THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Utilizing Blockchain Smart Contracts as Legal Contracts

Michael Robertson

October 2020

Submitted as part of the assessment for ENGEN582-20D
within
The School of Engineering
and
The School of Computing and Mathematical Sciences

Declaration of Authorship

I, Michael Robertson, declare that this thesis, titled ‘Utilizing Blockchain Smart Contracts as Legal Contracts’, and the work presented in it, are my own except as noted. I confirm that:

- This work was done wholly or mainly as part of ENGEN582 at this University
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Acknowledgements

I would like to thank Steve Reeves for his help on this project. His input and insight has been invaluable in guiding the direction of this project, as well as providing constant support and feedback throughout the year.

I would also like to thank Rebekah Te Rito for providing perspectives from her legal background, and helping to source participants for a survey.

Abstract

Legal contracts are an important part of modern society that has so far failed to adapt to modern technology. As a result, they now lack the security and guarantees that they once provided with potential fraud from digital signatures. Meanwhile, blockchain technology has created distributed networks of databases where information is immutable, with no one entity controlling the network. In particular, the Ethereum blockchain technology extends this by allowing arbitrary computation by the user on the network on top of all the benefits already provided by the blockchain.

This project seeks to create a system that allows users to sign legal contracts using blockchain technology. Such a system provides guarantees on authors of a signature as well as detection of any tampering of the signature or the original contract. A prototype was developed as an Ethereum Dapp interfacing with a React web client to achieve this, with all persistent data stored on the blockchain.

This prototype demonstrates the potential of such an application, as a cost-effective solution for contract signing without needing to trust a third party. However, specific details of implementation may have privacy implications due to the public nature of the blockchain, that would need to be solved with further research. Additionally, issues with blockchain networks such as unpredictable prices for their usage means that current public networks are not suitable for mainstream applications. Finally, one of the largest obstacles for adoption is the lack of understanding that the blockchain provides for the average user (in this case those with a legal background).

Contents

List of Figures

List of Tables

1. **Introduction**

Legal contracts are an important part of modern transactions and are used to create trust and guarantees between people. However, in a modern world contracts have not advanced significantly, and attempts to digitize the process have removed many of the actions that made contracts trustworthy. Solutions for remotely signing contracts often require an extended process of printing, scanning and signing documents in order to emulate traditional methods. However, these methods leave open significant vectors for fraud, invalidating many of the guarantees that a contract is designed to provide.

Meanwhile, blockchain technology has well tested systems for distributed consensus mechanisms. Allowing a user to perform an action which can be validated in a decentralised manner. Additionally, blockchain technology has also developed into the concept of smart contracts, allowing arbitrary actions to be coded into the network with a record of actions that can be trusted. This project seeks to create an application that allows a user to create and modify a legal contract digitized as a smart contract. The smart contract will then be able to allow other users to sign the contract using existing technology to sign transactions on the blockchain. These signatures will be able to better guarantee the identity of those signing, while providing a more modern and easier to use experience for signing contracts than the current status quo.

This project seeks to develop an application capable of running completely on the Ethereum network that is capable of storing legal contracts. As well as create a working system for extending ECDSA signatures used by Ethereum to be used on the contracts. The resulting system should be evaluated on its usability in both in terms of user experience as well as the practicalities of its use by legal professionals.

2. Background

Currently, most contracts still require a handwritten signature to indicate acceptance by both parties. Moving into digital documents, often the signing process has simply become printing, signing and scanning the document. Other systems such as pasting an image of a signature into a document add more convenience, but all these methods remove most of the guarantees that an in-person signature provides (primarily prevention of fraud).

Current digital signature technology, and particularly that surrounding blockchains could be used to help strengthen the security around signatures while also making them more convenient. In New Zealand, the Contract and Commercial Law Act 2017 provides some guidelines requirements for a valid signature. The requirements state that they must: adequately identify the signatory, indicate the signatory's approval of the contract, and be reliable given the circumstances where the signature is required. In particular, for a digital signature to meet these requirements there are several tests for it to be presumed reliable. The signature must be linked to only one person and be controlled by that one person, any changes to the signature after signing must be detectable, and any changes to the information/contract after signing must be detectable [1].

2.1 Blockchain

Blockchains in their various forms have existed, with one of the largest and well known systems being the Bitcoin Blockchain. Since then there has been significant research and variation within blockchain technology, one of these systems is the Ethereum blockchain. Blockchains themselves are distributed ledgers of data, with multiple nodes all owned by different parties. Each node usually contains a copy of the entire blockchain and communicates with other nodes in order to ensure the integrity of the data. When data is committed to a blockchain it is put into a new block and information from the previous block in the chain is used to digitally sign the next. This new block is distributed amongst all nodes until they agree that the information and signature are valid. This process of signing new blocks with previous ones is what forms the ‘chain’. If any data is tampered with it can be seen as if the signature from the previous block is no longer valid, and all data down the chain will also no longer be valid without recomputing the entire chain. Additional validity then comes from the fact that a majority of nodes must agree on the state of the blockchain. Overall this creates a distributed system where almost anyone can publish data - usually with a fee for the computation - but where no one can tamper or remove the data afterwards [2].

2.1.1 Ethereum

The Ethereum blockchain then introduces a new novel concept of smart contracts. Instead of a blockchain exchanging information of a fixed format - for example to exchange a cryptocurrency - Ethereum allows arbitrary programs to be stored and executed on the blockchain. This means applications can be developed to take advantage of blockchains benefits without the cost of setting up a new blockchain network. Applications can take advantage of the massive network of existing Ethereum nodes to perform the computations on the blockchain. These computations cost a small amount of the Ethereum cryptocurrency Eth [3].

The programming language Solidity has been specifically designed for writing these smart contracts to run on the blockchain. Supporting C-like syntax and a number of types and built in data structures such as arrays and hash maps. These programs are converted to bytecode capable of running the Ethereum Virtual Machine (EVM). This byte code can then be committed to the blockchain as a transaction itself. Afterwards, additional transactions can call methods in the code which may add or modify information in the blockchain for a completely distributed application state. Each EVM instruction costs a fixed amount of Gas to execute, with Gas costing a variable amount of Eth as prices fluctuate [4].

2.1.2 Ethereum Accounts

Ethereum relies on each user having a set of private and public keys that they generate in order to interact with the blockchain. Ethereum uses this to assign Eth to users, and users can securely identify themselves to spend their Eth. This is important to this project as being able to uniquely identify a user and verify their actions is integral to a digital legal contract.

In Ethereum a private key is a random 256-bit number. All 256-bit numbers are valid, and a user can create a new Ethereum account simply by generating a new number. The chance of two users generating the same address is likely to never occur assuming they use a suitable source of randomness. Once this private key is generated, the corresponding public key is computed using an Elliptic Curve Digital Signature Algorithm (ECDSA), specifically Secp256k1 [5]. This public key represents a point on the curve that can be derived from the private key. Lastly, the public key is hashed using the Keccak algorithm [6], with the right-most 160bits of the hash forming the user's 40 digit long hexadecimal account number [7]. This account number uniquely identifies an Ethereum user [8].

2.1.3 Signing Algorithm

The secp256k1 curve predefined a prime number n and generator point G as part of its specification [5], with h being the message being signed represented as a number. The public key itself is a coordinate generated by multiplying the private key with G. When signing data an ECDSA signature is generated by choosing a random number k, and using this with the coordinates G to generate a new point on the curve R. The x-coordinate of this point is denoted r. From here the signature s is calculated as specified in equation 2.1.

$$s = k^{-1} \times (h + r \times PrivateKey) \mod n \tag{2.1}$$

The signature is then given the two numbers r and s. From the signature r, s, the original message h, and the public key you can then verify the signature using equations 2.2 and 2.3.

$$s_1 = s^{-1} \mod n \tag{2.2}$$

$$R' = (h \times s_1) \times G + (r \times s_1) \times PublicKey \tag{2.3}$$

This should recover the point R' from the public key. If the x-coordinate of R' matches r (the x-coordinate of the original point R). Then you have verified that whoever provided the signature knows the private key to the public key used [9].

This algorithm also has the advantage that it is possible to recover the public key from just the signature r, s and the message h. However, there is sometimes ambiguity in the calculation with multiple points being possible without the public key. To resolve this Ethereum signatures add an extra parameter v to the signature forming r, s, v. This extra parameter specifies which of the possible solutions is the correct public key making the reverse calculation deterministic [10]. For this project this means when given an Ethereum signature using this method you can calculate the Ethereum address of the person who signed it without needing to keep track of extra information.

2.1.4 Wallets

Users must generate a private and public key set in order to interact with the blockchain. Applications that use the blockchain need access to these values in order to perform actions on the user's behalf. This creates a problem where the user is forced to expose their private key to a third party application that they might not trust. The way this is resolved is with Wallet's to manage a user's account. When applications want to spend a user's Eth or create a signature they must call an API to interact with the user's locally installed wallet. The wallet will then prompt the user to confirm any actions, perform any computation requested using the private key, and then pass only the result back to the application. This allows the user to interact with any third party applications that they may not entirely trust without the risk of exposing their private key.

An important current implementation of this is the Web3 API [11]. This is a Javascript API that web applications can use in order to make calls to an Ethereum network. It defined many actions such as signatures, as well as methods to interface with EVM ABI calls. By default, it provides some basic functionality for debug purposes, but many of its methods are non-functional. Instead, a user will install a wallet such as MetaMask [12] as a browser extension, providing it with their private key to store locally on their device. Whenever they load a webpage using Web3, MetaMask will inject itself into the webpage and intercept all the Web3 calls made by the application.

2.2 Alternative Systems

Another Blockchain based contract signing system already exists, OpenLaw [13] (OpenLaw.io specifically, not to be confused with a New Zealand based

‘OpenLaw’ website). OpenLaw already achieves most of the goals of this project, providing a legally backed method of creating and signing smart contracts on the Ethereum blockchain, all behind a user friendly interface. It also provides significant functionality and developer API’s for extending it and creating your own compatible smart contracts. However, this project seeks to distinguish itself from OpenLaw in terms of how reliant it is on the blockchain versus other services. While OpenLaw uses the blockchain to store some smart contract information and the signatures, it still relies on an external service being run in order to store the metadata and locations of the contracts. Meanwhile, this project seeks to achieve the same goals while storing all persistent information completely on the blockchain. Overall one of the points of evaluation for any developed system will be the advantages and disadvantages to different levels of centralization of data. It may be that running completely on the blockchain or completely centralized with a trusted source is best, or there are crucial advantages to OpenLaw’s hybrid data storage approach.

3. Approach

From background research into prior examples of implementing legal contracts on the blockchain it was decided that developing a prototype implementation could help contribute to ongoing research in the area. With the goal of overall evaluating an implementation and identifying where blockchain succeeds in this area - as well as further obstacles that may need to be overcome in the future. While other solutions such as OpenLaw exist, none currently are run purely on the blockchain without the need for any external services.

This prototype will be implemented using web technologies and the Ethereum blockchain. As discussed earlier Ethereum has its own language, Solidity, for implementing smart contracts and code that can be executed across the blockchain. The code for these smart contracts can be easily written in a normal imperative programming style, while still taking advantage of the distributed consensus of the Ethereum blockchain. Additionally, several suites of tools such as Truffle made it easy to interface with Ethereum smart contracts from existing web technologies such as React. These tools will allow for rapid iteration and development, allowing for the ability to change course and overcome obstacles as they appear.

Once a prototype has either been developed the following criteria will be used to help benchmark and determine the feasibility of the proposed solution. Firstly, smart contracts cost Eth to run, which has a real world value. The cost of managing contracts on the blockchain using this prototype should be estimated and should not be significantly more than third party mediators for online contracts (i.e DocuSign). The system should be able to scale to increasing numbers of contracts and users without increasing costs or processing time. The design of data structures will be important to achieving this. Finally - while not important in the overall goal of showcasing blockchains ability to manage contracts - the application should have a focus on usability. This usability should be present in both a practical humancomputer interaction design, but it should ensure that the final application actually meets the needs of those who would use it (i.e. lawyers).

4. Design

4.1 Overview

This project has designed and developed a prototype implementation of an application for signing legal contracts on the blockchain. While it is a prototype with a minimal set of features, it was made with the goal of being a usable tool by someone less unfamiliar with blockchain technology. As such, it utilizes a number of web technologies such as React to create a user interface on top of the developed Ethereum smart contracts. The web application then uses the Web3 interface and a Web3 compatible wallet implementation in order to interact with the Ethereum network on the users behalf. Management of private keys and authorization of its use is managed by an existing Ethereum wallet such as MetaMask. Exactly how these components interact can be seen in figure 4.1.

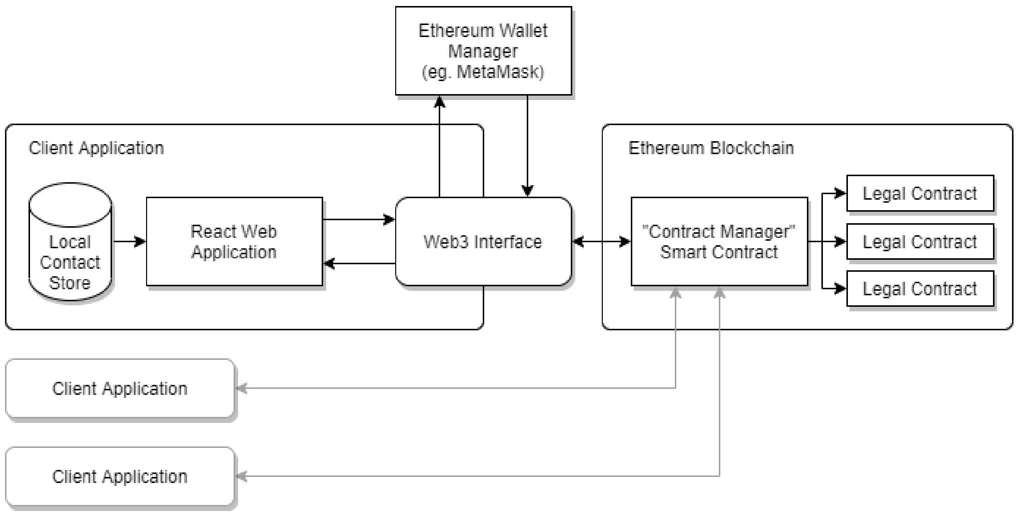


Figure 4.1: High level overview of interaction between client services and the blockchain

This prototype combines several existing frameworks, each of which was chosen for specific features or the ability to speed up prototype development. In particular, as previously mentioned, Ethereum is uniquely suited for blockchain development due to the versatility of its smart contracts. React and the corresponding JavaScript/Ethereum interface Web3 were chosen as they are packaged into the Truffle Suite of tools. This suite is well supported, and allowed for rapid development and iteration of the prototype.

This prototype is a minimum viable product, providing a very limited set of functionality while still demonstrating the practicality of signing documents on the blockchain. The initial goals for this application were to be able to: create a contract on the blockchain associated with a document, add other participants to the contract, have all participants be able to verify the authenticity of the document, and finally allow all participants to sign the document using their private keys.

4.2 Ethereum / Solidity

The most important component of this project was the design of the Solidity code that would manage computation on the blockchain. In particular, it needed to be able to execute the following workflow: create a contract, add other users as participants, allow all participants to sign the contract, and finally validate any signatures on the contract. However, the system should also be flexible and extensible to support a variety of different legal contracts. This extensibility needed to be balanced against having a consistent client interface capable of using these smart contracts. Therefore it was important to establish a strong data structure with stable EVM ABI (Ethereum Virtual Machine Application Binary Interface). Any client can interface using the standardised EVM ABI, while the smart contracts can add additional functionality. A UML class diagram of this Solidity design can be seen in figure 4.2.

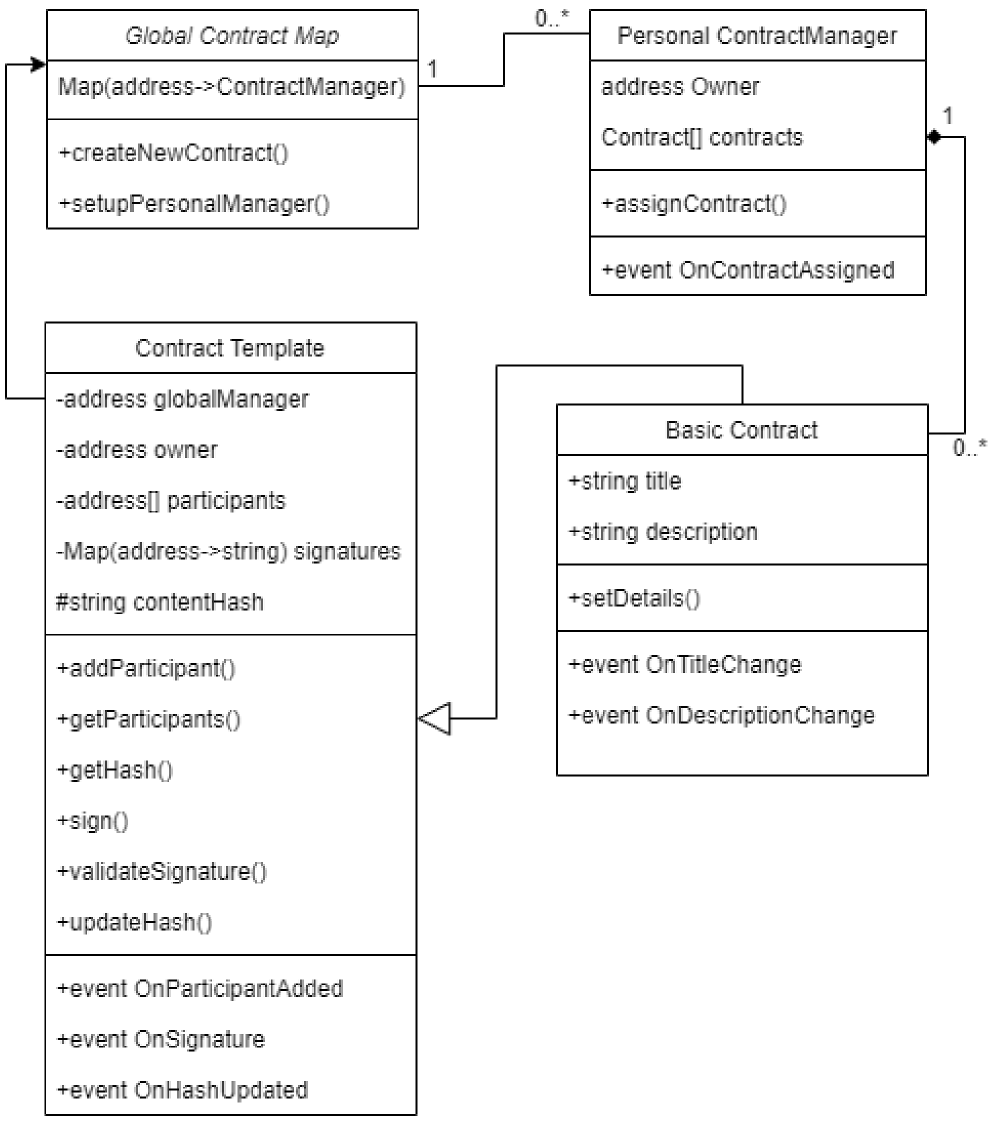


Figure 4.2: UML Class diagram of contract interfaces, with example 'Basic Contract' implementation

In this UML diagram there is a *Global Contract Map (GCM)*. There is a single instance of this on the blockchain that any client has a fixed reference to. This serves as a starting point for any look ups involving contracts. From this, there is an instance of the *Personal Contract Manager (PCM)* for every user that has created a contract, or is a participant to one. The mapping from the GCM allows users to find their contract manager from their Ethereum address. Where their contract manager contains all the details about contracts they are a participant to. The addition of the personal contract manager as a separate instance for each user, means that there is a setup cost the first time someone uses the system. Other systems such as storing various maps and data structures under the GCM were explored. However, the addition of the personal contract manager was a balance between initial setup costs and ongoing costs/speed of accessing data as the number of contracts in the system scaled.

Finally, the most significant part is the *Contract* itself. All contracts are instances of the base class *Contract Template*. This base class provides all the basic functionality for creating a contract, signing it, adding participants, and validation. This public interface for this template should remain consistent allowing client software to interact with any contract that implements it. *Basic Contract* is an example of this implementation, which simply adds some metadata for the UI in the prototype client. An example of how these methods and events are used to create and sign a contract can be seen in the time sequence diagram in figure 4.3.

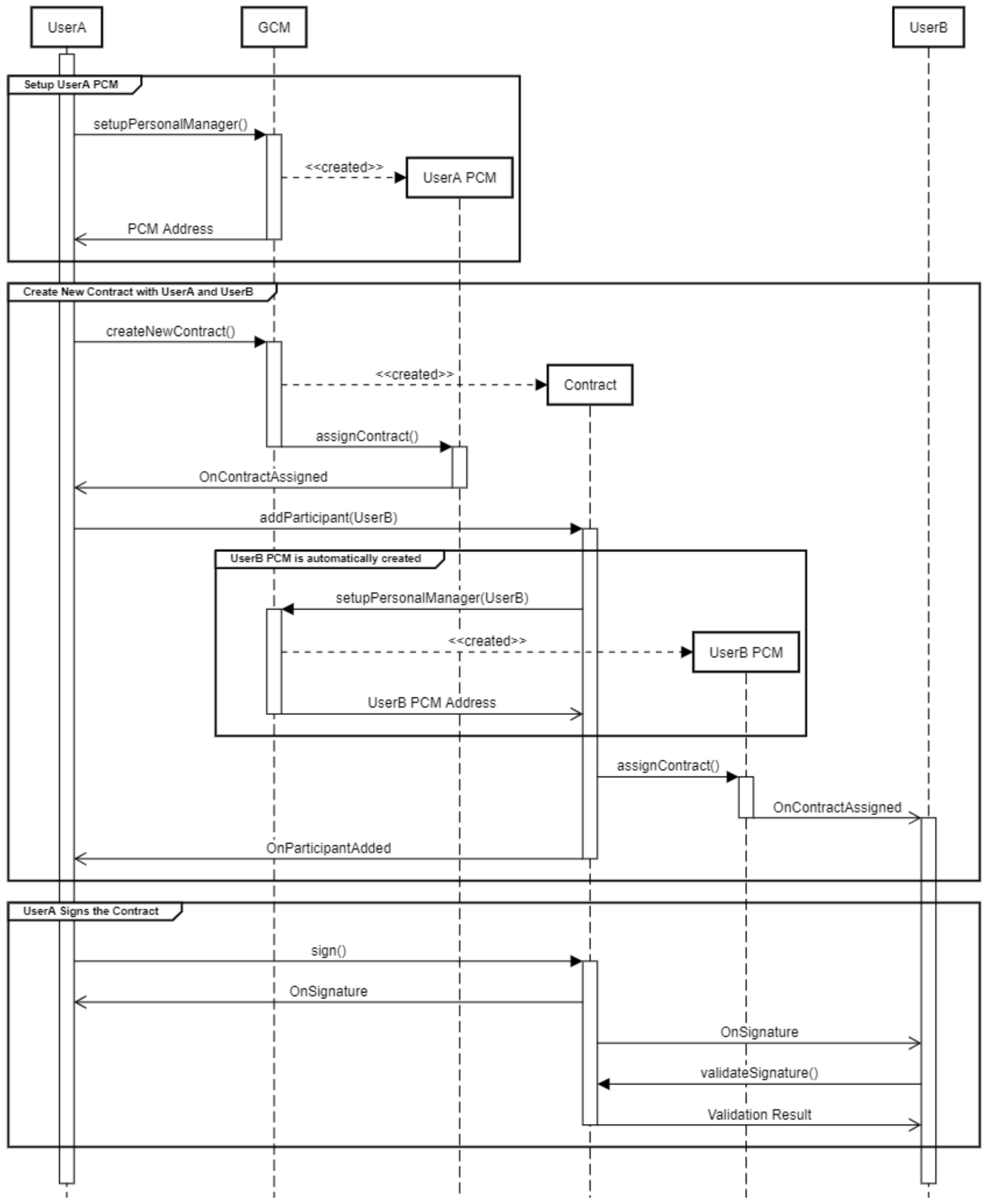
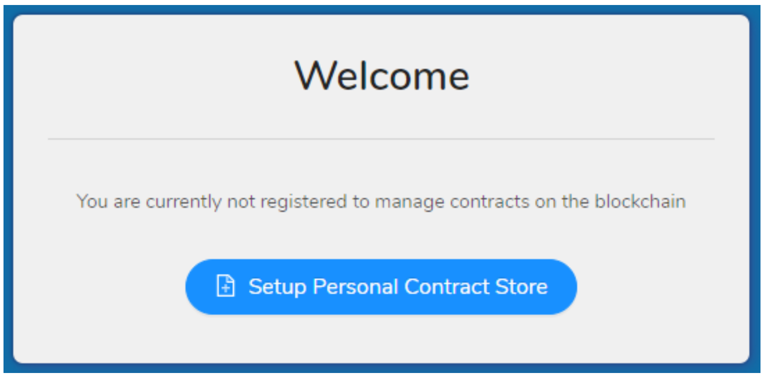


Figure 4.3: Time-Sequence diagram of two users without *Personal Contract Managers* creating and signing a contract

All of these classes have a base set of public methods, as well as events that broadcast changes. In particular most of these classes can be extended to add additional functionality. For example, standard contracts could be developed for specific purposes. A tenancy smart contract could be developed, that tracks additional parameters such as rent and payment. The Ethereum blockchain would then also allow enforcement of constraints in the code such as limiting rent increases to every 6 months. Additional functionality can be easily added due to the intentionally minimal standard interface. While not a feature intended for every user, the *Personal Contract Manager* can also be extended, to change up the data structures for potential power users, as long as it still exposes the same public interface. Allowing for more specialised management of large amounts of contracts.

4.3 Web Client

This project also saw the development of a web interface capable of interacting with an Ethereum network, with the above smart contracts deployed on it. While this can be deployed as a web application, it requires no central server and no connection to any services other than the blockchain itself. It was developed as a single page React application, that uses Web3 to interface with the blockchain. The address of the *Global Contract Manager* is hard coded into the application, with other aspects such as the user's account the blockchain network used are handled by the user's installed wallet. The wallet (such as MetaMask) provides a backend for Web3 calls, and will prompt the user for confirmation on actions that require their private key. This allows the user to use the application without worrying that they are exposing their private key.

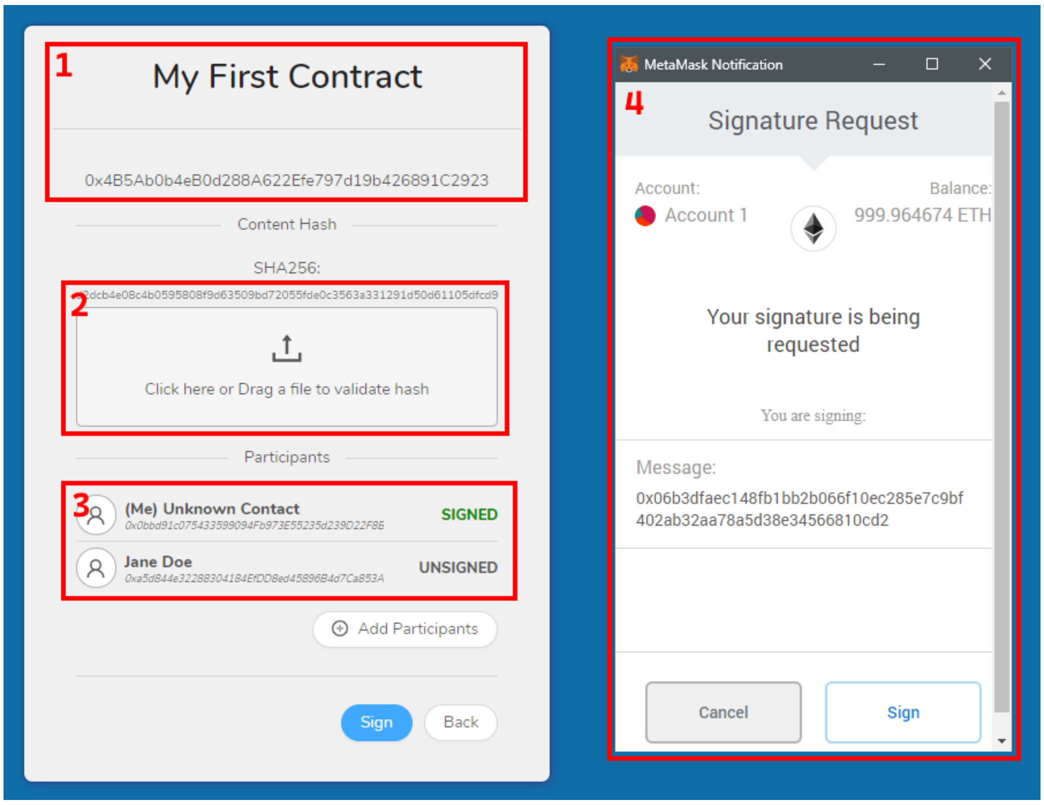

Welcome

You are currently not registered to manage contracts on the blockchain

Figure 4.4: Initial welcome screen for the web application for unregistered users

When users first open up the application the application searches for a *Personal Contract Manager* instance using their public key. If one is not found, they are presented with the welcome screen seen in figure 4.4. Users must pay a small upfront cost in order to generate the appropriate data structures on the blockchain and are prompted to do so. Although, the user can generate the data structures on behalf of another person automatically by adding them to a contract as a participant. Once this has been done users can access the main functionality of the application.

The application will query the *Personal Contract Manager* - once it is created - for any contracts the user is a participant to, displaying them in a list while also allowing the creation of a new contract through a short form. Users can then select contracts for a more detailed view.



My First Contract

0x4B5Ab0b4eB0d288A622Efe797d19b426891C2923

Content Hash

SHA256:
[illegible]2dcb4e08c4b0595808f9d63509bd72055fde0c3563a331291d50d61105dfcd9

Click here or Drag a file to validate hash

Participants

(Me) Unknown Contact 0x0bbd91c075433599094Fb973E55235d239D22F8B SIGNED

Jane Doe 0xa5d844e32288304184EfDD8ed45896B4d7Ca853A UNSIGNED

Signature Request

Account: Account 1

Balance: 999.964674 ETH

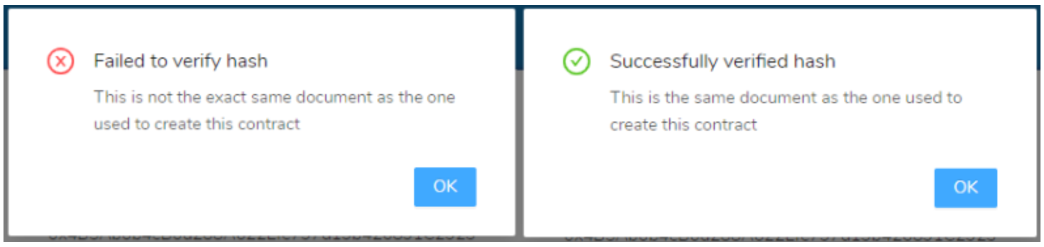
Your signature is being requested

You are signing:

Message:
0x06b3dfaec148fb1bb2b066f10ec285e7c9bf
402ab32aa78a5d38e34566810cd2

Figure 4.5: Screenshot of the contract signing interface, with MetaMask prompt

The screenshot seen above in figure 4.5 shows different parts of contract information from the *Basic Contract* template defined in Solidity, as well as the signing interface. The section labeled ‘1’ indicates some contract metadata such as title for organization, and the contracts address within the blockchain. Section 2 shows information about the contract itself. In this case a SHA256 of the original document designed, users may drag and drop another file onto the panel to validate that the hash of the file matches the original document. The conformation prompts for this can be seen in figure 4.6. Section 3 lists the addresses and names of all other participants to the contract, and details as to whether they have signed the contract or not. Any issues with validating the signature of a user is reported here. Below this is also controls for adding additional participants from a users contacts list. In this prototype contacts are stored locally on the device. Lastly, section 4 demonstrates the MetaMask prompt when a user attempts to sign the document. The application only sees the resulting signature and never the user's private key.


Failed to verify hash

This is not the exact same document as the one used to create this contract

Successfully verified hash

This is the same document as the one used to create this contract

Figure 4.6: UI for the user attempting to validate whether files as the same as the original document

Overall, users have the ability to create contracts, share them with other users. Any participants to a contract can validate whether a file hash matches that presented, and then submit a digital signature of that hash to the blockchain. Any participants can then validate that these signatures match the hash, and the public key of the user who supposedly signed them.

4.4 Alternative Designs

Throughout this process there were a few alternative approaches to various design decisions were tested. The first of these was the design of the Solidity data structures on the blockchain. Initial designs did not use a Personal Contract Manager (PCM), instead having all contracts stored in lists inside the Global Contract Manager (GCM). This had the advantage that users did not need to perform upfront computation - and in turn have an upfront cost - before using the system. Instead a contract could be assigned to any valid Ethereum address without creating any new data structures except a new contract instance. This can be seen in figure 4.7. However, making this efficient and scalable proved troublesome. The client would have to iterate over many contracts in order to find ones that the user was assigned to. A similar issue with events where users would not easily be able to subscribe to events relevant to them specifically and would have to filter through parameters of more generic events. Overall benefits of creating new data structures per user outweighed the upfront costs, and helped reduce long term computation costs. This also had the added benefit that users could in theory provide their own PCM implementation to customize their organization to their contracts within the blockchain.

Secondly, in this prototype the contract itself - at least in the BasicContract template - is represented as the SHA-265 hash of the document. Users can verify a document sent to them by other means such as email is the same, but currently the document itself cannot be stored in the blockchain. While some other alternatives were explored they were not implemented in the final prototype. The first was to store a reference to a document stored on the InterPlanetary File System (IPFS), a globally distributed file storage service. However, this is less practical for end users and requires significant effort for integration. The other alternative would have been to lean more heavily into code-as-law ideas, with the details of the contract defined in solidity code. While no specific example of this was made, the ability to create your own implementations of the contract template was made with this in mind.

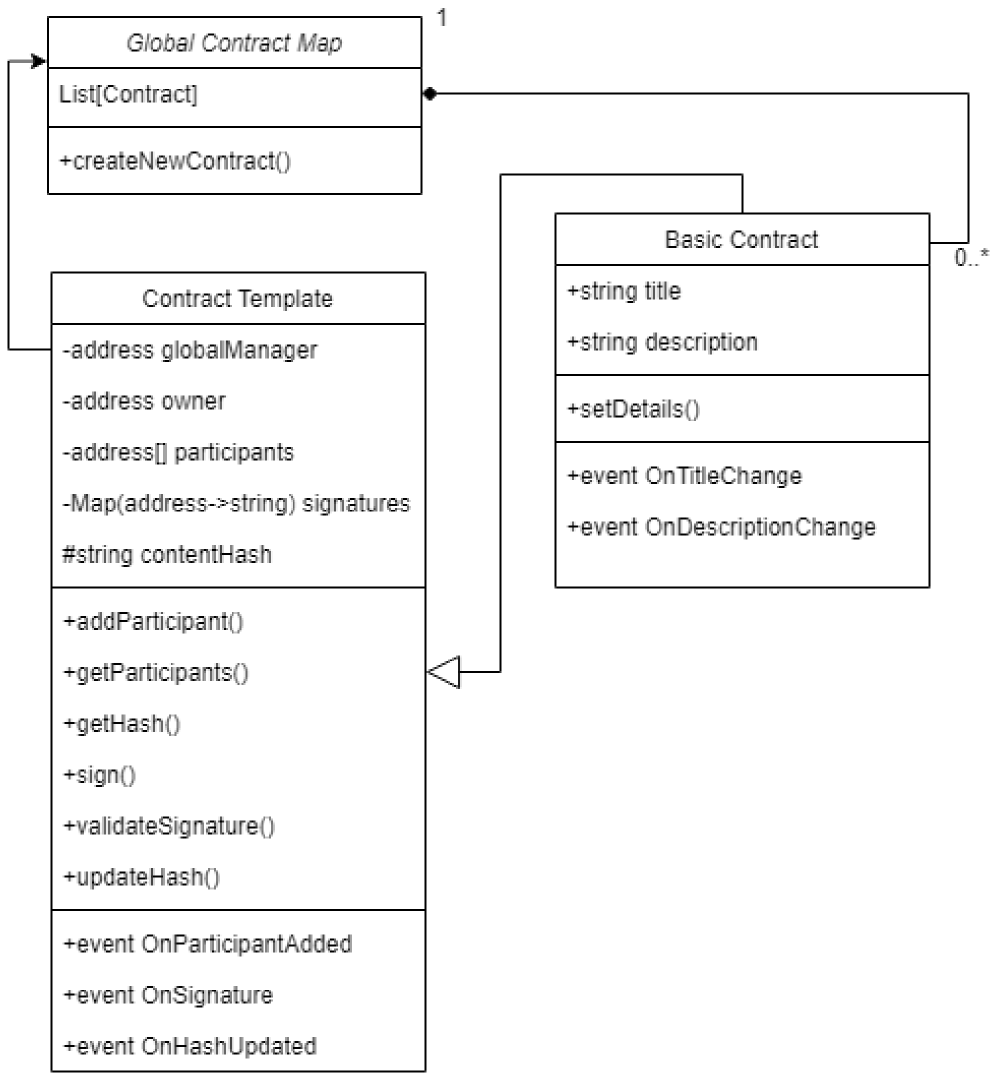


Figure 4.7: Alternative UML Class diagram of contract interfaces that was trialed

5. Results

The final system is a working Ethereum application with a Global Contract Manager and templates being able to be deployed to an Ethereum network. The web client when pointed at a GCM deployment can begin reading directly from the blockchain. A lookup checking whether the user has a Personal Contract Manager is performed, prompting the user to create one if it does not exist. If it does exist the application begins automatically reading a list of all contracts the user is a participant to. Users can then view details on any of these contracts or create a new contract from scratch providing details and a file to hash. When viewing the details of a contract the owner can add additional participants, and anyone who is a participant can sign the contract. Users are prompted for confirmation from their wallet provider for any action that requires their private key (signatures, or modifying data on the blockchain).

As described, the Solidity code stores all application state on the blockchain requiring no centralised services to operate. This includes data structures to look up all contracts a particular Ethereum address is participant to. A name for each contract, as well as a hash of the original file, any participants to the contract, and any signatures they have submitted. Verification of signatures can both happen on the blockchain as well as on the client. The only data important to the application that is not stored on the blockchain is the users' contacts (a mapping of Ethereum addresses to real names) so that they can easily see who is a participant to a contract. Instead this data is stored locally on the users' device. Overall this satisfies the original goals of the prototype from a technical perspective and demonstrates that a system to sign contracts running entirely on the blockchain works.

There are however a few areas in which the current prototype is lacking in terms of a complete end-to-end system. Firstly, there is currently no secure mechanism for confirming the owner of a particular Ethereum address. In order to be completely certain in the identity of a particular address they would have to be exchanged in person, or through another already established form of identity. Once the user has verified the owner of a particular Ethereum address, then this prototype allows the user to verify any signatures that they make on any contracts.

The second major limitation comes to the sharing of the contract definition itself, as a document. Currently the system expects the terms of the contract to be written in a document, then a SHA-256 hash of this document is what is stored on the blockchain and signed. In order for any participants to see the document they must be sent it via a different method such as email. Once all participants have the document, however, the system allows them to verify that its hash matches that of the original document.

5.1 Eth Costs

In order to create and manage the necessary data structures Eth must be spent in order to pay for the computation. Each instruction on the EVM costs a certain amount of gas. Some common actions for this application and their associated gas cost can be seen in table 5.1.

Tasks	**Gas**
Create Personal Contract Manager	231922
Create First Contract	974883
Create Additional Contract	959657
Sign (First Signatory)	146715
Sign (Additional Signatory)	52515
Add Participant (Existing Store)	106842
Add Participant (Create Store)	335913
Add Many Participants (2, Existing Store)	188453
Add Many Participants (2, Create Store)	643016

Table 5.1: Gas costs for executing common actions on the blockchain

The table above makes a distinction between the first time an action is performed and subsequent times. Some of Ethereum's data structures such as the map appear to have increased costs for the first element inserted as a form of initialization. These results also test the results of 'bundling' certain actions in order to measure the total overhead for each transaction relative to the cost of the computation itself. Extra solidity code was written to perform the same actions in a single transaction. For example, we can see that the cost of creating a Personal Contract Manager and then adding that person as a participant later only costs about 3000 gas more than if those actions were bundled into a single transaction (338,764 separate vs 335,913 bundled). This can be seen again when adding one participant at a time in separate transactions versus creating a method to bundle them all at once, although here the difference is much larger at 25,000 gas (213684 separate vs 188453 bundled). This difference in gas is likely that the bundling can take advantage of using loops rather than simply running the exact same instructions one after each other. However, the overall increase of gas costs in both cases is negligible compared to the overall gas costs.

These gas costs are a fixed value with specific EVM instructions costing a certain amount of gas. In order to actually have your computations run on the blockchain you need to bid on how much Eth you are willing to pay to have it executed. Nodes will then prioritise tasks with the highest bounty, essentially meaning that the more you pay the faster your transaction is processed. Due to the small costs associated with many transactions Ethereum defines a new unit Gwei as 10^{-9} Eth. The cost to have a transaction processed within a certain is relatively stable but does vary over time. Using Eth Gas Station [14] as of October 4th 2020 it costs about 38 Gwei per gas in order to have your transaction processed in 30 minutes or less. Additionally the current pricing for 1 Eth at the same time is approximately NZD$520 [15]. Therefore the Gwei and real costs of the above transactions can be seen in table 5.2.

Tasks	**Gwei**	**Cost (NZD)**
Create Personal Contract Manager	8813036	$4.61
Create First Contract	3704554	$19.37
Create Additional Contract	36466966	$19.06
Sign (First Signatory)	5575170	$2.91
Sign (Additional Signatory)	1995570	$1.04
Add Participant (Existing Store)	4059996	$2.12
Add Participant (Create Store)	12764694	$6.67
Add Many Participants (2, Existing Store)	7161214	$3.74
Add Many Participants (2, Create Store)	24434608	$12.77

Table 5.2: Costs of common actions in both Gwei and NZD for approximately 30 minute execution time

In a practical cost, a common scenario might be someone creating a contract, and adding two participants. It might be one of the participant's first time using the system and need a Personal Contract Manager set up on their behalf. All three (owner and two participants) will then need to sign the contract. The costing for this scenario can be seen in table 5.3 below. From this we can see that the total cost of this scenario would be approximately NZD$35, however this would fluctuate depending on Eth cost and current Gwei bidding.

Action	**Cost (NZD)**
Create Contract	$19.06
Add a Participant (existing PCM)	$2.12
Add a Participant (create PCM)	$6.67
First Signature	$2.91
Remaining 2 Signatures	$2.09
Total	**$32.86**

Table 5.3: Cost for various steps in a typical scenario for managing contracts

5.2 Survey

Finally, in order to better evaluate the success of this prototype from a user perspective a survey was designed explaining key features of the software and sent to people with a legal background. This survey asked them about common contracts that the tool might be used for, average number of participants, how much they might expect such a service to cost, and how long they would be willing to take for actions to process. Initially this was going to be a fairly widespread survey, but was instead limited to 8 participants in the end. This was due to the difficulties in describing the benefits blockchain provided to those without a technical background. In particular the concept of trust in a blockchain, as well as the link between costs and processing time in a legal context. Participants were able to understand with one on one questions but it would be too time consuming to extend the survey much further.

From this limited sample size some useful information was still found. Most common contract types are short form contracts such as employment agreements and rental agreements. The majority of contracts only ever have two parties involved. Overall those surveyed did not seem overly concerned with both costs and processing time, with 30 minutes being plenty fast enough in their opinion. Most were also largely unconcerned with price, although as those working in legal fields they intended to simply pass any costs onto the client.

6. Discussion

This project has succeeded in showing that you can create a completely decentralised system for signing contracts, albeit not without costs and limitations. Some of these limitations could be overcome with further development, while others may require additional research into potential solutions.

The current system stores all metadata and data structures for looking up contracts on the blockchain. This means the only information a user needs is their Ethereum address and an appropriate client to begin using the system. However, due to the public nature of the blockchain, this has the consequence that if you know anyone's Ethereum address you can look up any contracts they are a participant of. Given that this system is specifically for contracts, this can have significant privacy issues. It would in theory be possible to subscribe to the provided events in order to listen for whether particular people are participating in contracts together. For example, as a business, it would be possible to discover if competitors had agreements with each other, even if the remaining metadata was obfuscated. This could be solved by people having multiple accounts. However, this then introduces more complexities in being able to verify someone controls a particular account, a problem this prototype does not seek to solve. It could be possible to obfuscate metadata or not store legal contracts indexed from a single root smart contract (the GCM), however, this would require a complete redesign from the system proposed in this document.

These particular issues are solved by less decentralised systems like those of OpenLaw [13]. As mentioned, OpenLaw keeps the signing process on the blockchain but keeps the information about contracts and their metadata in a centralised system (albeit one that users can easily self-host). While this extra work on the end-user to set up and find these systems, it may be the best solution to balance between convenience, decentralisation and privacy.

This solution however does allow the user to take much better advantage of smart contracts and blockchain specific features. By implementing contract details in Solidity as a template you can enforce certain constraints on a contract. For example, a tenancy agreement might stipulate when rent must be paid, and only allow rent increases every 6 months. The Solidity could be written to calculate the rent due at certain times, with the contract stating that the solidity output is to be considered a source of truth. The blockchain could then enforce these additional constraints by only allowing specific parties to change the rent once the time frame has passed. All of this functionality is already possible to implement in the system designed by creating a new contract template. The client may need to be extended to display this additional information, and would currently require customization for all new contract types.

This customization of the client is not a significant issue, as currently it might be expected that anyone writing Solidity code could write an extension to the client. However, this exposes a wider issue that the current design does not provide any way for a particular contract to express its capabilities. A fairly complex contract could be designed, and the client in its current state would only report on basic info without indicating to the user any of the advanced functionality present. This is because clients can only rely on the basic interface methods being present. The client also has little information on what actions it is allowed to perform. Actions such as adding participants might be gated by the Solidity code, but the client may attempt to perform these actions anyway without providing useful feedback to the user. It could be useful for additional standard methods to allow the Solidity contracts to advertise which functions they support, however further work would be needed to investigate what would be the best way to approach this.

6.1 Missing Functionality

Some systems were omitted from this prototype with the idea that they could be better handled by another service. An example of this is managing contacts, a mapping of Ethereum addresses to someone's real name - or at least something the user can identify easier. Currently, this is stored locally on the user's device as part of the client. In the future a much better solution would be to create a separate browser extension acting as a form of address book for each user - it could be integrated into the user's wallet for example. This allows different applications to also take advantage of a user's contacts without having the user manually transfer them each time. It could also introduce an additional layer of privacy requiring an application to request permission to view a user's address book.

The other major component of this system that is not directly addressed by the prototype is the sharing of the contract document itself. Currently, the system uses an SHA-256 hash of the original document as the component that is actually stored on the blockchain. If both parties get copies of the document they can verify it is the same one before they sign. While SHA-256 is currently considered to be very resistant to collision attacks, other hashing algorithms such as SHA1 have examples of two very similar yet slightly tweaked documents that generate the same hash [16]. The chances of this occurring in this system are very unlikely but something to consider with mainstream use. Ideally, future work would integrate a service to easily share the document itself. As an example, the document could be stored on the InterPlanetary File System (IPFS) [17].

6.2 Transaction Costs

Overall costs of running a single contract on the blockchain came in at around NZD$32 as of October 2020 in order to run the program on the Ethereum Mainnet. This is a fairly significant cost, however, its value varies based on the type of contracts. For contracts handling large amounts of money or assets, this could be a worthwhile cost in exchange for the trust and security of the blockchain. Instead, the major issue with this is how much the cost varies over time. This price as listed is dependent on both the price of Eth relative to the local currency and the Gwei cost per gas for timely computation. The price of Eth on the Mainnet has nearly doubled since this time last year [15], a trend that may make any applications reliant on it too expensive to operate. Additionally, Gwei per has estimations fluctuate heavily throughout the day between 20 and 50 at the present time with massive spikes for some periods of time. This overall makes the Ethereum Mainnet seem poorly suited to this type of application long term. Hopefully, as the technology develops these costs will stabilize to perform a more accurate cost analysis.

This project also investigated the costs of individual actions and in particular the act of bundling actions into a single transaction to avoid the overhead costs associated. Overall it was found that bundled actions were cheaper, but not significantly so. As bundling actions requires writing new Solidity methods to perform them together it may be sensible for an application to avoid doing so. Every extra piece of code pairing actions adds complexity, makes the system more difficult to update and opens up more vectors for bugs and other issues. Instead, it seems that the primary benefit of bundling is to ensure that actions are executed in a sequence. Nodes can choose the order to execute transactions in. This means that a user of this system may have to wait up to 30 minutes for a contract to be created before they can add participants, rather than queuing the actions. Currently, Ethereum provides no method of specifying that a particular transaction is dependent on another.

6.3 User Feedback

Lastly, the results from the survey helped to evaluate some of these limitations in context. Due to the limited sample size, these results are not empirical evidence, however, can be used for some insight into real-world use cases. Those surveyed listed a very small number of contracts as those used most frequently - such as employment or rental agreements. These often follow a standard format and are reused frequently. This means that a templating system would work quite well for the system in general. A handful of templates could cover the majority of use cases for users.

As a bonus, it was found that those with legal backgrounds were largely unconcerned with processing time in most cases. Costs at the current price would simply be considered the costs of business for an important contract, and most contracts are signed over a period of days meaning processing time is less of an issue. This means that in real-world scenarios users will likely have more flexibility in how much they pay to use the system. As cost and processing speed tend to have an inversely proportional relationship on the blockchain, this means users who are willing to wait long periods of time can save a significant amount on cost and vice-versa.

The result that most contracts only ever have two participants, the owner and one other, might have implications if the system were to be redesigned. By creating a new contract template that only every has two participants would reduce the complexity of the data structures in the smart contract instance. This in turn has the consequence of reducing the cost of creating a new contract as mappings are quite expensive data structures.

Finally, the survey revealed interesting insights into the issues with potentially making this prototype into a product. There were difficulties in explaining how the system works and the benefits the blockchain provided to the problem. In particular, those with limited technical background were not able to grasp certain computing concepts such as decentralization without one on one explanations of the concepts. Hopefully, as more people become more aware of blockchain outside of cryptocurrency, many of its related concepts will become common knowledge.

7. Conclusions

Signing legal contracts on the blockchain overcomes many of the issues with traditional digital contracts. It provides the necessary requirements for a valid digital signature under New Zealand law. In particular, it satisfies the conditions that a signature must be linked to a person, and that any changes to the signature or document are detectable. The nature of linking signatures with a user's Ethereum address and the immutability of blockchain data makes this possible. Specifically, this prototype demonstrates that you can have a system where all persistent data is decentralised on the blockchain while maintaining an interface accessible to the average user.

The system as designed has some limitations in terms of the data it can represent. Storing entire contract documents on the blockchain at this time is not cost-effective, and must be shared through another method. However, the designs for Solidity code are designed to be extensible so more functionality can be built upon this prototype, in particular specific contract templates implementing aspects of code-as-law. Future work however should be done on developing how different implementations can advertise specific functionality to avoid incompatible clients.

While storing all the data on the blockchain is possible, it requires a lot of extra data in order for the client to be able to discover contracts for a particular user. This dramatically increases costs, which while still affordable would be a barrier to widespread adoption. Cost, in particular, is highly variable when considering the price of Eth on the Ethereum Mainnet. However, by storing all the necessary lookup tables on the blockchain there are significant privacy implications. If you know anyone's Ethereum address you can quickly look up all contracts they have signed using the system. Future systems may find it more practical to store more metadata outside of the blockchain in order to obfuscate information.

Overall, blockchain is a promising technology in the future of law and legal contracts. Especially as slow processing time could be seen as an acceptable trade-off for the proposed benefits to trust and security. However, the proper balance of how much information and data to offload onto the blockchain needs to continue to be experimented with through further prototypes. In the long run limitations specific to the Ethereum Mainnet and other current public blockchains, such as fluctuating costs, will hopefully stabilize. Instead, the largest barrier to entry is the average user's understanding of the technology and benefits, which will hopefully increase over time as blockchain technology becomes more prevalent in everyday life.

Bibliography

[1] *Entering Contracts In The Digital Age*, Sep. 2017. [Online]. Available: https://www.jacksonrussell.co.nz/site/jacksonrussell/Articles/Electronic%20Transactions%20-%20Entering%20Contracts%20in%20the%20Digital%20Age.pdf (visited on 09/30/2020).

[2] J. Wang, S. Wang, J. Guo, Y. Du, S. Cheng, and X. Li, "A Summary of Research on Blockchain in the Field of Intellectual Property," en, *Procedia Computer Science*, 2018 International Conference on Identification, Information and Knowledge in the Internet of Things, vol. 147, pp. 191–197, Jan. 2019, ISSN: 1877-0509. DOI: 10.1016/j.procs.2019.01.220. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187705091930239X (visited on 09/30/2020).

[3] *What is Ether (ETH)?* en, Sep. 2020. [Online]. Available: https://ethereum.org (visited on 09/30/2020).

[4] etherscan.io, *Ethereum Gas Tracker | Etherscan*, en, Ethereum Gas Tracker, Sep. 2020. [Online]. Available: http://etherscan.io/gastracker (visited on 09/30/2020).

[5] *Secp256k1*, Wiki, Apr. 2019. [Online]. Available: https://en.bitcoin.it/wiki/Secp256k1 (visited on 09/30/2020).

[6] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, *Keccak specifications summary.* [Online]. Available: https://keccak.team/keccak_specs_summary.html (visited on 09/30/2020).

[7] D. G. Wood, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER," en, p. 32,

[8] N. Adams, *Ethereum Addresses*, en, Github Repo, May 2020. [Online]. Available: https://github.com/ethereumbook/ethereumbook (visited on 09/30/2020).

[9] S. Nakov, *ECDSA: Elliptic Curve Signatures.* [Online]. Available: `https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages` (visited on 09/30/2020).

[10] D. R. L. Brown, "SEC 1: Elliptic Curve Cryptography," en, p. 144,

[11] *Web3.js*, original-date: 2014-09-30T20:50:37Z, Sep. 2020. [Online]. Available: `https://github.com/ethereum/web3.js` (visited on 09/30/2020).

[12] *MetaMask Docs.* [Online]. Available: `https://docs.metamask.io/guide/` (visited on 09/30/2020).

[13] J. deLottinville, *OpenLaw Docs.* [Online]. Available: `https://docs.openlaw.io/` (visited on 09/30/2020).

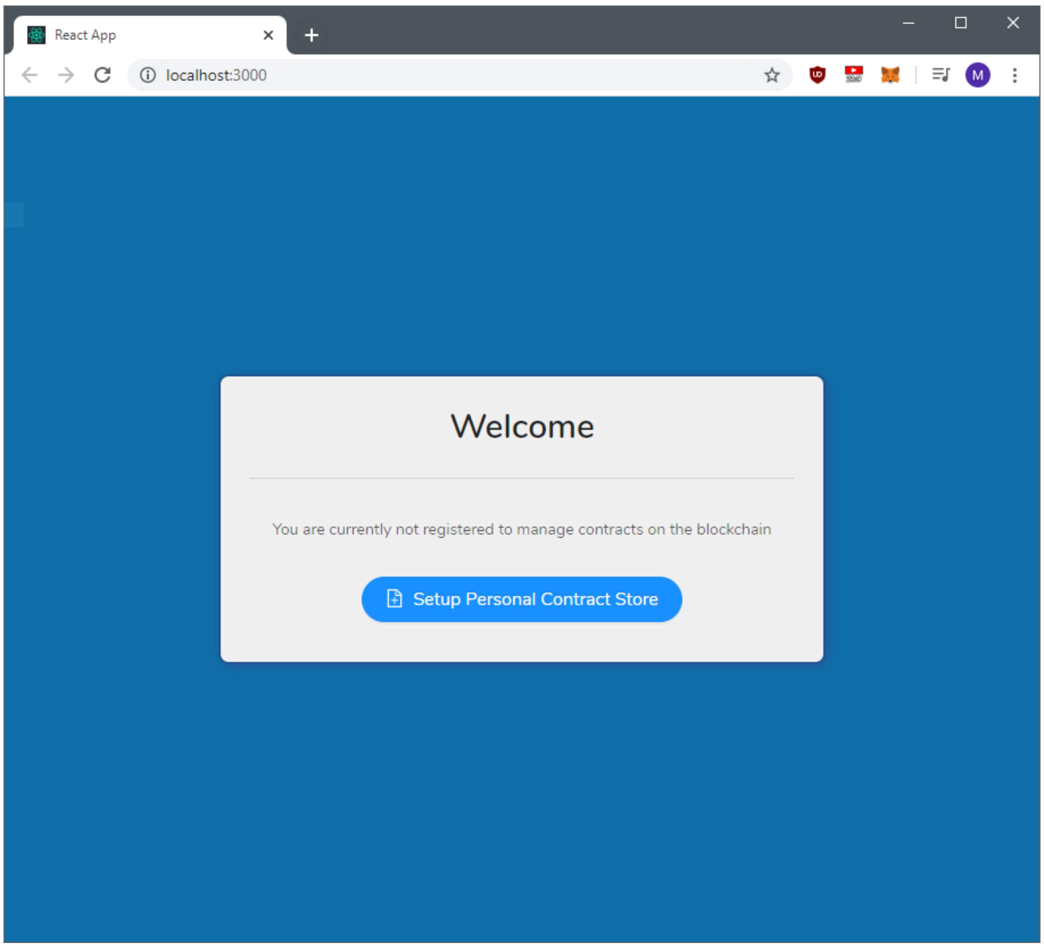
[14] *ETH Gas Station*, en. [Online]. Available: `https://ethgasstation.info/` (visited on 10/04/2020).

[15] *Ethereum (ETH) price, marketcap, chart, and info*, en. [Online]. Available: `https://www.coingecko.com/en/coins/ethereum` (visited on 10/04/2020).

[16] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The First Collision for Full SHA-1," en, in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds., vol. 10401, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 570–596, ISBN: 978-3-319-63687-0 978-3-319-63688-7. DOI: `10.1007/978-3-319-63688-7_19`. [Online]. Available: `http://link.springer.com/10.1007/978-3-319-63688-7_19` (visited on 10/06/2020).

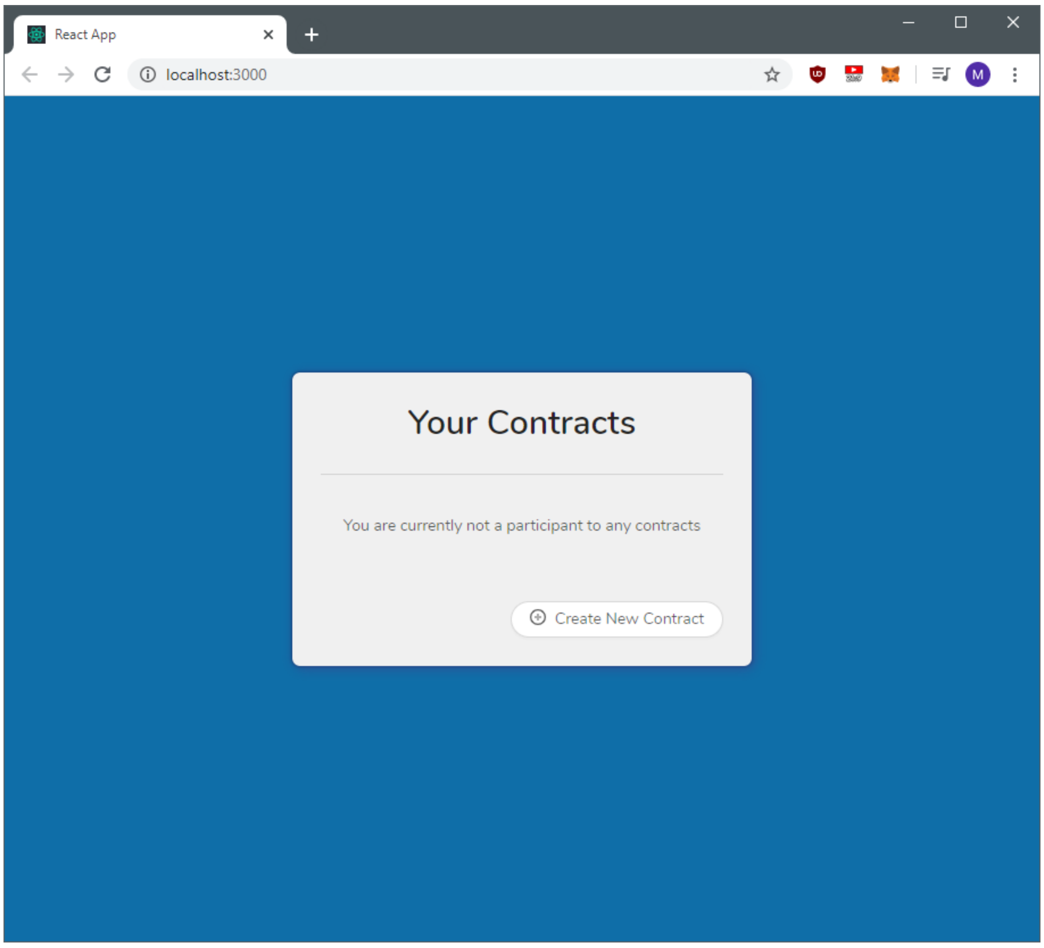
[17] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System," en, p. 11,

A. **Web Interface**



You are currently not registered to manage contracts on the blockchain

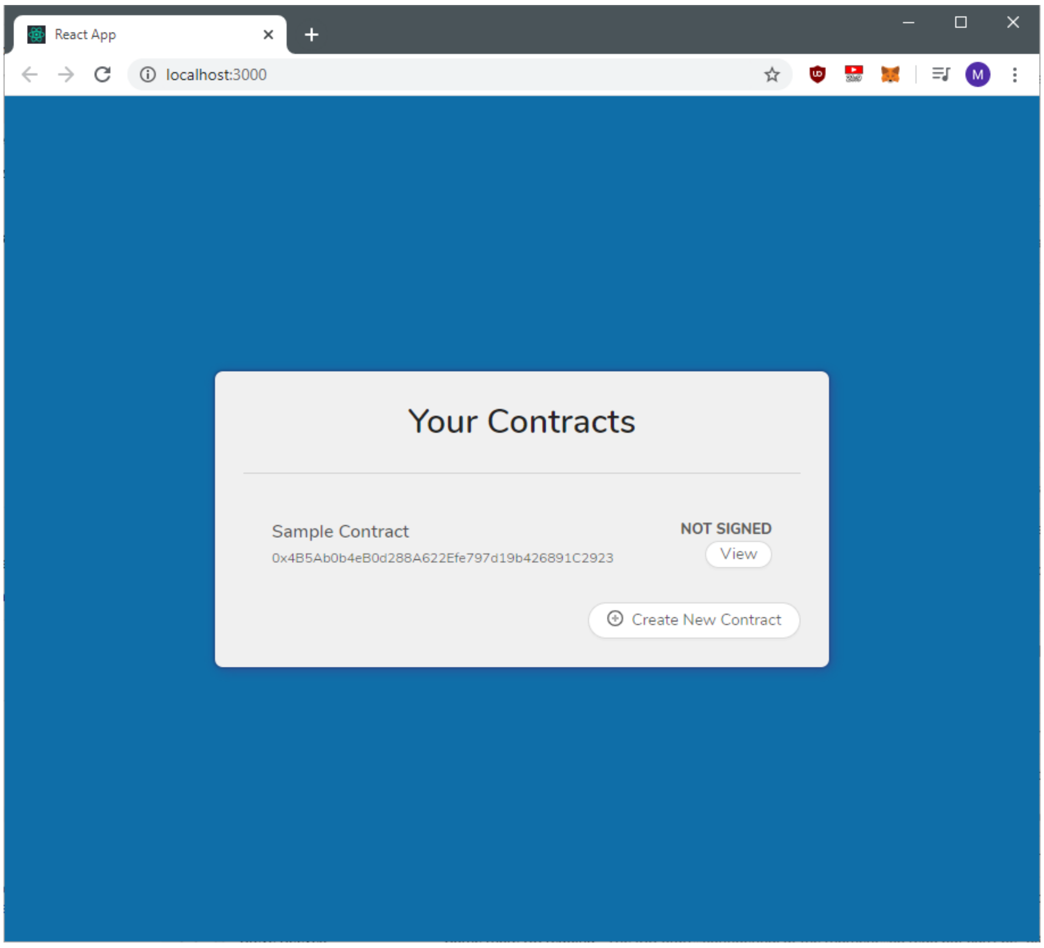
Figure A.1: Initial setup screen prompting users to create the necessary data structures



Your Contracts

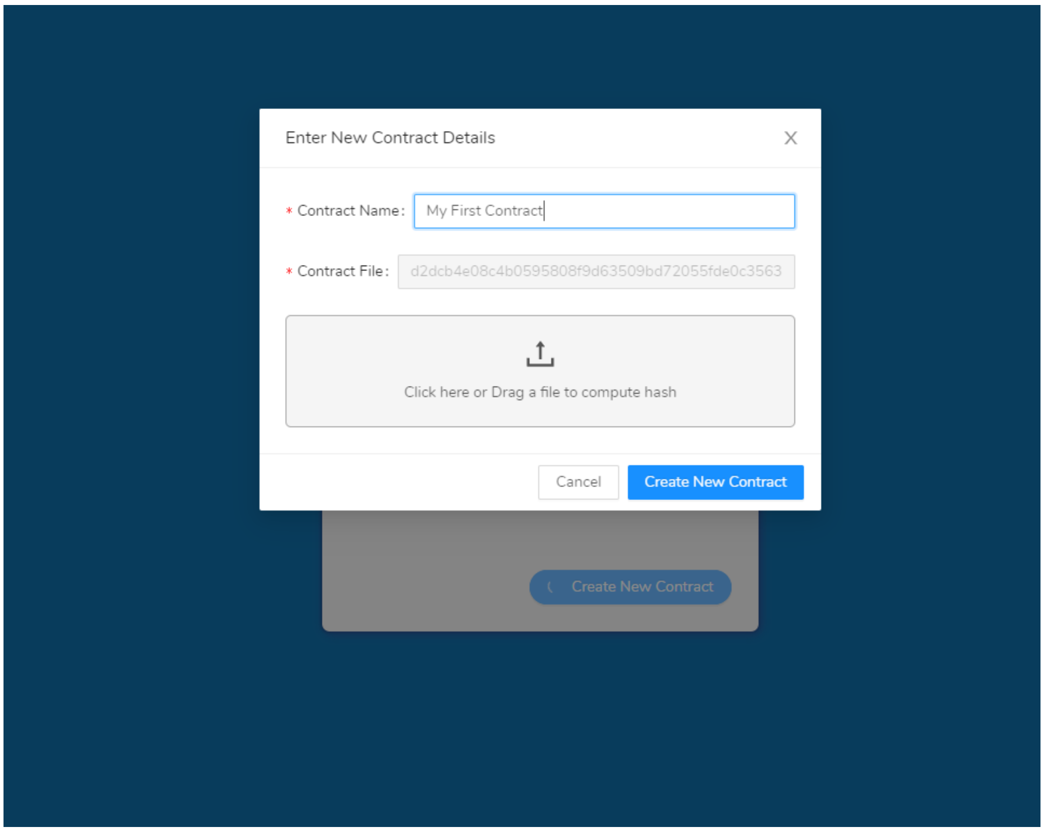
You are currently not a participant to any contracts

Figure A.2: Main screen without any contracts



Sample Contract
0x4B5Ab0b4eB0d288A622Efe797d19b426891C2923
NOT SIGNED

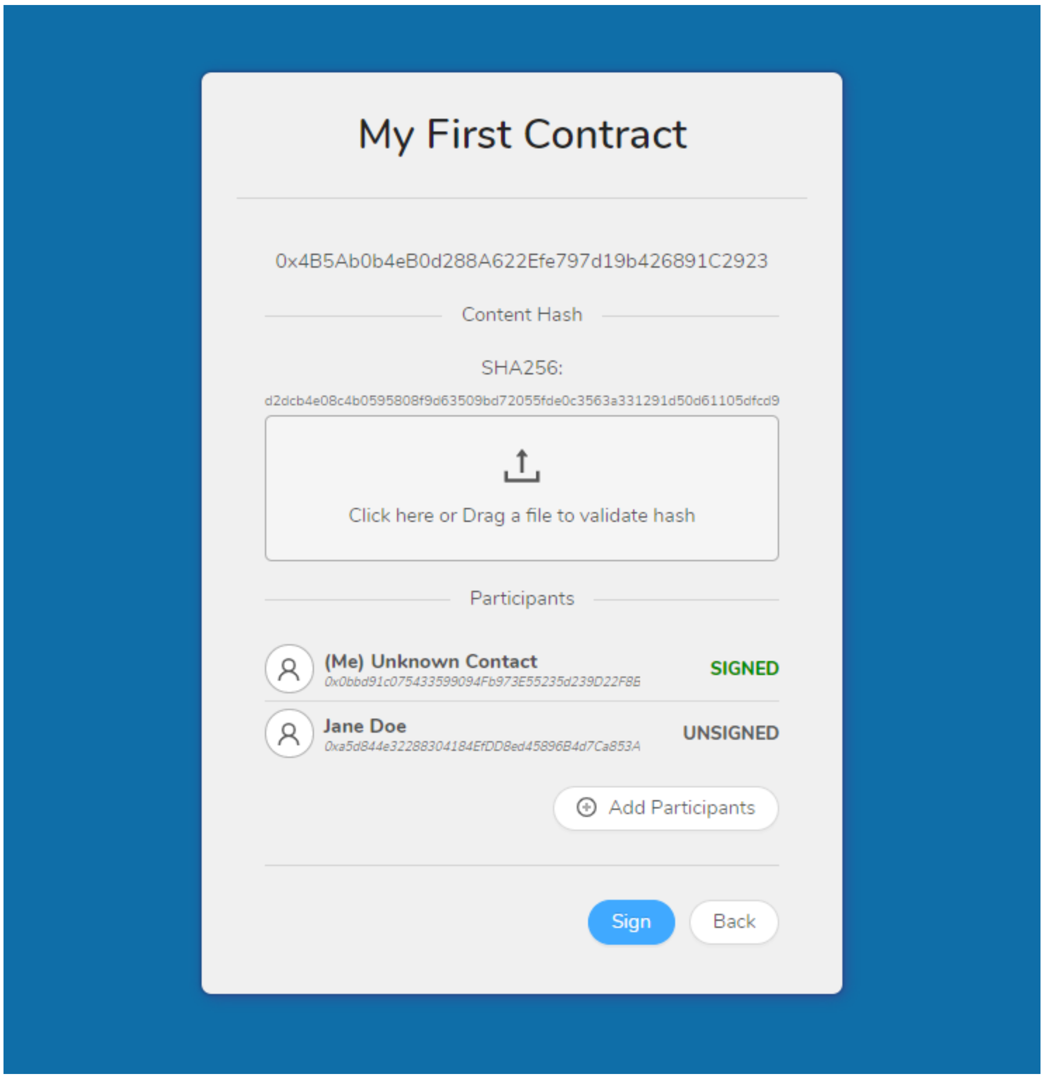
Figure A.3: Main screen with a contract the user is participant to


Enter New Contract Details

* Contract Name: My First Contract

* Contract File: d2dcb4e08c4b0595808f9d63509bd72055fde0c3563

Figure A.4: Basic Contract creation screen


My First Contract

0x4B5Ab0b4eB0d288A622Efe797d19b426891C2923

Content Hash

SHA256:

d2dcb4e08c4b0595808f9d63509bd72055fde0c3563a331291d50d61105dfcd9

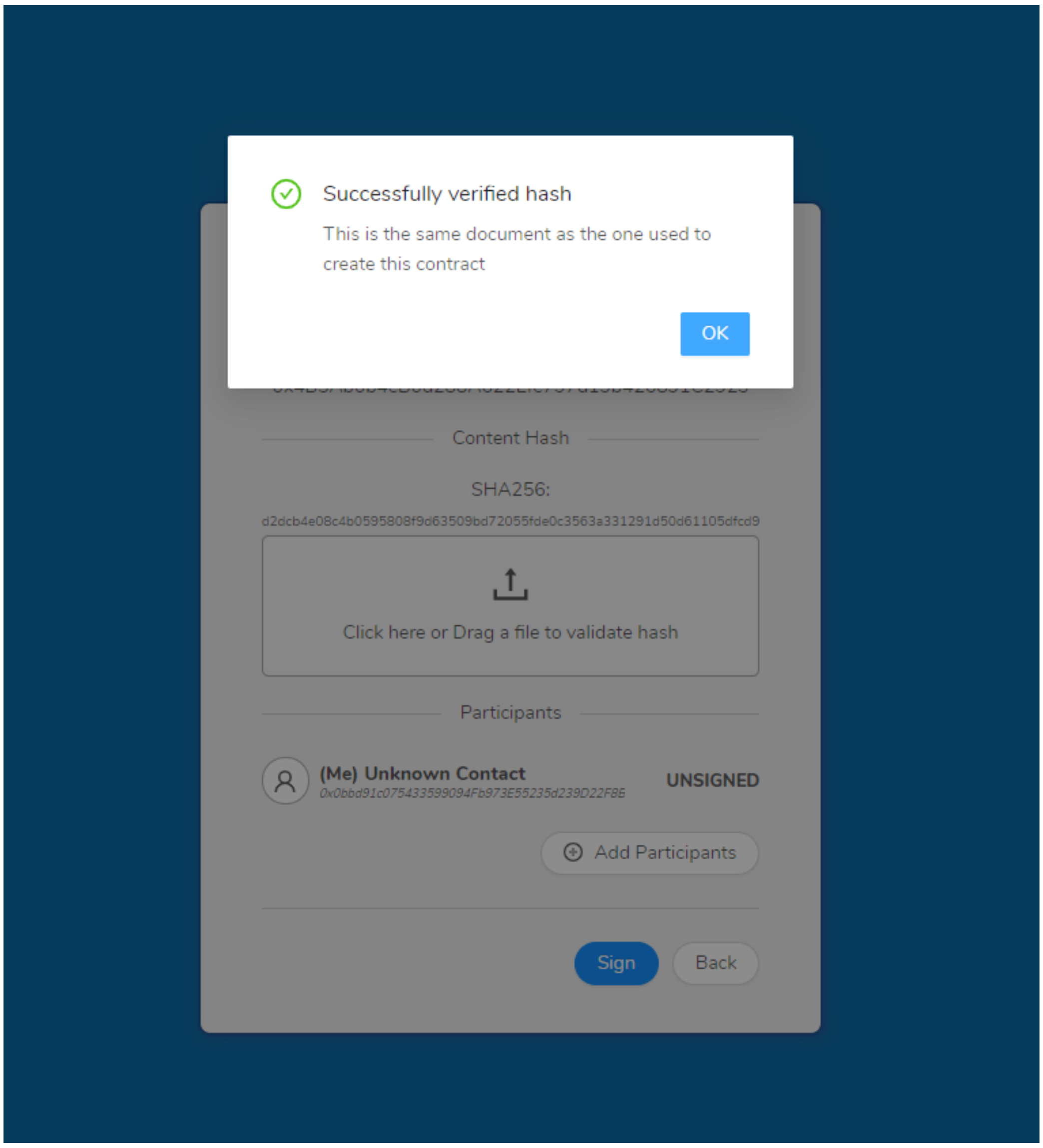
Click here or Drag a file to validate hash

Participants

(Me) Unknown Contact
0x0bbd91c075433599094Fb973E55235d239D22F8E SIGNED

Jane Doe
0xa5d844e32288304184EfDD8ed45896B4d7Ca853A UNSIGNED

Figure A.5: Detailed view of a specific contract


Successfully verified hash

This is the same document as the one used to create this contract

Content Hash

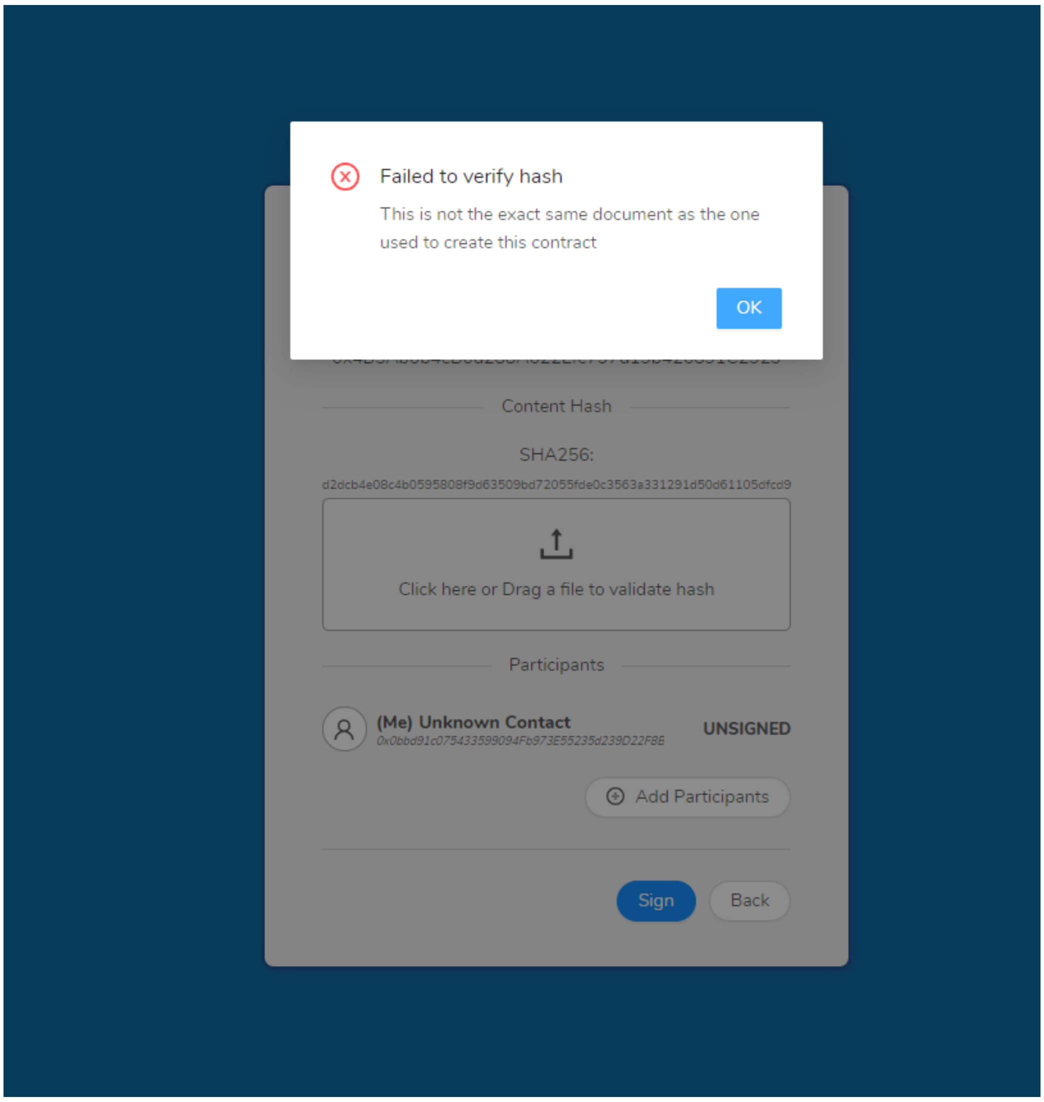
SHA256:

d2dcb4e08c4b0595808f9d63509bd72055fde0c3563a331291d50d61105dfcd9

Participants

(Me) Unknown Contact *0x0bbd91c075433599094Fb973E55235d239D22F8E* **UNSIGNED**

Figure A.6: View when successfully verifying a file



Failed to verify hash

This is not the exact same document as the one used to create this contract

Content Hash

SHA256:

d2dcb4e08c4b0595808f9d63509bd72055fde0c3563a331291d50d61105dfcd9

Click here or Drag a file to validate hash

Participants

(Me) Unknown Contact UNSIGNED
0x0bbd91c075433599094Fb973E55235d239D22F8E

Figure A.7: View when failing to verifying a file

B. Solidity Code

```
pragma solidity >=0.4.21 <0.7.0;

contract BasicContract {
  ContractManager manager;
  address owner;
  address[] participants;
  mapping(address => bool) permissions;
  string content;
  mapping(address => string) signatures;
  string title;

  constructor(address creator, string memory t, ContractManager caller) public {
    manager = caller;
    owner = creator;
    content = "";
    title = t;
    permissions[owner] = true;
    participants.push(owner);
  }

  function addParticipant(address participant) public {
    // Make sure not to readd  the same person
    if (msg.sender == owner && !permissions[participant]) {
      permissions[participant] = true;
      participants.push(participant);
      manager.addParticipant(participant, this);
    }
  }

  function addManyParticipants(address[] memory newParticipants) public {
```

```
  for (uint i = 0; i < newParticipants.length; i++) {
    addParticipant(newParticipants[i]);
  }
}

function sign(string memory signature) public {
  // Check they are allowed to sign the contract
  if (permissions[msg.sender]) {
    // TODO: Validate using ECRecover

    signatures[msg.sender] = signature;
  }
}

function setContent(string memory newContent) public {
  content = newContent;
}

function getSignature(address signee) public view returns (string memory) {
  return signatures[signee];
}

function getParticipants() public view returns (address[] memory) {
  return participants;
}

function getOwner() public view returns (address) {
  return owner;
}

function getContent() public view returns (string memory) {
  return content;
}

function getTitle() public view returns (string memory) {
  return title;
}

function getHash() public view returns (bytes32) {
  return keccak256(abi.encodePacked(content));
}
```

```
}

contract PersonalContracts {
  address owner;
  BasicContract[] contracts;

  event ContractAssigned(address creator, address location);

  function addContract(BasicContract c) public {
    contracts.push(c);
    emit ContractAssigned(msg.sender, address(c));
  }

  function getContracts() public view returns(BasicContract[] memory) {
      return contracts;
  }

  constructor() public {
    owner = msg.sender;
  }
}

contract ContractManager {
  mapping(address => PersonalContracts) personalContracts;

  function getPersonalContracts() public view returns (address) {
    return address(personalContracts[msg.sender]);
  }

  event PersonalContractsCreated(address owner, address location);

  function createPersonalContracts(address owner) public {
    PersonalContracts pContracts = new PersonalContracts();
    personalContracts[owner] = pContracts;
    emit PersonalContractsCreated(owner, address(pContracts));
  }

  function getOrCreatePersonalContracts(address owner)
    private returns (PersonalContracts) {
    if (address(personalContracts[owner]) == address(0)) {
      createPersonalContracts(owner);
    }

    return personalContracts[owner];
  }


  function addParticipant
        (address newParticipant, BasicContract contractAddress) public {
    PersonalContracts pContract = getOrCreatePersonalContracts(newParticipant);
    pContract.addContract(contractAddress);
  }


  function createContract(string memory title, string memory content) public {
    BasicContract newContract = new BasicContract(msg.sender, title, this);
    newContract.setContent(content);

    // Automatically assign the contract to the creators personal contracts
    PersonalContracts pContracts = getOrCreatePersonalContracts(msg.sender);
    pContracts.addContract(newContract);
  }

  function createContractWithParticipants
        (string memory title, string memory content,
        address[] memory participants) public {
    BasicContract newContract = new BasicContract(msg.sender, title, this);
    newContract.setContent(content);

    // Automatically assign the contract to the creators personal contracts
    PersonalContracts pContracts = getOrCreatePersonalContracts(msg.sender);
    pContracts.addContract(newContract);

    newContract.addManyParticipants(participants);
  }
}
```