

Evaluation of Migrating to Decentralised Applications using Blockchain Technology

A dissertation submitted in partial fulfilment of
the requirements for the degree of
BACHELOR OF SCIENCE in Computer Science

In
The Queen's University of Belfast

by
Blaine Malone
09/05/2017



Acknowledgements

Firstly, I would like to express my appreciation to the Ethereum community. Even during their busiest times, they offered support to any questions that I had regarding the platform.

Special thanks go to my supervisor Desmond Greer, who provided me with outstanding resources and a wealth of knowledge to aid my completion of this dissertation.

Finally, I would like to thank Fergal Downey, Martin Bell and Emer McVeigh who represent Rakuten Blockchain Lab, for their infectious passion and expert advice on various design choices that I had proposed over the course of my project.

Abstract

This paper provides an investigation into the development of decentralised applications (*DApps*) for the Ethereum network.

Initially, the fundamental ideas of blockchain technology are introduced. This is an essential chapter as it aims to build an understanding of core concepts, before progressing to further chapters, where a prototype system named ‘Unilog’ is developed on top of the protocol. The application aims to provide an autonomous service for detecting fraudulent academic credentials on the peer-to-peer network. A systematic approach is taken to migrate the current centralised solution to a decentralised equivalent. Consequently, the new solution can function successfully without the need for any intermediaries.

Unilog not only stands as a working prototype, but in the process, it defines a framework for businesses and institutions alike to adopt blockchain into their daily workflows. For entities to embrace this budding technology with confidence; the solutions need to be secure, reliable and their implementation details straightforward. Likewise, for consumers interacting with the platform, it needs to be user-friendly and transparent. The learning curve for a standard web user is kept to a minimum as they transition from the internet’s classic architecture style to a decentralised peer-to-peer alternative.

Institutions who wish to adopt blockchain technology are given two options, both of which define approaches for leveraging the power of *DApps* using blockchain technology.

Creating a decentralised application serves as a basis for exploring the design decisions that a *DApp* developer will likely face. Opting against a solution which is predominantly based on the familiar client-server architecture style, means that some traditional perceptions must be cast aside. Suggestions are made to help the developer become as proficient on this platform as they are on others. The evaluation discovers certain limitations and concerns that spawn from hosting *DApps* on Ethereum. With this, it is recommended to refrain from hosting mission critical applications until these are addressed.

Contents

ACKNOWLEDGEMENTS	1
ABSTRACT	1
CONTENTS	2
1 INTRODUCTION AND PROBLEM SPECIFICATION	4
1.1 THE DECENTRALISED WEB	4
1.2 BLOCKCHAIN AT THE CORE	5
1.3 CURRENT ALTERNATIVE SOLUTIONS	5
1.4 RESEARCH QUESTION AND METHODOLOGY	6
2 BACKGROUND	7
2.1 BITCOIN'S PROPOSAL	7
2.2 MINING AND THE DOUBLE-SPEND PROBLEM	8
2.3 ETHEREUM	10
2.3.1 <i>State and Accounts</i>	11
2.3.2 <i>Smart contracts</i>	12
2.3.3 <i>Transactions and Messages</i>	13
2.3.4 <i>Crypto fuel</i>	13
2.3.5 <i>Mining and Block Time</i>	15
2.3.6 <i>Ethereum Virtual Machine (EVM)</i>	16
2.3.7 <i>Storage Cost Analysis</i>	17
2.4 INTERPLANETARY FILE SYSTEM (IPFS)	18
2.5 PROTOCOL IMPLEMENTATIONS	19
3 SYSTEM REQUIREMENTS SPECIFICATION	20
3.1 MIGRATION	20
3.2 UNILOG PROTOTYPE	21
3.2.1 <i>Hybrid Approach</i>	21
3.2.2 <i>Total Decentralisation Approach</i>	22
3.2.3 <i>Functional and Non-Functional Requirements</i>	22
4 SYSTEM IMPLEMENTATION	23
4.1 ARCHITECTURE OVERVIEW	23
4.2 WHAT EXACTLY DO WE DECENTRALISE?	24
4.3 THE TEST NETWORK	24
4.3.1 <i>Unilog Nodes</i>	25
4.3.2 <i>Local Client Ethereum Node</i>	26
4.4 DECENTRALISED CODE AS SMART CONTRACTS	28
4.5 TRANSCRIPT STORAGE WITH IPFS	29
4.6 PROVABLY HONEST	30
4.7 DECENTRALISED APPLICATION – THE SMART CONTRACT INTERFACE	31
4.7.1 <i>Fraud Detection in the Verifier Portal</i>	31
5 SYSTEM EVALUATION	33
5.1 TESTING	33
5.1.1 <i>Automated Smart Contract Unit Tests</i>	33
5.1.2 <i>The Graphical User Interface</i>	33
5.2 LIMITATIONS AND CONCERNS	34
5.2.1 <i>Scalability</i>	34
5.2.2 <i>Difficulty Bomb</i>	35
5.2.3 <i>Immutability</i>	35
5.2.4 <i>Smart Contract Design</i>	36
5.2.5 <i>Data Encryption</i>	37

6 CONCLUSION.....	38
6.1 SUMMARY	38
6.2 EVALUATION	38
6.3 FUTURE WORK	40
REFERENCES.....	41
TERMS AND ABBREVIATIONS	44
TABLE OF FIGURES.....	46
APPENDIX 1 – USER MANUAL	47
APPENDIX 2 – THE UNILOG FRAMEWORK USE CASES.....	56
APPENDIX 3 – SMART CONTRACT SOURCE CODE	59
UNILOG REGISTRY SMART CONTRACT	59
UNILOG TRANSCRIPT OWNER CONTRACT.....	60
APPENDIX 4 – SMART CONTRACT TEST CASES	62

1 | Introduction and Problem Specification

1.1 | The Decentralised Web

The internet has become increasingly centralised since its birth. There is a handful of large entities holding our personal data. They have ownership over sizeable data centres that are responsible for a significant portion of today's internet traffic¹. Exposing these large, single points of failure is a hindrance for security professionals worldwide. Throughout this paper, a solution is built that's purpose is to give users control of their data and to decentralise what was previously an entirely centralised process. Along the way, we get the desirable properties of a network with a decentralised consensus at its core.

When considering many of today's internet services, as users we place complete trust on their inner workings, in the hope that they responsibly process our requests. Trust in any capacity exposes vulnerabilities on the trusting party. It presents undesirable opportunities for the trustee to ignore or falsify what you've instructed the web server to do [4].

The internet as we know it, is saturated with this notion of trust. We have our data dispersed throughout the internet, being held by arbitrary entities. We believe that it is their responsibility to be honest, safe and accountable when it comes to handling our data. This belief is fragile as it relies upon an opaque architecture. One that we inherently trust without any guarantee that honesty, security and accountability are being adhered to.

Events in recent years portray this model to be fundamentally broken. For example, the NSA's program PRISM demonstrated how the opaque web of today can be exploited without users knowing [39]. To address the shortcomings of this current model, blockchain technology promises to offer a more transparent web, one with desirable properties built-in by default. Users will not need to worry about trusting central authorities, as their inner workings will become fully transparent and immutable.

Today it is common for application code to be executed on a central server. However, this leaves a service susceptible to many failings. Including, but not limited to, the reliance on hardware and applications security. With the blockchain, these properties are built into the basics of the protocol meaning that all applications inherit them. If a central server crashes, then the service would be down permanently until it is restored. Contrast this with a node crashing in a peer-to-peer network, and you observe that all functionality will still remain.

Table 1. Properties of a decentralised blockchain architecture compared with a typical client-server architecture

Decentralised Blockchain	Centralised Server
New data is added via consensus	New data is added via administrator methods
Can only insert new data, old data is immutable	No restrictions on data modifications
Distributed	Single point of failure
Decentralised	Single point of control
Peer-to-peer	Largely client-server architecture
Cryptographic Authentication and Authorization	Server performs actions on user's behalf
Cryptographic verification	Cryptography is implemented separately, as an afterthought
Resiliency and availability increases with peer count	Backups and contingency plans are manually implemented

The following chapters will explore how these properties materialise.

¹ <https://www.sandvine.com/trends/global-internet-phenomena/> [Accessed: 11-04-2017]

1.2 | Blockchain at the Core

With this in mind, we can look at blockchain technology and how it aims to decentralise the internet.

Blockchain is commonly known as the underlying technology of the well-known cryptocurrency, Bitcoin. It was first introduced in a white paper titled, "Bitcoin: A Peer-to-Peer Electronic Cash System"[5] which was published in 2008 by the enigmatic Satoshi Nakamoto. As outlined in the abstract of the paper:

"A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending."

Here we learn the initial idea of removing the reliance on third trusted parties. Since then many pioneers in the field have been focused on the general uses blockchain technology. The introduction of "alt-coins" as separate blockchains from Bitcoin was the start of a movement to improve and innovate with the technology. They had their own cryptocurrencies and set to advance on the original bitcoin protocol by adding new capabilities.

In late 2013, the Ethereum co-founder, Vitalik Buterin proposed a new kind of blockchain platform. One that had applications at the forefront of design. Ethereum is a programmable blockchain. Rather than giving users a set of predefined operations (e.g. bitcoin transactions), Ethereum allows users to create their own operations of any complexity [6]. In this way, it serves as a platform for not only a cryptocurrency but for many different types of decentralised applications. Essentially offering a solution to the limitations seen in the web today. Ethereum have dubbed this solution 'Web 3.0'.

Throughout this paper, the term 'node', is relied upon to describe a machine that is running a blockchain protocol implementation. In other words, it's simply a partaking peer in the decentralised network. A node's participation can take many forms, including the signing of transactions, mining blocks and storing the actual blockchain data.

1.3 | Current Alternative Solutions

With this new, customisable platform, we can begin to think of use cases where trusted third party interactions can be bypassed and/or removed completely. The verification of academic assets currently relies on various avenues of verification. Universities are commonly known to handle requests for verification directly. It is also common for them to offload the task to third parties². Even more recently, research has been conducted around using the Bitcoin blockchain as a way of issuing and verifying academic transcripts [8]. The latter solution is a step in the right direction if we want to decentralise this process. It does not however, delve into the possibilities that the Ethereum protocol has to offer, such as the added functionality of smart contracts. There are also issues around centralisation due to the use of application-specific integrated circuits (ASICs) being used to mine in the Bitcoin network. Though if the intention was to build for a blockchain that is more mature, then Bitcoin would be the path to pursue because of its wide public adoption. The Bitcoin blockchain is the most tested and reliable blockchain to date³.

² <http://www.studentclearinghouse.org/> [Accessed: 11-04-2017]

³ <https://coinmarketcap.com/> [Accessed: 11-04-2017]

Either way, a large amount of time and resources are dedicated to such tasks as they usually need to be handled in a timely manner. This is mainly due to external companies and recruiters having their own deadlines.

Selecting this use case provides an opportunity to improve the current workflow that is in place, but it also makes for a solid base to explore the feasibility, implementation and benefits of such decentralised applications.

It is unnecessary to exhaustively list the potential mishaps that can occur if a dishonest candidate is mistakenly admitted. A fraudulent employee given responsibly to perform acts normally delegated to a qualified practitioner could end in a tragedy. Fraudulent activity in education is a real and growing problem [7]. A problem that is manifested by the increasing number of diploma mills in recent years [32].

As well as forgery, another problem is if the current systems are not performing as expected. Consider the worst-case scenario where an applicant is under a deadline to present some verification of prior academic achievement. If they miss a given deadline and are not able to deliver what is required then their admission would be denied. One can imagine the different outcomes a scenario like this could lead to.

1.4 | Research Question and Methodology

How can the Ethereum blockchain be utilised to develop an autonomous decentralised application, that provides a way for users to verify the legitimacy of a candidate's academic assets, while minimising the involvement of centralised institutions?

Throughout this paper, the above question will be addressed by answering the following sub-questions:

- How does the Ethereum blockchain work?
- How can the current solutions of academic transcript verification be reengineered to be hosted on the Ethereum network?
- How can data be stored in the new solution architecture?
- Can a user-friendly solution be made to help promote user adoption?
- Is the Ethereum platform ready to host decentralised applications and if not what risks come with early adoption?

Given a set of research questions, it is common to follow an organised way of exploring them. After some considerations, it was decided that the best approach was most likely an 'Implementation Driven Research' technique [9]. This approach advances over time by iteratively building better and better systems. With this, it was important that during development, the next best steps were always taken to ultimately provide a better analysis of the research questions in mind.

The idea is to implement a solution to stand as 'Proof by Demonstration' [10].

2 | Background

This section provides background material which has been studied throughout the development of Unilog. Understanding the foundations of relevant core technologies was critical to ensure a working and valuable system could be constructed.

2.1 | Bitcoin's Proposal

It is seldom that any prelude to Blockchain fails mention the Bitcoin network. It is common practice because blockchain is its underlying, core technology. Blockchain, an idea first introduced in 2008, quickly became revolutionary. Bitcoin provided a new transparency to the finance industry.

“The traditional banking model achieves a level of privacy by limiting access to information to the parties involved and the trusted third party. The necessity to announce all transactions publicly precludes this method” [5]

It removed the need for central authorities who were in control over the transactions occurring between parties. It set the premise for new ideas around decentralised autonomous organisations (DAOs). Think of DAO's as organisations that exist solely on the blockchain [11]. In Bitcoin's case, it removed the need for physical banks while delivering their fundamental organisational functions built into the blockchain. These functions are clearly and publically defined for all to see.

Furthermore, Bitcoin was the first to implement a probabilistic solution to the Byzantine Generals problem. In other words, in a distributed network a desired consensus can be obtained, all while disregarding every malicious actors' efforts to misinform partaking nodes.

“[Imagine] a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement.” [12]

However, Bitcoin does not solve the Byzantine Generals problem in the common case. It only works if one important condition remains true, i.e. the attackers are computationally limited. They must have less than 51% of the networks mining hash rate, as mentioned in section 2.2.

Blockchain technology provides a way for untrusting parties on a peer-to-peer network to agree on the contents of a vastly replicated database. The technology relies heavily on its cryptographic properties to serve this purpose. It is immutable in nature, meaning there is one, universally accepted version of the chains history which cannot be easily changed. During the implementation of Bitcoin in 2009, there were essentially two experiments happening at the same time. The first was to successfully develop a distributed currency with cryptographically enforced digital scarcity [13]. The second was the idea of using a blockchain to secure distributed consensus.

Both are dependent upon each other. For a distributed currency to exist you need a decentralised consensus. This is because all participants need to agree upon the blockchain's state i.e. order of transactions, without trusting a central authority. For a decentralised consensus to exist you need to provide an economic incentive, in the form a distributed currency.

2.2 | Mining and The Double-Spend Problem

Generally speaking, Bitcoin is concerned with the transfer of value between untrusting parties. Thus, the correct state of the distributed ledger (blockchain data) needs to be maintained for each new transaction. Its contents need to be perceived as valid by all nodes on the network, that is, a consensus must be obtained. The ledgers state is altered via a state transition function, which can be roughly interpreted as;

Given the state of the current ledger (S), apply the request for state change (TX - transaction). The result can be either a successful (S') or an unsuccessful state change ($ERROR$).

$$APPLY(S, TX) \rightarrow S' \text{ or } ERROR [1]$$

The result of the above function is dependent upon certain conditions. These are as follows,

- The sender of the transaction should not be attempting to send coins that do not exist. i.e. Bob tries to send 30 BTC when he only has 10 BTC in his wallet.
- The sender of the transaction should not be attempting to spend other people's coins. i.e. Bob attempts to send Alice's coins to Steve. This is enforced using asymmetric keys for producing digital signatures.
- The sender must have supplied enough value to the transaction inputs field. This is achieved by summing all owned unspent transaction outputs ($UTXO$, coins that are not yet spent) so that it is equal to or greater than the sum of the output.

For many years, the state transition function discussed was implemented as part of a trusted third party (TTP). It was their responsibility to provide the correct state of internal data, so that users didn't need to reach a consensus. As previously mentioned, this exposed a single point of failure, which is undesirable in any system that strives for high availability. It also increased the networks reliance on the TTP to behave properly.

The goal of building distributed consensus into the Bitcoin protocol, meant that this state transition function could be incorporated in such a way that everyone would agree on the state change (S') without a TTP . For a cryptocurrency, this was consistent with saying that everyone would agree on the order that transactions.

To achieve this, it is vital that participating nodes contribute to the role of mining. The process of mining requires that nodes incessantly attempt to group transactions into what are known as "blocks". By agreement, the Bitcoin network is intended to produce roughly one block every ten minutes, with each block containing a timestamp, a *nonce*, a reference to the previous block (i.e. a hash) and a list of all the transactions that have taken place since the previous block [1]. In this way, the term "blockchain" was coined. A continually growing chain of blocks that represent the new and agreed state of the Bitcoin distributed ledger.

Miners can create blocks through a process which is designed to be computationally intensive. To successfully create a block, a valid 'Proof of Work' (PoW) must be published. To accomplish this, a significant amount computation needs to be invested into solving a mathematical puzzle, where a solution can only be found through trial and error. A cryptographic hash function (SHA256) is the basis for these puzzles and a solution is only accepted if the output of the function is below a certain threshold value. It could take a typical computer a matter years to generate a valid proof of work for a block, but with all miners in the entire network trying to calculate the valid proof of work, it takes about 10 minutes on average. The difficulty of these puzzles is configured by the *block difficulty*.

The ‘winner’ broadcasts their block and receives a reward. This creates the needed incentive for nodes to continue with their participation in mining, and therefore contribute to the networks security. The *PoW* algorithm, thwarts attackers from promoting a dishonest blockchain that supports a malicious agenda. Even if an attacker was computationally equipped to do so, there is still a greater incentive to behave honestly on the network. This is because mining is likely to offer a greater return on investment than any attempts to defraud the platform.

Proof of work provides a way for honest nodes overcome Byzantine failures and to accept which block should be the next in the chain (i.e. reach a coherent global view of the system state [17]).

There is no way to look a transaction and cryptographically determine when it was created. Therefore, reaching a consensus for accepting blocks is arguably the most difficult part of implementing the protocol as this is where many attack vectors present themselves.

The double-spend problem provides us with an example of how an attack can occur while demonstrating how the Bitcoin network reaches a consensus about the state of the blockchain. Double-spending describes when a bad actor (traitor in the Byzantine Generals Problem [12]) could spend the same funds twice, denying the first transaction ever occurred. Let’s look at the following example of how a consensus is reached and how the network is protected against a double spend.

1. Payment of 50 BTC is sent from Bob to a retailer in exchange for goods.
2. Bob receives the goods after waiting around one hour.
3. Bob creates another transaction of 50 BTC and sends it to himself in the hope that he doesn’t lose money from the first transaction. This is where Bob will attempt to change the order of transactions to convince the network that the second transaction came before the first.

Once Bob sent the first transaction, he waited one hour before sending his second transaction. During this time, the network had moved on to successive blocks and Bob’s first transaction would have been placed into a block along the way. This block would then also be validated and accepted by the network. Block propagation continues around four more times before the hour expires. Bob now sends his second transaction. At this stage, Bob has a mammoth task ahead of him. He needs to convince the network that his second transaction occurred before the first one. Bitcoin can be described as a first-to-file system, where it will attempt to securely maintain the ordering of transactions by employing the proof of work scheme. The reason Bob sends the transaction to himself and doesn’t release it to the network is because as soon as another miner attempts to validate the transaction using the state transition function they will receive an error and abort.

To succeed, Bob needs to create a fork in the chain at the block where his first transaction was accepted. He needs to replace his first transaction with the new one. This newly created block now requires Bob to provide a proof of work, which is significantly different than before because of the contents update. Not only does a new proof of work need to be calculated for the current block, but it also needs to be calculated for the four successive blocks because of the updated block headers. This leads to Bob creating a separate chain to that of the original. For this new chain to be accepted in place of the original, it needs to be longer. This is because all nodes are programmed to settle on the chain with the largest investment into proof of work i.e. the longer chain.

Bob may be able to complete such a feat if he secures a 51% share of the networks computing power. If not, he is unlikely to succeed at double spending due to the nature of solving the proof of work puzzle. Having more than a 50% share of the networks computing power exposes the *51% attack* vector.

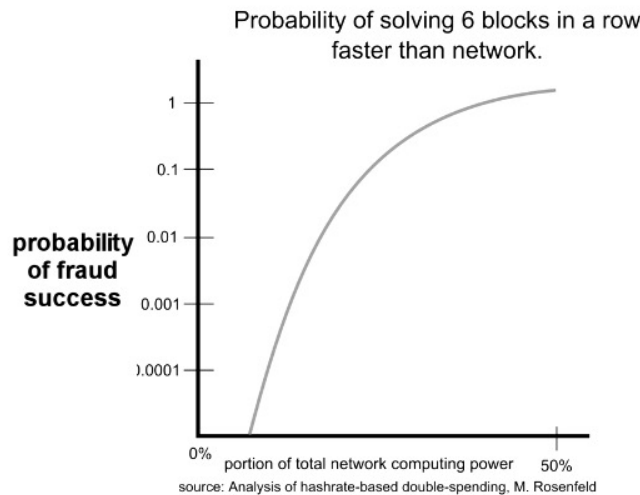


Figure 1. Graph showing the relative probabilities of solving consecutive blocks to execute a double-spend attack

2.3 | Ethereum

Decentralised consensus technology is facilitating approximately 300,000 transactions per day through Bitcoin⁴. A common misconception is that people think Ethereum is trying to compete with Bitcoin in the currency space.

To understand Ethereum, one needs to think of how decentralised consensus has revolutionised the finance industry. It leads to the question, can this technology be used for more general applications and not just a cryptographic currency?

Ethereum was founded on the thought of taking decentralised consensus and using it as a base for hosting a myriad of applications. In order to achieve this, it was realised that a general platform needed to be assembled that addressed both Bitcoin's consensus shortcomings and provided a blank canvas for application developers. Bitcoin was known to provide basic scripting functionality which seen many new applications emerge on top of the Bitcoin blockchain (e.g. Namecoin). Nevertheless, this feature was limited and the Ethereum founders made the decision to build a separate network as opposed to building a protocol on top of Bitcoin [1].

The overall paradigm of Ethereum is synonymous with a well-known example. The Android operating system and the Google Play Store. Engineers at Google provided a platform for developers to create mobile applications that could be run on the Android OS. The same philosophy applies to Ethereum.

At a glance, the Ethereum architecture can be digested naturally when one has prior experience with comparable technologies.

For novices however, the learning curve can be quite different. Understanding how the blockchain's state is updated across all nodes is the first of a few hurdles to overcome.

⁴ <https://blockchain.info/charts> [Accessed: 11-04-2017]

2.3.1 | State and Accounts

Ethereum adopted the approach of an account-based blockchain. The design rationale originated from a belief that if they were building a platform that would host decentralised applications which contain arbitrary state and code, then the benefits of accounts would greatly outweigh Bitcoins *UTXO* system.

To build a better conceptual understanding, we can say that all account states reside locally on the Ethereum node in the form of “state data” (this is common for performance reasons and is assumed that it will be stored in a Merkle Patricia tree, but the protocol specification doesn’t require it. [3]). So, in addition to the blockchain itself, we find ourselves dealing with a second state. State data can be described as implicit, meaning it can be calculated from the actual blockchain data. Transactions contain all the appropriate fields to determine new state data. Unlike Bitcoin, Ethereum blocks contain a copy of both the transaction list and the Merkle root hash of the entire state tree.

Let’s look at a high-level example to stress the importance of this new concept.

There exists a network which is hosting three nodes (A, B and C) that share the same blockchain. All nodes are capable of mining transactions into blocks. Every node stores two states, the actual blockchain and the state data containing the list of accounts and their associated states. For simplicity, each node contains one account also named ‘A’, ‘B’ and ‘C’ respectively.

At a point in time every account contains a balance of 100 ETH. If account ‘A’ needs to send 50 ETH to account ‘B’, we get the following sequence of events:

1. Node ‘A’ creates a transaction that has the intention of updating the state data of every node on the network. The transaction is concerned with sending 50 ETH from account ‘A’ to account ‘B’.
2. Node ‘A’ will submit the transaction to the network in the hope that it will be included into a block by a miner.
3. Node ‘C’ executes this transaction locally and validates that it is an acceptable state transition. Node ‘C’ is also the first to generate a valid proof of work for this new block containing our transaction. A new block announcement is made.
4. Nodes ‘B’ and ‘A’ pick up on the latest block announced. After block validation checks out to be true (i.e. calculated state root matches newest blocks state root etc.), the block is accepted and the transactions enclosed will be executed. Following execution, the account balances of accounts ‘A’ and ‘B’ will be 50 and 150 respectively. The state data on all nodes will now reflect the updated balances of accounts ‘A’ and ‘B’. Every node agrees on these new balances.

See Figure 2 for a visual aid on the above events.

There are two kinds of Ethereum accounts. These are externally owned accounts (EOAs), which are controlled by private keys and contract accounts, controlled by their contract code (often referred to as Smart Contracts, discussed in section 2.3.2).

In addition to having an account address, the Ethereum account contains four fields:

- Nonce - a counter used to make sure each transaction can only be processed once.
- Balance - the account’s current amount of Ether.
- Code - an account’s contract code, if present.
- Storage - storage for the account which is empty by default.

An EOA has no code and the account owner can send messages from an EOA by creating and signing a transaction with the accounts private key. In a contrast, a contract account which contains code will only act once it receives a message. It is not uncommon to hear them being referred to as, “autonomous agents” [1]. This suggests that once deployed to the blockchain, they are their own entity. The actual code in the smart contracts now have control over the actions they take. When the contracts code is invoked by receiving a message it has the functionality to read from and write to internal storage. In addition to this, contracts can send “messages” to other contracts.

Regardless of whether the account stores code, the two types are treated equally by the Ethereum Virtual Machine (*EVM*) [1].

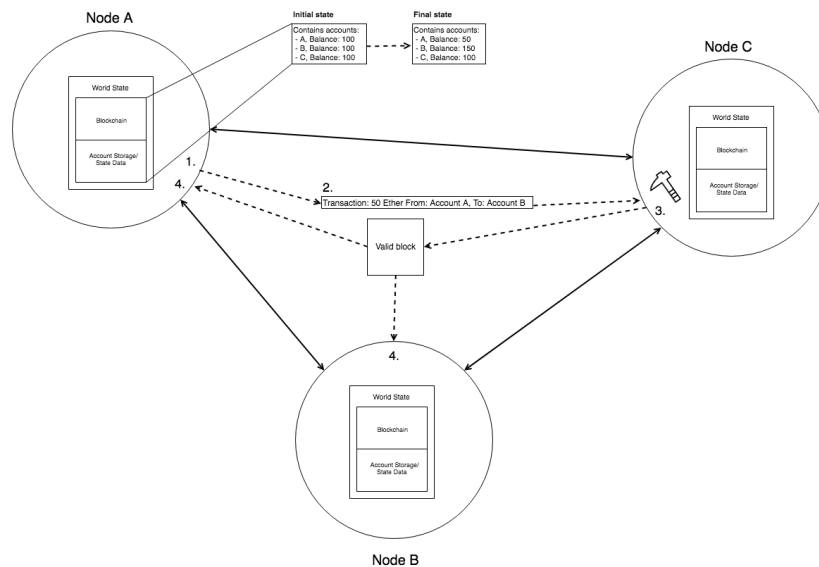


Figure 2. Diagram depicting events occurring in a simplified view of a network transitioning state on the Ethereum network

2.3.2 | Smart contracts

It has been discovered that smart contracts are simply just another kind of account that lives within Ethereum. That being said, they are a differentiating feature from most blockchain technologies.

To execute a contract, it needs to be compiled into *EVM* bytecode. There are many different high-level languages that have been created that can be compiled to the specific bytecode of the *EVM*.

The community has created the following Turing Complete programming languages:

- Mutan - A Golang-like language which has since been deprecated in March 2015.
- LLL - A Lisp-like language.
- Serpent - A Python-like language.
- Solidity⁵ - A JavaScript-like language. Currently the most prominent of all implementations. It was built with the intention of possibly replacing all three other languages.

Having the property of Turing completeness allows developers to implement more complex solutions that were not possible with Script⁶, Bitcoin’s deterministic scripting system.

⁵ <http://solidity.readthedocs.io/en/latest/> [Accessed: 12-04-2017]

⁶ <https://en.bitcoin.it/wiki/Script> [Accessed: 12-04-2017]

In fact, making the *EVM* Turing-complete was a decision made not solely through the need for Turing completeness, but because of the way accounts can interact with each other. Ethereum talks about the *first-class-citizen property*, where contracts can do anything that external actors can, this includes calling other contracts to perform actions. This exposes the potential problem of infinite recursion. A malicious actor could write contract code in a way that it causes nodes on the network to enter an infinite loop and halt [1]. Design decisions were made to prevent these events from occurring. It was these design decisions that allowed the *EVM* to be Turing-complete. There is an inherent constraint on the number of computational steps transactions/messages can perform (discussed section 2.3.4) so why not give the added benefit of Turing-completeness anyway? For these reasons, Ethereum is often labelled as Quasi-Turing-complete⁷ because of the guaranteed finality of all executions.

2.3.3 | Transactions and Messages

Transactions have been alluded to throughout this paper. In Bitcoin, they were necessary for the state transition function. Again, the same applies with Ethereum. Looking at the structure of a transaction, the differences between Ethereum and Bitcoin become evident. Transaction objects contain the following [1]:

- Recipient - Receiving account of the transaction(null on contract creation transaction).
- Sender - A signature identifying the sender.
- Ether - The amount of ether to transfer from the sender to the recipient (ETH).
- Data - An optional data field, which could contain data contract code if account is creating a new contract.
- Gas - A start gas value, representing the maximum number of computational steps the transaction execution can take (unused gas is refunded to the account sender). Can think of this as the total amount of gas a sender is willing to purchase for the transaction to be executed.
- Gas Price - A value, representing the fee the sender pays per computational step.

When a transaction is signed, and submitted to the network it needs to be validated. In Bitcoin, there are three validation steps for the state transition function (section 2.2). In Ethereum there are six, due to the extra parameters that are available in a transaction object. Messages are essentially the same as transactions, but differ by the origin of where it comes from. Messages are only produced by contracts. It is common for contracts to communicate with other contracts and this is done using messages. The signing of transactions facilitates the notion of ‘zero sign-on’ (see section 2.3.7). Only the owner holding the private key for an account can send transactions from that account. Therefore, applications can identify the user, removing the need for registration or logging in.

2.3.4 | Crypto fuel

All transactions in a new block need to be validated and accepted by every node in the network. This means that state changes which are triggered by these transactions need to be replicated on every node. Consequently, there are costs incurred for every redundantly executed operation. These are enforced by assigning the different operations that can be executed with a charge. Ethereum calls this ‘gas’ and can be purchased using Ether. Note that

⁷ <https://blog.ethereum.org/2014/12/17/ethereum-d%E0%BEv/> [Accessed: 13-04-2017]

these costs cannot be avoided. If you send a valid transaction to the network that is executing some arbitrary contract code or not, you will still have to provide gas along with your transaction.

If a developer creates a smart contract as part of a *DApp*. It is only fair to assume that every user executing code on the smart contract will not incur wildly different costs each time they return. If transaction fees were dependent on a volatile underlying currency then this would indefinitely happen.

For this reason, gas and ether are decoupled. To calculate the amount of gas needed to execute a transaction, we can use the following formula:

$$\text{Total gas cost} = \text{gas} * \text{gasPrice}$$

Where,

- ‘gas’ is a constant value that can be derived from totalling all the operations executed by a transaction. The values of each operation are predetermined and Ethereum protocol specifies them. The gas used by a transaction.
- ‘gasPrice’ price is not constant. It can be fine-tuned by the network to compensate for fluctuations in the value of Ether. With this model, it can be ensured that the total gas cost of sequential interactions with a *DApp* will remain relatively constant.

In practice, for a user to execute a transaction they will need to specify the maximum amount of ‘gas’ that they’re willing to buy (gas limit/gasUsed) and the ‘gasPrice’. These parameters are optional and if not explicitly stated then the client implementation will determine them automatically for the user by making an estimation on what it thinks the transaction gas should be (note that this is not entirely safe⁸ because in many cases you cannot be certain on the gas that will be bought by executing a transaction) [3].

If a user specifies a gas value that is too low, then it’s common for clients to throw an error telling the user that they’re not purchasing enough gas for this transaction to be executed. In contrast if the user sets the gas too high then they may be effectively spending more gas than what was originally intended.

It is important that the sender ensures they have thought about the gas limit that they’re using because in the event of an unexpected computational steps, they could lose a lot of funds or even bankrupt themselves [20].

Now we can understand why Ethereum is thought to be Quasi-Turing-complete⁶. If transactions did not have a gas limit then attackers could expose the vulnerability of the *EVM*’s Turing-completeness property. This would cause nodes that validate transactions to enter an infinite loop and thus causing a denial-of-service [20].

As previously stated, gas price is not constant and the choice is given to the network users to specify its value. It can be thought of as the amount in *wei* that the transaction sender is willing to pay per unit of gas.

If a transaction needs 100 gas to execute successfully and the default gas price for the network is 0.05×10^{12} *wei* then the total gas cost is:

$$\begin{aligned}\text{Total gas cost} &= 100 * (0.05 * 10^{12}) \\ \text{Total gas cost} &= 5 * 10^{12}\end{aligned}$$

For a sender to increase the chances of getting their transaction included into a block they would increase the gas price above the network default. Every block that gets created by

⁸ <https://ethereum.stackexchange.com/questions/266/> [Accessed: 14-04-2017]

miners also has a gas limit⁹ like transactions. As a result, this means that the miners will pick the transactions with the higher gas price per unit of gas used to get the best return for using their computing resources [20].

The gas limit acts as an upper bound on amount of computation a transaction can perform on its journey in the network. Similarly, if a car doesn't have enough gas for a journey the it will not reach its destination. This comparison is why Ethereum dubbed ether as the platform's crypto fuel.

2.3.5 | Mining and Block Time

As of April 2017, the current release of Ethereum is Homestead¹⁰. It was predicted that in a future release (*Serenity*) the core developers intend on moving away from their Proof of work consensus algorithm (*Ethash*¹¹) to favour what's been called '*Casper*', a Proof of stake (*PoS*) alternative. This is to address some of the concerns that have developed over time in Bitcoin's proof of work scheme. A notable concern is the energy consumption of mining in the Bitcoin network. It was calculated in 2014 to consume a comparable amount of electricity with that of Ireland [22]. The changeover was forecast to be sometime in the latter half of 2017¹² but may be later and run into 2018. Several projects today like the Gridcoin¹³ cryptocurrency have based their consensus algorithm on *PoS*.

Upgrading to a *PoS* scheme has raised many questions in the community, many of which come from the dedicated miners on the network. One frequently asked question was whether the hardware miners purchased would become obsolete¹⁴ if *PoS* was adopted. However, most mining equipment could still be used to mine other cryptocurrencies. In addition to this, there are projects like Golem¹⁵ which will pay users to 'rent' out their computing power for various computational tasks. An element of this project is built on Ethereum and uses the idea of a lottery based smart contract to pay users for renting their resources.

This paper will not explore the inner workings of *PoS*, as knowledge of this subject isn't currently needed to build a *DApp* on Ethereum, and therefore not needed to answer the research questions in mind.

So why did Ethereum not use the PoW algorithm deployed by the Bitcoin network?

As previously mentioned, Ethereum wanted to solve the limitations/concerns of Bitcoin by eventually bringing in *PoS*. In the meantime, they developed *Ethash* to enhance limitations that they could currently improve upon [23].

Ethash was primarily developed to tackle the problem of increasing centralisation of miners in Bitcoin. Mining pools, provide a way for highly profitable gains through sharing the effort of computing a valid *PoW* for a block. This means that the more computing resources available in a pool, the higher the chance of the blockchains consensus rules being broken. This could include a *51% attack*.

In general, centralisation has made many independent miners jobless. It is no longer profitable for independent miners to compete to solve the *PoW*. *Ethash*, is known as an "ASIC-resistant" *PoW* algorithm [23]. This property tackles the centralisation of mining by

⁹ <https://ethstats.net/> [Accessed: 14-04-2017] (Currently 4,007,729 gas at the time of writing.)

¹⁰ <http://ethdocs.org/en/latest/introduction/the-homestead-release.html> [Accessed: 14-04-2017]

¹¹ <https://github.com/ethereum/ethash> [Accessed: 14-04-2017]

¹² https://www.reddit.com/r/ethereum/comments/5lfugz/eip_186_modified/ [Accessed: 14-04-2017]

¹³ <https://en.wikipedia.org/wiki/Gridcoin> [Accessed: 14-04-2017] (hybrid consensus algorithm in which *PoS* plays a part)

¹⁴ <https://forum.ethereum.org/discussion/11408/casper-too-late-to-mine> [Accessed: 14-04-2017]

¹⁵ <https://golem.network> [Accessed: 14-04-2017]

proposing a memory hard aspect to the algorithm. This means that miners using *ASICs* do not gain a huge advantage over the miners using an affordable GPU.

The Ethereum network because of this reason, is not reliant on ASIC producers being devote advocates of decentralised technology (they may decide not to sell on their hardware, but instead use it themselves). In other words, the Ethereum community does not need to worry about *ASIC* producers holding a 51% share in the network.

Another way that Ethereum tackled centralisation was by decreasing the incentive for miners to join mining pools, as this is the only feasible way to receive a profit in Bitcoin. A protocol called Greedy Heaviest Observed Subtree (*GHOST* [26]) allowed this to happen. From the outset, Ethereum wanted to allow for faster block times. The main reason for this was to make decentralised applications more responsive for users [24]. Specifically, the community predicted that *GHOST* would allow them to hit a block time of around 12-15 seconds (Currently at 14.73 seconds¹⁶). With a fast block-time comes reduced security, therefore, it is expected for users to wait for multiple block confirmations to secure the transactions from double-spending. In general, blockchains with a faster block times produce more ‘stale’ blocks (also known as uncles). This is a result of the increased likelihood of two nodes mining a block around the same time, but only one is accepted because it spread through the network faster [1]. It is more likely for an independent miner to produce a stale block than a mining pool due to differing shares of hash-power. That is, a mining pool is more likely to mine a block faster and therefore they can start work on mining the next block before others. Effectively gaining a head start on their competitors in the ‘race’ for the next block [1]. If stale blocks were rewarded also then it would make miners think twice about the advantages of entering a mining pool. The *GHOST* protocol did exactly this. Miners now get an extra reward (approximately 4.375 ether of the total block reward [23]) if they include stale blocks into the successful block. This means increased chain security, more profit for the miners and more incentive for independent miners to contribute to the security of the network through mining as they too receive a reward even if their block is stale.

As mentioned in section 2.2, *block difficulty* can be used to configure the block time so that it remains around the desired 12-15 seconds.

2.3.6 | Ethereum Virtual Machine (*EVM*)

The Ethereum Virtual Machine is an execution environment for running transaction code to obtain a consensus. We have established that Ethereum allows for a more complex set of instructions to be completed in the form of smart contracts. For nodes on the network to be able to validate state changes relating to these contracts, they’ll need to be able to run their code and check if it’s a valid state transition. Again, this imposes a large computational redundancy on the network but it necessary to obtain consensus.

Creating a new virtual machine allowed Ethereum to target the issue of developer centralisation that was apparent in the Bitcoin community. This is where one code base/implementation was deemed to be the protocol¹⁷.

¹⁶ <https://ethstats.net/> [Accessed: 15-04-2017]

¹⁷ <https://github.com/ethereum> [Accessed: 15-04-2017] (Repository linking to many different protocol implementations.)

2.3.7 | Storage Cost Analysis

Many useful applications need to persist data to be beneficial for end users. If one is to migrate their application from a centralised approach to a decentralised alternative, then they'll need to consider where their users persistent data is stored.

A common mistake is to think that because Ethereum hosts decentralised applications then they must also provide the platform for storing an application's persistent data. In previous sections, we have encountered Ethereum accounts and established the fact that they are stored locally on the node (account data being derived from the blockchain itself). If a user has an Ethereum account then it discontinues the notion of having multiple accounts for many different websites/applications in today's centralised web. One account, providing network wide access to all *DApps*, referred to as 'zero sign-on' by Stephan Tual, a former CCO at Ethereum. If we now know that account data is inherently managed by the Ethereum protocol then all we need to consider is the storage of other kinds of data that an application needs.

Why shouldn't I store my applications data on the blockchain?

First and foremost, a blockchain by nature is replicated on every node in the network. Storing confidential information on the blockchain is advised against, even if it is encrypted. Because of the permanent nature of blockchains it means that advances in cryptography/cryptanalysis could potentially break encryption algorithms that we thought were close to unbreakable at the time of publishing to the blockchain [28].

Not only would it exponentially increase the size of the blockchain and thus affect user experience, it would cost an unsustainable amount of money.

By way of illustration, we can examine the cost of an example academic transcript being stored on the Ethereum blockchain.

A simplified degree transcript contains the following fields populated with sample data:

- Accredited Institution: *Queen's University Belfast*
- Degree Title: *Computer Science*
- Recipient: *John Smith*
- Classification: *First Class Honours*
- Date: *3rd day of July 2017*
- Additional Information: *Including Professional Experience*

Strings are represented in their UTF-8 encoding in a smart contracts storage¹⁸. This means one character can be represented by 8 bits (1 byte). If we total the number of characters in our sample data and add four to allow for a comma separated list we get 129. This means to store this transcript we need to effectively buy 129 bytes worth of storage in a smart contract.

```
contract TranscriptStorage {
    byte[129] data;
    function TranscriptStorage() {
        for (uint i = 0; i < 129; i++)
            data[i] = 'C';
    }
}
```

¹⁸ <https://solidity.readthedocs.io/en/develop/frequently-asked-questions.html> [Accessed: 15-04-2017]

This contract, written in solidity, stores 129 bytes of data when it is deployed to the network. Deploying this contract locally used 793852 gas. With a current gas price⁸ of 20 gwei (20000000000 wei), this transaction would have a total cost of:

$$\begin{aligned} \text{Total gas cost} &= 793852 * 20000000000 \\ \text{Total gas cost} &= 1.58 * 10^{16} \text{ wei (0.0158 Ether)} \end{aligned}$$

In today's market, 1 ether is £38.15 so, 0.0158 ether would be equivalent to £0.60283. To conclude, it would cost around 60p to store one transcript on the Ethereum blockchain. If there are an average of 100 students enrolled in each degree and there are 100 degrees at the university, then the total cost of storing all the transcripts for a university would be $100 * 100 * 0.6$. Equating to approximately £6,000. To adopt this approach permanently would cost an absurd amount of money and it isn't predictable either because gas prices fluctuate depending on the current state of Ethereum's ecosystem. Thankfully there are current technologies for decentralised data storage and they're free to use. This means that we can still build a decentralised application and not have to rely on a central database for persistent storage. One of these technologies in particular is *IPFS* and it will be the focus of the next section.

2.4 | Interplanetary File System (*IPFS*)

As previously discovered, it is too costly to store data directly on the Ethereum blockchain. This means an alternative persistent data source is required; one that is fully decentralised, to successfully migrate a centralised application to a fully-fledged *DApp* on Ethereum.

The Interplanetary File System (*IPFS*) is a peer-to-peer distributed file system [29]. It gives network users access to any file that was published, provided they know its content address (hash of files and objects using Multihash¹⁹ format and Base58 encoding). Like blockchain technology, it has no single point of failure and nodes do not need to trust each other [29]. *IPFS* speaks about the idea of a 'permanent web', what they mean is any published data to the network it will be available forever. This property is particularly desirable for cases where the loss of data can be catastrophic for users and businesses [30].

The more times a piece of data is accessed in *IPFS* the more copies of it co-exist on the network. Therefore, if the data is useful in anyway then there will always be a copy of it residing on some node in the network. In the example of degree certifications, the amount of copies would increase with the number of verifications on the document. This happens because once a node requests data from the network, it will not only receive it, but it will act as a host for other nodes looking the same data. Object pinning is used to describe when nodes ensure the survival of objects i.e. they may pay third parties with a cryptocurrency called FileCoin to pin their object so that it's always available [29]. Though this would go against properties of a *DApp*.

Adding new data to *IPFS* doesn't propagate to every node on the network like it would using Ethereum. Instead it broadcasts a message to the network stating that you have the data corresponding to a particular content address. If a node would like to access it then they can find it here. The data is only sent when a node requests it.

It's worth noting that *IPFS* builds on successful earlier peer-to-peer systems. To name a few, BitTorrent and Git. For example, it versions files using a Git inspired solution.

¹⁹ <https://github.com/multiformats/multihash> [Accessed: 16-04-2017]

2.5 | Protocol Implementations

There are many different implementations of the Ethereum protocol, which was specified in the yellow paper [3]. These implementations are nodes that can parse, verify the and interact with the blockchain. They can be written in many different languages and are often referred to as clients. It is important to study them and then choose the best implementations based on the purpose of the *DApp*.

The Ethereum Foundation endorses several reference implementations that support the protocol:

- Ethereum C++
- Go Ethereum (Geth, written in Go)
- EthereumJ (Java)
- Pyethereum (Python)

Bitcoin too has numerous implementations but at the beginning there was just one, Bitcoin Core²⁰, which was mostly written in C++. The benefit of having multiple clients at the inception of a protocol offers resilience and other advantages for the network. It comes in the form of:

- Faster issue detection and correction - The more people interpreting the protocol specifications, the less chance of errors staying undetected.
- More diversity offers increased security - If there is an attack vector or bug in any of the Ethereum implementations it means the network is usually fine as there is a bigger diversity of clients available. Currently, Geth has the majority share of the network but as Ethereum grows this should decrease.
- Different languages offer different benefits - Java, widely known and highly adopted in enterprise solutions. C++, very fast but more difficult to build. Go, easy to understand and read but loses some speed.

Though, it has been argued that multiple implementations do not offer as much benefit previously thought. Satoshi Nakamoto posted on an ‘bitcointalk²¹’ forum in 2010 saying he would advise against it.

“I don't believe a second, compatible implementation of Bitcoin will ever be a good idea. So much of the design depends on all nodes getting exactly identical results in lockstep that a second implementation would be a menace to the network.” – Satoshi Nakamoto

Since then blockchain technology has gathered a sizeable following. Even though blockchains are still a relatively esoteric field, as time progresses, maintaining multiple code bases has become more realistic because of this wider adoption.

The Unilog framework will interface with existing Ethereum protocol implementations. The exact implementations are discussed in following sections.

²⁰ <https://github.com/bitcoin/bitcoin> [Accessed: 16-04-2017]

²¹ <https://bitcointalk.org/index.php?topic=195.msg1611#msg1611> [Accessed: 16-04-2017]

3 | System Requirements Specification

To explore the remaining research questions defined in this paper, it was necessary to build a working prototype concerned with the verification of academic assets. However, from a high-level, the prototype defines a framework for entities looking to reap the benefits of blockchain technology. Adoption of a new technology into a business's workflow can be a huge undertaking, yet it is still desirable for businesses to embrace disruptive technology to avoid being left behind.

It is well known that today's application architecture on the internet is mainly based upon a client server style. The developed prototype, named Unilog, sets to provide a streamlined mechanism for entities to migrate their current architecture to one that is completely decentralised. It also offers a hybrid approach for users who are less concerned about the trust issue of a central server. The hybrid approach is more of an incremental way for an entity to adopt blockchain technology. Instead of a complete architecture overhaul, they may just integrate with an existing Unilog system. This means that they'll get many benefits of decentralisation like zero downtime, but they'll still be susceptible to certain vulnerabilities that stem from centralisation.

3.1 | Migration

It is important to fully realise the term 'migrate' within the context of this paper. Similar to the notion of migrating an application to from one environment to another e.g. The Cloud. It is defined exclusively here as the process of transforming an application from a centralised architecture (i.e. client-server) to a decentralised architecture using blockchain technology.

Section 1.3 discussed the current, alternative solutions for verifying academic transcripts. It was revealed that one solution was to offload the task of verifying transcripts to a trusted third party. Such a centralised approach is a suitable starting point for migrating to a *DApp* on the Ethereum network.

Let's look at a simplified architecture of a third trusted party verifying academic transcripts for a university.

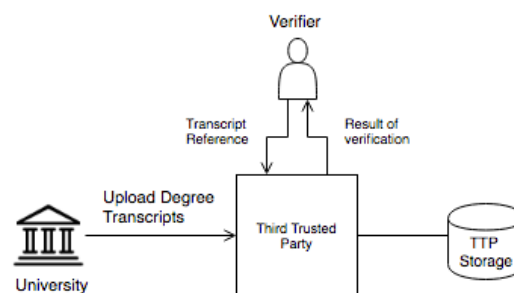


Figure 3. Diagram showing a simple architecture overview of a common solution for academic verification

1. Uploading transcripts to a central server so that they can be stored in a database owned by the *TTP*.
2. *TTP* handles requests from verifiers and provides them with relevant information regarding the existence of a candidate's transcript.

As discussed in the problem specification, a university would need to fully trust the *TTP* to handle their transcripts with care and to honestly carry out the requests made to their service. Three fundamental entities can be highlighted in the current system. Regardless of the system architecture, these three entities will remain.

- *Verifier* - A party that wants to check if a transcript is legitimate. An example of a typical scenario would be when a qualified person submits a CV to a company accepting applications. The company wants to verify that the information on the CV is accurate; so, they will act as a verifier in the system.
- *Issuer* – Could be an accredited institution who awards transcripts e.g. university.
- *Owner* – Transcript owner, they have gained the accreditation and have control over who they wish to see it e.g. student.

3.2 | Unilog Prototype

To migrate the current solution to a *DApp* we need to ensure that no core functionality is lost, all while maintaining the benefits of decentralisation. We can explore two options for accomplishing this.

3.2.1 | Hybrid Approach

It is common place for enterprises within the blockchain space to assist users when interacting with the technology. MetaMask²² for example, enables users to connect to the blockchain with zero synchronisation time. It does this by using an intermediary server that has already synced with the network. Again, this requires a level of trust that the server you are speaking to is relaying your transactions to the network and that it is providing you with the real state of the blockchain. This option exposes users to the *Sybil attacks*.

Other examples of this approach are the idea of ‘web wallets’²³ in Bitcoin. A web wallet is managed by a *TTP*. This means that it holds the public and private key pairs of the users registered to its system. Such control means that the *TTP* could potentially run away with a user’s funds or unwantedly act on their behalf. Increased usability is the main benefit from this trade-off.

It may seem like a contradiction to provide an intermediary to a network that is intrinsically meant to be decentralised, and it is. The main reason for this design choice is to aid the adoption of the technology. Ensuring usability is a large driving force for any new technology to prosper.

In this paper, the above technique is referred to as the ‘hybrid approach’. If the intermediary is truthful and operates as expected then its users are still able to profit from the same benefits as users employing the ‘*total decentralisation*’ approach.

Unilog will provide a hybrid approach by hosting a server that acts as a full node on the Ethereum network. It will manage keys and hold a remote copy of the entire blockchain, so that issuers do not need to worry about their key’s security or operating a full node locally. Key components of Unilog’s behaviour will be mathematically verifiable using a cryptographic hash function as a proof. These verifiable properties are also facilitated by open sourcing the software. This is discussed further in section 4.6.

²² <https://metamask.io/> [Accessed: 21-04-2017]

²³ <https://www.cryptocompare.com/wallets/guides/what-is-a-bitcoin-web-wallet/> [Accessed: 21-04-2017]

3.2.2 | Total Decentralisation Approach

In contrast to the hybrid approach, Unilog also provides a framework for parties to circumvent the usage of a *TTP* while offering the same functionality.

In essence, a hybrid approach provides a simpler way of interacting with the Ethereum smart contracts. Total decentralisation offers more enthusiastic users two opportunities:

- **Fork Unilog's open sourced code** - The issuer is in control of their own keys and blockchain/*IPFS* interactions. They will effectively act as a Unilog administrator without any involvement from Unilog e.g. A university would host their version of the code base. They could publish transcripts directly to the blockchain. It is not within the universities interest to publish illegitimate transcripts. Therefore, the feasibility of such an architecture design still holds.
An advantage of this approach is that the code base can be tailored to suit the needs of a specific issuer. In addition, they can keep the user interface that Unilog supplies, which greatly reduces the need for in-house technical capability.
- **Bespoke Implementation** - The issuer decides how transcripts are uploaded to the blockchain and how interactions with a decentralised file system of their choice are managed. Undoubtedly, this approach would require a significant amount of technical knowhow. One benefit is, if they opt to use *IPFS* and conform to the interfaces of the Unilog smart contracts, then all verifiers using the Unilog service would still be able to interact with an issuers bespoke implementation. They would also need to provide a way for verifiers to interact in a completely decentralised way, possibly by creating their own *DApp*. An advantage of this approach is complete control, while being given the choice to tap into Unilog's user base.

In both opportunities, agreeing to use a common interface will allow all implementations to be interoperable.

It is assumed that the system described hereafter is of the hybrid approach. The reasons are that this approach covers every aspect of the Unilog system and so offers a comprehensive evaluation of the migration that can be made. After understanding how the hybrid approach works, it is trivial for an issuer to employ a totally decentralised approach.

3.2.3 | Functional and Non-Functional Requirements

To capture the intended behaviour of the Unilog system, functional and non-functional requirements have been created in the form of use cases. The baseline functionality that allows Unilog to compete in this domain is defined. Features that differentiate the system from competitors' products are also detailed throughout.

Four actors are involved, these include the three mentioned previously, i.e. verifier, issuer and owner and the Unilog system administrator, who has control over various events that occur in the system.

The entire set of functional and non-functional requirements can be found in Appendix 2.

4 | System Implementation

In this chapter, the implementation of the Unilog prototype will be discussed. Design decisions are clarified and a tour through the system demonstrates the usability of the overall solution.

A key takeaway from this chapter will be an understanding of the migration occurred between the current and new Unilog system.

4.1 | Architecture Overview

The architecture diagram below will be a reference point for various discussions in this chapter. The main actors of the system are depicted and labelled. Their interactions with the system are also shown at a high level. As previously mentioned, this implementation follows the hybrid approach.

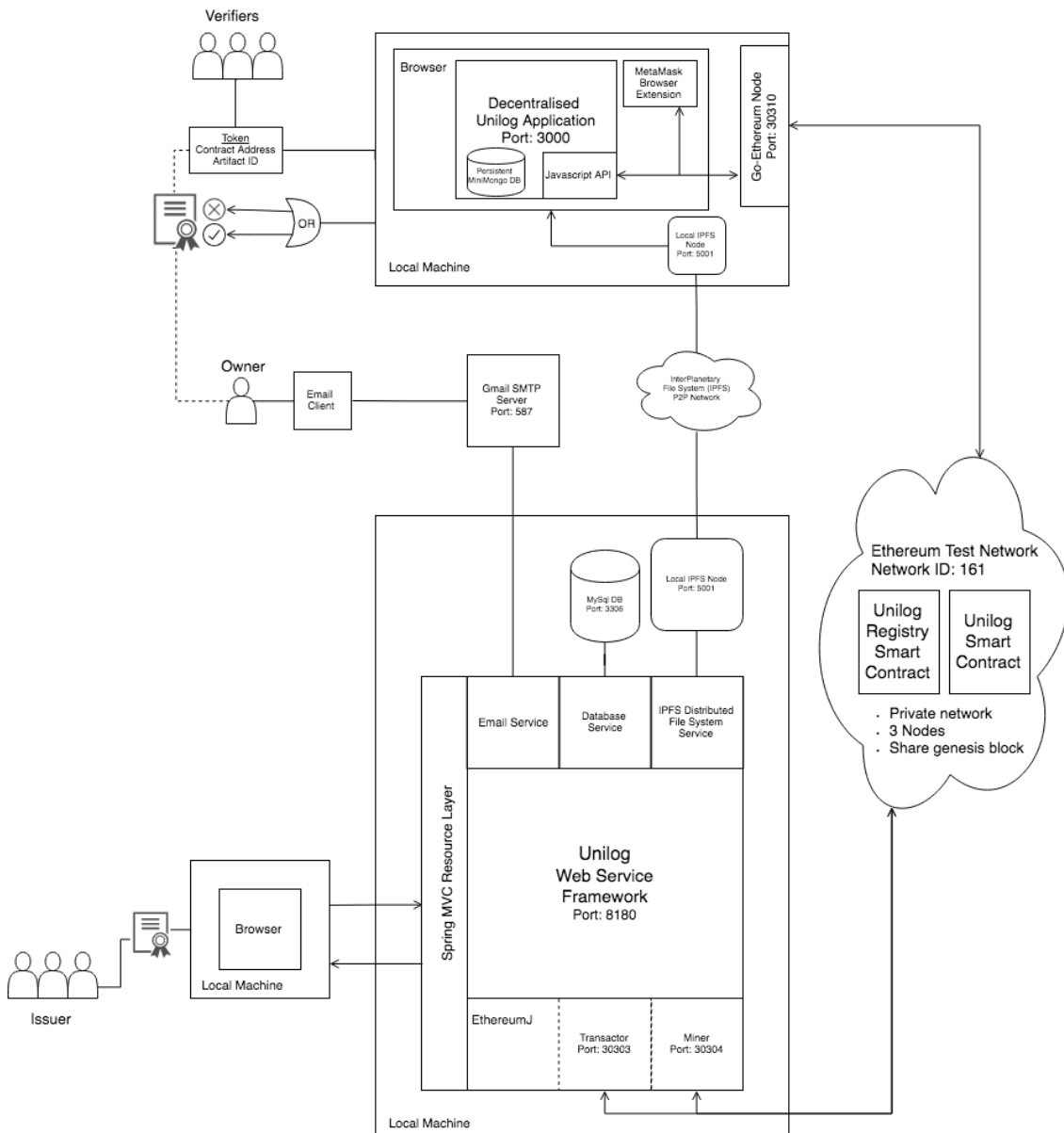


Figure 4. Architecture diagram showing the main components of the Unilog hybrid approach implementation.

4.2 | What exactly do we decentralise?

After studying how the current solution works for verifying academic transcripts. We need to know which components of the current architecture must be changed. In other words, how can the current solution of academic transcript verification be reengineered to be hosted on the Ethereum network? More specifically, what core functionality needs to be decentralised to make the new solution work.

There are several constraints that need to be fulfilled for an application of this nature to be adequate. The more of these constraints that can be emulated on the Ethereum platform the higher degree of decentralisation we will obtain. If any of them require an intermediary, then the application is only partially migrated.

The following are constraints that need to be fulfilled to produce a minimum viable academic verification service, of any architecture style:

Role Restrictions

A transcript owner should not be able to add their own transcripts. Only external, accredited institutions may do so. This is to prevent bogus transcripts being advertised as legitimate. More generally, all parties involved must be confident that the data being verified can only have ever been published by one source. Non-repudiation is the desired assurance that an issuer cannot deny having issued a transcript.

Ownership

A party that is a recipient of a transcript must be able to have complete control over its distribution. They must also be able to remove a transcript that is associated with their identity.

Exclusivity of Data

There should only be one, trustworthy copy of the transcript data in the network. This means that the same data should be referenced with every verification and there is no ambiguity around which copy is the legitimate copy. Efforts to trick verifiers into using a malicious imitation of the service must also be protected against under the *exclusivity of data*.

A decentralised application concerned with the verification of academic assets must satisfy the above constraints fully to be deemed viable.

4.3 | The Test Network

In previous chapters, the Ethereum platform was introduced as a blockchain that is publically accessible for anyone to use. In reality, there are blockchains where access permissions are tightly controlled. These are dubbed private blockchains (sometimes referred to as test networks by developers) [31]. When implementing Unilog, it was vital to use a test network because it is free to execute transactions. Testing smart contracts functionality by purchasing ether with real money would not be cost-effective. To obtain ether to use in development, a party can mine blocks on the network and earn the associated rewards. Block difficulty is lowered so that ether can be earned quickly.

Figure 4 shows that the Unilog prototype was built on top of a test network containing three separate nodes. For a test/private network to be assembled, all nodes need to share the same genesis block and be connected to the same network ID (161 in our case).

The *genesis block* is the first block in the blockchain. Among many modifiable parameters, you can specify the initial account balances of the earliest accounts to be made on the network. Under the controlled conditions of the Unilog test network, it was anticipated that there would one account concerned with creating state changing transactions. This was the Unilog account, more generally the admin controlling the Unilog server. To save time, it made sense to load the Unilog account with funds via the *genesis block*. Enough was deposited so that the account would not need to be funded again throughout development.

```
{
  "alloc": {
    "31e2e1ed11951c7091dfba62cd4b7145e947219c": {
      "balance": "1606938044258990275541962092341162602522202993782792835301376"
    }
  },
  "nonce": "0x0000000000000000",
  "difficulty": "0x100000",
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "timestamp": "0x00",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extraData": "0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbb7a38e1e50b1b82fa",
  "gasLimit": "0x1000000000"
}
```

Figure 5. The genesis block of the test network that was used to develop Unilog. It includes the initial balance allocated to the various accounts in the network, along with the mining difficulty, gas limit and other parameters.

The hex string 40 characters long, “31e2e1ed11951c7091dfba62cd4b7145e947219c” is the Ethereum account address of the Unilog account and it has been funded with a large amount of ether which is denoted in *wei*.

4.3.1 | Unilog Nodes

As outlined in Figure 4, the Unilog web service is an application written in Java. Many of its functional and non-functional requirements have been discussed section 3.2.3. Knowing that the service must act as a full node, a protocol implementation must be chosen. Previous research suggested that the obvious choice is the reference implementation offered by Ethereumj. The Unilog Spring Boot web application can now act as a full node on the network.

To create a separation of concerns between the process of mining and creating transactions, the choice was made to run two nodes in the Java application. One which was solely responsible for mining transactions on the test network and the other for creating transactions. Both nodes use the same *genesis block* and network ID, therefore they will be configured to connect to the same test network. It was also configured so that each node has exactly one account residing in its state and each node will use these accounts for any network interactions. Like Bitcoin, a transaction can only be deemed valid if it was cryptographically signed by the accounts private key, effectively producing digital signature.

The *role restrictions* constraint that was introduced in section 4.2 can now be reviewed. A benefit of Ethereum is that public key cryptography is implemented in the protocol by default. To satisfy the *role restrictions* constraint for Unilog this convenient property needs to be exploited. The digital signature associated with the transaction provides proof of origin.

Therefore, transactions that are signed only by an accredited institution or Unilog should ever be accepted to add a new transcript. This is the central security procedure in delivering trustworthy transcript data. However, it poses an important question, what exactly oversees the acceptance these transactions? The answer lies with Smart contracts (section 4.4).

4.3.2 | Local Client Ethereum Node

To participate in the Ethereum peer-to-peer network, one must either host a node locally on their machine or trust a central server to do so for them. It is advised that verifiers host a local node so that they can be sure that their blockchain is genuine. The architecture in figure 4 allows verifiers to completely bypass the central Unilog server for the verification of a transcript. This means that, for a verifier, the availability non-functional requirement is realised. A reliance on the resilience of the Ethereum network ensures this. So, for a verifier to successfully verify a transcript using the Unilog framework they'll need to:

- **Create an Ethereum account** – A verifier must own an account to interact with the blockchain. Currently the Unilog server has the capabilities to provide all users with Ethereum accounts. But in the interest of security and minimising reliance on a *TTP*, the Unilog framework suggests that users create their own accounts. Also, to enhance usability of the framework, Unilog endorses the usage of the *MetaMask* browser extension. This transforms the verifiers normal web browser (i.e. Google Chrome) into a *DApp* explorer. Therefore, it is not necessary to download Ethereum's official *DApp* explorer (Mist²⁴) to view content on the decentralised web. Which would, without a doubt, increase complexity and thus hinder the user experience.

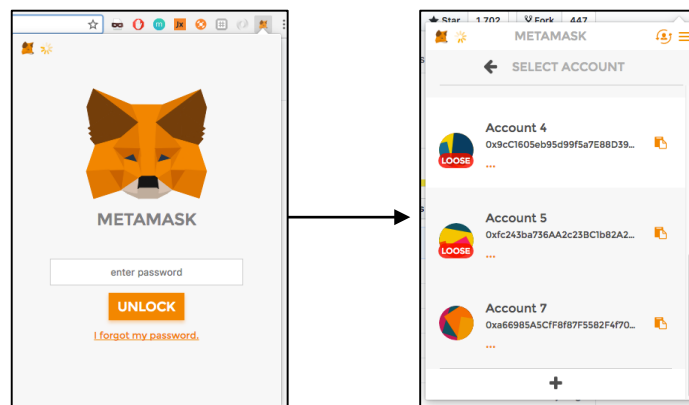


Figure 6. Shows the login process to a MetaMask vault. You can easily create a new account by clicking the plus icon at the bottom of the accounts list. You can quickly switch accounts by clicking on already created accounts.

A user would download the *MetaMask* extension from the Chrome web store, setup a vault and then create accounts that are all recoverable from the seed phrase that the extension provides. The account vault is encrypted and locally stored in the browser, so no account data is shared with any intermediaries. Verifiers now have hassle free account management.

If a non-admin account attempted to add a transcript to the system, it would be disallowed because of the *role restrictions* enforced by smart contracts (section 4.4).

²⁴ <https://github.com/ethereum/mist> [Accessed: 23-04-2017]

- **Choose their local client node implementation** – Again to operate the Unilog decentralised application, a verifier must be connected to a node. The implemented architecture shows that a Go-Ethereum (Geth) node is running locally and is acting as the access point to our Ethereum test network. This Geth node is configured to operate on the same test network as the Unilog nodes to mimic what it would be like on the public network. After specifying the same *genesis block* to the Geth node, it is configured further with more parameters to start operating.

```
geth --identity "GethNodeUnilog" --rpc --rpcport "8545" --rpccorsdomain "*" --datadir
Library/Ethereum/dataDir --nodiscover --port "30310" --rpcapi "db,eth,net,web3" --networkid 161 --unlock 0
console
```

The above command injects several configuration parameters to the local Geth node before it syncs with the test network. Geth was chosen as the implementation because it is currently the most commonly used. Accounts can also be made through the Geth console and then imported to the *MetaMask* vault as ‘loose accounts’. Note that the seed phrase will not recover these ‘loose accounts’.

When a transaction is constructed within the Unilog decentralised application, it will be signed by whatever account is currently ‘active’. If the *MetaMask* extension is disabled, the active account will be the unlocked account in the Geth node. If the extension is enabled then it will override a global JavaScript object (web3) causing the active account to be the selected account in *MetaMask*.

A verifier can choose to bypass *MetaMask* entirely if they know what they’re doing and are comfortable with the technology.

For users who aren’t as technically inclined, *MetaMask* offers their own nodes on a remote server so that verifiers do not even need a local client implementation.

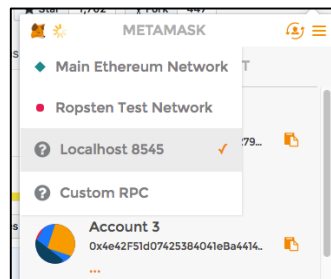


Figure 7. Shows the network connection options that *MetaMask* provides to users. Choosing localhost 8545 ensures that you hold your own version on the blockchain locally on the machine. We can also see that you can connect to the Main Ethereum network and also Ropsten, Ethereum’s test network.

Again, in the interest of security and reducing the reliance on a *TTP*, this prototype chooses ‘Localhost 8545’ as the blockchain access point. Further inspection of the command used to start the our Geth node shows that it exposes a HTTP *JSON-RPC* API at port 8545. The *MetaMask* extension is now connected to the Geth node.

To synchronise the blockchain on our private network, the Geth node must connect to at least one full node via a predefined discovery protocol²⁵. Specifically, it is configured to connect to the Ethereumj node concerned with creating transactions. This node holds the complete state of the blockchain, therefore it is an ideal peer to connect to.

²⁵ <https://github.com/ethereum/wiki/wiki/%C3%90%CE%9EVP2p-Wire-Protocol> [Accessed: 23-04-2017]

4.4 | Decentralised Code as Smart Contracts

Smart contracts host and execute the vital operations that would normally be delegated to a *TTP*. This allows for the core application code to become fully decentralised, every node in the test network must agree on the current state of the smart contract storage. As a result, node can reach a consensus about the transcripts and their ownership properties on the chain.

Smart contracts offer a large part of the solution for building a decentralised academic verification service. They impose *role restrictions* as well as *ownership* constraints. Therefore, they satisfy two out of the three checkboxes needed to pass as a minimum viable solution. It is imperative that only certain parties can add transcripts and that transcript owners have complete control over the information held about them.

To achieve this, the Unilog framework defines two different smart contracts which rely heavily on the ‘Authorization’ design pattern [37] which can be found in Appendix 3. They’re extensively annotated with helpful explanations.

The way in which a Unilog smart contract decides whether an account can invoke a function successfully is by inspecting the sender of a transaction. If the sender is who it expects, then it’ll allow the function to be executed. For example,

```
address private unilog = msg.sender;
```

this statement is executed when the smart contract is successfully published to the blockchain. The idea is analogous with constructors in Java, an objects preliminary state is initialised in the objects constructor when it is created. This means that the contract creators account address (*msg.sender* stores the current account address of the calling transaction) is saved in the state storage of the contract against a variable called *‘unilog’*.

With this functionality, the smart contract can enforce powerful restrictions on the operations that can be executed by different accounts.

```
function addArtifact(string artifactAddress, uint validFor) onlyBy(unilog) returns(uint)
```

The above defines the type signature for the *‘addArtifact’* function on the *‘Unilog.sol’* smart contract. The declaration is somewhat intuitive to understand but the use of function modifiers may be a new paradigm i.e. the *‘onlyBy(unilog)’* segment of the declaration. It defines a function that must be invoked before the *‘addArtifact’* function can be executed. This is a common pattern for restricting access in smart contracts. The same logic that is contained within the body of the *‘onlyBy’* function can be extracted and placed in the body of the *‘addArtifact’* function. Though because it is used in numerous functions, it makes sense refactor the code into its own method and use Solidity’s function modifiers feature.

```
modifier onlyBy(address _account) {  
    if (msg.sender != _account)  
        throw;  
    _;  
}
```

A simple check is performed to see if the parameter is not equal to the sender of the transaction. For the example of adding a transcript, this function will check if the sender of the contract is not Unilog.

If this is true then the sender will fail at adding a transcript, otherwise they will succeed. Again, it is important to remember that you can only send a transaction from an account if you are able to create a valid digital signature; By holding the private key of that account. It is crucial that Unilog keeps their private key safe to stop malicious users adding transcripts which would reduce the integrity of the entire solution.

The admin of Unilog is required to publish two smart contracts. We've established that the account publishing Unilog contracts, will be given access rights to successfully invoke restricted functions. In addition, smart contracts can also enforce *ownership*. It is the responsibility of Unilog to give the correct access rights to the owner of the transcript. This functionality is provided using the same restrictions discussed above. Unilog would execute this function to set the transcript owner.

```
function assignArtifactOwner(address artifactOwnerAddress) onlyBy(unilog) returns(bool)
```

Once set, the transcript owner has the access rights to successfully control the information stored about them. In other words, they can delete any data association.

```
function deleteArtifact(uint artifactId) onlyBy(artifactOwner) returns(bool)
```

The '*Unilog.sol*' smart contract is accountable for the storage of transcripts belonging to a transcript owner in the system. Therefore, one smart contract is created for each transcript owner. The '*UnilogRegistry.sol*' smart contract acts as a whitelist for all valid '*Unilog.sol*' contracts. Only Unilog admin account can change the storage associated with the registry. Therefore, an imitation of the '*Unilog.sol*' smart contract containing a phoney transcript cannot be considered legitimate by any verifiers who query the registry before verification. This is because the address of the phoney contract will not be included into the registry as Unilog never accepted the transcript to begin with.

Issuers also have the option of adding an expiration date to transcripts, this utilises Solidity's 'Time Units' feature which relies on block timestamps. When a verifier checks a transcript, its expiration date is inspected and the relevant data is returned.

4.5 | Transcript Storage with IPFS

All entities in the test network need to be connected to an *IPFS* node. This is to either publish new content to the network, or to retrieve existing content. The issuer node, being the Unilog admin, is responsible for publishing new content to the distributed file system. The general algorithm for publication and retrieval is as follows:

1. Issuer uploads a qualification containing transcripts.
2. Each transcript is extracted into a Java object.
3. The Java object is serialised into its equivalent *JSON* representation.
4. Unilog admin publishes the *JSON* string as a file to the IPFS network.
5. The hash of this transcript file acts as its content address. It can be used to retrieve the transcript from the distributed file system at any time.
6. The hash is added to the '*Unilog.sol*' smart contract via a transaction. This represents a transcript associated with an owner.
7. Verifiers can now use the content address of a transcript to salvage the authentic transcript data from the network. The verifier portal, which is accessed through the *DApp*, will parse the *JSON* returned from the network so that it is displayed within the verifiers browser.

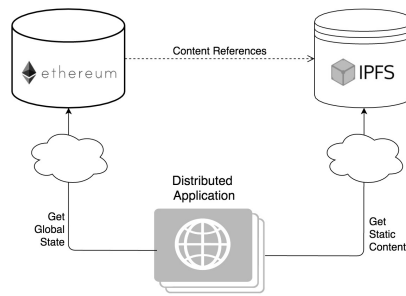


Figure 8. Shows the basic architecture of a DApp and how the content reference is stored in Ethereum so that the data can be retrieved from IPFS²⁶.

Storing the unique Multihash¹⁸ identifier in each smart contract allows the *exclusivity of data* constraint to be satisfied. It is highly unlikely that there will be a collision with the hashing function that *IPFS* uses. Under the hood, it uses SHA256 which means that there are 2^{256} different possible hash values. It would be highly impractical to discover a file that has the same hash as a given transcript, so much so that it's acceptable to ignore the possibility such an event ever occurring.

For a verifier to identify the exact location of the content address on the blockchain, a transcript owner must supply them with the information. The Unilog framework specifies that transcript owners are given tokens for each transcript. These tokens contain all required data to find the transcript in question, including the contract address, artifact ID and registry address. See the '*verifyArtifact(uint artifactId)*' function in appendix 3.

4.6 | Provably Honest

Thus far, this chapter has been exploring the implementation details of a hybrid approach for creating a decentralised application. It was briefly mentioned that for this approach to be trusted, users should be able to verify the server's actions whenever they like.

To facilitate such a feature, the Unilog server should be able to provide proof that they acted honestly with the data that was uploaded by an issuer, hence the term provably honest (see the Oraclize²⁷ Network Monitor for an example of something similar). Suppose an issuer is sceptical of the Unilog server and they are of the opinion that the uploaded transcript data, has been altered in transit to the blockchain and *IPFS*.

How can an issuer check if Unilog tampered with the data?

Open sourcing the code base does not suffice, because the data can still be susceptible to interception and therefore, any modifications to the data may remain undetected.

We know that the *IPFS* content address is a summarised representation of the transcript in the form of a *JSON* object. We also know, that if even one character in the *JSON* object was altered then the generated hash value would drastically change.

With this in mind, it is then reasonable to assume that a transcripts hash value can be predetermined by an issuer. Provided the process is well documented by Unilog and the code is open source, then the effort involved for an issuer to check the Unilog server's honesty is negligible.

Steps required to check the honesty of Unilog server:

1. Given a transcript, apply steps 2 & 3 from section 4.5 on its data.
2. Use an *IPFS* node to generate the predetermined hash value of this data. This can be done without publishing to the network with the following command.

²⁶ <https://karl.tech/content/images/2016/03/dapp-basic-structure.png> [Accessed: 23-04-2017]

²⁷ <http://app.oraclize.it/service/monitor> [Accessed: 23-04-2017]

ipfs add --only-hash <Enter transcript file containing data here>

3. Check if this hash is the same as the hash that was published to the blockchain. If they match then the server was honest. If they don't then the issuer knows that the server or somebody else tampered with the data.

The issuer may opt to ditch the hybrid approach at this point and stick to the totally decentralised approach. Where they have total control over the transcript data that is published.

4.7 | Decentralised Application – The Smart Contract Interface

To allow users to interact with Ethereum smart contracts, it is common to develop an application interface. This front-end would also provide additional functionality that the Ethereum blockchain does not offer e.g. connecting to an *IPFS* node. The combination of the blockchain and the distributed front-end code can collectively be thought of as the decentralised application (*DApp*).

The decentralised Unilog application runs in the client's browser as seen in Figure 4. It has local storage in the form of a MiniMongo DB, this helps with saving state for user sessions. Each instance of the application can connect to an *IPFS* node which assists a verifier on the retrieval of data from the *IPFS* network.

Only when the *DApp* is entirely decentralised can it actually be thought of as a *DApp*. If the front-end code is hosted by a central server in the final solution and not by a distributed network like *IPFS*, then it can only be considered partially decentralised. During development, it is advised to host this code on the localhost as it requires much less effort. If the prototype was going into production tomorrow, the next logical step would be to upload the front-end code to *IPFS* in the exact same way as our transcripts were uploaded in the previous sections. This would mean that the code would be on an entirely distributed network and thus breakdown the notion of a partly decentralised application. *IPFS* also provides a decentralized naming system called *IPNS*, which gives users a way to easily access web applications via human-readable names.

Without a front-end solution, users would have to manually interact with the Unilog smart contracts on the Ethereum blockchain. It is desirable to provide a user-friendly way around this. It is made possible with HTML, CSS and a JavaScript API that communicates the local Geth client, namely *web3.js*. Being able to communicate with the Geth client allows the *web3* object to act on your behalf because Geth knows your private key.

If you're using the *MetaMask* extension for connecting to a remote node on their server, then global *web3* object will be overridden. This is so the new *web3* object redirects communications to *MetaMask* and not a local Geth instance.

The next section gives an insight into how the *DApp* looks and works. For a more comprehensive overview of the Unilog interfaces, refer to the user manual in Appendix 1.

4.7.1 | Fraud Detection in the Verifier Portal

Given the scenario where the Unilog admin has uploaded a transcript for a transcript owner. To verify this transcript without a *DApp*, the verifier would need to perform many fine-grained tasks that are susceptible to human error. Verifications can be performed within a matter of seconds through the verifier portal.

The transcript owner has submitted their CV to a verifier with the Unilog token attached. The verifier wants to see if the information shown on the CV is correct.

The CV contains the following details:

- Qualification: *Degree in Computer Science from QUB, 1st class honours.*
- Email Address: *jsmith@qub.ac.uk*
- Unilog Token:
Content Address: *0ebac36cc4cd8e6824ca0affb239f04704cdf86d*
Artefact ID: *0*
Unilog Registry: *53277dd58eade034d9e32fdaf8df7b0d5cfa21d1*

The verifier takes the following steps:

1. Navigates to the verifier portal on the *DApp*.

Verifier Portal

Dashboard

Want to verify a candidates token against the block chain?
Enter their token information and simply click verify.

Token

Contract Address:

Artifact ID:

Unilog Registry:

Result

Please enter a valid token above to view the artifact.

2. Enters transcript owners token data.

Verifier Portal

Dashboard

Want to verify a candidates token against the block chain?
Enter their token information and simply click verify.

Token

Contract Address:

Artifact ID:

Unilog Registry:

Result

Accredited Institution:

Title:

3. Inspects the result returned from the verification.

Verifier Portal

Result

Accredited Institution:

Title:

Qualification Code:

Transcript Reference:

Recipient EmailAddress:

Classification:

Additional Information:

Overall Result:

This data has been obtained from a credible Unilog source on the block chain. Check it against whatever data the token was attached to. You can now determine the integrity of the information you were given.

4. Realises that John Smith lied on their CV.

John said that he obtained a 1st classification in his Computer Science degree. However, the verifier portal states that he in fact received a 2.2 classification.

We can see that the application is very user-friendly. The verify button acts as an abstraction for many of the topics discussed throughout this paper. Prior knowledge of blockchain technology is not necessary to be able to operate this application.

5 | System Evaluation

Up until this point we have seen how an application which is historically centralised can be migrated to a solution that doesn't rely on any intermediaries. This chapter will objectively evaluate the Unilog framework.

5.1 | Testing

To provide a meaningful evaluation of the hybrid approach implementation, it is important to show that the system is working as intended.

5.1.1 | Automated Smart Contract Unit Tests

Verifying that the smart contract code is functioning as expected allows for further evaluation of the system. Section 4.4 demonstrated how smart contracts provide the core functionality at the heart of the decentralised application. It is therefore crucial, to test this module extensively.

At the time of writing, there are several different ways for testing smart contracts that are intended for Ethereum. One of the more popular choices among *DApp* developers is Truffle. Truffle describes itself as a testing framework and asset pipeline for Ethereum²⁸. However, it was decided to deviate from this popular choice and choose something that was more in line with the current development environment of Unilog.

As well as offering a client implementation of the Ethereum protocol, Ethereumj provides what they call a 'Standalone Blockchain'²⁹. It accelerated development time, by allowing for automated unit tests to be written. The Unilog smart contract code was constantly confirmed to be working as expected every time the Java web service code was built.

For this reason, any errors that may have altered the behaviour of the smart contract were always found and fixed early.

The exact test cases and results of these automated unit tests can be found in Appendix 4. In the interest of brevity, all test cases passed, meaning everything is working as expected.

5.1.2 | The Graphical User Interface

There are two graphical user interfaces in the Unilog framework originating from the Java web service and the *DApp*. A key research question was to see if a user-friendly solution could be constructed using the deliberated technologies. This was on the thought that if the system was easy to use, then the technology would be more attractive for new users. The user manual in Appendix 1, shows a detailed journey of the system. It is clear from this that the usability of the interfaces is no different the from user-friendly websites found on the internet today.

Again, the solution that was built defines a prototype framework that is open to interpretation by users. For instance, components can be changed to suit the specific needs of an issuer/university. Testing the usability of the user interfaces was therefore seen as unnecessary. At the end of the day, the solution is only a prototype.

²⁸ <https://truffle.readthedocs.io/en/latest/> [Accessed: 25-04-2017]

²⁹ <src/main/java/org/ethereum/util/blockchain/StandaloneBlockchain.java> [Accessed: 25-04-2017] (Ethereumj code base)

5.2 | Limitations and Concerns

The Unilog framework is susceptible to some limitations. Whether these are inherited through the core technologies at Unilog's foundation, or newly spawned from the design choices during implementation. Both are discussed.

5.2.1 | Scalability

Synchronisation with the blockchain is an arduous task. Spinning up a node can take days, even weeks to complete on the main network. Sections of the Ethereum blockchain, due to the *DDOS* attacks in 2016³⁰, are not advised to be synchronised at all on a hard drive because it takes too long. This problem has raised questions over Ethereum's scalability. If the platform became the widespread norm for hosting enterprise applications, would the blockchain become too big, and thus hinder the user-experience?

Opting for an account-based blockchain was a step towards managing the scalability of Ethereum as it used less space when sending many transactions from a single account. The Bitcoin *UTXO* approach wasn't as efficient at this task. The core problem with scalability is that every node on the network must verify every single transaction. This means that a blockchain cannot process more transactions than a single node can handle. Ethereum is currently at somewhere between 7 and 15 transactions per second [34]. To be competitive, nodes need to be able to handle a much greater throughput.

Such capabilities are possible, we see services like PayPal that can cope with a very high throughput of transactions (in the region of hundreds of thousands). However, it requires a large amount of resources to set up, such that only a handful of participants can do so. This situation would be a backward step in achieving a decentralised model. Smaller nodes would end up having to trust that the blockchains state is valid because they do not have the resources to check themselves [33]. Larger nodes therefore have the power to manipulate blocks and perform changes to the state that isn't in accordance with the consensus protocol.

Sharding is a potential solution that can be introduced when Ethereum has implemented *Casper*, their *PoS* consensus algorithm. It sets to split the blockchain into separate states called shards without losing security. Sharding randomly selects nodes to process different transactions in parallel, allowing the network to become more powerful as the number of nodes increases. The Ethereum platform would then be able to process enough transactions to compete with systems like PayPal and clients wouldn't need to be exposed to slow synchronisation times. In a recent interview, Buterin said that it will take years to complete [34].

In the future, if this limitation isn't tackled then issuers on the Unilog *DApp* may have the burden of waiting a long time to have their request processed. Increased activity on the Ethereum network would slow down the response times of any application built on the network. It would also favour parties willing to pay more per computational step on the network i.e. stating a higher gas price.

³⁰ <https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/> [Accessed: 26-04-2017]

5.2.2 | Difficulty Bomb

The block difficulty on the current Ethereum blockchain is set to increase and at a specific point, ‘explode’ so that it is extremely difficult to create any blocks at all.

As previously established, Ethereum intends to implement *Casper*, their *PoS* consensus algorithm. To move from the old *PoW* algorithm, a hard fork of the blockchain needs to happen. This means that the community will need to switch from the old blockchain to the new one to maintain its security. To encourage this switchover, the Ethereum developers decided that there would be an imminent barrier to profiting from the old blockchain. This was in the form of what they dubbed an ‘Ice Age’ whereby block time would gradually increase [35]. That is, it would become increasingly difficult to produce new blocks up until a point where it was impractical to even try. Miners would then not be able to profit from the network and thus, shift to the new Ethereum blockchain, which is exercising the *Casper* algorithm.

Hosting a *DApp* on the old network will eventually become unrealistic. The block time will be so slow that the applications response times will leave them inferior to their centralised counterparts.

5.2.3 | Immutability

In June 2016, Ethereum was the subject of negative publicity when a programming error in a well-known smart contract, dubbed ‘*The DAO*’³¹, resulted in the huge loss of investors’ money. Its purpose was to operate like a venture capital fund for the crypto and decentralised space [36]. Having no central authority was an attractive thought for investors and this led to the project’s huge success. It raised over \$150 million through 11,000 members, making it the largest crowd funded project ever.

A hack that exploited the error caused \$70 million worth of ether to be drained from ‘*The DAO*’ within a few hours.

Note that the hack wasn’t due to a vulnerability in the Ethereum protocol, it was induced via a human programming error. The scenario is comparable with an unsecured website that is hosted on the internet, an attack on the website does not mean that the internet itself exposed the attack vector, but instead the negligence of the developers.

To reverse the hack, the Ethereum core developers got involved along with most of the community. At this point it is important to remember that blockchains are meant to be censorship resistant. They’re meant to be immutable whereby no one authority can change the events that come to pass. This is exactly what happened with regards to *The DAO* attack. In the end, core developers decided to hard fork the blockchain just before *The DAO* attack occurred. Essentially replacing the funds back to *The DAO* and in effect, changing history that was meant to be immutable. Many believe that the hard forked blockchain where ‘*The DAO*’ attack never happened is symbolic of a blockchain that is clearly not immutable. Ethereum now has two blockchains. The hard forked blockchain and Ethereum classic, which was the original chain before the hard fork. Ethereum classic supporters believe the old blockchain needs to be used for the following reason.

*“Blockchain-systems should always adhere to three characteristics: openness, neutrality, and immutability. Without these characteristics, a blockchain is nothing but a glorified database.”*³²

³¹ [https://en.wikipedia.org/wiki/The_DAO_\(organization\)](https://en.wikipedia.org/wiki/The_DAO_(organization)) [Accessed: 26-04-2017]

³² <https://bitcoinmagazine.com/articles/rejecting-today-s-hard-fork-the-ethereum-classic-project-continues-on-the-original-chain-here-s-why-1469038808/> [Accessed: 26-04-2017]

If Unilog was hosted on a blockchain that lacked immutability, then transcripts could potentially be revoked from genuine recipients. This is not desirable for such an application.

5.2.4 | Smart Contract Design

An investigation into the classification of smart contract design patterns found that there are two notable design choices very much prevalent in most smart contracts [37]. These are the ‘Authorization’ and ‘Termination’ patterns. The former is a core part of the Unilog solution, where it enforces the previously stated requirement of *role restrictions*. However, Unilog has not opted to use the latter ‘Termination’ pattern. Considering the immutable nature of blockchains, a smart contract cannot be deleted. No central authority can act as an administrator and erase them. Therefore, instead of removing a smart contract, it is common to disable its functionality. Solidity’s built in constructs can handle this but developers can also manually program such a feature. The Unilog registry would benefit from the ‘Termination’ pattern, because there is currently no way to disable it. It is seldom that Unilog would need to disable a registry contract, but having the option for such an eventuality is desirable. The same pattern could also be used in the transcript contracts (*Unilog.sol*) but it’s not as necessary. The relationship between it and the registry allows for efficient disabling of contracts by one party (Unilog admin). Removing a contract’s address from the registry will automatically deem it to be unauthentic and thus unconsidered by future verifiers.

Any state changing transaction that is executed cannot return a value to the user asynchronously without using an ‘event’ [38]. For example, the ‘*addArtifact(X,Y)*’ function currently changes the smart contracts state by adding a transcript. The method’s type signature says the it returns a ‘uint’, this is currently used only for unit testing the functions with the Ethereumj library. When executing this transaction in situ, the only value returned would be the transaction hash.

Using ‘events’ in the short term may increase complexity for the developer, but the benefits from a user-experience perspective would eventually outweigh any initial setup hassle. A *DApp*, as a result, would become more responsive to happenings inside the smart contract.

Events are a topic that have not been extensively covered as part of this paper. Mainly because an equally effective solution could be achieved without them. Their benefits are not restricted to offering a seamless way to achieve better usability. They can also act as a cheaper storage mechanism [38]. However, this option was decided against, accounting for the nature a transcript verification use case. It would have required an intricate solution to ‘delete’ single transcripts when they’re stored as events, as opposed to when they reside directly in contract storage.

5.2.5 | Data Encryption

A user's transcript data is currently stored in plaintext on the *IPFS* network. Knowing the transcripts content address, allows anyone to interpret its data. It can be argued that this data isn't extremely sensitive because people normally want to advertise their qualifications. However, this general assumption cannot be made, so a more desirable solution should be crafted.

According to the *IPFS* white paper [29], the protocol intends to provide '*Object-level Cryptography*' where encryption and decryption is handled directly. After communicating with the core developers, it was discovered that this feature isn't likely to be available soon and they suggest that external tools should be used in the meantime. The topic is also listed as in an issue titled, 'Top Topics that Need Better Documentation'³³.

There are various methods to solve this problem and for our use case a solution would rely heavily on secure distribution of encryption/decryption keys. One proposal for the Unilog framework could be as follows:

1. Instead of the Unilog administrator publishing the transcript to *IPFS*, they'll publish a hash of the unencrypted transcript to the 'Unilog.sol' smart contract. This will act as a form of digital signature to ensure the integrity of a transcript.
2. The unencrypted transcript is then sent to the transcript owner. At this point it is their decision to upload the transcript to the *IPFS* public network. Using basic symmetric encryption, they can encrypt the transcript before publishing to *IPFS*. The generated content address would then be added alongside the hash that the Unilog admin posted.
3. Next, the symmetric key would then be given either directly in person to a verifier or via some medium which makes use of hybrid cryptography.
4. The verifier would retrieve the encrypted content from *IPFS* and then decrypt it with the symmetric key. To be sure that the data wasn't tampered with by the transcript owner, the verifier can query the hash submitted by the Unilog administrator. Using the same strong hashing algorithm, the verifier can tell whether the transcript was altered before it was submitted to *IPFS*.

This is only one suggestion to solve the encryption concern. Not implementing it as part of the Unilog framework, meant that usability was favoured over privacy. For the purposes of a prototype this is deemed acceptable, but in a production ready system it should be condemned.

³³ <https://github.com/ipfs/community/issues/199> [Accessed: 28-04-2017] (An issue that I was referred to after enquiring about Object-level Cryptography on public developer channel)

6 | Conclusion

6.1 | Summary

This paper has answered the main question defined in section 1.4, that is, “How can the Ethereum blockchain be utilised to develop an autonomous decentralised application, that provides a way for users to verify the legitimacy of a candidate’s academic assets, while minimising the involvement of centralised institutions?”. In the quest to provide the reader with the answers to this broader question, it was necessary to break it into sub-questions. Each sub-question acted as a stepping stone to gradually build a picture of how the migration should occur. Eventually it was discovered that any party wishing to adopt blockchain technology will encounter two possible options. Both, will in turn determine the degree of their applications decentralisation. These are embodied by the hybrid and total decentralisation approaches that the Unilog framework defines. It is up to the party to choose which best suits their needs and is stated that to encourage a user to trust the intermediary introduced by the hybrid approach, it’s vital for the server to become ‘provably honest’. This assumes that users who want to use the server and therefore blockchain technology, are buying into the decentralised ethos.

For a viable verification service to be made, the essential functionality is extracted and decentralised into an Ethereum smart contract. An important cost analysis revealed that storing transcripts directly on the blockchain was not cost-effective. Leading to the use of a distributed file system that was complementary to the philosophy of decentralisation. Unilog also showed that users can benefit from the properties of *DApps* while maintaining the same level of usability as a standard web application. After researching blockchain technology, it is now clear why *DApps* exhibit properties like zero downtime, immutability, zero sign-on and resistance to collusion.

6.2 | Evaluation

The developed prototype successfully showed how an application which was historically centralised could be migrated to a solution that doesn’t rely on any intermediaries. However, given the system evaluation, it was realised that there are quite a few limitations and concerns hanging over the greater potential of Unilog and the Ethereum platform. The successful induction of *Casper* will be Ethereum’s remedy to allow the platform to mature and thus enable the hosting of more serious, mission critical applications. Due to the imminent consensus algorithm changeover, it is advised to refrain from hosting applications that are essential to the survival of a business or organisation. If the changeover doesn’t go as planned and the old blockchain remains dominant, then *DApps* are susceptible to the difficulty bomb and scalability issues, which will inevitably become a significant hindrance.

An academic verification service such as Unilog is recommended to be hosted on the network, provided it adopts a hybrid approach that is ‘provably honest’. As shown in the migration, a service like Unilog can benefit from decentralisation. The hybrid approach will provide a layer of indirection. Consequently, contributing to an organisation’s ability to safely manage backups of the applications data, so that if anything unexpected goes wrong with the Ethereum network, the intermediary will still be able to act as a failover and manage user requests.

Privacy is an evident concern with the current Unilog solution. As mentioned in section 5.2.5, the Unilog framework has not encrypted any data. Even though a viable solution was proposed, it could negatively impact the applications usability. To avoid this, work was underway to implement a notifications portal, which can be seen in appendix 1. The idea was to use the smart contract to exchange public key and transcript data securely. Notifications would be used with the help of events, to signal transcript owners when a verifier wanted to access their transcript details. A verifier would request to view an encrypted transcript on the *IPFS* network by submitting their public key to the smart contract. If permission is granted by the transcript owner, then the verifier's public key will be used to encrypt either a symmetric key or the actual transcript data (depending on the size of the data to be encrypted). Effectively allowing the verifier to view the transcript in plaintext form. These steps allow for the manual exchange of key information to be abstracted and thus maintain the usability and privacy of the application.

Evidence suggests that early adoption may come at a price, that is, if organisations do not hire the right people to develop their *DApp* then they'll be vulnerable from the outset. One key difference between code on a centralised server and code on the Ethereum blockchain is its publicity. Code hosted on a server has the option to be open-sourced or completely privatised. On the other hand, all smart contract source code is always easily accessible for public viewing. If not implemented correctly, malicious users could look for human induced vulnerabilities and terrorise the application. A prime example of which was demonstrated in the previously discussed, 'The Dao' attack. This is just another reason for holding out until the network matures. Even though contract code is unlikely to be obfuscated³⁴ (due to the expensive process of all nodes having to interpret the data), you can be sure that a more mature platform will make it easier to develop more secure applications.

Accounting for the above, the recommended best option is to test the water, build a prototype for your service and find out how you could decentralise earlier rather than later. This is only advisable to parties who recognise that their current application could be disrupted by a decentralised equivalent. Once Ethereum and related technologies like *IPFS* totally mature then they'll be ready to migrate. In the meantime, Ethereum still provides a blank canvas to be creative. So regardless of the technology's infancy, there is still an endless amount of non-mission critical applications that could be accommodated.

It's now clear that the Unilog smart contracts are at the core of a decentralised academic verification service. A holistic examination of their functionality, indicates that the code can be interpreted in a very general way. That is, artifacts can be uploaded by parties and their corresponding unique hash stored on the blockchain, essentially offering more than just a verification service. It can act as a notary service, providing a proof of existence for any document. This is useful in many circumstances as it can represent data ownership without revealing any actual data. For example, copyrighted material and patents could be added to the Ethereum smart contracts. This concept is described as proof of existence and has already been implemented on the Bitcoin blockchain³⁵.

³⁴ <https://www.quora.com/Is-all-the-code-inside-of-an-Ethereum-contract-public> [Accessed: 26-04-2017]

³⁵ <https://proofofexistence.com/about> [Accessed: 02-05-2017]

6.3 | Future Work

To build on the contributions of this paper, the next logical step would be to bring the Unilog framework into a production ready state. At this point, further research could provide an insight into how these systems cope with realistic user interaction. Metrics of how quickly requests are processed, in comparison to standard web applications would be an interesting route to follow.

A live system would help with gauging the actual costs incurred for publishing academic transcripts. These findings, and the predictions made in section 2.3.7, could offer a basis for calculating the future costs, and thus the feasibility of hosting/using the *DApp*. It would also be beneficial to automate the process of proving the hybrid approaches honesty (section 4.6). Ideally, allowing for a more user-friendly way to be sure that the server is behaving as expected.

Future work should include, addressing the shortcomings of the current Unilog framework. One noteworthy area is maintaining privacy while migrating an application to Ethereum. Implementing a seamless way to preserve privacy is crucial to keep the application competitive against centralised equivalents.

Other areas could be that could be improved upon include, delving deeper into enhancing the user experience with proficient use of ‘Events’. Reviewing methods of peer-to-peer communication for the hybrid approach, as it currently relies on intermediary email servers to communicate with users. And once *Casper* is implemented, provide metrics that demonstrate the effect on user experience from using *PoS* over *PoW*.

Every similar open sourced blockchain project, will follow a definite architectural style. Inspecting each, and compartmentalising their styles into either of the two architectural approaches defined in this paper, could prove that the engineering techniques are not limited to these two architectures. A large sample size would allow for a comprehensive list of approaches to be defined. Afterwards, a study into how an application’s architecture affects user adoption could be drawn and the most ‘successful’ style proposed e.g. A study may conclude that a service adopting the hybrid approach, receives more monthly active users than another which provides the same functionality, but follows a different architectural style.

After looking at a large sample of open sourced blockchain projects, it would be intriguing to investigate the broader application domains that *DApps* are being used for. Categorising applications into their respective domains would help with understanding what areas are most likely to be disrupted by blockchain technology. This process of classification, could be assisted by studying public smart contract code.

References

- [1]"ethereum/wiki", *GitHub*, 2013. [Online]. Available: <https://github.com/ethereum/wiki/wiki/White-Paper>. [Accessed: 27- Mar- 2017]
- [2]J. Czepluch, N. Lollike and S. Malone, "The Use of Block Chain Technology in Different Application Domains", *lollike*, 2015. [Online]. Available: <http://lollike.org/bachelor.pdf>
- [3]D. WOOD, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER EIP-150 REVISION", *yellowpaper*, 2014. [Online]. Available: <http://yellowpaper.io/>
- [4]N. Szabo, "The dawn of trustworthy computing", *Unenumerated.blogspot.co.uk*, 2014. [Online]. Available: <http://unenumerated.blogspot.co.uk/2014/12/the-dawn-of-trustworthy-computing.html>. [Accessed: 09- Apr- 2017]
- [5]S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", 2008. [Online]. Available: <https://bitcoin.org/bitcoin.pdf>. [Accessed: 09- Apr- 2017]
- [6]"ethereum/homestead-guide", *GitHub*, 2016. [Online]. Available: <https://github.com/ethereum/homestead-guide/blob/master/source/introduction/what-is-ethereum.rst>. [Accessed: 09- Apr- 2017]
- [7]U. NARIC, "Fraud: a growing problem in education, and how to guard against it", *UK NARIC*, 2015. [Online]. Available: <https://uknaric.org/2015/06/19/fraud-a-growing-problem-in-education-and-how-to-guard-against-it/>. [Accessed: 09- Apr- 2017]
- [8]"What we learned from designing an academic certificates system on the blockchain", *Medium*, 2016. [Online]. Available: <https://medium.com/mit-media-lab/what-we-learned-from-designing-an-academic-certificates-system-on-the-blockchain-34ba5874f196>. [Accessed: 09- Apr- 2017]
- [9]C. Johnson, "Basic Research Skills in Computing Science", *Dcs.gla.ac.uk*. [Online]. Available: http://www.dcs.gla.ac.uk/~johnson/teaching/research_skills/basics.html. [Accessed: 10- Apr- 2017]
- [10]C. Johnson, "What is Research in Computing Science?", *Dcs.gla.ac.uk*. [Online]. Available: http://www.dcs.gla.ac.uk/~johnson/teaching/research_skills/research.html. [Accessed: 10- Apr- 2017]
- [11]"Decentralized autonomous organization", *En.wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Decentralized_autonomous_organization. [Accessed: 10- Apr- 2017]
- [12]L. LAMPORT, R. SHOSTAK and M. PEASE, "The Byzantine Generals Problem", *inst.eecs.berkeley.edu*, 1982. [Online]. Available: http://www-inst.eecs.berkeley.edu/~cs162/fa12/hand-outs/Original_Byzantine.pdf. [Accessed: 10- Apr- 2017]
- [13]"Controlled supply", *En.bitcoin.it*, 2016. [Online]. Available: https://en.bitcoin.it/wiki/Controlled_supply. [Accessed: 10- Apr- 2017]
- [14]"Double-spending - Bitcoin Wiki", *En.bitcoin.it*, 2015. [Online]. Available: <https://en.bitcoin.it/wiki/Double-spending>. [Accessed: 10- Apr- 2017]

- [15]"Mining", *En.bitcoin.it*, 2017. [Online]. Available: <https://en.bitcoin.it/wiki/Mining>. [Accessed: 10- Apr- 2017]
- [16]S. Driscoll, "How Bitcoin Works Under the Hood", *Imponderablethings.com*, 2013. [Online]. Available: <http://www.imponderablethings.com/2013/07/how-bitcoin-works-under-hood.html>. [Accessed: 10- Apr- 2017]
- [17]"Byzantine fault tolerance", *En.wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Byzantine_fault_tolerance. [Accessed: 12- Apr- 2017]
- [18]"First-class citizen", *En.wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/First-class_citizen. [Accessed: 12- Apr- 2017]
- [19]"Contracts and Transactions — Ethereum Homestead 0.1 documentation", *Ethdocs.org*, 2017. [Online]. Available: <http://ethdocs.org/en/latest/contracts-and-transactions>. [Accessed: 13- Apr- 2017]
- [20]"ethereum/wiki", *GitHub*, 2016. [Online]. Available: <https://github.com/ethereum/wiki/wiki/Design-Rationale#gas-and-fees>. [Accessed: 14- Apr- 2017]
- [21]"ASIC", *En.bitcoin.it*, 2017. [Online]. Available: <https://en.bitcoin.it/wiki/ASIC>. [Accessed: 14- Apr- 2017]
- [22]K. J. O'Dwyer and D. Malone, "Bitcoin Mining and its Energy Footprint", *karlodwyer.github.io*, 2014. [Online]. Available: https://karlodwyer.github.io/publications/pdf/bitcoin_KJOD_2014.pdf. [Accessed: 14- Apr- 2017]
- [23]"Mining", *Ethdocs.org*, 2016. [Online]. Available: <http://ethdocs.org/en/latest/mining.html>. [Accessed: 15- Apr- 2017]
- [24]V. Buterin, "Toward a 12-second Block Time", *Ethereum Blog*, 2014. [Online]. Available: <https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/>. [Accessed: 15- Apr- 2017]
- [25]"What is the GHOST protocol for Ethereum?", 2016. [Online]. Available: <https://www.cryptocompare.com/coins/guides/what-is-the-ghost-protocol-for-ethereum/>. [Accessed: 15- Apr- 2017]
- [26]Y. Sompolinsky and A. Zohar, "Accelerating Bitcoin's Transaction Processing Fast Money Grows on Trees, Not Chains", 2013. [Online]. Available: http://www.cs.huji.ac.il/~avivz/pubs/13/btc_scalability_full.pdf. [Accessed: 15- Apr- 2017]
- [27]"ethereum/wiki", *GitHub*, 2016. [Online]. Available: <https://github.com/ethereum/wiki/wiki/Glossary>. [Accessed: 15- Apr- 2017]
- [28]D. Bernstein, "Introduction to post-quantum cryptography", 2008. [Online]. Available: http://www.pqcrypto.org/www.springer.com/cda/content/document/cda_downloaddocument/9783540887010-c1.pdf. [Accessed: 16- Apr- 2017]

- [29]J. Benet, "IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)", *ipfs.io*, 2014. [Online]. Available: <https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf>. [Accessed: 16- Apr- 2017]
- [30]C. McGoogan, "Yahoo hack: What you need to know about the biggest data breach in history", *The Telegraph*, 2016. [Online]. Available: <http://www.telegraph.co.uk/technology/2016/12/15/yahoo-hack-need-know-biggest-data-breach-history/>. [Accessed: 16- Apr- 2017]
- [31]V. Buterin, "On Public and Private Blockchains", *Ethereum Blog*, 2015. [Online]. Available: <https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/>. [Accessed: 23- Apr- 2017]
- [32]"Diploma Mills", *Wes.org*. [Online]. Available: <https://www.wes.org/ewenr/diplomamills.htm>. [Accessed: 25- Apr- 2017]
- [33]V. Buterin, "Ethereum Scalability and Decentralization Updates", *Ethereum Blog*, 2014. [Online]. Available: <https://blog.ethereum.org/2014/02/18/ethereum-scalability-and-decentralization-updates/>. [Accessed: 26- Apr- 2017]
- [34]"Vitalik Buterin Interview on Ethereum in Daily Pioneer", *europaendiaconclave.com*, 2017. [Online]. Available: <http://europaendiaconclave.com/vitalik-interview/>. [Accessed: 26- Apr- 2017]
- [35]A. Castor, "Ethereum's Difficulty Bomb: All Smoke, No Fire?", *CoinDesk*, 2017. [Online]. Available: <http://www.coindesk.com/ethereums-difficulty-bomb-smoke-no-fire/>. [Accessed: 26- Apr- 2017]
- [36]"The DAO, The Hack, The Soft Fork and The Hard Fork", *CryptoCompare*, 2017. [Online]. Available: <https://www.cryptocompare.com/coins/guides/the-dao-the-hack-the-soft-fork-and-the-hard-fork/>. [Accessed: 26- Apr- 2017]
- [37]M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns", 2017. [Online]. Available: <https://arxiv.org/pdf/1703.06322.pdf>. [Accessed: 27- Apr- 2017]
- [38]"Technical Introduction to Events and Logs in Ethereum", *ConsenSys Media*, 2016. [Online]. Available: <https://media.consensys.net/technical-introduction-to-events-and-logs-in-ethereum-a074d65dd61e>. [Accessed: 28- Apr- 2017]
- [39]"PRISM (surveillance program)", *En.wikipedia.org*, 2013. [Online]. Available: [https://en.wikipedia.org/wiki/PRISM_\(surveillance_program\)](https://en.wikipedia.org/wiki/PRISM_(surveillance_program)). [Accessed: 03- May- 2017]

Terms and abbreviations

ASIC

An Application-specific integrated circuit is hardware that is customised for a particular use. Rather than intended for general-purpose use, it will outperform general hardware at a specific task e.g. mining in Bitcoin [21].

Block difficulty

Block difficulty defines a measure of how long it will take a miner to produce a valid block. If the difficulty is low then the easier it is to calculate a valid hash for the new block by trial and error.

Casper

Ethereum's *PoS* consensus algorithm.

DAO

Decentralised autonomous organisation.

DApp

Known as a shortened version of decentralised application in the Ethereum ecosystem.

DDOS

Distributed denial-of-service attack.

Ethash

Purpose built proof of work algorithm for Ethereum¹⁰.

EVM

The Ethereum Virtual Machine.

First-class-citizen Property

In programming language design, a first-class citizen in a given programming language is an entity which supports all the operations generally available to other entities [18].

Genesis Block

The first block in the blockchain. It contains several parameters including the transactions from the ether sale (i.e. initial account balances of the first accounts). When a user inputs it into the client, it represents their decision to join the network under its terms: it is the first step to consensus.

GHOST

The Ghost protocol in Ethereum (Greedy Heaviest Observed Subtree) was introduced in 2013 as a way of combating the way that fast block time blockchains suffer from a high number of stale blocks [25].

IPFS

The Interplanetary File System (IPFS) is a peer-to-peer distributed file system that seeks to connect all computing devices with the same system of files [29].

JSON

JavaScript Object Notation or *JSON* is an open-standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs³⁶.

MetaMask

Google chrome browser extension to provide the bridge that allows you to visit the distributed web of tomorrow in your browser today. It allows you to run Ethereum DApps right in your browser without running a full Ethereum node. You can also create Ethereum accounts and manage which nodes you'd like to connect to.

Nonce

In cryptography, a nonce is an arbitrary number that may only be used once. A nonce is used in proof-of-work systems to vary the input to a cryptographic hash function so as to obtain a hash for a certain input that fulfils certain arbitrary conditions.

PoS

Proof of stake, an alternative consensus algorithm to the more common proof of work.

PoW

Proof of Work consensus algorithm.

Serenity

An imminent hard fork that will is expected to introduce new Ethereum improvement proposals such as *PoS*.

Sybil Attacks

A Sybil attack³⁷ is an attack where a single adversary is controlling multiple nodes on a network. It is unknown to the network that the nodes are controlled by the same adversarial entity. For example, an adversary can spawn up multiple computers, virtual machines, and IP addresses. They can create multiple accounts with different usernames and e-mail addresses and pretend that they all exist in different countries.

TTP

Third Trusted Party

UTXO

Unspent transaction output

Wei

It is a common denomination used for representing the Ethereum cryptocurrency³⁸.

$$1 \text{ ether} = 1 * 10^{18} \text{ wei.}$$

51% Attack

If a single entity contributed the majority of the network's mining hash rate, they would have full control of the network and would be able to manipulate the public ledger (blockchain) at will³⁹.

³⁶ <https://en.wikipedia.org/wiki/JSON> [Accessed: 24-04-2017]

³⁷ <https://bitcoin.stackexchange.com/questions/50922/whats-a-sybil-attack> [Accessed: 10-04-2017]

³⁸ <http://ether.fund/tool/converter> [Accessed: 14-04-2017]

³⁹ <https://learncryptography.com/cryptocurrency/51-attack> [Accessed: 15-04-2017]

Table of Figures

Figure 1. Graph showing the relative probabilities of solving consecutive blocks to execute a double-spend attack	10
Figure 2. Diagram depicting events occurring in a simplified view of a network transitioning state on the Ethereum network.....	12
Figure 3. Diagram showing a simple architecture overview of a common solution for academic verification	20
Figure 4. Architecture diagram showing the main components of the Unilog hybrid approach implementation.	23
Figure 5. The genesis block of the test network that was used to develop Unilog. It includes the initial balance allocated to the various accounts in the network, along with the mining difficulty, gas limit and other parameters.	25
Figure 6. Shows the login process to a MetaMask vault. You can easily create a new account by clicking the plus icon at the bottom of the accounts list. You can quickly switch accounts by clicking on already created accounts.	26
Figure 7. Shows the network connection options that MetaMask provides to users. Choosing localhost 8545 ensures that you hold your own version on the blockchain locally on the machine. We can also see that you can connect to the Main Ethereum network and also Ropsten, Ethereum's test network.	27
Figure 8. Shows the basic architecture of a DApp and how the content reference is stored in Ethereum so that the data can be retrieved from IPFS. Source: https://karl.tech/content/images/2016/03/dapp-basic-structure.png	30

Appendix 1 – User Manual

This manual demonstrates how to use the Unilog framework. It is targeted at developers looking to fork the Unilog framework. The inner workings of the software are not described in detail here. They are however covered in the paper.

Before delving into the navigation of the user interfaces, the initial configuration of Ethereum nodes and environment setup are covered. The manual shows the verification of an academic transcript, from initial upload to the final verification.

Start the local IPFS node and MySQL server:

```
➔ ~ ipfs daemon
Initializing daemon...
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip4/192.168.1.9/tcp/4001
Swarm listening on /ip6:::1/tcp/4001
Swarm listening on /ip6/fd98:e7f5:2be5:3800:1898:8c36:1133:6097/tcp/4001
Swarm listening on /ip6/fd98:e7f5:2be5:3800:542:ef24:955e:9249/tcp/4001
API server listening on /ip4/127.0.0.1/tcp/5001
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8080
Daemon is ready
```

```
➔ ~ mysql.server start
Starting MySQL
. SUCCESS!
```

Configure new Geth client node. A remote MetaMask node could also be used, but this means that users need to place trust on a TTP. Running a local node also is essential to connect to our test network. In addition, MetaMask offers a way to connect to local nodes as described in the paper:

```
➔ ~ geth --datadir Library/Ethereum/dataDir --networkid 161 --nodiscover --port 30310
init unilog/app/main-blockchain/src/main/resources/genesis/sample-genesis.json
```

Start Geth node:

```
➔ ~ geth --identity "GethNodeUnilog" --rpc --rpcport "8545" --rpccorsdomain "*" --datadir Library/Ethereum/dataDir --nodiscover --port "30310" --rpcapi "db,eth,net,web3" --networkid 161
console
```

Create local Ethereum account on Geth console. For conciseness, this will act as the transcript owner and verifiers Ethereum account i.e. "0x7471ac9f01fd3ab36c4d0cfb9876b472c29c70b0". Set the Etherbase to be the newly created account. If node starts to mine then rewards go to this new account:

```
> personal.newAccount("password") > miner.setEtherbase(eth.accounts[0])
```

Obtain the Geth 'enode' address and add to boot nodes configuration in Java layer. Allows peers to connect on the test network:

```
> admin
{
  datadir: "/Users/macuser/Library/Ethereum/dataDir",
  nodeInfo: {
    enode: "enode://e8dc86cb8760667afa4e834cacf153eed33baca0a67c9703b58d05df2f65e28a252376abae
bb5ff38ce249195890b719feb77c93fc467195e9c20893939b0e28@[::]:30310?discport=0",
    id: "e8dc86cb8760667afa4e834cacf153eed33baca0a67c9703b58d05df2f65e28a252376abae5ff38ce24
9195890b719feb77c93fc467195e9c20893939b0e28",
```


Add Geth 'enode' address into configuration for Ethereum Java client:

```
peer {  
    # Boot node list  
    active = [  
        { url = "enode://26ba1aada59d7607ad7f437146927d79e80312f026cfa635c6b2ccf2c5d3521f5812ca2beb3b295b14f97110e64ae48622a8766ee6" },  
        { url = "enode://e8dc86cb8760667afa4e834cacf153eed33baca0a67c9703b58d05df2f65e28a252376abaebb5ff38ce249195890" }  
    ]  
}
```

Restart Geth node, this time unlocking the new account that was created:

```
→ ~ geth --identity "GethNodeUnilog" --rpc --rpcport "8545" --rpccorsdomain "*" --datadir Library/Ethereum/dataDir --nodiscover --port "30310" --rpcapi "db,eth,net,web3" --networkid 161 --unlock 0 console
```

Start the Java and Unilog DApp applications:

```
→ dappunilog git:(master) ✗ meteor  
[[[[[ ~/unilog/dappunilog ]]]]]  
  
=> Started proxy.  
=> Meteor 1.4.4.1 is available. Update this project with 'meteor update'.  
=> Started MongoDB.  
=> Started your app.  
  
=> App running at: http://localhost:3000/
```

```
→ app git:(master) ✗ gradle bootRun  
Starting a Gradle Daemon (subsequent builds will be faster)
```

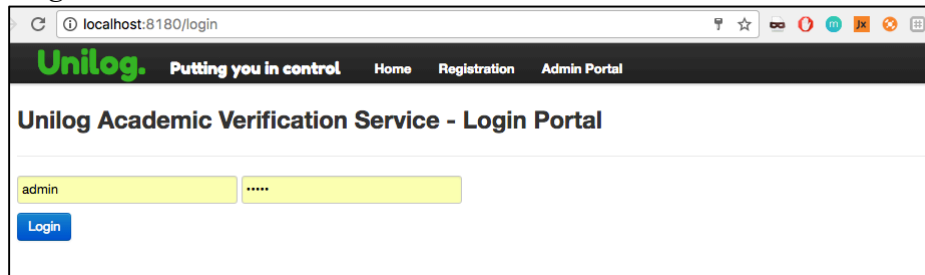
Ensure that the Geth node has been added to the list of peers in the Java web service logs:

```
[c.u.app.UnilogServiceApplication] Started UnilogServiceApplication in 18.173 seconds (JVM running for 19.125)  
[net] New peers processed: [3973cb86 | /127.0.0.1:56417], active peers added: 1, total active peers: 1  
[net] New peers processed: [26ba1aad | localhost/127.0.0.1:30304, e8dc86cb | localhost/127.0.0.1:30310], active peers added: 2, total active peers: 2  
[c.u.b.client.ClientListenerAdapter] peer e8dc86cb | localhost/127.0.0.1:30310 has been added to transaction pool
```

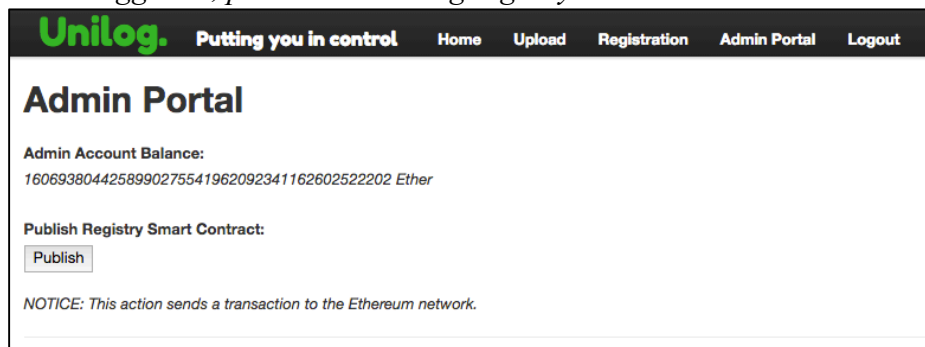
Check that the Geth node has connected to a peer, specifically the Java services Ethereumj node:

```
> admin.peers  
[  
  {  
    caps: ["eth/62"],  
    id: "3973cb86d7bef9c96e5d589601d788370f9e24670dcba0480c0b3b1b0647d13d0f0ffed115dd2d4b5ca1929287839dcd4e77bdc724302b44ae48622a8766ee6",  
    name: "Ethereum(J)/v1.5.0/Mac/Release/Java/Dev",  
    network: {  
      localAddress: "127.0.0.1:30310",  
      remoteAddress: "127.0.0.1:56418"  
    },  
    protocols: {  
      eth: {  
        difficulty: 1048576,  
        head: "0x742ff9348db11357952fd84880cbb3bcc07fd253e574c5764ca3828b412af9dd",  
        version: 62  
      }  
    }  
  }  
]
```

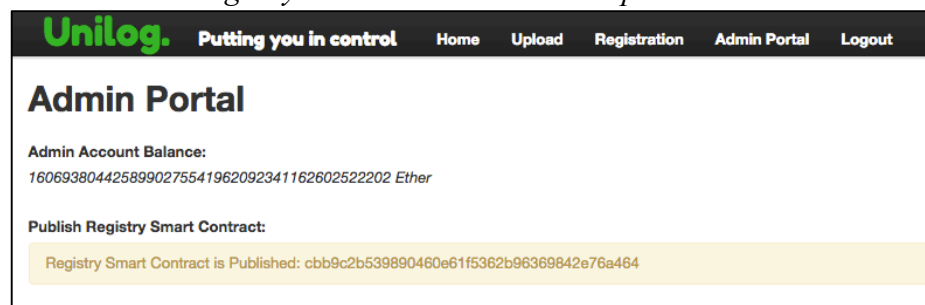
Navigate to localhost:8180 in the browser, the Java Spring Boot web service is hosting static content that is available at this port.
Login to the Admin Portal:



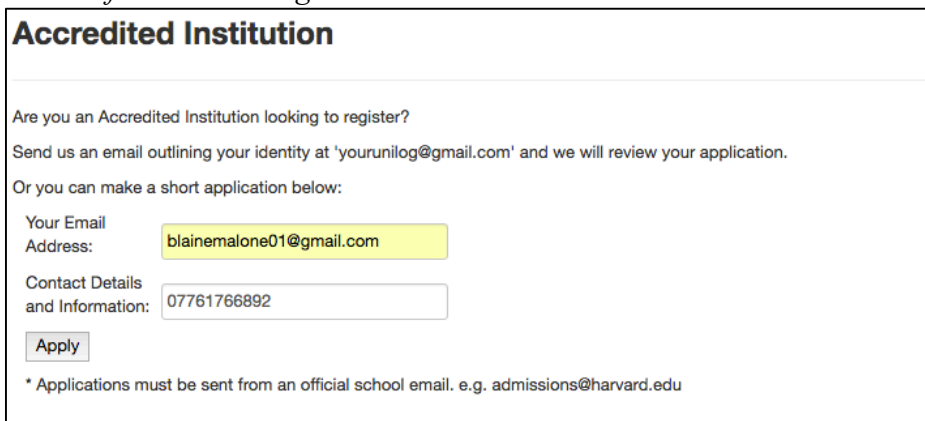
Once logged in, publish the Unilog registry smart contract:



Wait until the registry smart contract has been published:



Logout of the Admin account and submit an application. This application would usually be a university or other accredited institution. This is found in the registration tab:



Wait until the application has been sent:

Apply

* Applications must be sent from an official school email. e.g. admissions@harvard.edu

Your application has been sent successfully.

Once an institution applies, the Unilog admin account will receive an email. At this point the admin should investigate the party to prove they are who they claim to be. Once this happens, the admin should login to the Unilog admin portal and add the institution to the system:

Unilog - Application Review

yourunilog@gmail.com 12:46 AM (2 minutes ago)

to me

Admin,
The following party has asked to be registered in the Unilog application.
Take care when investigating the legitimacy of their request. Below are the details of their application:
Email: blainemalone01@gmail.com
Message: 07761766892

Register an Account:

Enter Email of New Institutions account:

blainemalone01@gmail.com

Register

The institutions email address must adhere to a correct format:

Register an Account:

Enter Email of New Institutions account:

Register

New Account has been registered!

Register an Account:

Enter Email of New Institutions account:

@example.com

Register

Invalid email address entered. If the problem persists,

The institution will receive an email from the Unilog admin to active their account:

Unilog - Activate your account

yourunilog@gmail.com 12:50 AM (3 minutes ago)

to me

Congratulations, your request for a Unilog account has been accepted.
Please activate your account with the following code:

iq0c2p3vhp8ur0tbdpd9k1suhp

Institution activates their account to receive their login details:

Already Applied?

Are you an institution who has already applied and successfully received their activation code?
If so, activate your account below.



Your Email Address:




Activation Code:

Activate

Using the login details below, the institution should now login to the Unilog server to being uploading transcripts:

Unilog - Your Login Details Inbox x

 **yourunilog@gmail.com** 12:55 AM (0 minutes ago) ☆  

to me ▾

Below are your login details for Unilog.
Make sure to enjoy our features available to you!
Using block chain technology we can reduce the work load of your staff by handling the academic verification of transcripts!

username: blainemalone01@gmail.com
password: v3sdqo5r1vj749kbol51rlm3tf

Unilog Academic Verification Service - Login Portal

Institution uploads transcripts alongside their associated qualification:

Unilog. Putting you in control Home Upload Registration Admin Portal Logout

Upload Qualification

Qualification Code:

Title:

Accredited Institution:

• Student Number

Pass ☒

Student Email

Classification

Additional Information

• Student Number

Pass ☒

Student Email

Classification

Additional Information

Qualification will successfully be uploaded provided all the fields are entered correctly, data validation is important at this stage:

Unilog. Putting you in control Home Upload Registration Admin Portal Logout

Upload Qualification

Your qualification was uploaded!

The transcript owner will receive an email telling them to activate their account. The account in the example is bmalone05@qub.ac.uk:

☆ yourunilog@gmail.com Inbox - Exchange

Unilog - Complete Setup!

To: Blaine Malone

An accredited institution uploaded a transcript associated with you.
If you already have an Ethereum Account then you're ready to finish registration!

Otherwise, create an account and come back to us to complete your registration!

Bring the following activation code with you when you're completing registration:

dloo8ldo1jia9fgrnp92o7idj9

If the admin account attempts to authorise the transcripts to be published to the blockchain before the transcript owner has complete their account they'll get the following error:

Transcript owner transactions awaiting authorisation:

- ID: 1
Email Address
bmalone05@qub.ac.uk
- ID: 2
Email Address
cgrant14@qub.ac.uk

Enter ID of account to authorise:

NOTICE: Please be patient. Transactions will be sent to network and may take a while due to block propagation delays.

Upon authorising an account, the following happens:

- Unilog.sol Smart contract created for transcript owner
- Transcript owner is sent their token via email
- Transcript data is published to the IPFS network
- Transcript's IPFS content address added to smart contract
- Transcript owner is set as the smart contract's owner
- Smart contract address is added to the Unilog registry contract

Error occurred when publishing. Please check server logs.
Have you published the registry?

Therefore, the transcript owner must complete their account:

Complete Registration

To finish registration please enter the activation code sent to your email and if you haven't already, create an Ethereum Account.
Successful registration will provide you with a token that you can share with employers/verifiers!

Email Address:

Ethereum Account:

Activation Code:

The transcript owner transactions can now be authorised in the admin portal:

Transcript owner transactions awaiting authorisation:

- ID: 2

Email Address
cgrant14@qub.ac.uk

Enter ID of account to authorise:

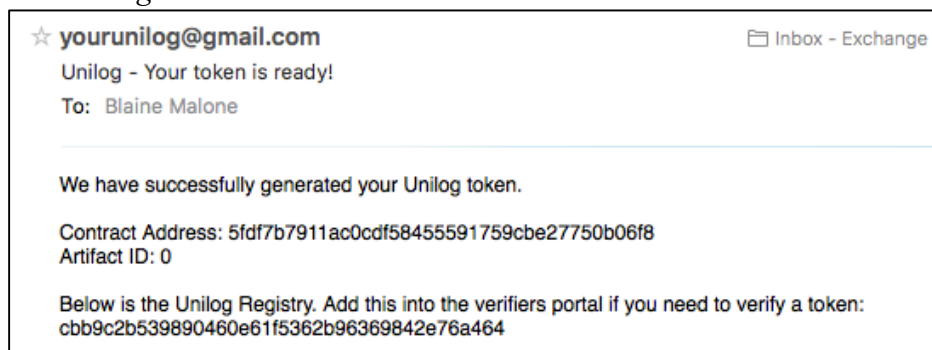
NOTICE: Please be patient. Transactions will be sent to network and may take a while due to block propagation delays.

Upon authorising an account, the following happens:

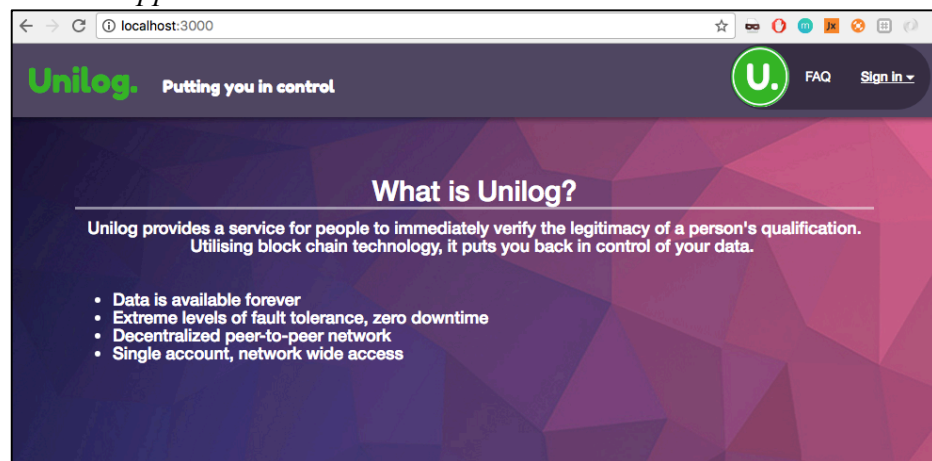
- Unilog.sol Smart contract created for transcript owner
- Transcript owner is sent their token via email
- Transcript data is published to the IPFS network
- Transcript's IPFS content address added to smart contract
- Transcript owner is set as the smart contract's owner
- Smart contract address is added to the Unilog registry contract

Account Authorised. Token has been sent!

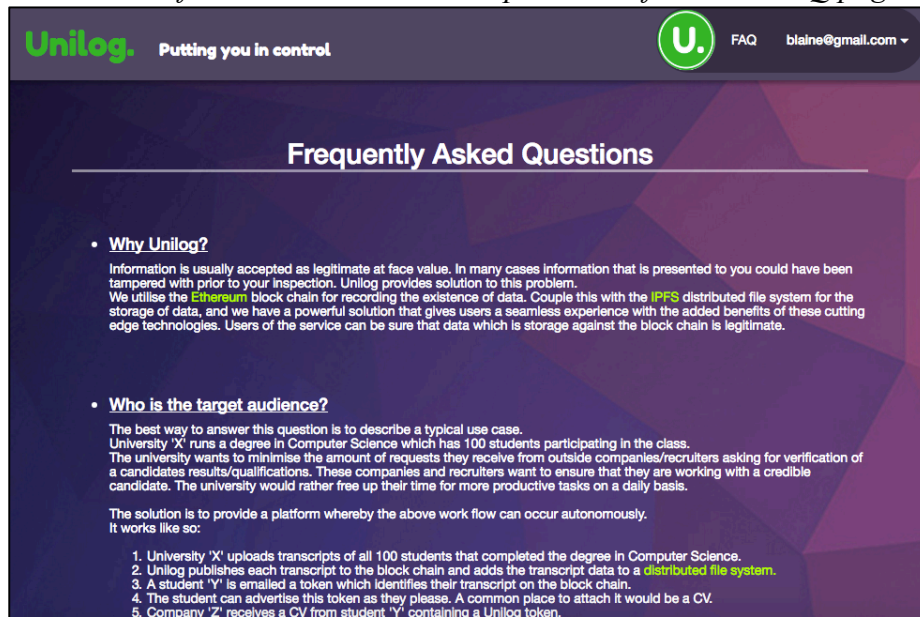
The transcript owner (bmalone05@qub.ac.uk) receives an email containing their token:



The verifier should open the ÐApp and create an account. This functionality is not necessary because the login details are not stored in a central server. All it provides in an extra layer of security for the local ÐApp.



For more information on the current process, refer to the FAQ page:



The screenshot shows the Unilog website's FAQ page. The header includes the Unilog logo with the tagline "Putting you in control", a user profile icon with the letter 'U', and links for "FAQ" and "blaine@gmail.com". The main heading is "Frequently Asked Questions".

- Why Unilog?**

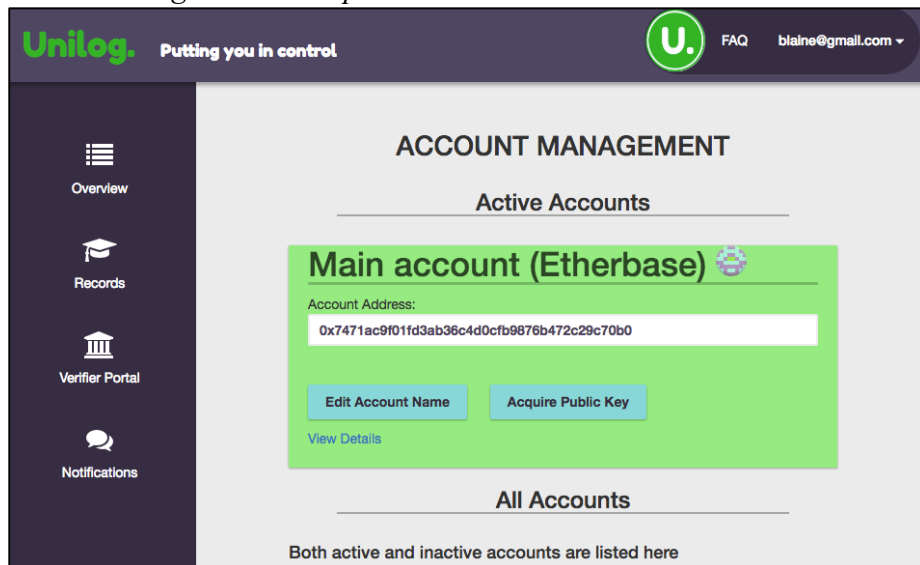
Information is usually accepted as legitimate at face value. In many cases information that is presented to you could have been tampered with prior to your inspection. Unilog provides solution to this problem. We utilise the **Ethereum** block chain for recording the existence of data. Couple this with the **IPFS** distributed file system for the storage of data, and we have a powerful solution that gives users a seamless experience with the added benefits of these cutting edge technologies. Users of the service can be sure that data which is storage against the block chain is legitimate.
- Who is the target audience?**

The best way to answer this question is to describe a typical use case.
University 'X' runs a degree in Computer Science which has 100 students participating in the class. The university wants to minimise the amount of requests they receive from outside companies/recruiters asking for verification of a candidates results/qualifications. These companies and recruiters want to ensure that they are working with a credible candidate. The university would rather free up their time for more productive tasks on a daily basis.

The solution is to provide a platform whereby the above work flow can occur autonomously.
It works like so:

 1. University 'X' uploads transcripts of all 100 students that completed the degree in Computer Science.
 2. Unilog publishes each transcript to the block chain and adds the transcript data to a **distributed file system**.
 3. A student 'Y' is emailed a token which identifies their transcript on the block chain.
 4. The student can advertise this token as they please. A common place to attach it would be a CV.
 5. Company 'Z' receives a CV from student 'Y' containing a Unilog token.

Access the DApp and browse some of the available functionality.
Note that the current user is the account that was created at the start of the user manual, this account is going to be used for verifying as well as being the transcript owner:



The screenshot shows the Unilog website's Account Management page. The header is identical to the FAQ page. A left sidebar contains navigation links: Overview, Records, Verifier Portal, and Notifications. The main content area is titled "ACCOUNT MANAGEMENT" and has a sub-heading "Active Accounts".

The "Main account (Etherebase)" section displays the account address: `0x7471ac9f01fd3ab36c4d0cfb9876b472c29c70b0`. Below the address are buttons for "Edit Account Name", "Acquire Public Key", and a link for "View Details".

Below this section is the "All Accounts" heading, followed by the text: "Both active and inactive accounts are listed here".

To verify a transcript, open the verifier portal. Using the token received from a candidate, enter the details and click verify:

Unilog. Putting you in control U. [FAQ](#) [blaine@gmail.com](#)

Verifier Portal

Dashboard

Want to verify a candidates token against the block chain?
Enter their token information and simply click verify.

Token

Contract Address:

Artifact ID:

Unilog Registry:

Result

Please enter a valid token above to view the artifact.

Say this candidate stated that they received a 1st class honours in their Computer Science degree. The verifier can immediately tell that they're being dishonest as the result returned says that the actual classification is a 2.1:

Unilog. Putting you in control U. [FAQ](#) [blaine@gmail.com](#)

Result

Accredited Institution:

Title:

Qualification Code:

Transcript Reference:

Recipient EmailAddress:

Classification:

Additional Information:

Overall Result:

This data has been obtained from a credible Unilog source on the block chain. Check it against whatever data the token was attached to. You can now determine the integrity of the information you were given.

The 'Notifications' option contained within the navigation bar above, was intended to be a place holder for one of the proposed privacy solutions discussed in section 6.2.

Appendix 2 – The Unilog Framework Use Cases

Use Case	Description	Actors	Non-Functional	Assumptions
1: Potential issuer can register	An issuer can submit an application to register with the Unilog system.	Issuer	Usability: The system should provide a user-friendly interface via a registration page so that applying is simple.	N/A
2: Admin can login to an admin portal	There exists an admin portal that provides a dashboard containing all the functions that an admin can perform. An admin can gain access to this portal based on their role privileges in the system. Access is granted by entering an acceptable username and password.	Admin	Usability: Dashboard should clearly outline all available tasks and functions that an admin can complete.	Role of user is 'ADMIN'.
3: Admin can accept or decline an application	After performing an internal investigation, the admin should accept or decline an application. The outcome should be based on the identity of the applicant, e.g. if a university allegedly applied then Unilog must verify that the application was submitted by the actual university. If accepted the applicant should receive an activation code to activate their account.	Admin	Usability: Admin should be able to see all pending applications and be able to easily accept or decline based upon identity checks. Performance: Internal investigations should be performed as soon as possible. Maximum delay should only be a few working days.	Accepted if applicant is who they claim to be.
4: Issuer can activate their account	After receiving their activation code, an issuer can activate their account by entering their email address that they applied with. They'll also need to provide the activation code that Unilog sent them. This acts as another level of security to prove the identity of an applicant. If the activation code matches then the issuer should be sent their login details.	Issuer	Usability: The system should provide a user-friendly interface so that activating an account is simple. Security: Activation code is secure enough that a brute force attack is impractical.	Use case 3 resulted in an accepted action by Admin. Activation code is secure.
5: Issuer can login to their account	Upon receiving their login details, an issuer can login to their account. They now have access to features that only registered users could've previously accessed.	Issuer	Usability: Simple login portal for users that will automatically redirect them to their desired location.	Role of issuer is 'USER'. Use case 1, 2,3 and 4 all occurred successfully.
6: Publish smart contract registry	Admin can publish a registry smart contract to the Ethereum network. This acts as a whitelist for all accepted tokens by the Unilog system and should only ever have to be performed once i.e. the smart contract will 'live' on the Ethereum blockchain forever.	Admin	Security: State changing functions on the registry smart contract can only be invoked by the Unilog Ethereum account. Usability: Admin can easily publish a smart contract through the admin portal.	This should always be performed before any users are registered. Avoids unpredicted events from occurring.

The Unilog Framework Use Cases – 2

Use Case	Description	Actors	Non-Functional	Assumptions
7: Issuer can upload a qualification	An issuer can upload a qualification along with its associated transcripts. System should alert users if incorrect data is entered at this stage. Activation code should be sent to the transcript owners listed on the qualification once upload completes.	Issuer	Robustness: If an error occurs during an upload the system will alert the issuer about the problem is, e.g. email addresses for transcript owner is not the correct format. Email should not be sent in this case Usability: Easily add qualifications to the Unilog system. Response time: Unilog should process these requests quickly. Issuers should not be exposed to Ethereum network latencies.	Use case 1,2,3,4,5 and 6 all occurred successfully.
8: Owner can complete registration	After receiving the activation code from Unilog, an owner can complete their registration. Three fields need to be completed: - The email address relating to their transcript e.g. school email. - An Ethereum account address. - The activation code that was sent by Unilog. If the activation code matches then the admin must authorise their account.	Owner	Usability: A simple portal for owners to complete their registration with Unilog. Security: Activation code is secure enough that a brute force attack is impractical.	The upload in use case 7 contained the email address of the owner.
9: Users can create Ethereum accounts	Unilog system has the capability for creating Ethereum accounts. It should also accept accounts that are created with other platforms.		Interoperability: Unilog should be able to accept Ethereum accounts that are created with other services.	Standard Ethereum address is 40 hex characters long i.e. 20 bytes
8: Admin can authorise a qualification upload on an individual transcript basis	Admin can authorise an individual transcript associated with a qualification. When the transcript is authorised, four transactions are sent to the Ethereum network. Unilog's account is used to send the transactions therefore this step is necessary to stay in control of Unilog's Ether. First transaction creates the transcript owners smart contract. Second adds transcript reference address to smart contract. Third sets the transcript owner as the owner of the smart contract. Fourth adds the owners smart contract address to the Unilog registry whitelist contract. Unilog also publishes the transcript to a distributed file system. Afterwards, the transcript owner should receive their Unilog token.	Admin	Usability: Dashboard should clearly outline all available tasks and functions that an admin can complete. Transparency: Users should be able to verify that the server is acting honestly. Open-sourcing can help with this. Unilog should document the steps that they take to publish a transcript so that issuers can verify their actions if they're in any doubt. Availability: Once a transcript is uploaded, it should always be available for verification.	Role of user is 'ADMIN'. Admin account has enough ether to complete the transactions.

The Unilog Framework Use Cases - 3

Use Case	Description	Actors	Non-Functional	Assumptions
9: Admin can authorise a qualification upload without having to accept each individual transcript	As in use case 8, an admin will authorise an upload. Instead of having to accept the qualifications associated transcripts one by one. The admin can authorise the whole qualification at once.	Admin	See use case 8	See use case 8
10: Verifier can successfully verify an owner's transcript without any central server	<p>The transcript owner can advertise their Unilog token after it is received. A common place for a token would be contained within a CV. A transcript owner would submit their CV to a hiring company (verifier). The verifier would use the Unilog <i>DApp</i> to open the verifier portal. Using the token, they can check the legitimacy of the owner's CV/transcript. Results can be either:</p> <ul style="list-style-type: none"> - Transcript Expired - Fictional Transcript (token does not exist) - Or all the data of a legitimate transcript corresponding to a token. <p>If transcript data is returned then the verifier can be sure that it is trustworthy. This is because it could've only been published by the university/Unilog (enforced by cryptographic signatures). A verifier would then manually scan the CV to check it against the trusted data. A decision can then be made on the credibility of the CV.</p>	Verifier	<p>Fault tolerance: System should be able to maintain core functionality, even if core components of the system break. In the event of the Unilog server being switched off, verifiers should still be able to verify transcripts.</p> <p>Usability: Simple interface for Ethereum smart contracts via a <i>DApp</i>. User shouldn't feel like they're using a <i>DApp</i>. Ideally, familiar tools should be used to access the application e.g. Standard web browser.</p> <p>Response time: Verification of transcripts should be as fast as or faster than the current <i>TTP</i> solution.</p>	A transcript has been authorised by the Unilog admin and published to the blockchain and a distributed file system.
11: Decentralised application can interface with browser extensions to help users	The decentralised application can interface with extensions that are purpose built to turn standard web browsers into Ethereum <i>DApp</i> explorers e.g. <i>MetaMask</i> helps with Ethereum account management and network connections.	Verifier, owner, issuer	<p>Usability: Targeted user demographic should easily be able move from old system to new system without a steep learning curve. There whole process should be intuitive. Even while using a new technology, they shouldn't feel like they are.</p> <p>Performance: The process of account creation should be no more that 5 minutes.</p>	

Appendix 3 – Smart Contract Source Code

The smart contracts used in the Unilog framework to allow for a decentralised academic verification service to function on the Ethereum network.

Unilog Registry Smart Contract

```
pragma solidity ^0.4.0;

contract UnilogRegistry {

    // Value will be assigned during creation of contract
    address private owner = msg.sender;

    mapping(address => bool) private repository;

    /*
        Only unilog (the owner) can add a contract address to the repo.
        This prevents users from making their own contracts with forged
        artifacts and adding those so that they will be accepted by verifiers.
    */
    function addContract(address contractAddress) onlyBy(owner) {
        repository[contractAddress] = true;
    }

    /*
        Disable a contract. This could be a reactive procedure for
        potential errors when granting a contract eligibility. Only
        unilog can invoke this function.
    */
    function disableContract(address contractAddress) onlyBy(owner) {
        repository[contractAddress] = false;
    }

    /*
        Allows verifiers to see if the contract they are querying is listed by
        unilog.
    */
    function contains(address contractAddress) returns (bool) {
        return repository[contractAddress];
    }

    modifier onlyBy(address _account) {
        if (msg.sender != _account)
            throw;
        // _; is replaced by the function that onlyBy(x) is invoked upon
        _;
    }

    function version() constant returns (uint) { return 1; }
}
```

Unilog Transcript Owner Contract

```
pragma solidity ^0.4.0;

contract Unilog {

    //Value will be assigned during creation of contract
    address private unilog = msg.sender;
    address private artifactOwner;

    /*
     * Struct presenting properties of an artifact.
     */
    struct Artifact {
        string artifactAddress;
        /*
         * A time in the future that the artifact expires. Not all artifacts
         * need expire. In those cases we will pass 0.
         */
        bool active;
        uint expirationDate;
    }

    //Array to hold the artifacts and their current status
    Artifact[] private artifacts;

    /*
     * Function to add artifacts to the artifact owners contract.
     * Only the unilog (the creator of the contract) can access this function.
     * Allows verifiers to be sure that if the artifact exists, then it is
     * legitimate. This is because unilog is the only party able to
     * alter the contents of this state variable.
     * The artifact of this operation should be returned in an event.
     * Returning value statically because its easier to test with Ethereumj.
     */
    function addArtifact(string artifactAddress, uint validFor) onlyBy(unilog)
        returns (uint) {
        //Could add check if the artifactAddress already exists and return '-1'
        //if it has. This would require an iteration of artifacts array and
        //thus increases the amount of operations performed by the EVM. This
        //causes an increase in the transaction cost for unilog.
        uint expirationDate;
        if(validFor == 0) { //common case, artifact doesn't have expiration date.
            expirationDate = validFor;
        } else {
            //Change weeks to seconds for testing purposes.
            expirationDate = now + (validFor * 1 weeks); // now is the current time.
        }
        //Note that unilog-service must derive the index of the new value
        //for the artifact id to be collected. ((length-1) == artifactId)
        //Perform that operation in unilog-service to reduce costs.
        return artifacts.push(Artifact(artifactAddress, true, expirationDate));
    }

    /*
     * This function is accessible to anyone looking to inspect if a user
     * legitimately received an artifact.
     * It returns the artifactAddress of an artifact if it exists and if it
     * hasn't expired by the time of verification. The function may return
     * "Expired" if an artifact has expired or "Fictional Aritfact!" if the
     * artifact is not contained in the state array.
     */
    function verifyArtifact(uint artifactId) constant returns (string) {
        if(artifactId < artifacts.length) {
            Artifact current = artifacts[artifactId];
            if(current.expirationDate == 0 || now < current.expirationDate) {
                //Artifact is still valid or doesn't have an expiration date.
                return current.artifactAddress;
            }
        }
    }
}
```

```

        } else {
            //Artifact has expired.
            return "Expired!";
        }
    } else {
        return "Fictional Artifact!";
    }
}

/*
Only the artifact owner can delete an artifact associated with them.
Puts them in control of what artifacts are linked to their collection.
The boolean result of this operation should be returned in an event.
Returning result statically because its easier to test with Ethereumj.
*/
function deleteArtifact(uint artifactId) onlyBy(artifactOwner) returns(bool) {
    if(artifactId < artifacts.length) {
        delete artifacts[artifactId];
        return true;
    } else {
        return false;
    }
}

/*
Only unilog can assign the artifact owner.
The boolean result of this operation should be returned in an event.
Returning result statically because its easier to test with Ethereumj.
*/
function assignArtifactOwner(address artifactOwnerAddress) onlyBy(unilog)
    returns(bool) {
    //Unilog does not have permission to be artifact owner.
    if (artifactOwnerAddress != unilog) {
        artifactOwner = artifactOwnerAddress;
        return true;
    }
    return false;
}

/*
Function to return the latest artifact Id from the contract.
Called immediately after an artifact has been added and
its transaction included into a block.
Can be replaced if events are implemented.
*/
function latestArtifactId() constant returns(uint) {
    return artifacts.length;
}

modifier onlyBy(address _account) {
    if (msg.sender != _account)
        throw;
    // _; is replaced by the function that onlyBy(x) is invoked upon
    _;
}

function version() constant returns (uint) { return 1; }
}

```

Appendix 4 – Smart Contract Test Cases

Preconditions:

Smart contracts are deployed to a Sample blockchain. Simulates actual environment where *EVM* would execute operations.

Dependencies:

UnilogRegistry smart contract must be deployed before any other user's smart contract and accounts that set to invoke functions must have at least the minimum balance to do so.

Module Name: Unilog Registry Smart Contract – UnilogRegistry.sol						
Test	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Submit Unilog Registry contract to the blockchain. Check if the version function returns the expected result.	The compiled Solidity contract corresponding to 'UnilogRegistry.sol'.	'version()' should return 1.	1	Pass	Should be the first contract to be submitted to the network.
2	Submit Unilog user contract to the blockchain. Check if the version function returns the expected result.	The compiled Solidity contract corresponding to 'Unilog.sol'.	'version()' should return 1.	1	Pass	This contract, if submitted by Unilog, should be added to the registry immediately after inclusion into a block.
3	Add contract to repository with 'owner's' identity. Then see was the contract added.	Use the identity that the contract was created from. User contract address as function argument.	'contains(x)' should return True.	True	Pass	This test demonstrates the 'contains(x)' functionality works as expected also.
4	Add contract to repository with 'hacker's' identity. Then see was the contract added to the repository.	Create new Elliptic curve key that will represent 'hackers' identity. Create sample contract address as function argument. This is a byte array.	'contains(x)' should return False.	False	Pass	Hackers account must have a minimum of 0.25 ether to be able to invoke a contracts function.
5	Query registry to see if a contract that Unilog didn't register is present.	A random contract address that Unilog hasn't registered. This is a byte array.	'contains(x)' should return False.	False	Pass	
6	Test disable contract passes when called with 'owner's' identity.	User contract address as function argument. This is a byte array.	'contains(x)' should return False on a contract that has already been added.	False	Pass	
7	Test disable contract fails when called with 'hacker's' identity.	User contract address as function argument. This is a byte array.	'contains(x)' should return True on a contract that has already been added.	True	Pass	No person other than Unilog can disable a contract.

Module Name: Unilog Users Smart Contract – UnilogTestVersion.sol
(Changes to time units to make testing feasible)

Test	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Add artifact with no expiration clause from Unilog identity.	Sample IPFS content address 'x', with 'validFor' field set to 0 'y'.	'addArtifact(x,y)' should return the length of the artifact's array which is equal to 1.	1	Pass	The artifact id would be length - 1.
2	Add artifact with no expiration clause from 'hacker's' identity.	Sample IPFS content address 'x', with 'validFor' field set to 0 'y'. Sending account is not the Unilog account.	'addArtifact(x,y)' should not be invoked so default value 0 returned	0	Pass	Zero is returned because function is restricted only to Unilog.
3	Add artifact with 1000 second expiration clause from Unilog identity.	Sample IPFS content address 'x', with 'validFor' field set to 5 'y'.	'addArtifact(x,y)' should return the length of the artifact's array which is equal to 1.	1	Pass	Test to see if unit still functions correctly when supplied with a non-zero value.
4	Add artifact with expiration date then verify before it expires.	Sample IPFS content address 'x', with 'validFor' field set to 1000 'y'. Retrieve artifact id as 0 'z' and verify.	'verifyArtifact(z)' returns the sample IPFS content address 'x'	'x'	Pass	Add an artifact that does not expire by the time we verify the same artifact.
5	Add artifact with expiration date then verify after it expires.	Sample IPFS content address 'x', with 'validFor' field set to 1 'y'. Retrieve artifact id as 0 'z' and verify.	'verifyArtifact(z)' returns 'Expired!'	'Expired!'	Pass	Add an artifact that does expire by the time we verify it.
6	Add artifact then verify that it has expired twice.	Sample IPFS content address 'x', with 'validFor' field set to 1 'y'. Retrieve artifact id as 0 'z' and verify twice.	'verifyArtifact(z)' returns 'Expired!'	'Expired!'	Pass	This demonstrates that logic for the artifact's 'active' flag is working as expected.
7	Verify that bogus artifact id does not return an artifact address.	Artifact id as 7 'x' when there are no artifact's associated with the user.	'verifyArtifact(x)' returns 'Fictional Artifact!'	'Fictional Artifact!'	Pass	The artifact does not exist, therefore the only response from the contract is that it doesn't exist.
8	Assign artifact owner from the Unilog identity.	Artifact owner address 'x' as the function argument. 'x' is not the same as Unilog account address.	'assignArtifactOwner (x)' returns True	True	Pass	True is returned because function is restricted only to Unilog and the argument was not the same as Unilog's account address.
9	Assign artifact owner from the Unilog identity where artifact owner is Unilog.	Artifact owner address 'x' as the function argument. 'x' is the same as Unilog account address.	'assignArtifactOwner (x)' returns False	False	Pass	The nature of the contract is that the actual owner of the contracts artifact's cannot be Unilog.
10	Assign artifact owner from 'hacker's' identity.	Artifact owner address 'x' as the function argument. Sending account is not the Unilog account.	'assignArtifactOwner (x)' returns False	False	Pass	

Module Name: Unilog Users Smart Contract – UnilogTestVersion.sol
(Changes to time units to make testing feasible)

Test	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
11	Delete an artifact from the artifact owners account.	Artifact id 0, is valid 'x'. Sending account is the artifact owners account.	'deleteArtifact(x)' returns True	True	Pass	Minimum of 2500000000000000000 wei(0.25 ether) must be sent to the artifact owners account for successful interactions with the smart contract.
12	Delete an artifact from Unilog account.	Artifact id 0, is valid 'x'. Sending account is not the artifact owners account but the Unilog account.	'deleteArtifact(x)' returns False	False	Pass	
13	Delete an artifact from 'hacker's' account.	Artifact id 0, is valid 'x'. Sending account is not the artifact owners account but the 'hacker's' account.	'deleteArtifact(x)' returns False	False	Pass	Minimum of 2500000000000000000 wei(0.25 ether) must be sent to the 'hacker's' account for successful interactions with the smart contract.
14	Delete an artifact that does not exist	Artifact id 8, is invalid 'x'. Sending account is the artifact owners account.	'deleteArtifact(x)' returns False	False	Pass	
15	Delete an artifact when no artifact's exist.	Artifact id 0, is invalid 'x'. Sending account is the artifact owners account.	'deleteArtifact(x)' returns False	False	Pass	

**SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER
SCIENCE**

CSC3002 – COMPUTER SCIENCE PROJECT

Dissertation Cover Sheet

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

Student Name: Blaine Malone

Student Number: 40106309

Project Title: Evaluation of Migrating to Decentralised Applications using Blockchain Technology

Supervisor: Desmond Greer

Declaration of Academic Integrity

Before signing the declaration below please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

I declare that I have read both the University and the School of Electronics, Electrical Engineering and Computer Science guidelines on plagiarism - <http://www.qub.ac.uk/schools/eeecs/Education/StudentStudyInformation/Plagiarism/> - and that the attached submission is my own original work. No part of it has been submitted for any other assignment and I have acknowledged in my notes and bibliography all written and electronic sources used.

Student's signature

Date of submission