporting asset more than the existing systems. The project's scope works under the assumption that investors will employ the mechanisms enabled by the smart contracts in good faith to uphold the system. Figure 5.1 demonstrates a high-level overview of the design, which will use Ether (the native currency of the Ethereum blockchain) as the collateral currency.

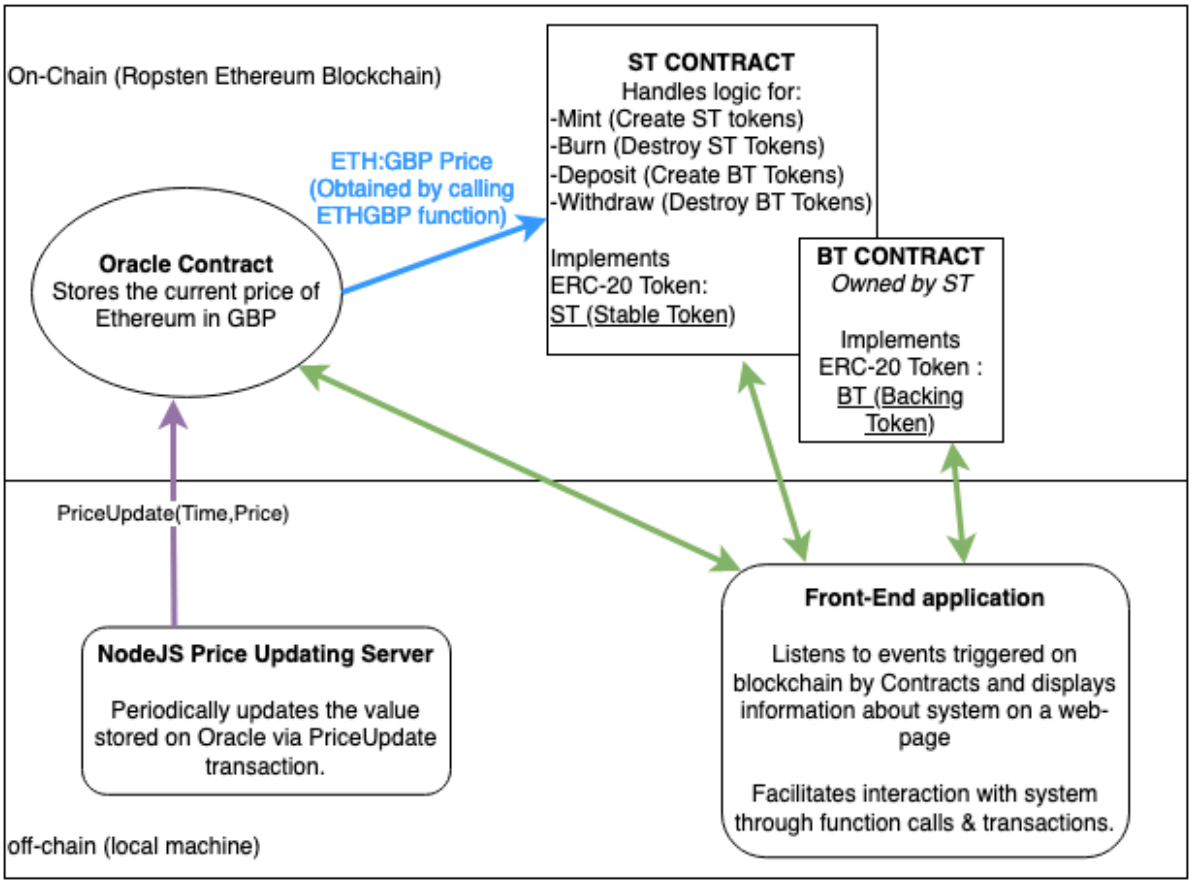


Figure 5.1: System Diagram for Ether Collateralised Stablecoin

5.2.1 Stabilisation Strategy

As described in the problem analysis and design section (3.3), the system proposed in this project is an example of a dual-coin design; the two tokens implemented are 'stable token' (ST) and 'backing token"(BT). ST is the primary token aiming to stabilise at 1 GBP, and BT is the secondary token that serves to de-risk ST tokens and brings stability to the system.

The mechanisms implemented within this system aim to stabilise the market price of the ST token through the holding of Ether collateral within a 'vault'. In a functioning market where there is ample liquidity, if the collateral being held is sufficiently high enough to fully back all of the ST tokens, then the price of one ST can never fall below the peg of 1 GBP. Suppose ST tokens are trading below the peg on an exchange; individuals will be incentivised to purchase the tokens at a discount and then redeem these tokens in return for the underlying collateral. The opposite is true in the case of the coin trading above the peg, individuals can deposit Ether into the system and obtain ST tokens to sell for a profit. This natural process of maintaining

the peg through the market activity of system participants is explained in the literature review section 2.4.2.1 and is the process that Tether and other fiat-backed coins employ. Of course, the challenge here lies in the fact that maintaining enough collateral to back the tokens entirely is difficult because the price of Cryptocurrencies can be highly volatile.

The secondary token of this system, BT, serves to raise more collateral for the vault. In the case of negative price deviations, excess collateral raised through the creation of BT tokens will provide an ample 'buffer' that can absorb the cost of the price falling. In the case of a rising Ether price, holders of the BT token stand to make large profits because they are essentially holding a leveraged position [109] on the collateral they have invested. BT will appeal to investors that believe in the long-term success of the Ether Cryptocurrency. Aside from offering an opportunity to take a leveraged position on the price of Ether, holders of the BT token will be financially rewarded for supporting the peg using proceeds from transaction fees charged during the minting or burning of the ST tokens.

The system proposed in this design will attempt to maintain a collateral ratio of 120%; the collateral ratio can be calculated as (vault value)/(total supply of ST tokens). This means that for every 1 ST token minted; the system wants the vault to hold at least 1.2 GBP worth of Ether. The 20% 'buffer' serves to pad the vault value and protect the token from the daily fluctuations that occur commonly with Cryptocurrencies. When a user deposits x GBP worth of Ether in return for ST, they will be always be rewarded with either 0.992*x or 0.998*x ST tokens; a small fee is charged on mint operations. This fee is determined by the current collateral ratio of the system. When the ratio falls below 120%, the fees are tripled to prevent users from further diluting the collateral ratio. On the other hand, if a user wants to burn x amount of ST tokens, they will be rewarded with either 0.985*x or 0.995*x GBP worth of Ether. Again, the base 0.5% fee is tripled to 1.5% when the collateral ratio falls below 120%.

However, the unit price of one BT token fluctuates depending on the current value of the vault and the outstanding debt to ST token holders. When there is a positive buffer (when the vault value is greater than ST total supply), the price of a single BT token can be calculated as (buffer value)/(BT tokens total supply). However, when the buffer is negative (ST tokens total supply is greater than the vault value), the unit price of a BT token is set to either 0.50 GBP or the (current collateral ratio * 1) in GBP, whichever is higher. If there is a zero supply of BT tokens or no buffer, the unit price is set at 1 GBP worth of Ether. The cost of BT tokens decreases as

the collateral ratio decreases because the system needs the Ether deposits even more to maintain the peg. Eventually, the price will lower to a value that is attractive for investors to deposit their funds for the chance to earn big profits, which will in turn raise the collateral ratio and secure the system. The floor price of 0.50 GBP has been selected to prevent severe dilution of the BT token value.

The final measure to uphold the collateral ratio is to freeze the burning of ST and BT tokens. When the collateral ratio falls below 120%, BT tokens can no longer be redeemed for Ether until the ratio rises again. ST tokens can be redeemed for the underlying collateral until the ratio falls below 100%. Once the ratio dips below 100%, the burning of both ST and BT tokens are frozen; this is a necessary step to prevent a bank run [110]. The likelihood of Ether's price never recovering is not very high. However, investors will need to understand this risk and the fact that even if the price is to recover, it may take a long time – during which the deposited Ether will be locked up.

Holders of ST tokens will always be able to exchange their tokens into BT at the current unit price for BT; this will save them Ether on Ethereum gas fees for moving the capital around. When the collateral ratio falls below 120%, there is no fee charged on the exchange; this will encourage holders of the ST token to transfer into the BT tokens that do not have any burn fees, which should, in turn, increase the buffer size. When the collateral ratio is above 120%, the regular burn fee of 0.5% is applied to prevent users gaming the system to withdraw without paying the fees when there are sufficient funds in the vault. When investors holding ST believe the price of Ether is likely to rise, they can transfer their tokens into BT to take a leveraged position without sending multiple transactions.

Four components must be developed to implement the mechanisms described above :

1. Pricing oracle which stores the ETH: GBP price on-chain.
2. Server(s) that keep the price information held by the oracle updated.
3. An ERC-20 Contract, Backing Token, which implements the supporting token BT.
4. An ERC-20 Contract, StableToken, which encapsulates the on-chain logic for the system to function and implements the ST token that will be stabilised.

5.2.2 Oracle

The mechanisms described above require up-to-date pricing data for Ether; this functionality will be delivered through the median price oracle system outlined in the design choices (section 3.4.6). The latest price update accepted by the oracle will be stored within a publicly accessible variable on the contract. The other contracts in the system can then access the information. To achieve the median pricing functionality shown in figure 3.3 the oracle contract must receive pricing updates via transactions from a set of trusted servers. The contract will keep track of addresses used by the pricing update servers, and when a trusted price update is received, it will be added to the list of pending price updates. Once price updates have been received from each of the trusted servers, the median value shall be adopted as the current price of Ether.

5.2.3 Pricing Server(s)

In a production environment, the price updating servers would be distributed and hosted so that they are very robust. However, to get the system running in a testing environment, a single server will execute a NodeJS script that periodically fetches pricing information from three industry-leading Cryptocurrency market API's and subsequently makes transactions to the oracle contract containing the pricing information. The three API's are:

- CoinGecko [111] allows unlimited queries for free without an API key.
- Coinbase [112] allows unlimited queries for free without an API key.
- CoinAPI [113] allows 100 queries daily for free with an API key.

The limitation of using CoinAPI is that the demonstration of the system is limited to the 100 free updates per day; in production, this could be navigated by purchasing the paid tiers, but it should not be a problem for the demonstrative system. As mentioned in the oracle design, there will be three different addresses that are allowed to send a price update to the contract; thus, the server will need access to the private keys for these Ethereum addresses to create and sign the transactions.

5.2.4 Backing Token smart contract

The Backing Token contract will implement the BT ERC-20 token, and have its functionality extended such that the contract can have an owner. The owner of the Backing Token contract will be the Stable Token contract, which will have the ability to call the mint and burn functions in response to system events. The game theory behind BT and its role in stabilising the price of ST can be found in the stabilisation techniques section above.

5.2.5 Stable Token smart contract

The Stable Token contract will encapsulate the bulk of code responsible for implementing the stabilisation techniques, and the ST ERC-20 token. Additionally, the contract will handle all deposited Ether collateral. The majority of the logic for enabling the system to operate can be split into four functions: mint, burn, deposit, and withdraw.

The mint function in this contract extends the functionality seen in the mint function from the base implementation; this will facilitate the creation of new ST tokens based on the GBP value of Ether being sent to the contract. The value will be calculated according to the current market price of Ether fetched from the pricing oracle. The burn function is also an extension of the initially implemented function of the same name in section 4.2.1. It will enable ST token holders to specify the number of tokens, x, that they wish to burn. These ST tokens will be burned, and in return, the function caller will be sent x GBP worth of Ether (minus fees as explained above); again, this is calculated using the market price of Ether according to the oracle.

The contract will also have deposit and withdrawal functions responsible for creating and destroying BT tokens. The Stable Token contract will be the owner of the Backing Token contract and can access permissioned actions such as minting and burning BT tokens. When a deposit transaction is received, the Stable Token contract will mint the correct amount of BT to the transaction sender. This amount is calculated according to the formula explained in section 5.2.1. Inversely, the contract will calculate the Ether value of the tokens being burned and send this amount of Ether to the transaction sender when a withdrawal transaction is received.

5.3 Implementation

5.3.1 Pricing Oracle

This oracle, of which the imports, state variables, events, and constructor code is shown in figure 5.2, is implemented according to the design explained above. The contract inherits from the OpenZeppelin Ownable implementation [114]. As a result, the sender of the deployment transaction is automatically assigned to an 'owner' variable within the contracts state; this owner will be one of the trusted addresses. The other two trusted addresses to send pricing information are implemented as private variables – Server2Owner, and Server3Owner. The other state variables for the contract are ETHGBP, updated, updating, and pendingUpdates.

"ETHGBP" is the public integer value that will store the current market price of Ether in terms of GBP. Other smart contracts, DAPP's, or individuals on the blockchain will be able to read this value off of the contract free of charge. When the price is updated, the current timestamp is saved into the "updated" variable, providing proof of the freshness. The variable pendingUpdates is an array of integers; when a pricing update is received from a specific server, it will be placed into the relevant index of pendingTransactions. Once all three pricing updates have been received, the median value can be calculated from the array. The Boolean variable 'updating' is a flag that will be set to True when the median value is being selected; this is important to prevent multiple transactions triggering the same function whilst the calculation is taking place. An event, PriceUpdate is also defined; this is triggered whenever the public variable ETHGBP is updated and serves to update DAPP's that rely on the pricing information. When the constructor is executed, it takes two addresses; these are assigned to the variables storing the trusted server addresses. It will also fill the pendingUpdates array with 0's and set the updating flag to false.

The code shown in figure 5.3 is an example of a updatePriceServer function. There are three of these; one for each of the trusted parties. Each of the functions has a 'modifier' that ensures that only the permissioned address can use the function. Suppose a server is sending a price update to the oracle contract; they would call their specific update function on the current Ether price in string format. If the server has permission and has called the correct function, the string will be converted into integer format via the safeParseInt function from the usingOraclize library. This integer value will then be passed into the addToPending function (figure 5.4 alongside the

```
pragma solidity ^0.5.0;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.5.0/contracts/owne
import './oraclizeAPI_0.5.sol';
contract Oracle is Ownable{
    uint public ETHGBP;
    uint public updated;
    uint[3] private pendingUpdates;
    address private Server2Owner;
    address private Server3Owner;
    bool private updating;

    event PriceUpdate(uint _time, uint _value);

    constructor(address _owner2, address _owner3) public{
        pendingUpdates = [0,0,0];
        Server2Owner = _owner2;
        Server3Owner = _owner3;
        updating = false;
    }
...}
```

Figure 5.2: Pricing oracle source code: variables, imports, and constructor

```
function updatePriceServer1(string memory _value) public onlyOwner returns(bool) {
    addToPending(0, usingOraclize.safeParseInt(_value,18));
    return true;
}
```

Figure 5.3: Solidity source code for a sample updatePrice function.

allocated index for this server. The decision to implement three different functions for updating the prices, rather than performing a logical test on-chain to determine the relevant index, was

made such that there are fewer gas fees for updating the price. This is an important consideration because transactions to update the price will be made continuously, which could become costly very quickly. Of course, the trade-off here is that the up-front cost of deploying the contract will be slightly higher.

```
function addToPending(uint8  _id, uint  _value) internal {
    pendingUpdates[_id] = _value;
    if (updating != true && pendingUpdates[0] != 0 && pendingUpdates[1]!= 0 && pendingUpd
        updating = true;
        uint[3] memory sorted = insertion(pendingUpdates);
        ETHGBP = sorted[1];
        updated = now;
        pendingUpdates = [0,0,0];
        updating = false;
        emit PriceUpdate(updated, ETHGBP);
    }
}
```

Figure 5.4: Solidity source code for addToPending function

The addToPending function is responsible for assigning an incoming price update to the correct index of the pendingUpdates array; once the value has been added to the array, the function checks if all three price updates have been received. If an update has been received from all three servers, the updating flag is set to true, and the process of adopting the median value begins. The array is sorted numerically, and the median value is adopted. Unlike many modern programming languages, Solidity has no built-in sort function, and computation on the EVM comes at the cost of gas. Therefore it is vital to select an efficient sorting algorithm. The sorting algorithm is supplied by the insertion function, shown in figure 5.5

Figure 5.5 displays the solidity implementation for the insertion sort [115] algorithm, which was specifically selected because it can be implemented in a few lines of code and is very efficient for small data sets. The pendingUpates array is only three elements in length, and we want to keep the code as short as possible to lower the cost of deploying the transaction, a perfect use

```
function insertion(uint[3] memory data) internal pure returns(uint[3] memory){
    uint length = 3;
    for (uint i = 1; i < length; i++) {
        uint key = data[i];
        uint j = i - 1;
        while ((int(j) >= 0) && (data[j] > key)) {
            data[j + 1] = data[j];
            j--;
        }
        data[j + 1] = key;
    }
    return data;
}
```

Figure 5.5: Solidity source code for insertion sort algorithm

case. Once the array has been sorted, the centre-most element can be adopted as the current market price of Ether. Following the price adoption, the pendingUpdates and updating flag are restored to their default state, and a PriceUpdate event is triggered.

5.3.2 Price updating server

The NodeJS script responsible for updating the pricing oracle employs the same NodeJS packages as the back end implemented in the exploratory implementation (section 4.2.4); lightwallet and Web3. The script uses an additional package; axios [116]. Axios is a promise-based client for making HTTP web requests, and promises are a unique feature of JavaScript designed to deal with asynchronous operations. A promise is a guarantee that a result will eventually be returned from an asynchronous task. Within this script's scope, the axios library allows a series of asynchronous HTTP GET requests to be made to the pricing API's detailed above. These requests fetch Ether's current market value in GBP, which can then be passed on to the oracle contract by calling the relevant functions. The code for such functionality can be seen in figure 5.6.

The oracle price is only updated when it has accepted a price update from each of the trusted

```
function getPrice() {
    return axios.get(url).then(response => response.data)
}
function myFunc() {
    getPrice().then(data => {
//CODE WHICH MAKES UPDATE PRICE TRANSACTION TO ORACLE HERE
})}
```

Figure 5.6: Using Axios to fetch the latest pricing data

addresses. The script, therefore, needs access to the private keys for each of the addresses. Equipped with these private keys, the lightwallet and Web3 packages are used in the same fashion as the base implementation to make transactions that call the correct functions.(section 4.2.4)

Making a GET request to the API's and then updating the on-chain values runs in an infinite loop via recursion; as soon as an update transaction has cleared, the server will make another HTTP web request and begin the cycle again. There is a slight delay whilst waiting for a transaction to be mined, which means that the oracle's pricing data is not perfectly up to date. However, it is sufficient for this project.

5.3.3 Backing Token Smart Contract

The Backing Token smart contract inherits from three OpenZeppellin contracts: ERC20 [117], ERC20Detailed [118], and Ownable. The Ownable inheritance implements the same ownership variable and helper functions as for the oracle contract. The two ERC20 parent contracts provide all of the base ERC20 functionality described in section 2.7. The design section above explains that the contract must meet the following criteria:

1. Implements an on-chain token that is ERC-20 compliant.

2. Implements a contract owner as the only entity able to mint and burn the tokens.

The source code for this contract is shown in figure 5.7. The entity that deploys the Backing To-

```
pragma solidity ^0.5.0;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.5.0/contracts/toke
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.5.0/contracts/toke
//Allows token to be owned by another address.
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.5.0/contracts/owne
contract BT is ERC20, ERC20Detailed, Ownable{
    constructor() public ERC20Detailed("BackingToken", "BT", 18){}
    function mint(address _to, uint _quantity) public onlyOwner returns(bool success){
        _mint(_to, _quantity);
        return true;
    }

    function burn(address _from, uint _quantity) public onlyOwner returns(bool success){
        _burn(_from, _quantity);
        return true;
    }
}
```

Figure 5.7: Solidity source code for Backing Token contract.

ken contract will automatically be assigned as the 'owner', granting them access to the mint and burn functions protected by the onlyOwner modifier. During the execution of the Backing Token constructor, the ERC20Detailed() function is called with some parameters; these parameters are responsible for setting the contract name, the token symbol, and the number of decimal places, respectively. The source code above does not display any of the necessary ERC20 functions; they are implemented automatically through the inheritance from the OpenZeppellin contracts. The functions implemented through this inheritance can be accessed by calling the function name with a '_' as the prefix; for example, the mint function shown above calls _mint() to create new BT tokens. The ability to call mint() and burn() is restricted to the owner of the contract, meaning that users can trace the minting of all tokens to the contract owner.

5.3.4 Stable Token Smart Contract

The implementation of this smart contract can be split into several key stages:

1. Imports, constants, and the constructor
2. Mint and Burn functions
3. Deposit and Withdraw functions
4. Deploying the contract on a local blockchain
5. Deploying the contract on the Ropsten Ethereum blockchain
6. Verifying the code for the contracts within system

5.3.4.1 Imports, constants, and the constructor

The ST smart contract implements the ST ERC-20 token and encapsulates all of the pricing logic and supply control mechanisms that stabilize the token's value. The code containing the imports for this contract is displayed in figure 5.8.

```
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.5.0/contracts/toke
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.5.0/contracts/toke
//SafeMath library, which prevents overflow when carrying out arithmetic.
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.5.0/contracts/math
//Local file import of contracts: Mock Oracle (Oracle) and Backing Token (BT).
import "./Oracle1.sol";
import "./BT.sol";
import './oraclizeAPI_0.5.sol';
```

Figure 5.8: Imports for the ST ERC-20 contract.

These imports include the previously seen OpenZeppelin contracts, oracle contract, BT contract alongside the SafeMath and oraclizeAPI libraries. The OpenZeppelin contracts are imported to abstract away the ERC-20 interfaces, whilst the oracle and BT contracts must be imported to

facilitate interfacing with them successfully. The SafeMath library is used when carrying out arithmetic and protects against memory overflows; this avoids the problems discussed in section 4.2.1

```
contract ST is ERC20, ERC20Detailed{
using SafeMath for uint;
Oracle private oracle; //The oracle which provides pricing data.
BT public bt; //Instance of BT Contract responsible for ERC-20 compliant backing token, o
uint constant private WAD = 10 ** 18; //10^18, used in pricing arithmetic.
uint private vault; //Stores the ammount of collateral held by the contract... used inste
uint private constant minCollatRatio = 12 * (10**17); //120% Minimum collateral %
uint private constant mintFee = 2*(10**16); //2% Minting fee
uint private constant burnFee = 5*(10**16); //5% Burning fee
constructor(address _oracle) public ERC20Detailed("StableToken", "ST", 18) {
oracle = Oracle(_oracle); //Update the oracle and load into interface.
bt = BT(new BT()); //Deploy a new BT contract.
}
...
}
```

Figure 5.9: ST Contract declaration and constructor

Figure 5.9 shows the contract declaration which inherits from both ERC20 and ERC20Detailed like the BT contract. Within Solidity, it is possible to specify for a library to be used on all variables of a particular type. In this code sample, the SafeMath library is applied to all uint values. Variables for the oracle and bt contract instances are defined, and additional constants for the stabilisation are also stated. The decision to make the BT contract address public was purposeful so that investors can audit the total supply. The vault variable maintains the count of Ether collateral being held by the contract; during the execution of a payable function, the Ether sent with the transaction is considered to be in the contract's balance; this means that this(address).balance can not be used when calculations want to consider the balance prior to the transaction being sent. A constant named WAD is created; this is instrumental when performing

any fixed-point arithmetic on the contract (as explained in the literature review section 2.6.2.

The stabilisation strategy section above discusses the application of fees within the system; there is a base 0.2% fee charged on a mint transaction, and 0.5% charged on any burn transaction. The fees are defined in WAD format alongside the minimum collateral ratio (minCollatRatio) of 120%, and the minimum unit price (floor price) for a BT token at 0.50 GBP. The two different fee values are used to charge a premium on mint or burn operations, these percentages will be subtracted from the number of ST minted, or the value of ETH paid upon ST burning. The funds obtained from these fees will be kept in the vault and simply add to the buffer, which will subsequently increase BT tokens' value – rewarding BT holders.

Other important constants are the mintFee, burnFee and the minimum collateral ratio. The minimum collateral ratio refers to the minimum collateral: ST tokens ratio at which withdrawal BT tokens will be enabled. The two different fee values are used to charge a premium on mint or burn operations, these percentages will be subtracted from the number of ST minted, or the value of ETH paid upon ST burning. The funds obtained from these fees will be kept in the vault and simply add to the buffer, which will subsequently increase BT tokens' value.

The constructor for the ST contract takes the oracle's address as a value; once the token has been deployed on the blockchain, the oracles address is saved in the state and can never be changed again. Within the constructor, a new instance of the BT contract is deployed, and the address is stored in the bt variable. Deploying the BT contract in the constructor ensures that the ST contract is instantiated as the owner of the contract forever.

5.3.4.2 Mint and Burn

The mint and burn functions within this system have complete control over the supply of ST tokens. The code for the mint function can be seen in figure 5.10. The function declaration itself states that it is "payable", as discussed in section 2.6.1 this allows the function to take in Ether payment when the function is called. The amount of Ether being sent can be accessed through "msg.value", and the value is reflected in the smart contracts balance immediately. To build up an initial collateral buffer, a deposit into BT must be processed before any ST tokens can be minted at the unit price of 1 GBP. The require statement will revert the transaction and provide the error message if the condition does not evaluate to be true. The next step in processing a mint operation is to calculate the GBP value of the Ether sent; this is key because the amount

```
function mint(address _to) public payable returns(bool success){
    require(bt.totalSupply() != 0, 'Requires deposit into BT tokens to initiate system.')
    uint feeMultiplier = collateralRatio() < minCollatRatio ? 3 : 1;
    uint transactionValue = calcMinusFees(ETH_TO_GBP(msg.value), mintFee.mul(feeMultiplie
    _mint(_to, transactionValue);
    vault += msg.value;
    return true;
}
```

Figure 5.10: Solidity code for mint function

of ST tokens that will be created will be equivalent to this value (minus fees).

The fees are dynamically calculated by multiplying the base fee (stated as a constant) by the the feeMultiplier. The multiplier for minting is equal to 3 when the collateral ratio falls below 120%, and 1 when the collateral ratio is above 120%. Once the GBP value of the transaction has been calculated, the fees are subtracted from the number of ST tokens to be minted. The number of ST tokens to mint is output by the calcMinusFees() function, this quantity is immediately minted with the _mint function and the Ether value of the transaction is added to the vault counter.

```
function ETH_TO_GBP(uint _eth) private  view returns (uint _GBP){
    return (oracle.ETHGBP().mul(_eth)).div(WAD);
}
```

Figure 5.11: Helper functions for handling Ether paid for mint function

The calculation to find the GBP value of the Ether sent with a transaction is: (msg.value) * (Market value of 1 ETH in GBP). This calculation is computed by the ETH_TO_GB function that is shown in figure 5.11. The current market price of Ether is read directly from the oracle contract by calling oracle.ETHGBP().

Figure 5.12 contains the burn function; this function carries out the inverse of the mint function.

```
function burn(uint _value) public returns(bool success){
    uint feeMultiplier = collateralRatio() < minCollatRatio ? 1 : 3;
    uint burnValue = calcMinusFees(GBP_TO_ETH(_value), burnFee*feeMultiplier);
    require(vaultValue() >= burnValue, 'Not enough collateral to cover burn');
    require(adjustedCollateralRatioBurn(_value)>=WAD, 'Minimum collateral ratio of 100% v
    _burn(msg.sender, _value);
    msg.sender.transfer(burnValue);
    vault -= burnValue;
    return true;
}
```

Figure 5.12: Solidity code for burn function

The input for a burn is the number of ST tokens that are being redeemed, represented in the code as _value. The current market value of these tokens in Ether is calculated using GBP_TO_ETH, and the fees are subtracted in the same fashion as in the mint function; in this case the feeMultiplier is set to 1 when the collateral ratio is below 120%, and 3 when above. There are two require statements in this sample: the first makes sure the number of tokens being withdrawn has enough collateral backing. The second makes sure that the burning does not violate the collateral ratio of 100%. As stated in the design section, disabling the withdrawal of ST from the system when it is below 100% is necessary to prevent a bank run and make sure all ST holders are treated fairly.

Two helper functions are shown in figure 5.13; vaultValue(), and adjustedCollateralRatioBurn() are used to check that the system's requirements are upheld. vaultValue() returns the current market value of the vault in GBP, whilst adjustedCollateralRatioBurn computes how the collateral ratio will be affected by the burning of some number of ST tokens. If the result of a Burn does not violate the collateral rules, the ST tokens are deleted. The correct amount of Ether collateral is returned to the function caller, and the vault value is adjusted accordingly. When calculating the adjusted collateral ratio, there is a check that ensures that the divisor of the calculation is not zero; this prevents the transaction from reverting under a division by zero error.

```
function vaultValue() public view returns(uint value){
    return (vault.mul(oracle.ETHGBP())).div(WAD);
}

function adjustedCollateralRatioBurn(uint _burning) internal view returns(uint ratio){
    if(vaultValue() == 0 || vaultValue() == _burning || totalSupply() == 0 || totalSupply
        return WAD;
    }else{
        return((vaultValue().sub(_burning)).mul(WAD)).div(totalSupply().sub(_burning));
    }
}
```

Figure 5.13: Solidity code helper functions used by burn function

5.3.4.3 Deposit, Withdraw, and Exchange

BT tokens play a major role in raising the vaults collateral by providing incentives to holders of the tokens. The three functions which enable the creation and destruction of BT tokens are "deposit", "withdraw", and "exchange". Deposit is implemented as a payable function, which allows it to accept an Ether payment and reward the depositor with BT tokens.

```
function deposit(address _to) public payable returns(bool success){
    uint transactionValue = ETH_TO_BT(msg.value);
    bt.mint(_to, transactionValue);
    vault += msg.value;
    return true;
}
```

Figure 5.14: Solidity code deposit function

The deposit function in figure 5.14 utilizes the helper function ETH_TO_BT to calculate the correct amount of BT tokens to reward the depositor. The number of tokens to award is obtained

from the calculation: (market value of Ether deposited in GBP) / (unit price of BT token in GBP). The process of calculating the unit price for BT is shown in figure 5.15. The withdrawal

```
function unitPriceBT() public view returns(uint price){
    uint unitPrice = 0;
    if(bt.totalSupply() == 0 || buffer() ==0 ){
        unitPrice = WAD;
    } else if(buffer() > 0){
        unitPrice = (uint(buffer()).mul(WAD)).div(bt.totalSupply());
    } else {
        //Buffer must be negative if execution ends up here....
        unitPrice = collateralRatio();
    }
    return unitPrice >= floorPriceBT ? unitPrice : floorPriceBT;
}
```

Figure 5.15: Solidity code for calculating the unit price of BT tokens

function (figure 5.16) has several parallels with the burn function. A helper function determines the market value of the BT tokens, there are some requirement statements to ensure the collateral ratio rules are not violated, and the BT tokens are destroyed followed by a payment of Ether to the address calling the function. Similarly to handling deposits, the BT_TO_ETH uses the unit price of BT to determine how much Ether to reward for the destroyed tokens.

Computing the unit price for BT is not always as simple as dividing the buffer value by the total supply of BT tokens. There are three cases when the calculation must be changed: buffer being negative, buffer being zero, or no BT supply. When there is no supply of BT, or the buffer is zero, the unit price is fixed to 1 GBP. When the buffer is negative the price is set to the collateral ratio multiplied by 1 GBP. If the resulting unit price of any of the previous calculations is less than 0.50 GBP then the price will be rounded up to the floor price of 0.50 GBP.

The exchange function provides a pathway to convert ST tokens into BT tokens. The quantity of ST tokens being converted is passed into the function as _st, of which the BT value is

```
function withdraw(uint _bt) public returns(bool success){
    uint withdrawEth = BT_TO_ETH(_bt);
    require(buffer() > 0, 'No Collateral to cover withdrawal');
    require(adjustedCollateralRatioWithdraw(ETH_TO_GBP(withdrawEth)) > minCollatRatio ||
    bt.burn(msg.sender, _bt);
    msg.sender.transfer(withdrawEth);
    vault -= withdrawEth;
    return true;
}
```

Figure 5.16: Solidity code for withdraw function

calculated through dividing the amount by the BT unit price and applying the relevant fees. When the collateral ratio falls below 120%, there is no burn fee charged on the exchange, and when the ratio is above 120%, the usual burn fee of 1.5% applies. The code (shown in figure 5.17 demonstrates how a ternary statement is used to calculate the correct quantity of BT to mint depending on the current collateral ratio.

```
function exchange(uint _st) public returns (bool success){
    uint btValue = (_st.mul(WAD)).div(unitPriceBT());
    uint afterFees = collateralRatio() < minCollatRatio ? btValue : calcMinusFees(btValue
    _burn(msg.sender,_st); //Burn the ST tokens
    bt.mint(msg.sender, afterFees);
    return true;
}
```

Figure 5.17: Solidity code for exchanging ST tokens into BT tokens

5.3.4.4 Deploying on local blockchain

This implementation consists of multiple smart contracts and must be deployed in the correct order to function as intended. The local VM supplied by Remix explained in 4.2.2 can be used to deploy this system and weed out various issues. However, it's a slightly different process. The oracle is deployed in the same fashion as the SimpleStable contract during the base implementation. Following its deployment, a mock value for ETH-GBP can be sent to the local oracle for testing. The oracle's address is then inputted into the ST contract's deployment arguments (figure 5.18).Additionally, the transaction to deploy ST requires more gas than the default 3000000 WEI limit; 6000000 is appropriate. This increase in gas usage is in part due to the number of contracts imported, but mostly because there must be enough gas to pay for the sub-contract BT, which is deployed by ST during constructor execution. Deploying the ST contract calls

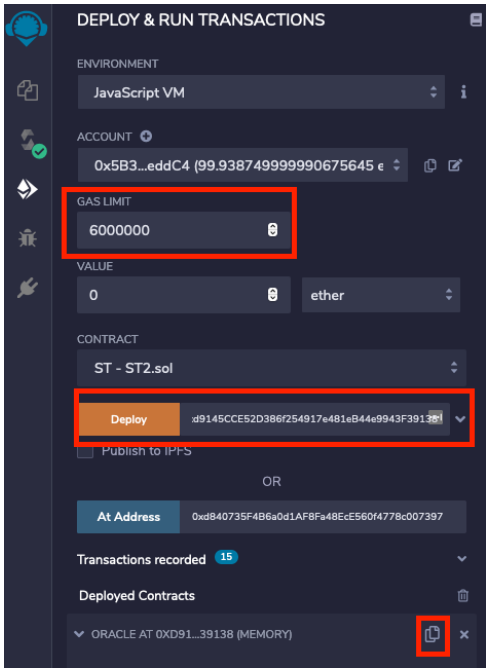


Figure 5.18: Configuring the deployment variables for the ST contract within Remix.

the constructor, and as described previously, the constructor deploys a new instance of the BT contract. Remix does not automatically add the interface of sub-contracts to the "deployed contracts" section of the user interface; however, it is possible to load the contract from its address in memory. The "bt" variable on the ST contract is public and holds the address of the BT contract being used; copy pasting this address into the "At Address" field in the figure above

will allow access to the interface in Remix.

5.3.4.5 Deploying on the Ropsten blockchain

Deploying this implementation onto the Ropsten blockchain involves following the same procedure from section 4.2.3 regarding the sourcing of some Ether and then using MetaMask with Remix. However, there are multiple contracts to be deployed. The oracle contract is the only contract with an external owner that can change the on-chain code content after deployment. As a result, the same owner account used in the base implementation can be used here.

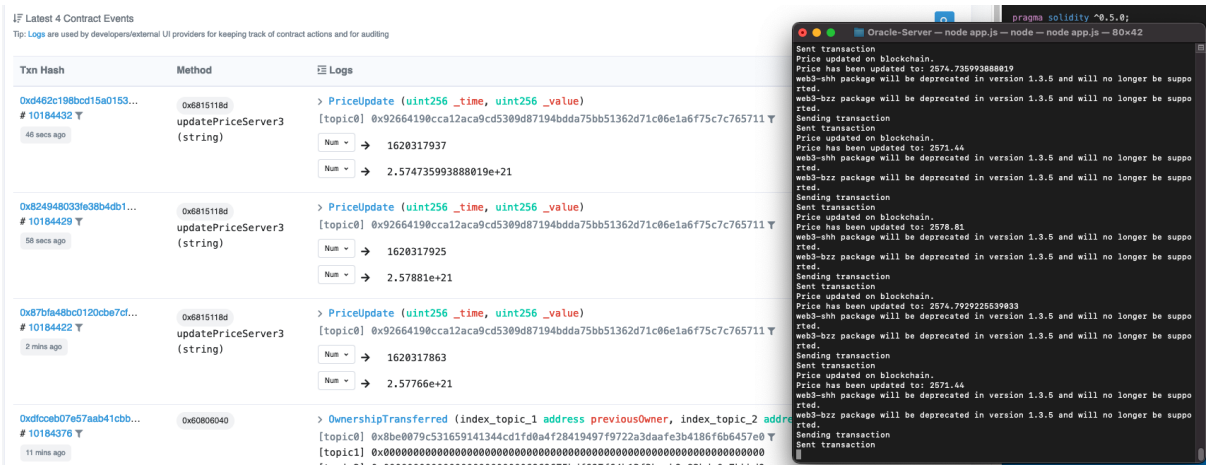


Figure 5.19: Oracle server updating the oracle's pricing data on the blockchain

Following the oracles deployment, the ST contract was also deployed using Remix and MetaMask. The ST contract automatically creates the BT contract on Ropsten during the execution of its constructor. Once these transactions have been mined on the network, the final piece of the system which needs to be running for everything to function is the oracle updating server. This is acheived by running the server (app.js file) using NodeJS. The server will then constantly update the ETHGBP value on the blockchain, which is used by the ST contract for pricing calculations. Figure 5.19 demonstrates the server running and the price updates being added to the blockchain.

5.3.4.6 Verifying the contracts on Etherscan

As described in section 3.4.5, it is best practice to verify the source code of active contracts with a platform like Etherscan. Unfortunately, EtherScan cannot verify the deployed contracts using the exact same code used during the deployment from Remix. This is because when Remix deploys the contract, it will flatten all of the imports (in this case, BT, Oracle and all of the OpenZeppelin contracts for ST) into a single file. When the code containing import statements is pasted onto Etherscan, they cannot reconcile the code with all of the on-chain content, and the verification fails. To reproduce the same flattened Solidity source code, the truffle package[119] for NodeJS had to be installed. Following this, the OpenZeppelin contracts were downloaded locally; it was crucial that the versions downloaded matched precisely with the code deployed to the blockchain. The ST contract was also downloaded, and the import statements were amended such that they referenced the local contracts (shown in figure 5.20 for the ST imports).

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20Detailed.sol";
//SafeMath library which prevents overflow when carrying out arithmetic.
import "@openzeppelin/contracts/math/SafeMath.sol";
//Local file import of contracts: Mock Oracle (Oracle) and Backing Token (BT).
import "./Oracle.sol";
import "./BT.sol";
import './oraclizeAPI_0.5.sol';
```

Figure 5.20: Refactored Solidity source code for verification of ST contract.

Once the code was refactored, the truffle-flattener [120] that is a part of the truffle suite was used to obtain a file containing the flattened source code. Before the contract can be verified there is a final piece of information that is needed; the oracle's address which was a constructor argument for the ST contract. A slight complication arises from the fact that this address must be ABI encoded. To produce this it is possible to use an online encoding utility such as HashEx [121]. Providing the flattened Solidity source code alongside the constructor argument, compiler version, and licensing information (MIT), will allow Etherscan to verify the contract's

source code successfully.

5.4 Evaluation

This implementation achieves a strong starting point for a StableCoin that is deployed on a live Ethereum blockchain. The deployed system stores Cryptocurrency assets as collateral in a transparent and verifiable smart contract where no single entity has complete control over the supply of tokens. To support the smart contracts that execute the system's core functionality, a median pricing oracle contract and oracle-updating server have been built to facilitate access to reliable real-world data on the blockchain. Furthermore, the source code for all of the contracts is verified on Etherscan to achieve maximum transparency and allow some user-testing. It is safe to say that this chapter has met the criteria for completing phase two of the project (as stated in chapter 3.1).

The functionality of the deployed system can be tested using the user interface provided by Etherscan; with the MetaMask browser extension installed and some Ether on the Ropsten blockchain, it is possible to interact with the smart contracts. Before testing the system, the price-updating server must be executing in the background. Otherwise, the pricing data could be wildly inaccurate. An example scenario that captures the system's functionality is demonstrated below; this is not extensive and will be expanded on in chapter 7 once the Dapp's user interface has been deployed (the web application).

1. Visit a Ropsten faucet to get some Ether for testing out the payable functions (`https://faucet.dimensions.network/` works as of 16042021).

2. Navigate to the Oracle contract's address on the Etherscan Ropsten block explorer (`https://ropsten.etherscan.io/address/0xe8db3272f9424e72714c06feb180a2b9157399c2#code`), inspect the code and the different values stored in the contracts state.

3. Visit the 'Events' tab of the Etherscan page and monitor as the server sends price update transactions and ensures that the price updates register and emit a 'PriceUpdate' event.

4. Navigate to the ST contracts address on the Etherscan Ropsten block explorer (`https://ropsten.etherscan.io/address/0x80f2ec1be261d74da1c3010a8ac5e9d2e898652d#code`). Inspect the verified code and explore the different values stored in the contracts state by

pressing the 'Read Contract' button.

5. Press the 'Write Contract' button; this will load up a page displaying all of the functions on the contracts interface that require transactions to call. Connect the MetaMask wallet with the application by selecting the 'Connect to Web3' button located above the list of all functions. Accept the MetaMask pop-up request to connect to the WebApp.

6. Add the tokens to the MetaMask wallet so that the balance can be viewed through MetaMask (click on "Add Token" under the Assets tab of the Ethereum account, paste the ST and BT contract addresses into the custom token address field separately).

7. Click the 'mint' dropdown from the list of functions, enter the amount of Ether to exchange for ST tokens alongside the address to receive the tokens. Press 'Write'; the transaction status can be followed by clicking the 'View your transaction' button, which will open a new tab.

8. Once the transaction has been processed into the next block, return to the previous tab and refresh the page. Go back to the 'Read Contract' section and check that the right amount of tokens have been minted.

9. Follow steps 7 and 8 with the deposit, burn and withdraw functions to make sure they all work. Note: When burning or withdrawing tokens, the quantity must be expressed in WAD format (multiplied by 10^18).

10. Send some ST and BT tokens to any Ethereum address using the MetaMask extension interface; it is then possible to check the balance of this address using the balanceOf function in the 'Read Contract' section.

The ability to see the transactions being processed in real-time on EtherScan, and to monitor the state changes on the blockchain provides ample evidence that the logic is being carried out according to the design. The screenshot of EtherScan in figure 5.21 shows the contracts transaction log resulting from a complete run-through of this scenario.

The base implementation was undoubtedly a stepping stone towards this implementation. However, the system is comprised of three different smart contracts which must interface with one another; this comes with some inherent complexities. The base implementation was purely an exercise to gain familiarity with development on an Ethereum blockchain and did not require

Figure 5.21: List of transactions with ST contract from testing the functionality.

any off-chain data such as the price of Ether. To transform the work completed in chapter 4 into a working product, a considerable amount of effort would need to be spent researching the laws, setting up bank accounts, and handling card payments through a payment gateway such as Stripe. However, this Crypto-collateralised implementation is a much more complete system and could easily be deployed on the Ethereum mainnet and become a financial product of real-life value. Of course, the logic behind the system remains an experiment of game theory, but the on-chain logic will not lie. It will explicitly perform the system's functions according to the design.

The components within this system have been implemented with the time constraints of this project in mind. There are several areas of development that could be improved upon in future iterations of the product. The on-chain logic for the oracle is solid. However, three different price sources are not that extensive; increasing the number of responses required to five or seven would be a good improvement. Furthermore, the current oracle system would fail if any of the three selected API's were to stop answering; this could be addressed by having seven price API's and updating the price when responses have been received by any five of the group. If two of the API's crash, the system will still function as intended.

During the development of the price-updating server, there were complications from the handling of large integers; initially, the price updates were being sent into the Oracle in an integer format. This causes problems because JavaScript can only safely represent integers up to the value of 2^53 – 1; thus, the numbers being sent into the oracle were not accurate representations of the price. The oracle was adapted to receive the price updates in String format to solve this issue.

The ST contract that has been implemented in this chapter deploys a set of mechanisms intending to stabilise the market value of the ST token. Whilst each of these mechanisms is built upon sound economic principles such as QTM; there are no guarantees on how the system would perform in a real-world scenario. In future iterations of the project, it would be beneficial to employ financial experts or people who are familiar with the banking industry. A tricky part

of the design process was selecting a method for pricing BT tokens when the buffer is negative; it was challenging to choose an appropriate valuation method that did not penalise existing investors through severe dilution of their funds. The selected method means that the price of BT becomes more attractive as the need for investors increases, and the 0.50 GBP floor price prevents the value of BT from becoming overly diluted.

Chapter 6

Implementing a demonstrative frontend

6.1 Introduction

The final phase of the project involves the implementation of a web application that provides a frontend for using the system. The front end will display an overview of the system and complete the project by facilitating user interactions with the stablecoin implementation completed in chapter 5. This section will encompass the design and implementation of the frontend, and explain how it can be applied to test the system's functionality.

6.2 Design

The front end must demonstrate the functionality of the stablecoin system and provide evidence that the various mechanisms outlined in the stabilisation strategy are applied correctly. The proposed design for this frontend is shown in figure 6.1. This design involves a single web application that listens out for different events emitted by the three contracts that comprise the system. When an event has been detected, the frontend will re-render the latest information about the system in a table . The data of interest is the following:

- Latest oracle pricing information
- Supply details of the ST and BT tokens
- Collateralisation data: the value of the vault, current buffer, and collateral ratio
- Unit price for BT tokens

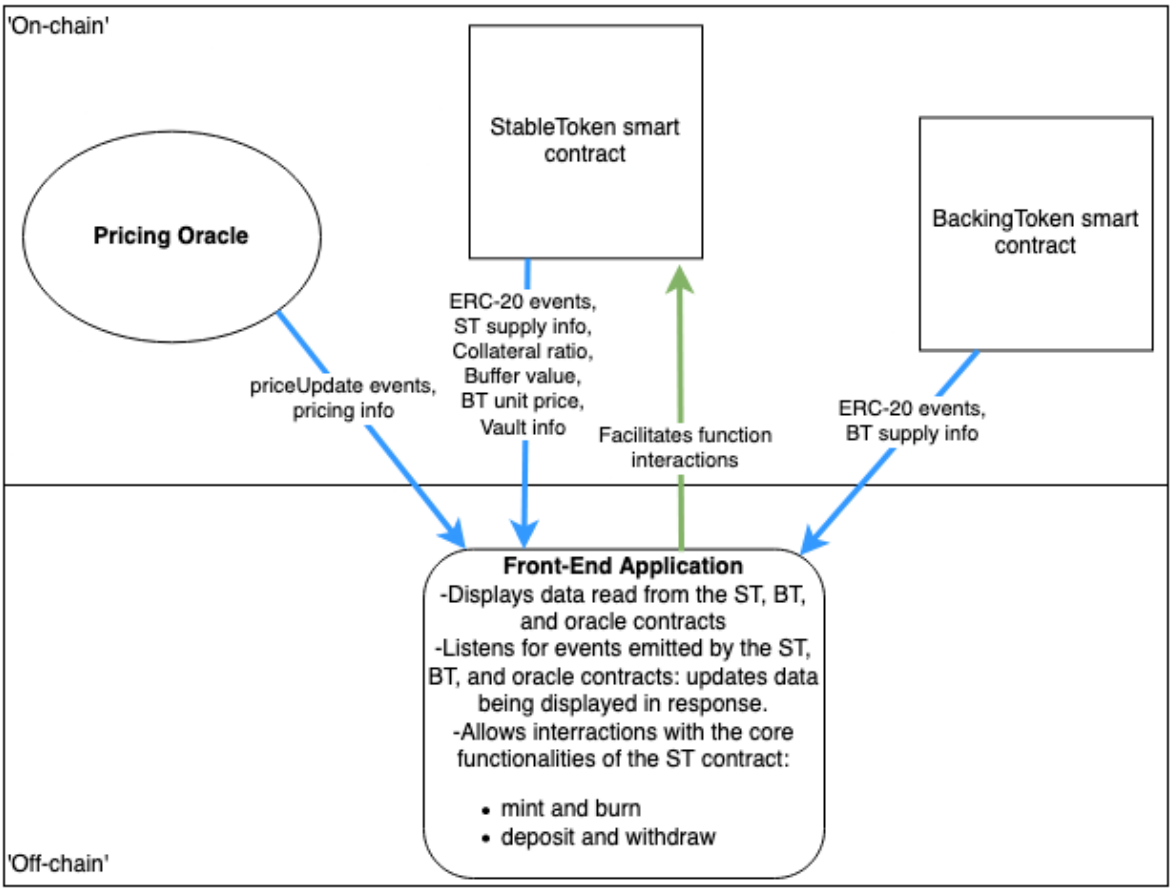


Figure 6.1: Proposed design for frontend application

Aside from rendering the above data directly from the blockchain, the frontend will also need to display the different functionalities of the system that should be enabled. For example, if the collateral ratio is below 120%, the frontend will notify users that the withdrawal of BT tokens is disabled; this provides an avenue for users to test that the displayed behaviour matches the design outlined in section 5.2.1.

The frontend will also need to display forms that allow interactions to take place with the smart contracts functions. The forms will interface with the MetaMask browser extension to execute transactions. Once a transaction has been made using MetaMask, the application will detail the transaction information and wait for it to be confirmed; upon confirmation, the tables displaying the system information will be re-rendered. The React library for NodeJS will be employed to build a responsive web application capable of delivering the necessary functionality outlined above.

6.3 Implementation

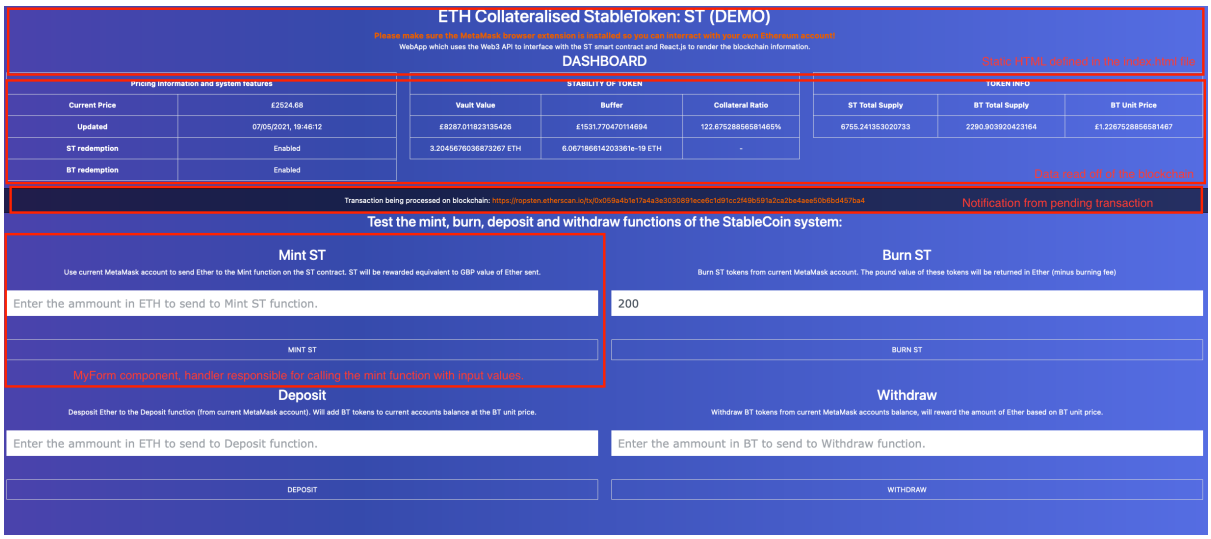


Figure 6.2: Completed frontend for this implementation

The development stack for implementing this web app is different to that within the base implementation; a client-side rendering library has been used so that the content displayed can be constantly updated with the latest information without needing to refresh the page. The Web3 library for NodeJS is employed to create subscriptions to the on-chain events of interest and propose transactions to the users MetaMask wallet in a similar fashion to the base implementation. The main body of this report is focused on the on-chain stability mechanisms and the off-chain infrastructure needed to uphold the 1 GBP peg. The completed frontend can be seen in figure 6.2, and the complete implementation documentation for the frontend is located in the appendix section A

6.4 Evaluation

The design and creation of a React application, which displays critical information about the system's on-chain components, has been completed within this section. The finalisation of this piece of software marks the end of the three-phase development approach to the overarching problem. When any of the relevant on-chain data is changed, the application will be notified. It will then leverage the client-side power of the React library to render the latest data in a table

layout for readability. Within the blockchain development community, such a web application would be labelled as a Dapp (see section 2.6).

As stated in the design above, the terminal purpose of this application is to facilitate a demonstration of the implementation completed in chapter 5. This purpose has been achieved by displaying crucial information in a 'dashboard' and facilitating interactions with the on-chain smart contracts using the Web3 API and an Ethereum account connected to the MetaMask browser extension.

Chapter 7

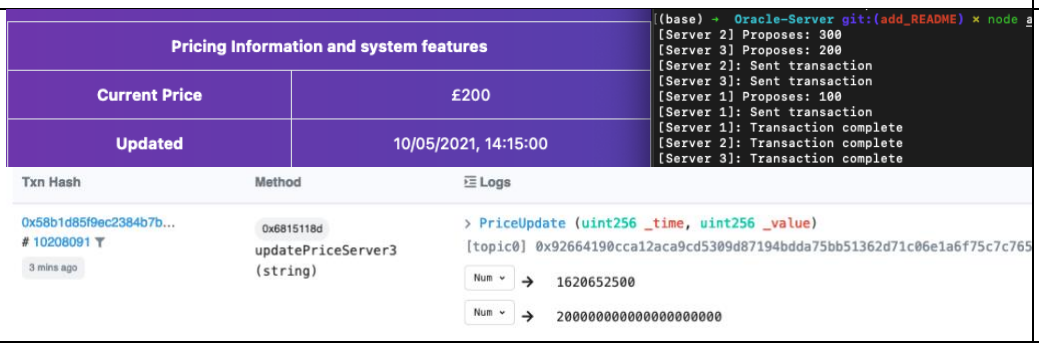
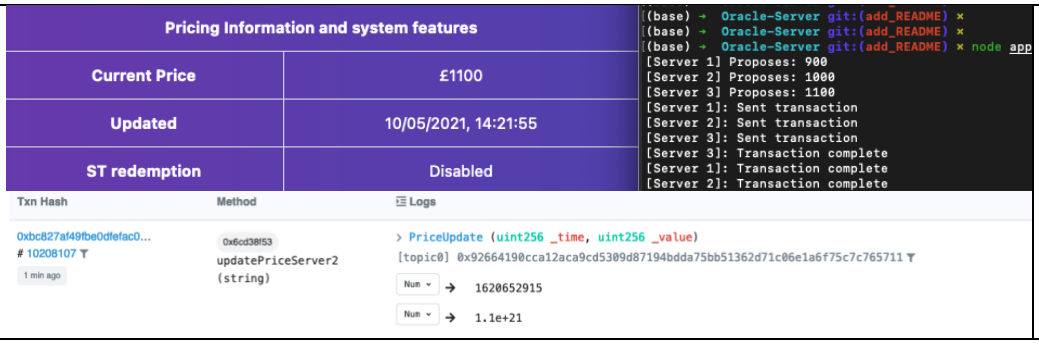
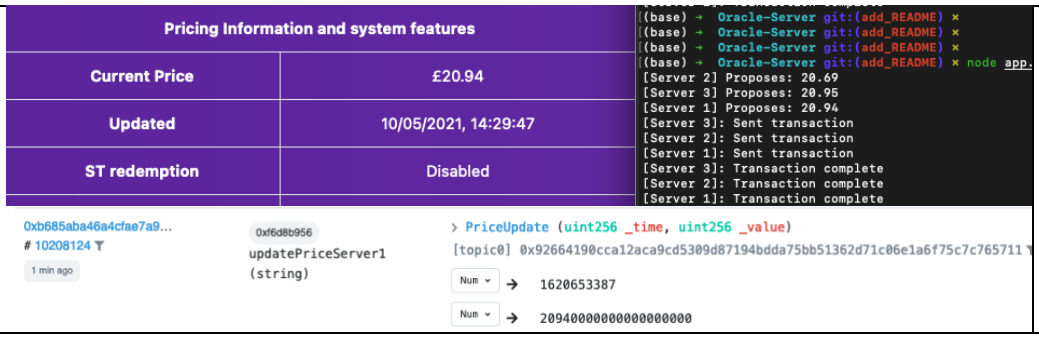
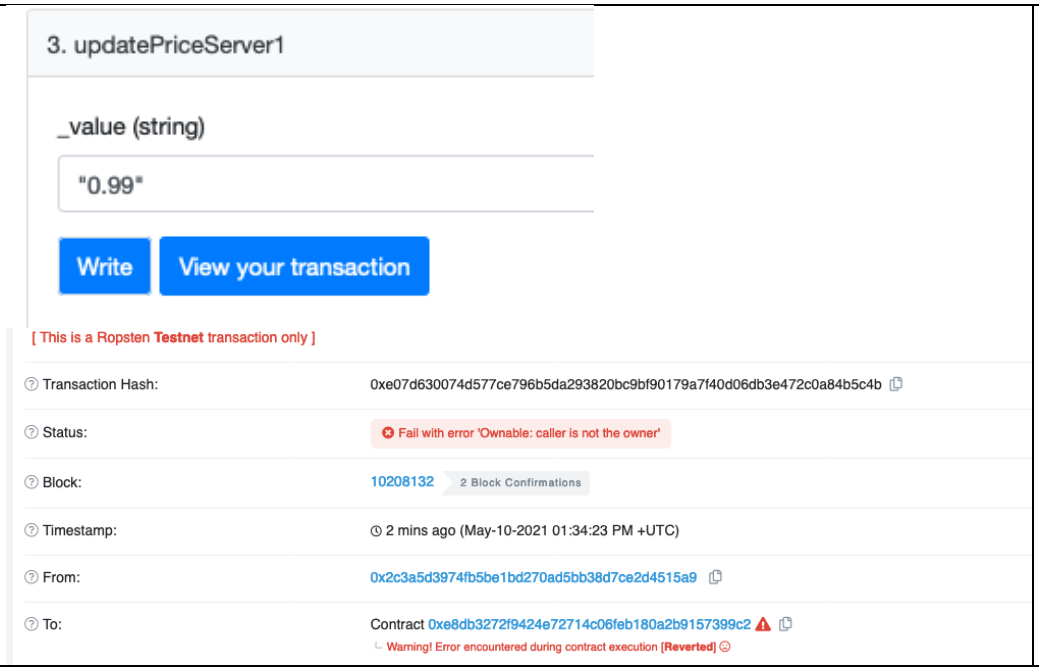
Testing

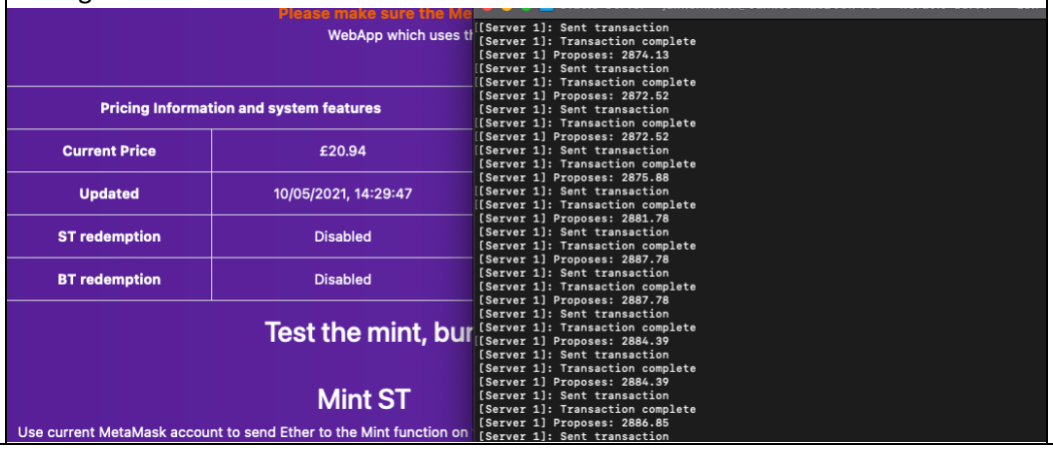
7.1 Introduction

Testing the different components of the system is an essential part of producing any piece of software. In this case, the testing is significant because the project proposes a financial product where a single incorrect line of code can result in a considerable expense for investors. This section will cover various test cases that have been performed on the completed system from chapters 5 and 6, focusing on the oracle and ST contracts in particular. There is no specific testing for the front end because it serves the essential purpose of facilitating these test cases; its correctness can be observed while testing the other components. The results from the testing will be used to evaluate the project's successes at the end of this section.

7.2 Testing the Oracle system

The following test cases test the functionality of the oracle system, ensuring that the price update permissions are enforced and that the median price is selected correctly. S1, S2, and S3 refer to the three servers trusted with updating the price information (sending transactions signed with the trusted private keys)).

Event: 1. S1 proposes £100, S2 Proposes £300, S3 Proposes £200		
Expected Behaviour	Exhibited behaviour	Passed
Oracle should accept £200 as the current price and reflect this on the ETHGBP variable. (will be 2 * 10^18 because in WAD format).		Yes
Event: S1 Proposes £900, S2 Proposes £1000, S3 Proposes £1100		
Oracle should accept £1000 as the current price and reflect this on the ETHGBP variable. (will be 1000 * 10^18 because in WAD format).		Yes
Event: S1 Proposes £20.94, S2 proposes £20.69, S3 proposes £20.95		
Oracle should accept £20.94 as the current price and reflect this on the ETHGBP variable. (will be 20.94 * 10^18 because in WAD format).		Yes
Event: Untrusted entity, User A attempts to send a pricing update of £0.99 to the oracle contract.		
Transaction should fail as transaction not signed with private key of a trusted address.		Yes

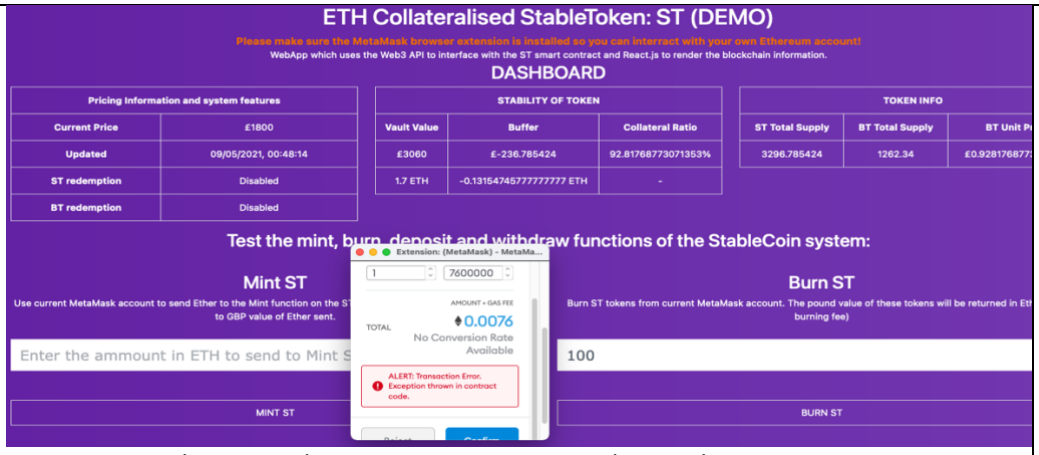
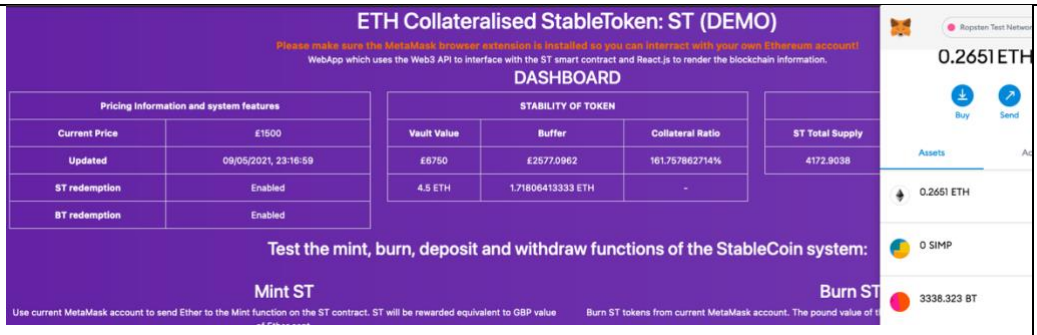
Event: Same server sends multiple pricing updates (i.e. a scenario when only one server remains up and the other two have crashed.)		
Oracle doesn't adopt any value; requires responses from each server before updating.	Single server sent 7 price update transactions; the value presented by oracle didn't change. 	Yes

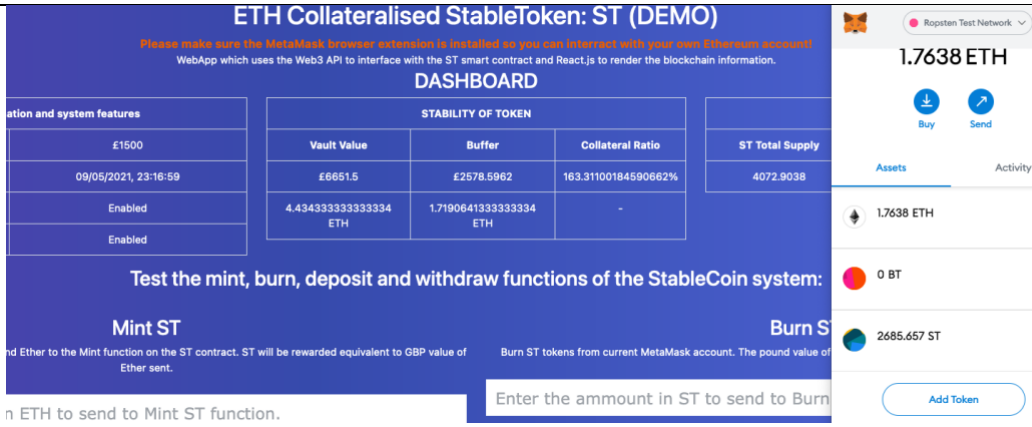
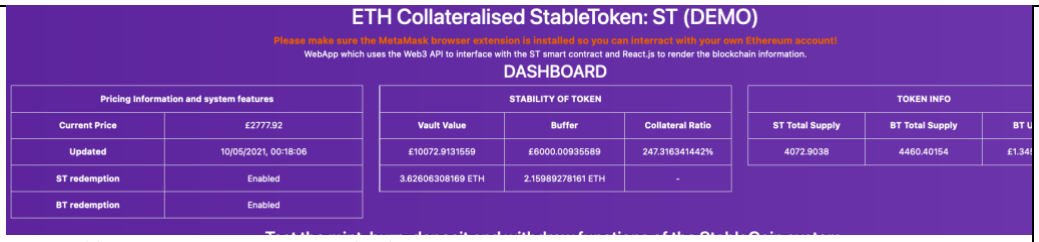
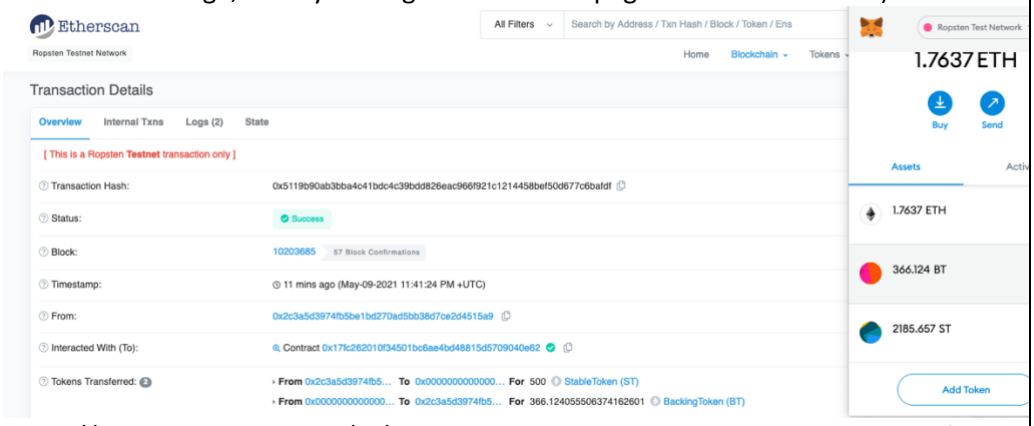
The events detailed above were selected because they test all of the different aspects of the design specified for the oracle (sections 3.4.6 and 5.2.2). The ability to correctly select the median value from a set of proposed values is evidenced through three events; one in which the correct value comes from each server. Furthermore, there is evidence that the contract can select a median value even if the difference is as little as 0.01 GBP. The final two events prove that the ability to update the price is only granted to the trusted servers and that the contract will await a proposal from all three servers before selecting a median value. Where possible, the EtherScan page displaying the price update event being transmitted is present, and the complete ledger of all these tests being carried out can be viewed on the EtherScan block explorer on the following URL: `https://ropsten.etherscan.io/address/0xE8DB3272f9424e72714c06Feb180A2b9157399c2`.

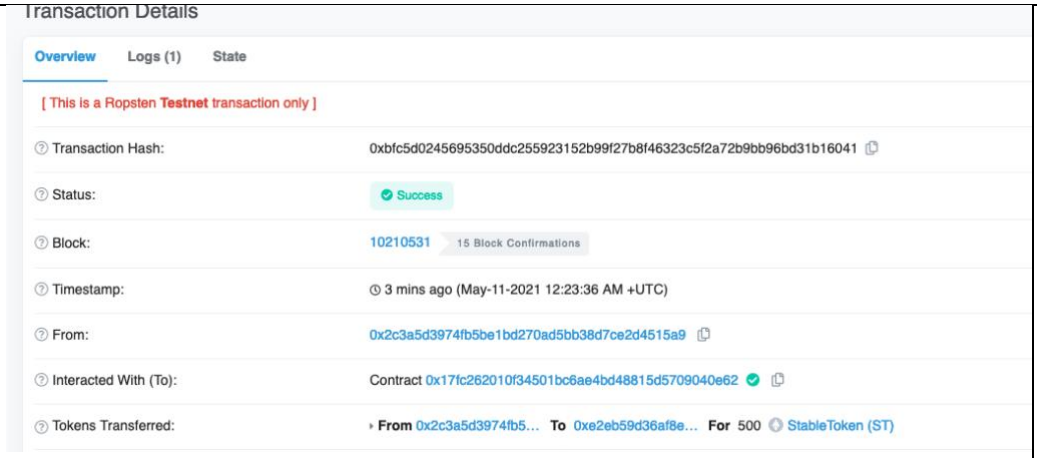
7.3 Testing the ST contract

Below is a single scenario that encompasses several 'events'. These events were tested consecutively and provide a reasonable path of activities that allow all aspects of the ST token to be tested. Following each event, the witnessed behaviour of the system is compared to the behaviour outlined in the design (section 5.2.1). The initial state of the system is essential to know so that the behaviour following the first event is understood:

1. Ether market price of £2791.24
2. Initial vault of 1.5 Ether (4186.86 GBP)
3. ST Token total supply of 1387.24628
4. BT Token total supply of 2791.24

Event: User A mints 1 ETH worth of ST tokens.		
Expected Behaviour	Exhibited Behaviour (screenshot and link to Etherscan transaction if possible)	Passed
1 Ether is valued at 2791.24. The 0.2% fee should be applied because the buffer is greater than 120%, meaning that 2785.65752 ST should be minted. Vault should increase to 2.5 ETH (£6978.10) Collateral ratio should adjust to ~167%.	 https://ropsten.etherscan.io/tx/0x60d3d9bca2ef9be3e44596be09ba6e91b862deeeab11cc2c3155800e6b1e1c8e	Yes
Event: Value of Ether falls to £1500, User A attempts to burn 100 ST		
Vault value will decrease to £3750, buffer will be negative, collateral ratio falls to ~89.865%. BT Unit price updates to 0.89865 … GBP, ST redemption will be disabled so burn attempt will fail.	 Attempting to burn ST tokens raises an exception shown above.	Yes
Event: User B deposits 2 Ether into BT, Ether price remains £1500		
Vault value will increase to £6750, buffer will increase to £2577.0962. B will be rewarded with 3338.32304 BT tokens. Collateral ratio increases to ~161.8%	 https://ropsten.etherscan.io/tx/0xb92f159c8b48e15d0cbf359b208ad2f04b88db4e2e4b13ff28c44f0b84e8c66b	Yes
Event: User A attempts to burn 100 ST tokens again.		

1.5% burn fee means that A will receive 0.06566666666 ETH. Vault will decrease to 4.43433... ETH (£6651.5). ST total supply reduced by 100, buffer increases by 0.001 ETH	 https://ropsten.etherscan.io/tx/0xb881bcd31d9dc85f9f2289aed615af44ce349f5fb7559eaf1e11efb0affb113a	Yes
Event: Price of Ether rebounds to £2777.92		
Vault of 4.43433… ETH now worth ~£12318.2, collateral ratio increases to ~302.44%		Yes
Event: User B withdraws half of their BT Tokens (1669.1615 BT)		
At a unit price of 1.3451724698035463, User B should receive £2245.31009746 worth of Ether (0.80827025164 ETH); vault will decrease to ~3.62 ETH. BT total supply will reduce to ~4460.40154.	 https://ropsten.etherscan.io/tx/0x96db38b23c70de88184bd0851e1d9c177a454a356cacf7904a2087869ed6a32b	Yes
Event: User A Exchanges 500 ST into BT tokens		
Vault value remains the same. 1.5% burn fee applies because above 120% collateral: A Receives £492.5 worth of BT tokens (366.124) .	(No user interface on web-app for the exchange function, Proof can be seen in User A's balance change, and by visiting the etherscan page for the transaction)  https://ropsten.etherscan.io/tx/0x5119b90ab3bba4c41bdc4c39bdd826eac966f921c1214458bef50d677c6bafdf	Yes
Event: User A transfers 500 ST to User B		

User A’s balance of ST decreases by 500, User B’s increases by 500. Total supply and vault values remain constant.	 https://ropsten.etherscan.io/tx/0xbfc5d0245695350ddc255923152b99f27b8f46323c5f2a72b9bb96bd31b16041	Yes

The single scenario above is sufficient to test the ST contracts functionality; the following features are tested:

- Ability to call the mint, burn, deposit, withdraw, and exchange functions.
- Dynamic mint, burn, and exchange fees based on the current collateral ratio – detailed in section 5.2.1.
- Reading of ETH:GBP pricing information from the oracle contract.
- The freezing of withdrawals and burning according to the current collateral ratio.
- Ability to send the Cryptocurrency assets (tokens) to other Ethereum addresses.

During these test scenarios, some minor errors in code were detected and corrected; this is a good sign that the testing regime achieved its goals. In particular, the precise comparison between the expected and witnessed numerical values was critical. For example, one error was that during a withdrawal, the Ether value was miscalculated and the entity making the withdrawal was paid more Ether than intended. The source of this miscalculation was the incorrect ordering of the function's logic; the BT tokens were being burnt before calculating the Ether value of the withdrawal. Thus, when the unit price of BT was being calculated, it was returning a higher value (as there were fewer BT tokens to divide the buffer by).

The testing carried out here focuses on ensuring the system's design specified in section 5.2.1 is implemented correctly. It was impossible to put together an unbiased test for the tokens stability strategy as a whole because it is impossible to reasonably predict how holders of the tokens would act in a free market. The mechanisms that have been implemented are grounded on strong economic principles, and within the scope of this project, the assumption is made that the incentives provided are strong enough to manipulate the stablecoins supply successfully and subsequently stabilise the price.

7.4 Evaluation of results

Following the correction of minor issues, all mechanisms implemented by the ST contract performed flawlessly during the execution of the tests; evidence to prove this has been presented. To create some of the 'events' manual pricing information had to be sent to the pricing oracle.

However, most of the tests were carried out with actual market prices. An EtherScan transaction link has been provided for each of the tests wherever possible, alongside screenshots of the front end's relevant dashboard sections. Whilst it was not the primary objective of testing, the scenario above also demonstrates that the frontend is sufficient for demonstrative purposes and provides the ability to interact with and evaluate the completed implementation.

The oracle contract also successfully passed all of the tests documented above. One issue encountered during testing was that the price is only updated when the price update transactions sent from the servers are mined; the timing for this can be inconsistent depending on the current average gas fee. Whilst the correct median value would be selected, the updates were more irregular than desirable for tracking the price accurately. More regular price updates could be achieved by sending the transactions with a much greater gas price, but this comes at the cost of long-term sustainability and viability of the implementation, especially when considering the high value of Ether.

Chapter 8

Project Evaluation

8.1 Introduction

This chapter reflects on what has been achieved over the course of the project, and evaluates its successes in regards to the objectives and aims stated at the very beginning. This chapter also details considerations for how the system could be improved on in the future. A comprehensive and coherent evaluation of the project can be obtained through the combination of implementation section specific evaluations, the evaluations of results, and the evaluation of objectives below.

8.2 Overview

Several critical pieces of software have been built on the quest to successfully solve this project's problem of implementing a Cryptocurrency-backed stablecoin on a blockchain. As stated in the problem analysis – section 3.1, the approach to this problem was split into three phases. This section will document all of the software created within these phases.

During phase one (Chapter 4), an ERC-20 contract, SimpleStable, that could represent the British pound was successfully developed and deployed onto the Ethereum Ropsten testnet. The source code was verified with EtherScan following successful deployment, and a server-side frontend was written using NodeJS, which demonstrates the functionality of this token with the Web3 JavaScript library.

Phase two (Chapter 5 involved developing several components, both on-chain and off-chain.

The first piece of software created for this phase was the pricing oracle, which was deployed on the blockchain. The pricing oracle, in this case, is a smart contract that allows three trusted addresses to propose price updates; once a proposal has been received from each address, the median value is accepted and publicised by the contract. Ideally, there would be three different servers sending price updates into the contract from unique price sources. However, this project utilises a NodeJS script that simulates the existence of multiple price updating servers. The script has access to the three trusted private keys and uses these to make valid price update transactions. Next, the BT and ST smart contracts were created and subsequently deployed onto the Ropsten testnet; the BT smart contract is essentially a base ERC-20 contract implementing the BT token. The ST contract implements the ST token and is where all of the stabilisation logic lives. This ST token is the final on-chain implementation of the project; it attempts to peg to the value of 1 GBP and is minted in return for the deposit of Ether into the smart contract.

Phase three encompassed the final piece of software development for the project - a frontend crafted using ReactJS. The frontend is a responsive web application that displays critical information regarding the completed on-chain system: pricing information from the oracle, supply information of the two tokens, strategy specific values such as the unit price for BT tokens, alongside other vital data. The frontend also interfaces with users MetaMask wallets to facilitate user testing of the core contract functionality; minting, burning, depositing, and withdrawing.

8.3 Evaluation against project objectives

Chapter 7 measures and evaluates how the final implementation fulfils the design crafted throughout the project (chapters 3.1, 5, and 6). However, the wider project aims and objectives stated in section 1.2.2 must be considered when discussing the projects overall successes; the following table details each of the project's objectives and whether they have been achieved or not.

Objective	Description	Fulfilled?
1	Examine protocols that allow blockchains to function at great depth.	Y
2	Perform a literature survey on existing stablecoins	Y
3	Research the feedback mechanisms employed by Dai and other stablecoins, which stabilize the Cryptocurrency; understand the economic principles which are applied.	Y
4	Select a blockchain on which to develop a stablecoin on.	Y
5	Learn how to write and deploy smart contracts on a blockchain.	Y
6	Implement a stablecoin token on the selected network.	Y
7	Develop a smart contract that will mint new tokens in response to the collateral being deposited and also burn tokens in response to the collateral being withdrawn.	Y
8	Design and implement a stabilization strategy using smart contracts.	Y
9	Test and evaluate the functionality and viability of the smart contract system	Y

Table 8.1: Evaluation against project objectives

Objective one has been accomplished through the literature review (chapter 2). A variety of research published by academics and industry leaders alike have been reviewed and studied to a great extent to gain a proper understanding of blockchain technology. In particular, this projects research on blockchains at large was focused on the different components that piece together to form a complete blockchain network. The protocol at the centre of all viable blockchains is known as a 'consensus' protocol; research conducted within the literature review explores the importance of consensus and how it may be implemented.

A thorough survey was conducted into the role that stablecoins play in the current Cryptocurrency market and the different implementations; this meets the second objective. There is a magnitude of distinct offerings, each with a different stability strategy. However, studying the whitepapers of existing stablecoins and academic papers made it possible to group the implementations into three core categories: asset-backed, algorithmic, and Cryptocurrency-backed. A fundamental discovery was that all Cryptocurrency-backed and algorithmic stablecoins require reliable pricing data to achieve stability; obtaining this data robustly and dependably presents a real challenge for decentralised systems. Different ways of constructing a data feed have been researched and compared; this information was valuable when selecting an oracle design for the project's implementation.

The completion of the literature review also achieves objective three; it reveals how all stablecoins must enact some form of supply control to maintain a stable price regardless of their type. Regarding Cryptocurrency-backed coins such as Dai or algorithmic coins such as ESD, this supply control is detailed within their whitepapers and implemented through a set of mechanisms enforced by smart contracts. The fundamental economic principle at the heart of these implementations is the Quantity Theory of Money. Understanding this was crucial when formulating the stablecoin design in chapter 5.

The selection of Ethereum as the blockchain for development is discussed in chapter 3.1, and meets objective four; Ethereum stood out as the platform of choice for all of the top stablecoins during the literature survey. Furthermore, it has a rich ecosystem of libraries and learning resources for the development of smart contracts. Of course, most of the work conducted in this project revolves around creating smart contracts on a blockchain. It would not have been possible to deliver the final implementation under the given time constraints without using Ethereum specific dedicated tools such as Remix or open-source contract implementations from OpenZeppellin.

Learning how to write and deploy smart contracts (objective five) was fundamental for completing this project. Initially, the official Solidity documentation was beneficial for gathering the basic ideas and getting a feel for the contract-oriented language. The first phase of development for this project involved implementing the foundational on-chain logic for a stablecoin token – the ability to create, destroy, and transfer a fungible token that could represent 1 GBP hosted on the Ethereum network. Realising this implementation was a cornerstone moment of the project; the act of completing the implementation achieved objective six, and the lessons learned whilst working on it indeed provided a strong understanding of how to write, debug, and deploy smart contracts on a live blockchain. In particular, the ERC20 standard was implemented manually rather than inheriting from an existing contract library. Simply reading the documentation or inheriting the functionality from an external library is never sufficient for gaining a working understanding of any programming language. The decision to create this narrow-scoped implementation was a vital stepping stone towards the final Cryptocurrency-backed product. The deployment of the base implementation was far less punishing when errors occurred, providing a perfect environment for learning; this is because it was a single smart contract.

The previously mentioned research and practice with the Solidity programming language al-

lowed a smooth transition to complete objective seven; an ERC-20 compliant smart contract, StableToken, was used to implement the token. The collateral selected for this project was the Ether Cryptocurrency - this was the ideal option because the Ethereum blockchain natively supports it. Extending the minting and burning functionality from the base implementation to accept Ether can be achieved by adding the single keyword 'payable' to the function declaration. However, the difficulty in meeting this objective lies in ensuring that the market value of Ether is known to the smart contract. Without this data, it is impossible to mint the correct number of tokens in reward for Ether or to pay the valid value of Ether in return for burning tokens. The stated goal of this project is to peg the value of the token to 1 GBP. Therefore, the current market price for Ether in GBP is needed. An external software oracle was built to provide pricing data into the smart contract; this pricing oracle attempts to minimise the risk of data outliers by selecting a median value from a set of trusted sources – inspired by the Dai protocol.

Furthermore, the knowledge gained during the literature review into Ethereum and the pricing problem was crucial for implementing an oracle that minimises gas fees whilst performing correctly. Following the completion of the pricing oracle, it was possible to access the market value of Ether and subsequently handle Ether payments correctly. The complete design and implementation documentation for achieving this objective can be found in chapters 3.1 and 5.

Objective eight was fulfilled within section 5.2.1, and 5.3.4. Designing the stabilisation strategy was extremely challenging; a dual-coin design was selected that employs mechanisms inspired by existing stablecoins. The core intention of the design is to maintain a vault of Ether that can at least fully back the value of all ST tokens created. A buffer of Ether is raised through the sale of BT tokens, which offload the risk from ST token holders. Dynamic fees are charged on interactions to provide financial motivations for market participants to behave in a certain manner. The smart contract will disable the withdrawal or burning of ST and BT tokens when the vault goes underwater. The strategy was successfully captured by the logic of the StableToken smart contract and deployed to the Ropsten testnet. The full documentation of this implementation can be found in section 5.3.4. While developing the smart contract, it was beneficial to segment the process into smaller chunks of functionality - for example, focusing on all logic to do with BT tokens or ST tokens individually.

To meet the final objective, a demonstrative frontend was deployed (chapter 6), and a comprehensive set of test scenarios have been executed and documented in section 7. A justification

of the selected procedures, alongside a critical evaluation of the outputs, is detailed within the section. There was unforeseen difficulty in proving the effectiveness of the stability strategy; it would be impossible to predict how market participants act in a free market. The testing, however, ensures that all the mechanisms discussed in the design are implemented correctly.

As outlined in table 8.1, and discussed in the text above, this project achieves all stated objectives. These objectives certainly laid out a logical pathway to follow for solving the overarching problem. If the project were to be attempted again, it would be a good idea to add a few smaller objectives to keep the workload more manageable; for example, the design and implementation of an oracle was a considerable challenge that justifies a separate objective.

8.4 Areas for improvement and future works

The blockchain industry is infamous for its rapidly evolving nature; new technology is constantly released that shifts the ground beneath previous products feet. One such addition to the Ethereum ecosystem is the concept of a permit [122]; permits allow the ERC-20 functions approve and transferFrom to be called in a single transaction rather than two – as of writing this section, the gas prices are incredibly high due to heavy network traffic. The Ether saved by enabling this functionality could provide significant savings to users of the system. Another such development is the introduction of layer 2 scaling solutions [123] such as the Polygon network [124], which enables Ethereum systems to offload some of their processing to a sidechain with massively cheaper gas prices.

Mature projects such as Dai employ several Cryptocurrency assets as collateral; this provides the benefit of diversifying the vault. As a result, the system would be less sensitive to decreases in the value of a single asset. Implementing this functionality would be achievable using other ERC-20 tokens on the network as smart contracts on the blockchain can hold balances of these tokens trivially.

Another exciting development would be deploying the system on the Ethereum mainnet; this would allow a truthful and realistic measurement into the viability of the stability strategy. Testing the strategy itself was not possible within this project due to the lack of an active market. Furthermore, if the system were to be deployed on the mainnet, it would be a good idea to hire a third-party auditing company. Auditing firms such as Consensys Dilligence [125]

extensively analyse contracts in an attempt to remove vulnerabilities.

If the projects time constraints allowed, it would have been beneficial to add extra functionality to the demonstrative web application. Better visualisations in the form of pie charts would help display the buffer value in a useful format. The ability to communicate information relevant to the application's user would be a significant improvement. For example, the web application could allow users to view their current balance, filter past transactions, or even initiate transfers from within the application.

As discussed in 2.4.2.4, there is a multitude of different options for the pricing oracle. The oracle implemented fulfils the needs of the project and was the best option given the constraints of the Ropsten testnet and the project as a whole. However, it would be interesting to experiment with not only the oracle improvements proposed in sections 5.4 and 7.4, but also with entirely different implementations such as the ChainLink and NEST [126] oracle networks.

Chapter 9

Statement of Ethics

9.1 Introduction

This chapter formally discusses the legal, societal, ethical and professional aspects of the project. Throughout the completion of the project, all of the relevant legislature and the British Computer Society (BCS) Code of Conduct (CoC) [127] have been followed. It is vital that the work completed is a positive contribution to society and upholds the reputation of the Computer Science industry and the professionals therein. Of course, blockchain technology, in general, is a scorching topic, and stablecoins even more so. As such, considerations have been made for both the narrow scope of this project and the broader stablecoin landscape. In times as recent as March 2021, the United Kingdom (UK) government have been making plans to "expand the regulatory perimeter to cover stable tokens used as a means of payment" [128], and have announced that "Britain will focus first on regulating stablecoins rather than the broader cryptocurrency market"[129].

The University of Surrey is affiliated with the BCS, and therefore the professional standards set out in the CoC were followed during the completion of the project. The following subsections are guided by the four fundamental principles of the CoC.

9.2 Public Interest: Ethical and Social responsibilities

The Public interest standards defined in the BCS CoC state that considerations must be made for "public health, privacy, security and wellbeing of others and the environment", this is something

that was highly regarded throughout the project. Standard 1.c, "conduct your professional activities without discrimination...", is extremely important in the industry at large; however, it did not apply to this project because all of the work was completed individually, and there was no opportunity for discrimination of any kind. Finally, 1.b, "due regard for the legitimate rights of third parties" has also been respected, as demonstrated through the correct handling of copyrighted materials and intellectual property.

9.2.1 Doing no harm

This project has not been developed to cause harm to any party. However, with the base implementation, there is a risk. As discussed in section 4.3, the ability to create new tokens is possessed by the central entity that controls the "owner" account assigned to the contract on the Ethereum network. The implementation here is only theoretical and not handling Fiat payment. Still, if a malicious party were to obtain the code and deploy the system, they would possess the ability to mint tokens at will. The minting of more tokens than collateral being held would lead to a colossal deflation in the token value on any decentralised exchanges. The original investors who placed trust in the system would be financially harmed.

The final implementation does not involve a central entity that can burn tokens. However, there is a weak link in the implementation. The mock Oracle, which is being used to fetch Ether's pricing data, is fetching data from a single source and being updated using a server with ownership over the Oracle. If a malicious user manipulates the pricing data or the server fails, and the price is not updated correctly, any stabilised asset holder could be financially harmed. Fully developed projects such as DAI address this problem using a set of decentralised pricing Oracles, so it is essential to clarify that this problem does not plague the entire landscape.

9.2.2 Social Responsibility

The introduction of an Ether collateralised stablecoin that is pegged to £1 has high social merit. The Ethereum network enables a permissionless, borderless, neutral and censorship-resistant payment mechanism; extending these properties to a stable asset is invaluable and has plenty of humanitarian use cases worldwide. For instance, policies enacted in Venezuala have lead to hyperinflation and heavy economic sanctions[130], making it impractical to give aid using Fiat currency. An existing token pegged to the value of $1, USDC (USD Coin), has been used

precisely for this purpose [131]. Hundreds of millions of people worldwide live in countries such as Venezuela or Nigeria, which suffer from an unstable currency. A stable currency is essential for people who want to make plans and save towards achieving their goals. With a monetary system being crippled by hyperinflation, it becomes impossible to make any plans or have a high standard of living; people are subject to extreme stress and cannot afford the bare necessities.

The Bitcoin cryptocurrency has received a lot of criticism regarding the huge energy use for its proof of work consensus model, which produces a massive carbon footprint. The Ethereum network also currently employs a proof of work model. The protocol's security relies on the computational difficulty of finding the golden nonce as discussed in section 2.3. Whilst the high energy consumption of such a model cannot be ignored, one must question the value that can be achieved due to this energy expenditure. The societal benefits of the Ethereum blockchain are far-reaching and should not be written off due to energy use. All significant advances in technology have come at the cost of justifiable higher energy usage. To put this into perspective, does it seem rational to suggest shutting down the internet because 2% of global CO2 production is a result of supporting the network? [132]. This project, in particular, has a high value to society, as argued in the paragraph above. Therefore it is an ethically sound decision to employ blockchain technology to create this system, which would otherwise be impossible. Furthermore, the upcoming Ethereum 2.0 network will greatly reduce the overall computing power required to secure the network; this will be achieved by introducing a new Proof of Stake consensus mechanism. [133]

9.2.3 Legitimate rights of third parties: intellectual property

In the final implementation, several smart contracts owned by OpenZeppelin are used as a starting point; they are a company that provide blueprints for the development of solidity smart contracts on the Ethereum blockchain. In particular, this project uses various ERC-20 interfaces and the SafeMath library, all of which are licensed under the MIT license. The reasons for using these third-party solutions are outlined in section 5.2. The MIT license is defined at the top of the contracts imported directly from the official OpenZeppelin GitHub [134]; the license effectively states that all kinds of use are allowed as long as the copyright notice is visible.

9.3 Professional competence and integrity

Section 2 of the BCS CoC, "Professional Competence and Integrity", states that the project must strive to meet good practice in the area of specialism and follow all the relevant government legislation. Throughout the implementation chapters 4 and 5, there are multiple instances where best practices are discussed and applied to the implementations. Furthermore, the UK legislation has been considered heavily due to the quickly evolving nature of the Cryptocurrency industry.

9.3.1 Handling of sensitive data

Due to the nature of the project, any data that is handled on-chain will be replicated across millions of network nodes worldwide. Within both the base implementation and the final implementation, no private or sensitive data is handled or subsequently stored on any system. The only information that is ever revealed is the public Ethereum address of the person interacting with the smart contracts; this information is not sensitive and can only be exposed on the blockchain when the owner of said address willfully signs a transaction using their private key. The only data available to view using this public address will be the ledger of all transactions made on the Ethereum network, leading to the conclusion that the project satisfies the requirements set out in the Data Protection Act 1998 and Data Protection Act 2018[135].

9.3.2 UK Cryptoasset Law

Whilst there are many ethically sound applications of Stablecoins, there is, of course, the risk that a malicious party could use the system to launder money. To combat this risk, the UK government announced "The Money Laundering, Terrorist Financing and Transfer of Funds" regulations, 2017 [136]. Any business that facilitates the exchange of Fiat money for crypto assets must carry out due-diligence on their customers; this usually involves applying a Know Your Customer (KYC) policy. The KYC process involves obtaining personally identifiable information from the customer to verify their identity and establish that they are using the system with a legitimate and law-abiding intent. The base implementation within this project simply covers the on-chain logic and demonstrates the functions that could be called upon successful financial transactions if a payment processor was implemented. Furthermore, the project is implemented on the Ropsten test network and not the Ethereum Mainnet. If one were to implement the

ability to deposit British pounds using a payment processor, they would indeed need to register with the Financial Conduct Authority (FCA) [137] and ensure that the previously mentioned regulations are followed. The final implementation in this project involves stabilising an ERC-20 token using on-chain crypto assets (Ether), such that its value remains pegged to £1. According to recent documentation released by Her Majesty's (HM) Treasury, this kind of stabilisation technique means that the token would be classified as an "unregulated exchange token" [138]. However, the stipulation is made that they "may be subject to regulation in future". Ultimately it would be challenging for regulations to control such an asset; the logic is carried out by a decentralised set of computers worldwide, and no Fiat currency is involved. If the various fiat on-ramps fulfil their due diligence, it will be difficult for malicious parties to gather the Ether needed to create the Stablecoin.

9.4 Duty to the relevant authority and duty to the profession

Professional judgement has been exercised during this project's completion; the relevant authority, in this case, is the University of Surrey. Any potential problems on the horizon have been discussed with the University such that the scope of the project could be adjusted. The knowledge and understanding gained through implementing two different variations of a StableCoin have been compiled within this document, which will be publicly available to encourage fellow BCS members and the wider Computer Science community to explore the technology.

Chapter 10

Final Conclusion

This project has been an overall success; the three aims have been met alongside all of the objectives set out to achieve them (section 8.3). The project has discussed the existing stablecoin solutions – why they are needed, how they can be classified, the economics behind the systems, and specific implementations. It has also explored the blockchain technology they are built upon at a great depth. The final system achieved in chapter 5 has shown how a smart-contract capable blockchain such as Ethereum can be leveraged to build a Cryptocurrency-backed stablecoin that attempts to maintain a peg at 1 GBP. The completed system presents the ST token as an option for a trustless digital token representing 1 GBP, alongside BT as an investment vehicle that allows investors to take a leveraged position on Ethereum.

The completion of this project has granted exposure to many different aspects of developing on a blockchain; the literature review has shone a light on how a blockchain may be split up into several standard components and how these components fit together. The process of tackling the project's problem has allowed experience to be gained in the languages and techniques necessary to develop and deploy smart contracts on a live blockchain. An understanding has been obtained of the limitations of blockchains and why the technology is not necessarily a silver bullet for all computing problems. However, it is a fantastic option for creating trustless, permissionless, and censorship-resistant systems.

A substantial array of technologies have been mastered by completing this project, both on-chain and off-chain. In particular, a proficiency in NodeJS and React for handling the off-chain logic and Solidity alongside the entire development pipeline for smart contract development. An important discovery has been that no successful stablecoin is entirely decentralised; retrieving

reliable pricing data from the real world requires some level of centralisation. Of course, this centralisation can be minimised but not eradicated with the current solutions. Aside from the technical experience, a fundamental understanding has been gained into the economic game theory that allows existing stablecoin systems such as Tether, Dai, and USDC to be successful.

If this project were to be attempted again, a potential improvement would be to develop the system on an alternative blockchain network; whilst the Ropsten testnet does most closely emulate the mainnet, the irregular gas prices and network congestion could be frustrating at times. Other options such as Rinkeby wouldn't have required any changes to the smart contract code; the only difference would be the underlying consensus protocol. Furthermore, other oracle options such as ChainLink would have been available on alternate networks. Regarding the stability strategy, it would be a significant improvement if some form of a testing regime could be created, perhaps a mathematical test model that doesn't require an active market.

Overall this project has been extremely worthwhile and has enabled a glimpse into the emerging world of decentralised finance and potentially the future of money. The system would be a good starting point for future works to experiment with alternative fee models and stabilisation mechanisms that rely on Ether collateral and a dual-coin design.

Appendix A

Implementation documentation for the frontend

A.1 Setting up the React project

ReactJS is the library used to render the client-side HTML dynamically, whilst the Web3 library (literature review, section 2.6) facilitates the communication between the on-chain data and the web app. To get the project up and running as quickly as possible, the create-react-app command-line tool was used to bootstrap a working frontend application. This can be seen in figure A.1.

Once the project directory had been created, the placeholder code automatically generated was stripped out alongside the default CSS. ReactJS is a NodeJS library; as such npm must be used to install other dependencies such as web3 and bootstrap. Once the project had been configured, and all the dependencies were installed, the files responsible for rendering the content could be worked upon.

The index.html file within a React app contains all of the static content and a HTML div element of the class "root"; the dynamic content of the application is all rendered inside the root element using client-side JavaScript. The static content of this application is the page title and a short description of the frontend.

There are two essential JavaScript files in regards to rendering the dynamic content; index.js and App.js. index.js (figure A.2 contains the JavaScript that attaches to the root element and

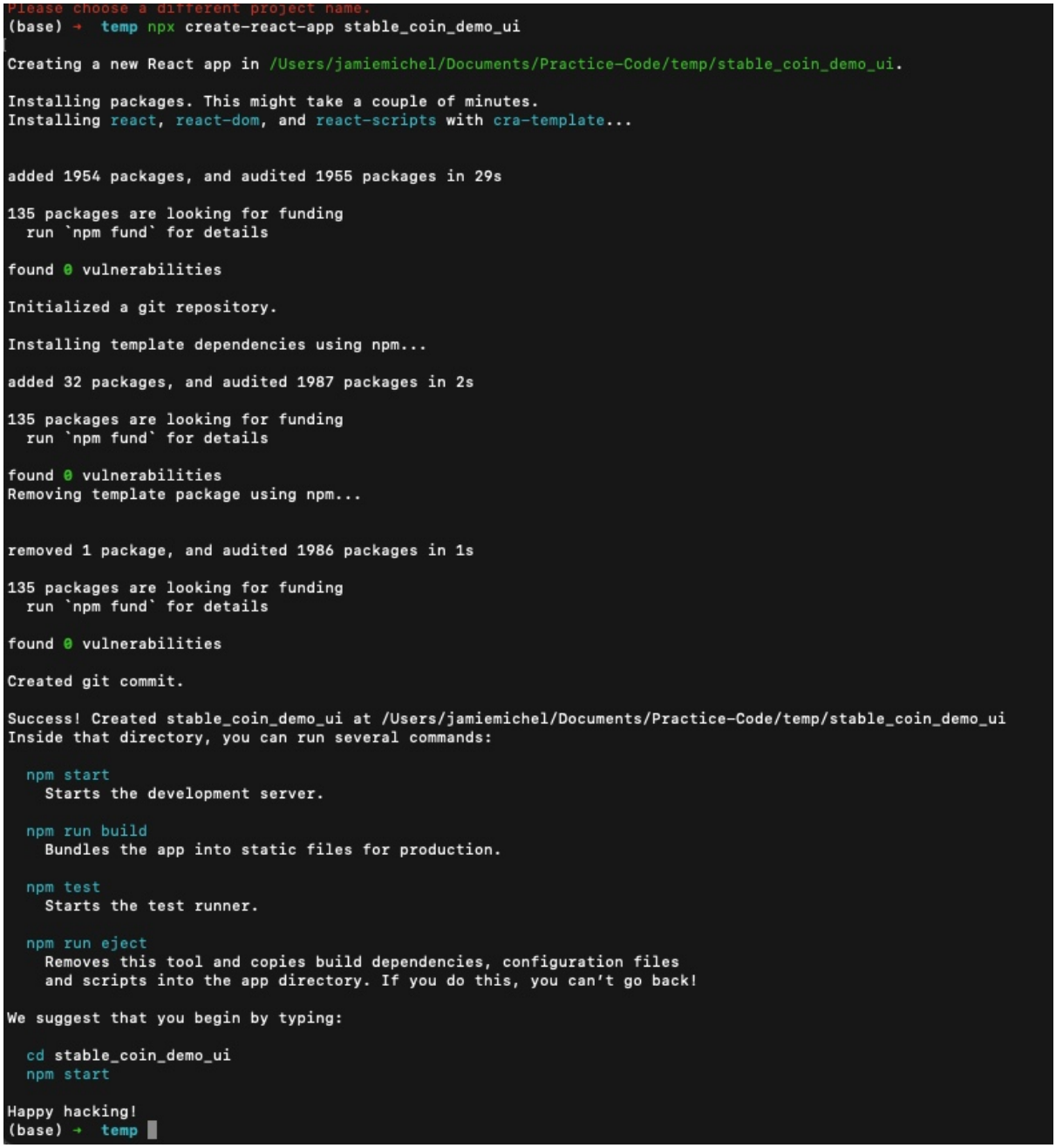


Figure A.1: Starting a React application using create-react-app

also states which React "components" to render within the root. Components are reusable bits of code that work in isolation and return HTML to be rendered client-side. Every component has a state object that is used to store its values; whenever this state is changed the component is re-rendered.

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Figure A.2: index.js file within React application.

A.2 Reading data from the blockchain and connecting with MetaMask

The App component's source code resides in the App.js file; this is where most of the app's functionality is defined. To use Web3 to interact with a smart contract, two pieces of information are needed: the contract's address and the contact's ABI (the ABI is explained in section 4.2.4). As shown in figure A.3, the information needed for each of the three contracts are defined as constants at the top of the file alongside the various imports. The code sample does not show the full JSON ABI for each contract because they are lengthy JSON objects with many different values.

The imports of Web3, React, and App.css have been explained previously, getWeb3 however, is a new piece of code. getWeb3.js (figure A.4 is a JavaScript file that contains the code which attaches the application to the MetaMask account of the app user (it also asks for permission if the Dapp connection has not been accepted previously). Previously, websites could interact with a users MetaMask wallet without approval; this was changed to prevent malicious parties from tracking peoples browsing activities via their exposed wallet address. The "_desc" constants

```
import React, { Component } from 'react'
import Web3 from 'web3'
import './App.css'
import getWeb3 from './utils/getWeb3.js';
const InfuraEndPointKey = "d0f7b76ae5724e82ab871bba3643dfd3";
const OracleABI = //ABI JSON FOR ORACLE HERE
const OracleAddress = "0x77da79f650c0ed49741912af14a1b1231fcd1009";
const ST_ABI = //ABI JSON FOR ST HERE
const ST_Address = '0x8c9cC8C438629626129dae4D1e18Df83c1c05a2A';
const BT_ABI = //ABI JSON FOR BT HERE
const BT_Address = "0x26fD98a585B14BaE4e529cf2461d20F5c499B60A";
const mint_desc = "Use current MetaMask account to send Ether to the Mint function on the
const burn_desc = "Burn ST tokens from current MetaMask account. The pound value of these
const deposit_desc = "Desposit Ether to the Deposit function (from current MetaMask accou
const withdraw_desc ="Withdraw BT tokens from current MetaMask accounts balance,  will re
const WAD = 1000000000000000000;
```

Figure A.3: Imports and constants within App.js

for each of the functions (mint, burn, deposit, and withdraw) contain a description that will be passed into the components responsible for the function testing later on.

App.js contains the code for two different components: MyForm and App. MyForm is a child of the App component and Is used to render a form for interfacing with each of the system's functions. Within React, componentWillMount() is triggered before the component is rendered client-side. The state within a React app contains all of the important variables that the app requires; each time the state is updated, any component that uses the updated value is re-rendered to reflect the change. Therefore, when the App component is first initialized, and it reads the necessary data off of the blockchain. It also subscribes to the relevant events from the smart contracts, and finally, it connects the app to the users MetaMask wallet.

The first line of code within the componentWillMount() function (Figure A.5) creates a new instance of web3 using Infura as the provider; this is similar to the base implementation. However,

```
import Web3 from 'web3';
const getWeb3 = () => new Promise((resolve) => {
  window.addEventListener('load', () => {
    let currentWeb3;
    if (window.ethereum) {
      currentWeb3 = new Web3(window.ethereum);
      try {
        // Request account access if needed
        window.ethereum.enable();
        // Acccounts now exposed
        resolve(currentWeb3);
      } catch (error) {
        // User denied account access...
        alert('Please allow access for the app to work');
      }
    } else if (window.web3) {
      window.web3 = new Web3(window.web3.currentProvider);
      // Acccounts always exposed
      resolve(currentWeb3);
    } else {
      console.log('Non-Ethereum browser detected. You should consider trying MetaMask!');
    }
  });
});
```

Figure A.4: getWeb3() function, used to connect with MetaMask Web Browser extension.

it is slightly different because the WebSockets version of the Infura API is used. It is essential to use the "wss://" URL because events can only be listened to using WebSockets. Once the web3Events instance has been created, local instances of each contract are created via web3 using the relevant ABI and address. These local instances for the contracts are passed into two functions: "loadBlockchainData" and "subscribeEvents". The keyword "this" is used to access

```
class App extends Component {
  componentWillMount() {
    const web3Events = new Web3('wss://ropsten.infura.io/ws/v3/' + InfuraEndPointKey);
    const oracleContract = new web3Events.eth.Contract(OracleABI, OracleAddress);
    const ST_Contract = new web3Events.eth.Contract(ST_ABI, ST_Address);
    const BT_Contract = new web3Events.eth.Contract(BT_ABI, BT_Address);
    this.loadBlockchainData(oracleContract,ST_Contract,BT_Contract)
    this.subscribeEvents(oracleContract,ST_Contract,BT_Contract);

    getWeb3()
    .then((result) => {
      this.setState({web3:result});// we instantiate our contract next
    })
    .then(() =>{

      this.state.web3.eth.getAccounts()
      .then((accounts) => {
        let ST_Instance = new this.state.web3.eth.Contract(ST_ABI,ST_Address);
        this.setState({account:accounts[0],ST_Instance});
      })

    }
    )
  }
...
}
```

Figure A.5: componentWillMount of the App component

the functions because they are defined within the App component.

Figure A.6 contains the code for the loadBlockchainData function; as the name suggests, this

```
  async loadBlockchainData(oracleContract,ST_Contract,BT_Contract) {
  // this.setState({ account: accounts[0] }
  const price = await oracleContract.methods.ETHGBP().call();
  const totalSupplyST = await ST_Contract.methods.totalSupply().call();
  const totalSupplyBT = await BT_Contract.methods.totalSupply().call();
  let time = await oracleContract.methods.updated().call();
  time = new Date(time*1000).toLocaleString('en-GB');
  const unitPriceBT = await ST_Contract.methods.unitPriceBT().call();
  //TODO: Get the rest of the data from the blockchain on mount.
  const collateralRatio = await ST_Contract.methods.collateralRatio().call();
  const bufferValue = await ST_Contract.methods.getBufferValue().call();
  const vaultValue = await ST_Contract.methods.getVaultValue().call();
  const vaultEth = await ST_Contract.methods.vault().call();
  const bufferEth = bufferValue/(parseInt(price)*WAD)

  this.setState({price:price, time, totalSupplyST:totalSupplyST/WAD, totalSupplyBT:totalSup
}
```

Figure A.6: Code for the loadBlockchainData function

function uses the passed in contract instances to fetch the latest relevant data from the contracts on the Ropsten blockchain. The function is declared as "async", meaning that it is asynchronous because it fetches data from an external source. Within the function body, local variables are populated for the data of interest (pricing information, token supplies, and the value of different system parameters). With the contracts instances, it is possible to access the methods using dot notation and the functions name, for example: oracleContract.methods.ETHGBP.call(). The concepts of read and write operations for the blockchain are explained in section 4.2.4. All of the operations to load the data are simply reading from the blockchain, meaning no transactions need to be made, and the data is obtained using the .call() method rather than .send(). The timestamp for when the ETHGBP price was updated on the oracle is in Unix timestamp format; this is the total number of elapsed seconds since the Unix epoch on January 1st 1970. To present this in a user-readable format, the time is converted into a JavaScript Date object.

The variables read from the blockchain are appended to the App's state via this.setState(), this causes a re-rendering of the frontend, which will display this information. It is worth noting that the values are all divided by the constant WAD before being served to the client, this is because all values on the contract are represented in WAD format (multiplied by 10$\hat{1}$8).

Calling subscribeEvents() within componentWillMount() executes the code shown in figure A.7. This function takes an input of the different contract instances and registers a subscription to the events of each contract. This subscription is constantly executing in the application's background using the Web3 API and the Infura WebSockets previously mentioned. On the oracle contract, the only event occurs in the case of a price update, so this event is listened to via dot notation specifically on the contract instance (contract.events.PriceUpdate()).

The ST and BT contracts have a multiple events implemented through the OpenZeppelin ERC20 inheritance; the application subscribes to all events from these two contracts. When any event is detected, the loadBlockchainData() function is called again to fetch and render the data.

Following the creation of the subscriptions, and loading the blockchain data into the application state, the getWeb3() function is called. Suppose the user accepts the request to connect, the output of this operation is a web3 object which can be used to make API calls to the users MetaMask wallet (such as proposing a transaction). This web3 object is appended to the applications state alongside the address of the active Ethereum account within the users MetaMask. Additionally, a new local instance of the ST contract is appended to the state; this is used to propose the user transactions with the correct encoding and destination.

A.3 Writing data to the blockchain

The App component has four "handlers" defined; these will be called in the event of particular forms being submitted on the application. There will be four different forms on the page: Mint, Burn, Deposit and Withdraw. When a form is submitted, it will call the relevant handler within the App component. The choice to keep the handlers within App was made because this is where the different contract instances are defined; it is also where the state must be adjusted due to the form submissions.

Each of the handlers follow a similar pattern to that of the handleMint shown in figure A.8. The input of a handler is a single variable; for handleMint and handleDeposit, this refers to the

amount of Ether that is to be sent to the payable function (the amount of Ether being exchanged for either BT or ST tokens). For handleBurn and handleWithdraw, this is the amount of ST or BT tokens being redeemed for the current Ether value. The ST contract MetaMask instance, (ST_Instance), which was saved into the application state during the execution of componentWillMount(), is used to propose a relevant transaction with the correct payload for the desired effect. In the case of the handleMint, the mint function is called with the destination address being set to the current Ethereum account within MetaMask, and the value of the transaction is set to the value input to the function. The input to the handleMint or handleDeposit function is assumed to be in Ether format. However, transactions must be configured in WEI; to solve this problem, the built-in web3 utility function "toWei" is used to convert it to the correct format.

The toWei function is additionally used in the handleBurn and handleWithdraw functions. This can be seen in figure A.9. The conversion of Ether to WEI is completed by multiplying the Ether value by 10Î18. As mentioned multiple times throughout this project, the number of tokens to be burned or withdrawn must be expressed in WAD format. Converting to WAD format requires the same multiplication, so this is acceptable.

Once a handler has sent a transaction, its transaction hash is appended to the state variable 'pending', an array of all pending transactions. When there is a pending transaction, the frontend will notify the user that the transaction is in progress with a link to the EtherScan page. Upon a transaction being successfully processed by the network, the hash will be removed from the pending array, which will dismiss the notification. Additionally, as soon as this transaction has been processed, the web app's content will be automatically re-rendered thanks to the previously implemented event subscriptions.

The MyForm component is defined within the App.js file and is a child of the App component. The component requires a few pieces of data, known as props, to be passed into it to be rendered correctly. These props are "desc", "name", "callback" The component itself renders a fieldset to the client with a title, description, and a text field. When the submit button within a form is pressed, the contents within the text field is sent to the supplied callback function.

A.4 Rendering the HTML content

The final piece of the App components code is the render() function; this is where all of the previously explained implementation efforts combine into a single product. The render function is what is rendered in the root element and is re-called every time the application's state is altered. Within this function, the HTML tables which render the "dashboard" of the system are defined (for example, the table for token information in figure A.10).

The variables saved into the applications state by the loadBlockchainData, such as totalSupplyST, can be accessed from the state and dynamically rendered client-side in the manner shown above. Furthermore, the render() function also contains the logic for rendering notification boxes of pending transactions, and is where the MyForm components are employed and populated with the relevant information. (Figure A.11).

To render the notifications, each element of the "pending" state variable is mapped to a div that contains the transaction hash linked to the transaction information on Etherscan.

Because create-react-app was used to bootstrap the project, running the command "npm run start" in a terminal inside the directory will start the server. It is possible to visit the application and use all of its functionality by visiting http://localhost:3000 in a web browser. Figure 6.2 displays the completed frontend.

```
async subscribeEvents(oracleContract, ST_Contract,BT_Contract){
  oracleContract.events.PriceUpdate()
  .on("error", (error) => {
    console.log("Error, ", error);
  })
  .on("connected",(subscriptionId) => {
      console.log(subscriptionId);
  })
  .on('data', async (event) => {
      console.log(event); // same results as the optional callback above
      this.loadBlockchainData(oracleContract,ST_Contract,BT_Contract);
  })

  ST_Contract.events.allEvents()
  .on("error", (error) => {
    console.log("Error, ", error);
  })
  .on("connected",(subscriptionId) => {
      console.log(subscriptionId);
  })
  .on('data', async (event) => {
      console.log(event); // same results as the optional callback above
      this.loadBlockchainData(oracleContract,ST_Contract,BT_Contract);
  })
  BT_Contract.events.allEvents()
  .on("error", (error) => {
    console.log("Error, ", error);
  })
  .on("connected",(subscriptionId) => {
      console.log(subscriptionId);
  })
  .on('data', async (event) => {
      console.log(event); // same results as the optional callback above
      this.loadBlockchainData(oracleContract,ST_Contract,BT_Contract);
  })
}
```

```
handleMint = (value) => {
  this.state.ST_Instance.methods.mint(this.state.account).send({from: this.state.accoun
  .on("error", (error) => {
    console.log("Error, ", error);
  })
  .on("transactionHash", (transactionHash) => {
    console.log('Transaction Sent: ', transactionHash);
    this.setState(prevState => ({
      pending: [...prevState.pending, transactionHash]
    }));
  })
  .once('confirmation', (confirmationNumber, receipt)=> {
    console.log('Transaction Approved:', receipt.transactionHash);
    this.setState(prevState => ({
      pending: prevState.pending.filter(transaction => transaction !== receipt.transact
    }));
  })
}
```

Figure A.8: Code for an example handler function within the App component (handleMint)

```
this.state.ST_Instance.methods.burn(this.state.web3.utils.toWei(ST_Quant, 'ether')).send(
```

Figure A.9: Code snippet from the handleBurn function

```
<div class="col-sm-4">
    <div class="table-responsive">
      <table class="table table-bordered">
        <th colspan="3" class="text-center">TOKEN INFO</th>
        <tr>
          <td><b>ST Total Supply</b></td>
          <td><b>BT Total Supply</b></td>
          <td><b>BT Unit Price</b></td>
        </tr>
        <tr>
          <td>{this.state.totalSupplyST}</td>
          <td>{this.state.totalSupplyBT}</td>
          <td>£{this.state.unitPriceBT}</td>
        </tr>
      </table>
    </div>
</div>
```

Figure A.10: Code for example dashboard table within App componenets render() function.

```
</div>
  {this.state.pending.map((transaction) => {
    return(
  <div key={transaction} class="row" style={{backgroundColor: '#00000090',paddingTop:'15p
    <div  class = "col-sm-12" style={{textAlign: 'center'}}> {"Transaction being processe
  </div>
    );
  })}
  <h2>Test the mint, burn, deposit and withdraw functions of the StableCoin system:</h2>
  <div class="row">
  <MyForm name="Mint ST" callback={this.handleMint} desc={mint_desc} currency="ETH"/>
  <MyForm name="Burn ST"  callback={this.handleBurn} desc={burn_desc} currency="ST"/>
  </div>
  <div class="row">
 <MyForm name="Deposit" callback={this.handleDeposit} desc={deposit_desc} currency="ETH"/
<MyForm name="Withdraw" callback={this.handleWithdraw} desc={withdraw_desc} currency="BT"
 </div>
```

Figure A.11: Code that renders notifications and the different forms for frontend

Bibliography

[1] "Bitcoin: A peer-to-peer electronic cash system," https://bitcoin.org/bitcoin.pdf, accessed on 08-11-2020.

[2] "Bitcoin block scanner, btc block 0," https://explorer.bitcoin.com/btc/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f, accessed on 08-11-2020.

[3] "Bitcointalk.org - pizza for bitcoins?" https://bitcointalk.org/index.php?topic=137.0, accessed on 08-11-2020.

[4] K. Matthews, M. Giuliodori, and F. S. Mishkin, *The Economics of Money, Banking and Financial Markets*. Harlow, United Kingdom: Pearson Education Limited, 2013.

[5] "Tether whitepaper," available at https://tether.to/wp-content/uploads/2016/06/TetherWhitePaper.pdf, accessed 08-11-2020.

[6] "The dai stablecoin system," https://makerdao.com/whitepaper/DaiDec17WP.pdf, accessed on 08-11-2020.

[7] "Ethereum: A nextgeneration smart contract and decentralized application platform," https://ethereum.org/en/whitepaper/, accessed on 28-03-2021.

[8] T. Zimwara, "Tether's stablecoin dominance drops below 80 percent as audit controversy lingers on," Sep 2020, https://news.bitcoin.com/tethers-stablecoin-dominance-drops-below-80-as-audit-controversy-lingers-on-on/, accessed 08-11-2020.

[9] "Introduction to smart contracts," https://ethereum.org/en/developers/docs/smart-contracts/, accessed on 28-03-2021.

[10] F. Glaser, K. Zimmermann, M. Haferkorn, M. C. Weber, and M. Siering, "Bitcoin-asset or currency? revealing users' hidden intentions," *Revealing Users' Hidden Intentions (April 15, 2014). ECIS*, 2014.

[11] P. Ciaian, M. Rajcaniova *et al.*, "Virtual relationships: Short-and long-run evidence from bitcoin and altcoin markets," *Journal of International Financial Markets, Institutions and Money*, vol. 52, pp. 173–195, 2018.

[12] "Stablecoins as a sector," https://cryptoslate.com/cryptos/stablecoin/, accessed on 08-11-2020.

[13] G. Hileman, "State of stablecoins (2019)," *Available at SSRN*, 2019.

[14] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain technology overview," *CoRR*, vol. abs/1906.11078, 2019. [Online]. Available: http://arxiv.org/abs/1906.11078

[15] "Hyperledger: What is hyperledger fabric?" https://hyperledger-fabric.readthedocs.io/en/release-2.2/whatis.html, accessed on 19-04-2021.

[16] "Blockchain nodes: An in-depth guide," https://nodes.com/, accessed on 19-04-2021.

[17] B. Preneel, "Analysis and design of cryptographic hash functions," 1993.

[18] C. Jaikaran, *Blockchain: Background and policy issues*. Congressional Research Service Washington, DC, 2018.

[19] "Cloudflare: How does public key encryption work? | public key cryptography and ssl," https://www.cloudflare.com/en-gb/learning/ssl/how-does-public-key-encryption-work/, accessed on 19-04-2021.

[20] E. B. Barker, L. Chen, A. R. Regenscheid, and M. E. Smid, "Sp 800-56b. recommendation for pair-wise key establishment schemes using integer factorization cryptography," 2009.

[21] "Blockchain address 101: What are addresses on blockchains?" https://blockgeeks.com/guides/blockchain-address-101/, accessed on 19-04-2021.

[22] "Bitcoininformation.org: P2sh address, pay to script hash," https://btcinformation.org/en/glossary/p2sh-address, accessed on 19-04-2021.

[23] "Blockchain architecture basics: Components, structure, benefits & creation," https://mlsdev.medium.com/blockchain-architecture-basics-components-structure-benefits-creation-beace17c8e77, accessed on 19-04-2021.

[24] "Geeksforgeeks: Blockchain and block header," https://www.geeksforgeeks.org/blockchain-and-block-header/, accessed on 19-04-2021.

[25] "Ibm: What is blockchain technology?" https://www.ibm.com/topics/what-is-blockchain, accessed on 19-04-2021.

[26] D. Mingxiao, M. Xiaofeng, Z. Zhe, W. Xiangwei, and C. Qijun, "A review on consensus algorithm of blockchain," pp. 2567–2572, 2017.

[27] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, Jul. 1982. [Online]. Available: https://doi.org/10.1145/357172.357176

[28] "Byzantine generals' problem - an introduction," https://academy.ivanontech.com/blog/byzantine-generals-problem-an-introduction, accessed on 19-04-2021.

[29] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols," in *Secure information networks*. Springer, 1999, pp. 258–272.

[30] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 3–16. [Online]. Available: https://doi.org/10.1145/2976749.2978341

[31] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," *self-published paper, August*, vol. 19, p. 1, 2012.

[32] C. Ganesh, C. Orlandi, and D. Tschudi, "Proof-of-stake protocols for privacy-aware blockchains," pp. 690–719, 2019.

[33] A. Berentsen and F. Schär, "Stablecoins: The quest for a low-volatility cryptocurrency," *Fatas A.(a cura di), Economics of Fintech and Digital Currencies*, pp. 65–71, 2019.

[34] "Frax is the world's first fractional-algorithmic stablecoin," https://frax.finance/, accessed on 23-04-2021.

[35] M. Friedman, *Quantity Theory of Money*. London: Palgrave Macmillan UK, 2017, pp. 1–31. [Online]. Available: https://doi.org/10.1057/978-1-349-95121-5_1640-2

[36] M. Mita, K. Ito, S. Ohsawa, and H. Tanaka, "What is stablecoin?: A survey on price stabilization mechanisms for decentralized payment systems," pp. 60–66, 2019.

[37] "døllar," https://github.com/emptysetsquad/dollar/blob/master/d%C3%B8llar.pdf, accessed on 22-04-2021.

[38] N. Al-Naji, J. Chen, and L. Diao, "Basis: a price-stable cryptocurrency with an algorithmic central bank," *Basis. io*, 2017.

[39] R. Sams, "A note on cryptocurrency stabilisation: Seigniorage shares," *Brave New Coin*, pp. 1–8, 2015.

[40] "Multicoincapital: An overview of stablecoins," https://multicoin.capital/2018/01/17/an-overview-of-stablecoins/, accessed on 22-04-2021.

[41] "Makerdao: The strengths & weaknesses of stablecoins," https://medium.com/@MakerDAO/part-3-the-strengths-weaknesses-of-stablecoins-62f13b592e3f, accessed on 22-04-2021.

[42] "Blockchain.com: The state of stablecoins," https://www.blockchain.com/ru/static/pdf/StablecoinsReportFinal.pdf, accessed on 22-04-2021.

[43] "Coininsider: What is a non-collateralized stablecoin?" https://www.coininsider.com/what-is-a-non-collateralized-stablecoin/, accessed on 22-04-2021.

[44] D. Bullmann, J. Klemm, and A. Pinna, "In search for stability in crypto-assets: are stablecoins the solution?" *ECB Occasional Paper*, no. 230, 2019.

[45] "Coingeek: Tether is still lying to you," https://coingeek.com/tether-is-still-lying-to-you/.

[46] "Usdc: the world's leading digital dollar stablecoin," https://www.circle.com/en/usdc, accessed on 22-04-2021.

[47] A. Kahya, B. Krishnamachari, and S. Yun, "Reducing the volatility of cryptocurrencies–a survey of stablecoins," *arXiv preprint arXiv:2103.01340*, 2021.

[48] "Investopedia: What is a commodity?" https://www.investopedia.com/terms/c/commodity.asp, accessed on 22-04-2021.

[49] "Pax gold: The safest way to own gold," https://www.paxos.com/paxgold/, accessed on 22-04-2021.

[50] "What are stablecoins?" https://www.gemini.com/cryptopedia/what-are-stablecoins-how-do-they-work, accessed on 22-04-2021.

[51] "Reserve stabilization protocol," https://reserve.org/whitepaper.pdf, accessed on 23-04-2021.

[52] A. Moin, E. G. Sirer, and K. Sekniqi, "A classification framework for stablecoin designs," 2019.

[53] "Celo: Prosperity for all," https://celo.org/about, accessed on 23-04-2021.

[54] "Coingecko pricing information: Celo dollar (cusd)," https://www.coingecko.com/en/coins/celo-dollar, accessed on 23-04-2021.

[55] "An analysis of the stability characteristics of celo," https://celo.org/papers/stability, accessed on 23-04-2021.

[56] "The market collapse of march 1213, 2020: How it impacted makerdao," https://blog.makerdao.com/the-market-collapse-of-march-12-2020-how-it-impacted-makerdao/, accessed on 23-04-2021.

[57] "Black thursday for makerdao: $8.32 million was liquidated for 0 dai," https://medium.com/@whiterabbit_hq/black-thursday-for-makerdao-8-32-million-was-liquidated-for-0-dai-36b83cac56b6, accessed on 23-04-2021.

[58] "Black thursday makerdao's multi collateral dai exploitation (and the plan to recover)," https://medium.com/linum-labs/black-thursday-makerdaos-multi-collateral-dai-exploitation-and-the-plan-to-recover-c083c0b81875#9154, accessed on 23-04-2021.

[59] "Chainlink blog: What is the blockchain oracle problem?" https://blog.chain.link/what-is-the-blockchain-oracle-problem/, accessed on 19-04-2021.

[60] "Blockchain oracles explained," https://academy.binance.com/en/articles/blockchain-oracles-explained, accessed on 19-04-2021.

[61] "Schellingcoin: A minimal-trust universal data feed," https://blog.ethereum.org/2014/03/28/schellingcoin-a-minimal-trust-universal-data-feed/, accessed on 19-04-2021.

[62] "On stablecoins and beauty pageants," https://hackingdistributed.com/2019/05/07/stablecoins-and-beauty-pageants/, accessed on 19-04-2021.

[63] J. M. Keynes, *The general theory of employment, interest, and money.* Springer, 2018.

[64] "Chainlink: Blockchain oracles for connected smart contracts," https://chain.link/, accessed on 07-04-2021.

[65] "Centralized oracles vs. decentralized oracles," https://www.blockchain-council.org/blockchain/centralized-oracles-vs-decentralized-oracles/, accessed on 19-04-2021.

[66] "What is chainlink? introduction to link token," https://cryptobriefing.com/what-is-chainlink-link-token/, accessed on 19-04-2021.

[67] "What is chainlink in 5 minutes," https://www.gemini.com/cryptopedia/what-is-chainlink-and-how-does-it-work, accessed on 19-04-2021.

[68] "The idea of smart contracts," https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html, accessed on 19-04-2021.

[69] "What are smart contracts? a breakdown for beginners," https://medium.com/coreledger/what-are-smart-contracts-a-breakdown-for-beginners-92ac68ebdbeb, accessed on 19-04-2021.

[70] M. Alharby and A. Van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *arXiv preprint arXiv:1710.06372*, 2017.

[71] "Solidity programming language." [Online]. Available: https://soliditylang.org/

[72] "What is the "unstoppable world computer"?" https://www.bitrates.com/guides/ethereum/what-is-the-unstoppable-world-computer, accessed on 19-04-2021.

[73] "Ethereum virtual machine (evm)," https://ethereum.org/en/developers/docs/evm/, accessed on 19-04-2021.

[74] "The ethereum virtual machine — how does it work?" https://medium.com/mycrypto/the-ethereum-virtual-machine-how-does-it-work-9abac2b7c9e, accessed on 19-04-2021.

[75] "Ethereum: Networks," hhttps://ethereum.org/en/developers/docs/networks/, accessed on 19-04-2021.

[76] "Finematics: Decentralized finance education," https://finematics.com/, accessed on 19-04-2021.

[77] "More institutional investors are buying ether, seeing it as a store of value," https://www.coindesk.com/institutional-investors-ether-rally, accessed on 19-04-2021.

[78] "Ethereum 2.0: Still not scalable," https://coingeek.com/a-ethereum-2-0-still-not-scalable/, accessed on 19-04-2021.

[79] "Ethereum review: Ethereum use cases, advantages & disadvantages," https://www.exodus.com/blog/ethereum-review/, accessed on 19-04-2021.

[80] "Ethereum casper: A comprehensive guide," https://www.skalex.io/ethereum-casper/, accessed on 19-04-2021.

[81] "Introduction to dapps," https://ethereum.org/en/developers/docs/dapps/, accessed on 23-04-2021.

[82] "Web3 foundation," https://web3.foundation/, accessed on 23-04-2021.

[83] "Solidity programming language." [Online]. Available: https://docs.soliditylang.org/en/v0.5.3/

[84] "Dapphub github: Ds-math." [Online]. Available: https://github.com/dapphub/ds-math

[85] "Web3.js ethereum javascript api," https://medium.com/coinmonks/web3-js-ethereum-javascript-api-72f7b22e2f0a, accessed on 23-04-2021.

[86] "Go ethereum," https://geth.ethereum.org/, accessed on 23-04-2021.

[87] "ethers.io: Providers," https://docs.ethers.io/v5/api/providers/, accessed on 23-04-2021.

[88] "Json-rpc 2.0 specification," https://www.jsonrpc.org/specification, accessed on 23-04-2021.

[89] "A crypto wallet & gateway to blockchain apps," https://metamask.io/, accessed on 23-04-2021.

[90] "Blockchain.com: What is an erc20 token?" https://support.blockchain.com/hc/en-us/articles/360027491872\protect\discretionary{\char\hyphenchar\font}{}{}What\protect\discretionary{\char\hyphenchar\font}{}{}is\protect\discretionary{\char\hyphenchar\font}{}{}an\protect\discretionary{\char\hyphenchar\font}{}{}ERC20\protect\discretionary{\char\hyphenchar\font}{}{}token, accessed on 19-04-2021.

[91] "Erc-20 interface: eip-20.md," https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, accessed on 19-04-2021.

[92] "Tokenmintblog: Mintable erc20 token explained," https://tokenmint.io/blog/mintable-erc20-token-explained.html, accessed on 19-04-2021.

[93] "Binance academy: What is a coin burn?" https://academy.binance.com/en/articles/what-is-a-coin-burn, accessed on 19-04-2021.

[94] "Cointelegraph: Erc-20 tokens, explained," https://cointelegraph.com/explained/erc-20-tokens-explained, accessed on 19-04-2021.

[95] "Ethereum wallets the key to your digital future," https://ethereum.org/en/wallets/, accessed on 23-04-2021.

[96] "Hot wallet vs. cold storage," https://learn.onemonth.com/hot-wallet-vs-cold-storage/, accessed on 23-04-2021.

[97] "Metamask: What it is and how to use it," https://decrypt.co/resources/metamask, accessed on 23-04-2021.

[98] "Forbes: Decentralized finance is building a new financial system," https://www.forbes.com/advisor/investing/defi-decentralized-finance/, accessed on 23-04-2021.

[99] "Binancesmartchain:aparallelbinancechaintoenablesmartcontracts," https://dex-bin.bnbstatic.com/static/Whitepaper_%20Binance%20Smart%20Chain.pdf, accessed on 21-04-2021.

[100] D. Geroni, "Blockchain interoperability : Why is cross chain technology important?" *101Blockchains.com*, 2020. [Online]. Available: https://101blockchains.com/blockchain-interoperability/

[101] "Remix: Solidity ide." [Online]. Available: https://remix.ethereum.org/

[102] "Ganache: One click blockchain," https://www.trufflesuite.com/ganache, accessed on 23-04-2021.

[103] "Etherscan: Ropsten testnet explorer," https://ropsten.etherscan.io/, accessed on 28-03-2021.

[104] "npm: react," https://www.npmjs.com/package/react, accessed on 22-04-2021.

[105] "ejs - npm," https://www.npmjs.com/package/ejs, accessed on 28-03-2021.

[106] "Openzeppellin: The standard for secure blockchain applications," https://openzeppelin.com/, accessed on 28-03-2021.

[107] "Web3 secret storage definition," https://github.com/ethereum/wiki/wiki/Web3-Secret-Storage-Definition, accessed on 23-04-2021.

[108] "Vanity-eth.tk: Open source ethereum address generator," https://vanity-eth.tk/, accessed on 28-03-2021.

[109] "Leverage and margin in bitcoin trading," https://www.tradingpedia.com/bitcoin-guide/leverage-and-margin-in-bitcoin-trading/, accessed on 12-05-2021.

[110] "Investopedia: Bank run," https://www.investopedia.com/terms/b/bankrun.asp, accessed on 12-05-2021.

[111] "Coingecko: Cryptocurrency prices and market capitalization," https://www.coingecko.com/en, accessed on 07-04-2021.

[112] "Coinbase digital api," https://developers.coinbase.com/, accessed on 03-05-2021.

[113] "Coinapi:all cryptocurrency exchanges integrated under a single api," https://www.coinapi.io/, accessed on 03-05-2021.

[114] "Openzeppellin docs: Ownership," https://docs.openzeppelin.com/contracts/2.x/api/ownership, accessed on 22-04-2021.

[115] "Bbc common algorithms: Insertion sort," https://www.bbc.co.uk/bitesize/guides/zjdkw6f/revision/6, accessed on 04-05-2021.

[116] "axios: Promise based http client for the browser and node.js," https://github.com/axios/axios, accessed on 22-04-2021.

[117] "Github: Erc20.sol," https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.5.1/contracts/token/ERC20/ERC20.sol, accessed on 12-05-2021.

[118] "Github: Erc20detailed.sol," https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.5.1/contracts/token/ERC20/ERC20Detailed.sol, accessed on 12-05-2021.

[119] "Truffle: Sweet tools for smart contracts," https://www.trufflesuite.com/, accessed on 28-03-2021.

[120] "truffle-flattener," https://github.com/nomiclabs/truffle-flattener, accessed on 28-03-2021.

[121] "Hashex: Abi encoder," https://abi.hashex.org/, accessed on 28-03-2021.

[122] "Ceip-2612: permit – 712-signed approvals," https://eips.ethereum.org/EIPS/eip-2612, accessed on 16-05-2021.

[123] "Layer 2 rollups," https://ethereum.org/en/developers/docs/scaling/layer-2-rollups/, accessed on 16-05-2021.

[124] "Polygon: Ethereum's internet of blockchains," https://polygon.technology/, accessed on 16-05-2021.

[125] "Blockchain security & ethereum smart contract audits," https://consensys.net/diligence/, accessed on 16-05-2021.

[126] "Nest: Securing decentralized oracles," https://nestprotocol.org/, accessed on 16-05-2021.

[127] "Code of conduct for bcs members," https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf, accessed on 07-04-2021.

[128] e. a. Harry Eddis, Michael Voisin, "Uk reveals plans to regulate stablecoins," *LinkLaters: Blockchain and digital assets*, 2021. [Online]. Available: https://www.linklaters.com/en/insights/blogs/fintechlinks/2021/january/uk-government-reveals-plans-to-regulate-stablecoins

[129] T. W. Huw Jones, "Britain will focus crypto rules on stablecoins, minister says," *Reuters: Banks*, 2021. [On-

line]. Available: https://www.reuters.com/article/us-crypto-currency-regulations/uk-to-focus-crypto-rules-on-stablecoins-says-minister-idUSKBN2BM11G

[130] e. a. Dominic Bailey, David Brown, "Venezuela: All you need to know about the crisis in nine charts," *BBC News*, 2019. [Online]. Available: https://www.bbc.co.uk/news/world-latin-america-46999668

[131] "Circle partners with bolivarian republic of venezuela and airtm to deliver aid to venezuelans using usdc," https://www.circle.com/blog/circle-partners-with-bolivarian-republic-of-venezuela-and-airtm-to-deliver-aid-to-venezuelans-using-usdc accessed on 07-04-2021.

[132] "Why your internet habits are not as clean as you think," https://www.bbc.com/future/article/20200305-why-your-internet-habits-are-not-as-clean-as-you-think, accessed on 06-04-2021.

[133] "Proof-of-stake (pos) whitepaper," https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/, accessed on 06-04-2021.

[134] "Github: openzeppelin-contracts," https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v2.5.0, accessed on 12-05-2021.

[135] "Data protection act 1998," https://www.legislation.gov.uk/ukpga/1998/29/contents, accessed on 06-04-2021.

[136] "The money laundering, terrorist financing and transfer of funds (information on the payer) regulations 2017, note = https://www.legislation.gov.uk/uksi/2017/692/contents/made, accessed on 07-04-2021."

[137] "Cryptoassets: Aml / ctf regime," https://www.fca.org.uk/firms/financial-crime/cryptoassets-aml-ctf-regime, accessed on 07-04-2021.

[138] "Uk regulatory approach to cryptoassets and stablecoins:consultation and call for evidence," https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/950206/HM_Treasury_Cryptoasset_and_Stablecoin_consultation.pdf, accessed on 07-04-2021.