Frontier Multisig

Technical Specification

Dara Lynch - 19324446
Alan McGrath - 19392951

Date of completion: 05/05/2023

1. Introduction

1.1 Glossary

DAO (Decentralised Autonomous Organization): An organisation that is governed by coded rules on a blockchain, without centralised control, and managed by its members through consensus mechanisms.

dApp (Decentralised Application): An application that runs on a distributed computing system, typically utilising blockchain technology and smart contracts.

Decentralised: A system or network that is not controlled by a central authority, distributing decision-making power across multiple nodes or participants.

ERC20: A widely-used token standard on the Ethereum blockchain that defines a set of rules for creating and managing tokens.

Escrow: A financial arrangement in which a trusted third party holds and regulates payment of funds, releasing them only when certain conditions are met.

Ethereum: A decentralised, open-source blockchain platform that features smart contract functionality.

Gas: The fee required to perform transactions and execute smart contracts on the Ethereum network.

Hardhat: A development environment and task runner for Ethereum that facilitates building, testing, and deploying smart contracts.

JIRA: A project management and issue tracking software used for organising and managing tasks in a collaborative environment.

Kanban: An agile project management methodology that visualises work, limits work in progress, and maximises efficiency.

Mempool: This is where unconfirmed transactions are stored on the ethereum blockchain. (memory pool)

MetaMask: A browser extension and mobile application that functions as an Ethereum wallet, enabling users to interact with decentralised applications directly from their browser or mobile device.

Multi-signature (Multisig) Wallet: A type of wallet that requires multiple signatures or approvals to authorise a transaction.

Private Key: A cryptographic key that is kept secret by the owner and used to sign transactions and control access to a wallet's funds on a blockchain network.

Proof of Stake (PoS): A consensus algorithm for blockchain networks in which validators propose and attest to blocks based on their holdings of a cryptocurrency.

Remix IDE: An Integrated Development Environment (IDE) used for developing, testing, and debugging smart contracts written in Solidity.

Signers: The individuals or entities who hold the private keys and are authorised to sign transactions in a multi-signature wallet.

Smart Contract: Self-executing contracts with the terms of the agreement directly written into code, typically deployed on a blockchain network.

Solidity: A programming language used for writing smart contracts on the Ethereum blockchain.

Testnet: A testing environment for a blockchain network that simulates the main network (mainnet) but uses valueless tokens to test and experiment.

1.2 Overview

The rapidly growing world of cryptocurrencies and decentralised finance (DeFi) has transformed the way we conduct transactions and manage assets. With the increasing adoption of blockchain technologies, the need for secure and efficient management of digital assets has become more important than ever. A major concern for users and businesses alike is the security of their digital assets.

A multi-signature cryptocurrency wallet, often referred to as a multisig wallet, is a type of digital wallet that requires multiple signatures or approvals from different owners to authorise transactions. This added layer of security makes it more difficult for unauthorised parties to access or move funds, as they would need control over multiple private keys.

The Frontier Multisig Wallet aims to address the aforementioned security concerns, by creating a wallet that can have multiple owners who must find consensus through a voting process in order to decide which transactions are completed or cancelled. A user can create a wallet, add funds, add/remove new owners, submit transactions to be voted on, vote on transactions, tag transactions with a title and description so other members can understand what they are voting for, change the number of votes required to approve/deny and also view the wallets transaction history.

1.3 Justification

The development of the Frontier Multisig is essential in today's digital age, as it addresses critical security concerns and offers a robust solution for various use cases, such as collaborative asset management, DAOs, escrow services, and estate planning. By providing an added layer of security and a flexible, user-friendly interface, the application empowers users and organisations to manage their digital assets with confidence, while mitigating risks associated with single-point failures. In an increasingly decentralised world, the Ethereum multisignature wallet application is a necessary and innovative tool for effective digital asset management.

As students, it also gives us an opportunity to explore blockchain technology. It currently does not have a dedicated module in the CASE program, so we wanted to take on the challenge and attempt to build something ourselves. By carrying out this project we were able to develop the skills to build blockchain and web3 applications and further our understanding of the technology.

1.4 Use Cases

1. *Enhanced Security*: One of the primary use cases of the Frontier Multisig Wallet is to provide an added layer of security for digital asset management. By requiring multiple signatures to authorise a transaction, the application reduces the risk of unauthorised access, theft, or loss of funds due to a single compromised private key.

2. *Collaborative Asset Management*: Businesses and organisations often need to manage their digital assets jointly, requiring a solution that allows multiple parties to securely access and control the same wallet. The application enables shared access to a wallet, while maintaining a high level of security through the necessitation of multiple signatures for transaction approval.

3. *Decentralised Autonomous Organizations (DAOs)*: DAOs are organisations that operate on the blockchain and require a decentralised approach to decision-making and fund management. The wallet is well-suited for DAOs, as it enables multiple members to collectively authorise transactions, providing a transparent and decentralised solution for managing their shared assets.

4. *Escrow Services*: Frontier Multisig can be used to facilitate escrow services, where a neutral third party holds funds until certain conditions are met. By requiring multiple signatures to release funds, the application ensures that both parties involved in the transaction agree to the terms and conditions before the transfer is executed. This is a particularly exciting possibility, due to the immutable, irreversible nature of the blockchain.

5. *Family Trusts*: Individuals can use the application to set up family trusts or plan their estate, ensuring that multiple family members or beneficiaries have access to the digital assets, while still maintaining control over the authorization of transactions.

1.5 Technologies Used

The Frontier Multisig Wallet uses cutting edge technologies to provide a secure and efficient solution for managing digital assets. Built on the Ethereum blockchain, the wallet utilises smart contracts to enable multisignature functionality. The smart contracts, written in the Solidity programming language, ensure secure transaction processing and allow for the seamless integration of various features, such as creating wallets, adding and removing owners, and setting the number of approvals or denials required to complete a transaction.

In order to allow seamless interaction with the smart contracts, the wallet utilises the Hardhat development environment and Ethers.js, a popular JavaScript library designed for interacting with the Ethereum blockchain. Hardhat simplifies the development, deployment, and testing of smart contracts, while Ethers.js provides a user-friendly interface for connecting to Ethereum nodes and sending transactions. The combination of these technologies ensures a robust and secure multisignature wallet application that caters to the diverse needs of users and organisations in the rapidly evolving world of decentralised finance.

On the web app side, Frontier Multisig employs a Node.js-based back end, ensuring a fast and scalable foundation for handling user requests and managing interactions with the Ethereum blockchain. This choice of back end technology allows for smooth integration with the aforementioned Ethers.js library and offers a wide range of community-driven packages to enhance the application's functionality.

The front end of the wallet application is designed using HTML and Tailwind CSS, a modern utility-first CSS framework that enables rapid development of responsive and visually appealing user interfaces. This combination of technologies results in a responsive, user-friendly, and visually appealing web application that streamlines the process of managing digital assets through the Wallet.

1.6 Deployment

During the development of the Frontier Multisig wallet application, we focused on creating a flexible and adaptable deployment process. We deployed the smart contracts on a locally hosted Ethereum node using Hardhat, which enabled us to take advantage of its powerful development environment and testing capabilities. This local deployment allowed for fast restarts and quick iteration as changes were made to the application.

In addition to the smart contracts, we also deployed the website locally on localhost. This made the most sense to allow for rapid deployment, It also meant we simply had to refresh the page after making changes in the codebase.

In our project last year, we proved we could deploy to the official EVM testnets, and also experienced the issues which can come with that. It is often slower and has more downtime than running a node locally. So this year we felt it wasn’t a necessity to go past deploying locally. However our deployment strategy was designed with flexibility in mind, allowing for the choice of transition to different Ethereum networks if necessary. By making some minor configuration changes, we could easily deploy the smart contracts on the Ethereum Sepolia testnet or even the mainnet.

2. Testing

To ensure that our app functioned as intended and was secure, it was crucial to conduct thorough testing. The following section outlines the various testing methods we employed, including testing with Remix IDE, utilising Hardhat unit tests, and manually testing the application.

2.1 Remix IDE

Remix IDE is an open-source tool that enabled us to write, deploy, and test smart contracts using the Solidity programming language. This browser based IDE is designed for quick prototyping and testing. We used Remix IDE to test our smart contracts in the following manner:

1. Wrote the smart contract code in Solidity and compiled it within Remix IDE.
2. Used the built in features to call functions and see how that changes the data.
3. Make changes and fixes if something isn't working as expected.
4. Repeat the process until the required functionality is reached.

By testing our smart contracts in Remix IDE, we easily identified and resolved any issues in the early stages of development. The speed in which code could be tested and changed was huge for us, saving a huge amount of time.

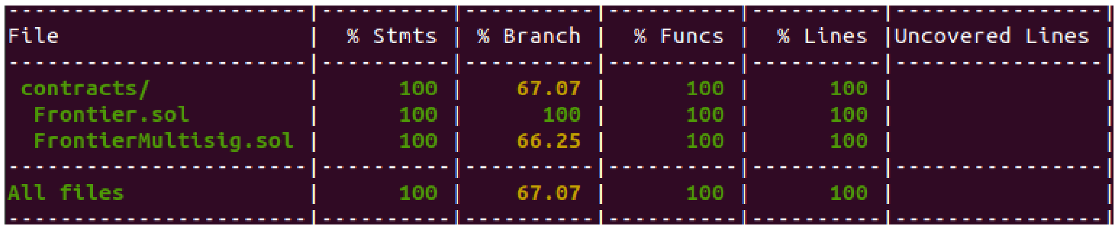
2.2 Hardhat Unit Testing

Hardhat is a development framework for Ethereum smart contracts that simplifies the process of building, deploying, and testing smart contracts. With Hardhat, we wrote and executed unit tests to ensure the correct behaviour of our smart contracts. We were also able to implement solidity coverage into our hardhat unit tests. This meant we could see which lines of code were being tested and which had not yet been covered. It provided us with a full in depth overview of the effectiveness of our unit tests.

To test our smart contracts with Hardhat unit tests, we:

1. Wrote test cases for each smart contract function, considering both expected and unexpected input scenarios.
2. Ran the tests using Hardhat's test runner, which automatically compiled and deployed the contracts to a local Ethereum network.
3. Analysed the test results to ensure that the smart contracts worked as intended and were free from vulnerabilities.
4. Checked the Solidity coverage statistics to learn where to target with further tests.

The image below shows our coverage statistics. As you can see in the table, we were able to test every line of our smart contracts.



```
------------------------|----------|----------|----------|----------|----------------|
File                    |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
------------------------|----------|----------|----------|----------|----------------|
 contracts/             |      100 |    67.07 |      100 |      100 |                |
  Frontier.sol          |      100 |      100 |      100 |      100 |                |
  FrontierMultisig.sol  |      100 |    66.25 |      100 |      100 |                |
------------------------|----------|----------|----------|----------|----------------|
All files               |      100 |    67.07 |      100 |      100 |                |
------------------------|----------|----------|----------|----------|----------------|
```

2.3 Manual Testing

In addition to using Remix IDE and Hardhat unit tests, it was essential to manually test the Frontier Multisig Wallet application. This involved interacting with the application as an end-user would to ensure that the user interface was user-friendly, responsive, and accurately reflected the smart contract's functionality.

Manual testing of the application included:

1. Deploying the smart contracts to a test network and connecting the frontend to the testnet.
2. Testing the application's various features, such as creating and interacting with multisig wallets, submitting transactions, voting on transactions, and executing transactions.
3. Observing the application's behaviour, noting any inconsistencies or issues that arose.

This process was allowed to be particularly effective due to the local deployment of the Ethereum node and website, meaning errors could be rapidly fixed.

3. Development Process

3.1 Version Control

During the development of our Frontier Multisig wallet application, we employed Git as our version control system to effectively manage the source code and ensure smooth collaboration between team members. We maintained a master branch that contained stable, tested, and working iterations of our application. The develop branch, which branched off from the master branch, served as an integration point for completed features and bug fixes.

To organise our work further, we created separate branches for different aspects of the application: smartContracts, backend, frontend, and testing. These branches were merged into the develop branch when necessary to assemble a fully functioning application. Only when the develop branch contained stable and tested code, we merged it back to the master branch.

3.2 Project Management

For project management, we utilised JIRA to create a board that facilitated task management, workload balancing, and tracking the progress of various tasks. The board helped us maintain a clear overview of work in progress, the backlog of tasks, and selection for development. It also allowed us to collaborate efficiently and monitor the time taken for individual tasks.

Using Kanban within JIRA enabled us to integrate these methodologies directly into our Git flow. By including relevant JIRA tags (e.g., FRON-1, FRON-2, etc.) in our commit messages, we linked our work to specific tasks on the JIRA board.

Furthermore, we used keywords such as COMPLETE or PROGRESS in our commit messages to provide a quick overview of the task status in the commit log. This practice allowed us to easily identify which tasks were completed or still required further work, enhancing our ability to manage and prioritise tasks effectively.

3.3 Agile Methodology

3.3.1 Scrum Framework

We incorporated the Scrum framework, which is a popular Agile methodology, to effectively manage the project. We divided the project timeline into sprints, each lasting one to two weeks. At the beginning of each sprint, we held a sprint planning meeting to discuss and prioritise tasks that needed to be completed during the sprint. The tasks were then added to the JIRA board and assigned to the relevant person.

Throughout the sprint, we had daily stand-up meetings, which allowed us to sync up on our progress, discuss any challenges or obstacles, and ensure that the tasks were on track.

3.3.2 Pair Programming

Another Agile practice we adopted was pair programming. During the development of critical features or when facing complex challenges, we worked together on the same code to ensure that the best possible solution was implemented. We often booked library rooms or worked in the computing labs. Pair programming allowed us to share knowledge, learn from each other, and enhance the quality of our code. We were able to switch between each being the driver and navigator, which allowed us to excel in the areas we were most proficient in and optimise the pair programming process.

3.3.3 Continuous Integration / Continuous Deployment

As part of our Agile methodology, we also focused on continuous integration and continuous deployment. We made sure to make regular commits and had a branching policy. We divided the branches into master, develop, frontend, backend, smart contracts and testing. We would work in the relevant branch then merge changes into the develop branch, only merging into master when there is key progress made.We frequently committed our code to the repository and merged changes to the develop branch. This practice enabled us to detect and resolve integration issues early in the development process.

4. System Architecture

4.1 Outline

The Frontier multisignature wallet application is built using a combination of Solidity smart contracts and a React based frontend. The system architecture can be divided into two main components: the blockchain based backend and the web based frontend.

Backend (Ethereum Smart Contracts):

Frontier.sol: This smart contract is the primary entry point for the application. It allows users to create multi-signature wallets and keeps track of these wallets in a mapping. It also stores a reference to the FrontierMultisig.sol smart contract, which contains the core multisignature wallet functionality.

FrontierMultisig.sol: This smart contract contains the core functionality of the multisignature wallet, including the ability to manage owners, submit transactions, approve/deny transactions, execute transactions, and manage approvals and denials. The smart contract utilises Solidity structs and mappings to manage the data efficiently.

Frontend (React-based web application):

State management: We used the power of react pageProps. This allowed us to let the user switch between the different pages using the sidebar, without refreshes being necessary for each page switch. This gives a much smoother and modern feel to the web app. The application also uses React hooks, specifically useState and useEffect, to manage the local state of the components and handle side effects, such as fetching data from the blockchain or triggering updates when certain values change

Pages: The web app uses Next.js as the framework and is organised into several pages (e.g., app, index, manage, pendingTransactions) that correspond to the different views and functionalities of the application.

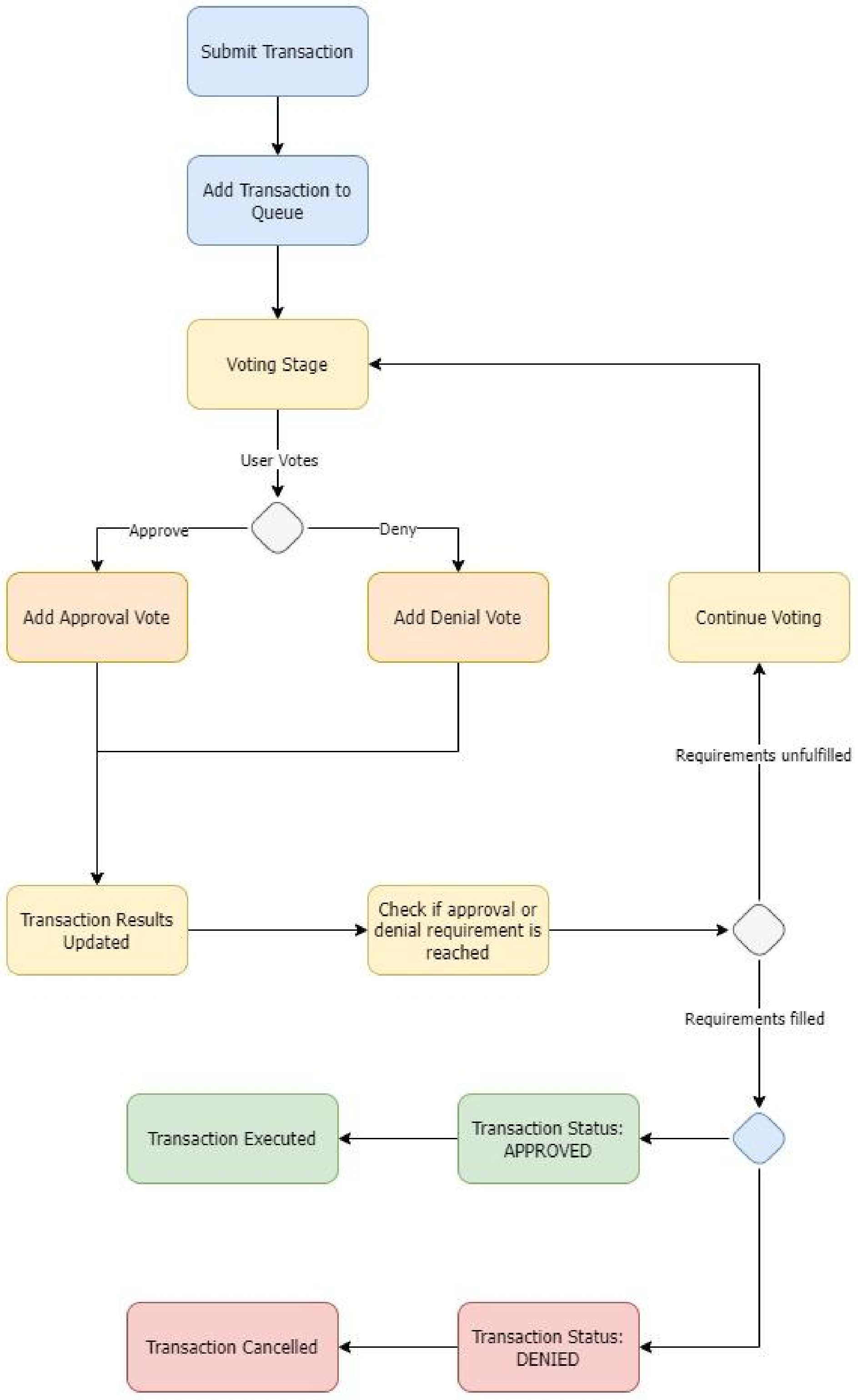
Components: Each page consists of multiple reusable React components that are responsible for rendering different parts of the user interface, such as transaction items and the wallet selector components

Ethers.js: The web application uses the Ethers.js library to interact with the Ethereum blockchain. It is used to create instances of the Frontier.sol and FrontierMultisig.sol smart contracts and call their respective functions using the JSON-RPC protocol..

Metamask integration: The application integrates with the Metamask browser extension to enable users to interact with the application using their Ethereum accounts. The application uses the window.ethereum object provided by Metamask to prompt users to connect their accounts and sign transactions.

Error handling: The application handles errors gracefully by displaying appropriate error messages to the user when issues arise, such as incorrect input, failed transactions, or problems fetching data from the blockchain.

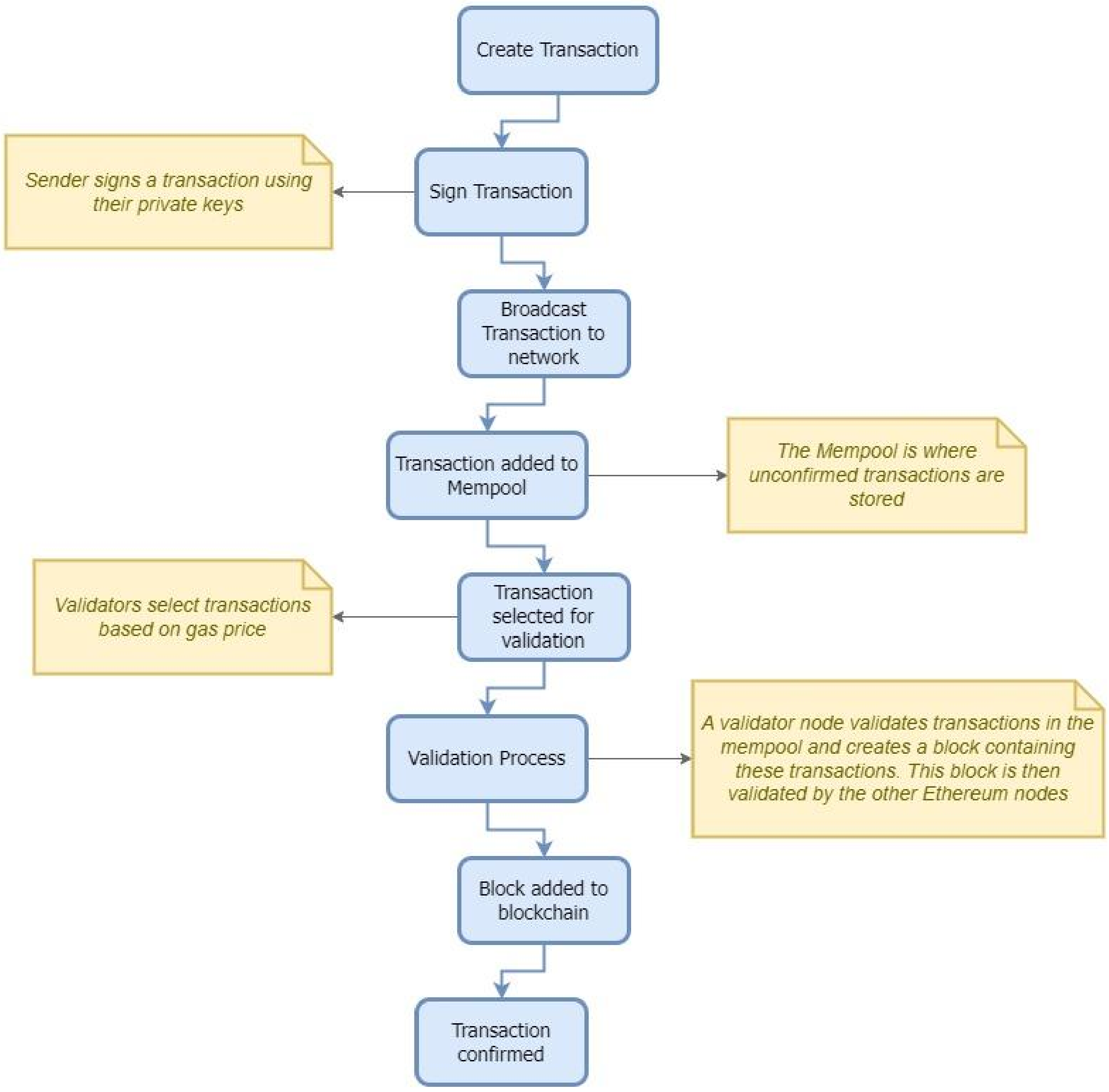
4.2 Frontier Transaction Flow *(Fig 1)*



(Fig 1) - The *Frontier Transaction Flow* diagram illustrates the full process of a transaction in the Multisig Wallet. First a transaction is submitted by any user who is an owner of the wallet. This transaction then shows as a pending transaction for all other owners of the wallet. The transaction is then subject to the voting process. A wallet will have a

configurable number of approvals or denials required to execute or cancel the pending transaction. When the required number of approvals/denials is hit, the respective status is applied to the transaction, then it is either executed and the funds are sent or it is cancelled and can no longer be executed or voted on. These transactions are then available to view on the *Analytics* tab of the application.

4.3 Ethereum Transaction Process *(Fig 2)*

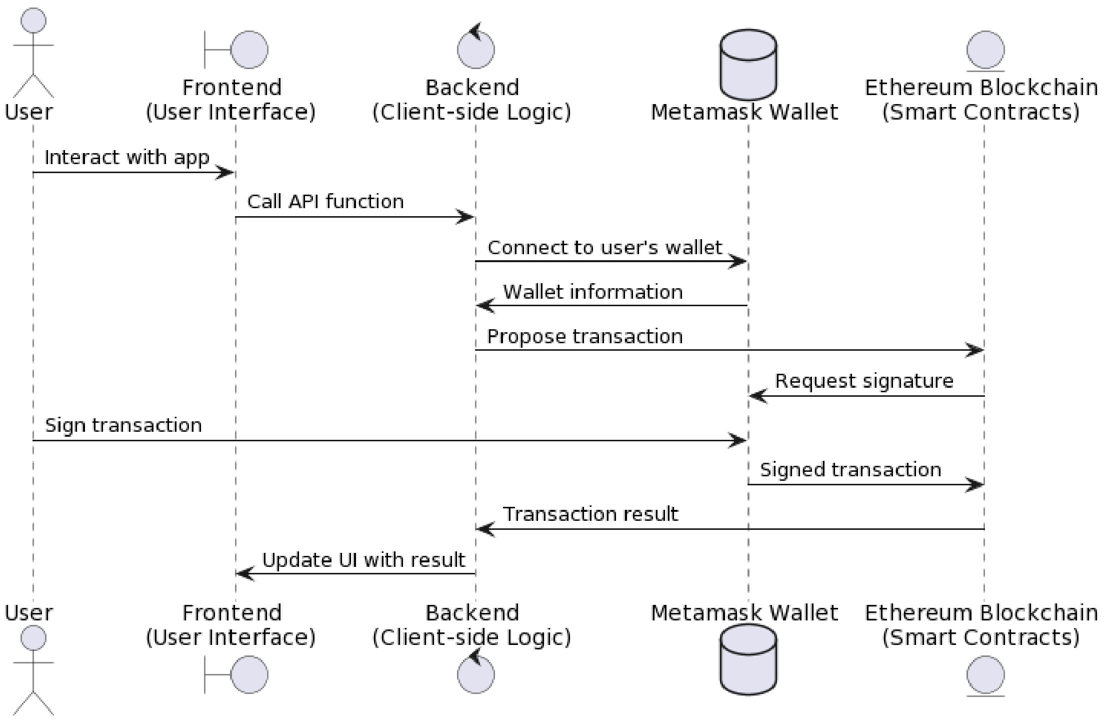

Create Transaction
Sign Transaction
Sender signs a transaction using their private keys
Broadcast Transaction to network
Transaction added to Mempool
The Mempool is where unconfirmed transactions are stored
Transaction selected for validation
Validators select transactions based on gas price
Validation Process
A validator node validates transactions in the mempool and creates a block containing these transactions. This block is then validated by the other Ethereum nodes
Block added to blockchain
Transaction confirmed

(Fig 2) - The *Ethereum Transaction Process* diagram provides a visual representation of the steps involved in executing a transaction on the Ethereum network. The process begins with the sender creating a transaction, specifying the recipient, value, and optional data. Next, the sender signs the transaction using their private key, ensuring its authenticity. Once signed, the transaction is broadcasted to the Ethereum network, where it is

propagated to various nodes. The transaction is then added to the mempool, which stores unconfirmed transactions.

Validators select transactions from the mempool based on their gas price and proceed to the validation process. In Ethereum's Proof of Stake consensus mechanism, validators propose and attest to blocks. When a block containing the transaction is validated and added to the blockchain, the transaction is considered confirmed. This diagram effectively showcases the key components and steps involved in processing an Ethereum transaction from creation to confirmation.

4.4 System Architecture *(Fig 3)*



(Fig 3) - The *System Architecture* diagram illustrates the flow of information and actions between the user interface (frontend), application logic (backend), and Ethereum blockchain. When users interact with the frontend, their actions trigger corresponding functions in the backend. These functions establish a connection to the user's Metamask wallet and interact with the deployed smart contracts on the Ethereum blockchain. The diagram demonstrates how the backend handles user requests for creating a multisig wallet, submitting transactions, approving transactions, and executing transactions. It highlights the seamless integration of the frontend, backend, and blockchain components to provide a smooth user experience while interacting with the Frontier Multisig Wallet application.

5. Future development

While the Frontier Multisig Wallet has been successful in providing a robust and secure solution for managing multisig wallets, there are several areas where the application could be improved in the future to offer a more comprehensive and user-friendly experience. In this section, we outline potential enhancements that could be made to enrich the user experience and add value to the application.

5.1 Transaction Visualisation

Integrating dynamic graphs and visualisations of transaction history would allow users to gain a better understanding of the wallet's activity over time. Users could view the transaction volume, frequency, and types in a visually appealing manner, making it easier to analyse the wallet's performance and financial trends.

5.2 Transaction Filtering and Sorting

By offering users the ability to sort and filter transactions based on various criteria such as date, amount, sender, recipient, or transaction status, it would become simpler to locate specific transactions and manage the wallet's activity more effectively.

5.3 Transaction Comments and Discussions

Implementing a feature that enables users to leave comments on pending transactions would facilitate communication and collaboration among wallet participants. Users could ask for clarification or provide additional details about the transaction, ensuring that all parties are well-informed before casting their votes.

5.4 Address Customization and Tagging

Allowing users to assign custom names or tags to wallet addresses would enhance usability and make it easier to identify counterparties at a glance. Users could create personalised labels for frequently used addresses, reducing the reliance on long, alphanumeric addresses and minimising the risk of errors.

5.5 Wallet Settings Customization

Introducing more customization options for wallet settings, such as the number of required signatures, notification preferences, and user roles, would enable users to tailor the application to their specific needs and preferences.

5.6 UI Improvements and Personalization

Investing in user interface improvements and personalization features, such as customizable colour schemes, font sizes, and display options, would make the application more visually appealing and adaptable to individual user preferences.

5.7 Mobile Application

Developing a mobile application for the Frontier Multisig Wallet would expand the accessibility and convenience of the platform, allowing users to manage their wallets and transactions on the go.

5.8 Integration with Third-Party Services

Expanding the application's compatibility with third-party services, such as hardware wallets, decentralised finance (DeFi) platforms, and cryptocurrency exchanges, would provide users with seamless access to a broader range of financial tools and services.

REFERENCES

1. Ethereum Foundation.

   https://ethereum.org

2. Ethereum Whitepaper

   https://ethereum.org/en/whitepaper

3. Hardhat Developer Environment

   https://hardhat.org

4. Solidity: Smart Contract Programming Language.

   https://soliditylang.org

5. Ethers.js: Complete Ethereum wallet implementation and utilities in JavaScript.

   https://docs.ethers.io

6. MetaMask: Ethereum Wallet and Web3 Provider.

   https://metamask.io

7. Remix: Ethereum IDE and Tools for Solidity.

   https://remix.ethereum.org