Contract Pallet Multisig Implementation: Backendless Architecture Document

A Journey with Protofire

By Luca Auet and Gabriel Gonzalez

Resume

This document outlines the architecture of a contract pallet multisig implementation without a backend. It provides a detailed view of the system's structure, components, and interactions, focusing on the advantages and disadvantages of a backendless approach.

Existing Implementation

Our research was based on the existing multisig pallet. This pallet provides the functionality for managing multisignature accounts in Substrate-based blockchains.

Core Functionality

Within this module, a comprehensive set of functionalities is available to facilitate the implementation of multi-signature dispatch. This empowering feature allows multiple signed origins (accounts) to collaboratively initiate a call from a pre-established origin. The process of determining this origin, specifically the account ID of the new multi-sig, is achieved through a deterministic approach that takes into account a set of account IDs and a designated threshold of approvals. By leveraging these capabilities, developers can enable seamless coordination and execution of actions among multiple signatories, bolstering security and flexibility within the system.

Pros

- Predictability: Deterministic account generation ensures that when the same input is provided, the same account will always be generated. This attribute becomes advantageous in scenarios where an action involves a multisig account that can be potentially created by other accounts and a specific threshold. This predictability guarantees consistent outcomes and enables the smooth execution of actions, even without the accounts' knowledge of their involvement.

- Reduced Storage: Due to the regenerability of multisig accounts based on the same input parameters, it becomes unnecessary to store each individual account separately. This feature leads to substantial savings in storage space, especially in scenarios

involving numerous accounts. By eliminating the need to store each account individually, storage requirements are minimized, optimizing resource allocation and efficiency.

- Hierarchical Structures: Deterministic account generation often allows for the creation of hierarchical structures, where child accounts can be derived from parent accounts. This can be useful for organizing accounts and managing permissions.

Cons

- Limited Flexibility: The inability to modify the threshold or change the multisig owners can pose a notable limitation when operational requirements evolve. In situations where a signatory departs from the organization or loses their key, the need to replace them with a new signatory or adjust the threshold becomes imperative. However, without the ability to make such modifications, the system faces constraints that can hinder adaptability to changing circumstances.

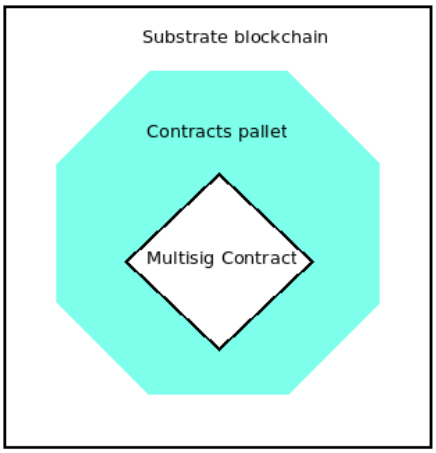
- Security Risks: If a signatory's key is compromised, not being able to remove them from the multisig owners can pose a security risk. The attacker could potentially approve transactions if they manage to compromise enough keys.

- Pallet Limitations: The use of the multisig pallet within a Substrate-based blockchain presents obstacles in terms of enhancing functionality and integrating custom modules. The complexity and potential disruptions associated with updating the pallet introduce impracticalities when attempting to improve multisig capabilities or incorporate custom features. Consequently, the ability to adapt the multisig implementation to meet specific project requirements or introduce new functionalities becomes constrained. This limitation hampers the overall flexibility and extensibility of the blockchain solution, potentially impeding its capacity to evolve and cater to evolving needs.

Our Implementation: System Architecture

The system architecture we defined is primarily composed of the on-chain component.

On-Chain Component



The on-chain component consists of the contract pallet and the multisig contract. The contract pallet is a module in the Substrate framework that allows us to write and deploy smart contracts on a blockchain. The multisig contract is a type of smart contract that requires multiple parties to sign off on transactions.

In a backendless architecture, the on-chain contract is more complex. It needs to store all proposed transactions and

have a logic for removing them. This increases the complexity and size of the on-chain contract.

Here's an overview of its main data structures and functions:

Data Structures

Transaction - Represents a transaction that can be performed when a threshold is met. It includes the receiver's address, the function selector, the input data, the value to be transferred, the gas limit, and a flag that indicates whether re entry is allowed.

MultiSig - The main data structure of the contract. It includes the list of owners, the threshold needed to approve transactions, a list of transactions, and a list of approvals and rejections. It also includes a mapping of transactions and approvals for constant-time access.

Constructors

new(threshold: u8, mut owners_list: Vec<AccountId>) -> Result<Self, Error>: This function is a constructor that creates a new instance of the MultiSig contract. It takes a threshold (the minimum number of approvals required to execute a transaction) and a list of owners. It ensures that the threshold and owners are valid, and initializes the contract with the given parameters. The function will fail if the threshold is greater than the number of owners, if the threshold is zero, or if the owners list is empty.
Possible Errors: *OwnersCantBeEmpty, ThresholdGreaterThanOwners, ThresholdCantBeZero.*

default() -> Result<Self, Error>: This is another constructor that creates a new instance of the MultiSig contract with default parameters. The default parameters are a single owner (the caller) and a threshold of 1. It shouldn't return an error due to all invariants being accomplished. The error is there to keep consistency in the constructors return types.
Possible Errors: -.

Functions

propose_tx(&mut self, tx: Transaction) -> Result<(), Error>: This function allows an owner to propose a new transaction. It ensures that the caller is an owner and that the maximum number of transactions has not been reached. If the threshold is reached when the transaction is proposed (i.e., the threshold is 1), the transaction is executed.
Possible Errors: *NotOwner, MaxTransactionsReached, TxIdOverflow.*
Emitted Events: *TransactionProposed,* if threshold is 1 *TransactionExecuted.*

approve_tx(&mut self, tx_id: TxId) -> Result<(), Error>: This function allows an owner to approve a transaction. It checks that the caller is an owner, that the transaction exists, and that the caller has not already voted on the transaction. If the threshold is met with this approval, the transaction is executed.

Possible Errors: *NotOwner, InvalidTxId, AlreadyVoted.*
Emitted Events: *Approve,* if threshold is met *TransactionExecuted.*

reject_tx(&mut self, tx_id: TxId) -> Result<(), Error>: This function allows an owner to reject a transaction. It performs the same checks as approve_tx. If the threshold cannot be met with the remaining approvals after this rejection, the transaction is removed.
Possible Errors: *NotOwner, InvalidTxId, AlreadyVoted.*
Emitted Events: *Reject*, if threshold can not be reached *TransactionRemoved.*

try_execute_tx(&mut self, tx_id: TxId): Execute a transaction if the threshold has been met and remove it from the storage. It's called try because nothing is done if the condition is not met. The external invocation of this function is permitted, although it is designed to be invoked automatically by the approve_tx. We have identified specific exceptional scenarios where manual invocation of this function is necessary.
Possible Errors: This function does not directly throw any errors.
Emitted Events: If threshold is met *TransactionExecuted*.

try_remove_tx(&mut self, tx_id: TxId): Remove a transaction if the threshold cannot be met with the remaining approvals. It's called try because the condition is checked too.
As mentioned in the description of try_execute_tx, this function is executed automatically. However, there are certain exceptional situations where manual invocation of this function becomes necessary.
Possible Errors: This function does not directly throw any errors.
Emitted Events: If threshold can not be reached, *TransactionRemoved*.

add_owner(&mut self, owner: AccountId) -> Result<(), Error>: This function adds a new owner to the contract. It checks that the caller is the multisig contract itself, that the number of owners is not greater than the maximum allowed, and that the new owner is not already an owner.
Possible Errors: *Unauthorized, MaxOwnersReached, OwnerAlreadyExists.*
Emitted Events: *OwnerAdded.*

remove_owner(&mut self, owner: AccountId) -> Result<(), Error>: This function removes an owner from the contract. It performs similar checks to add_owner and also ensures that the owners list will not be empty after the removal and that the threshold is not greater than the number of owners after the removal.
Possible Errors: *Unauthorized, NotOwner, OwnersCantBeEmpty, ThresholdGreaterThanOwners.*
Emitted Events: OwnerRemoved.

change_threshold(&mut self, threshold: u8) -> Result<(), Error>: This function changes the approval threshold. It checks that the caller is the multisig contract itself, that the new threshold is not greater than the number of owners, and that the new threshold is not zero.
Possible Errors: *Unauthorized, ThresholdGreaterThanOwners, ThresholdCantBeZero.*

EmittedEvents: *ThresholdChanged*

transfer(&mut self, to: AccountId, value: Balance) -> Result<(), Error>: This function transfers funds from the contract to another account. It checks that the caller is the multisig contract itself and then performs the transfer.
Possible Errors: *Unauthorized, TransferFailed.*
EmittedEvent: *Transfer*.

In addition to these, there are a number of getter functions to read the state of the contract such as getting the list of owners, the threshold, transactions, approvals, and rejections that will be used by the front end application.

The contract uses events for logging operations like transaction proposal, approval, rejection, addition and removal of owners, threshold change, and transfer of funds. This makes the contract traceable from an event subscriber.

Off-Chain Component

The off-chain component is a web application, primarily built using a technology stack that includes React, NextJS, and TypeScript.

React, a popular JavaScript library for building user interfaces, along with NextJS, a React framework for production-grade applications, are used to construct the frontend application. This application communicates with the on-chain component, submits transactions, and displays information to the user.

TypeScript, a statically typed superset of JavaScript, is used to ensure type safety and improve the development experience. It helps catch errors early in the development process and enhances code readability and maintainability.

Please note that this is the proposed technology stack and it may be supplemented with other libraries or tools as necessary to meet specific requirements or to enhance the functionality and performance of the off-chain component.

Advantages and Disadvantages

Advantages

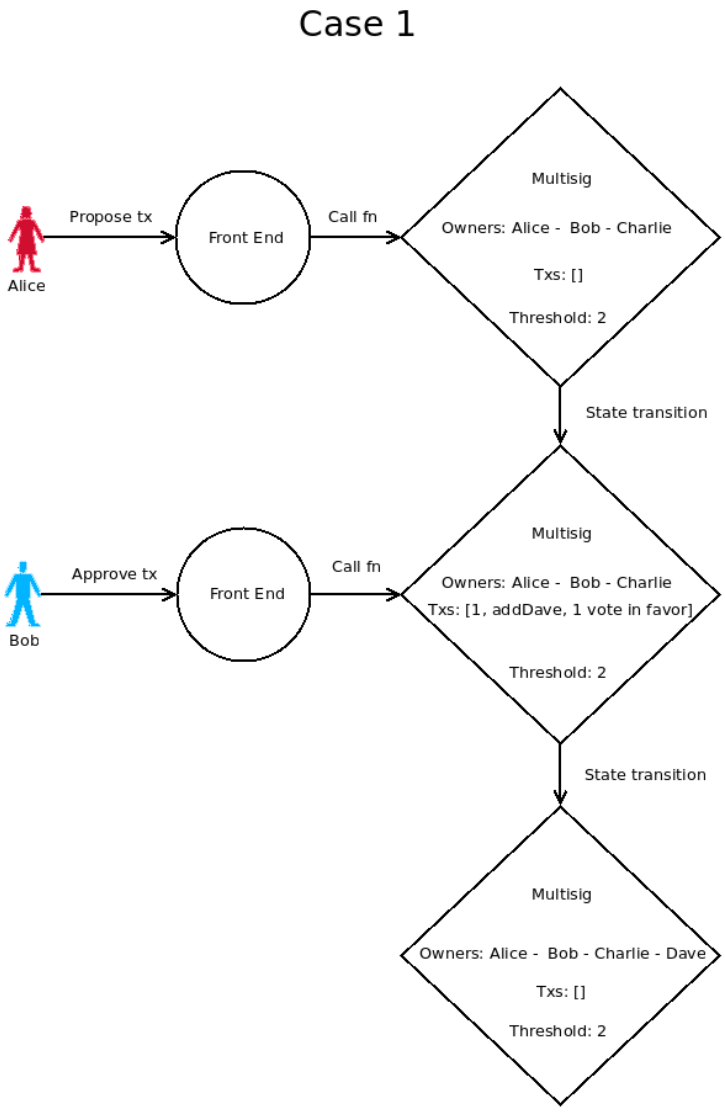
Decentralization: A backendless approach maintains the decentralized nature of blockchain technology. There's no need to trust a separate entity managing a backend.
Reduced Operational Costs: Without a backend, you save on the costs associated with maintaining and operating a backend service.

Disadvantages

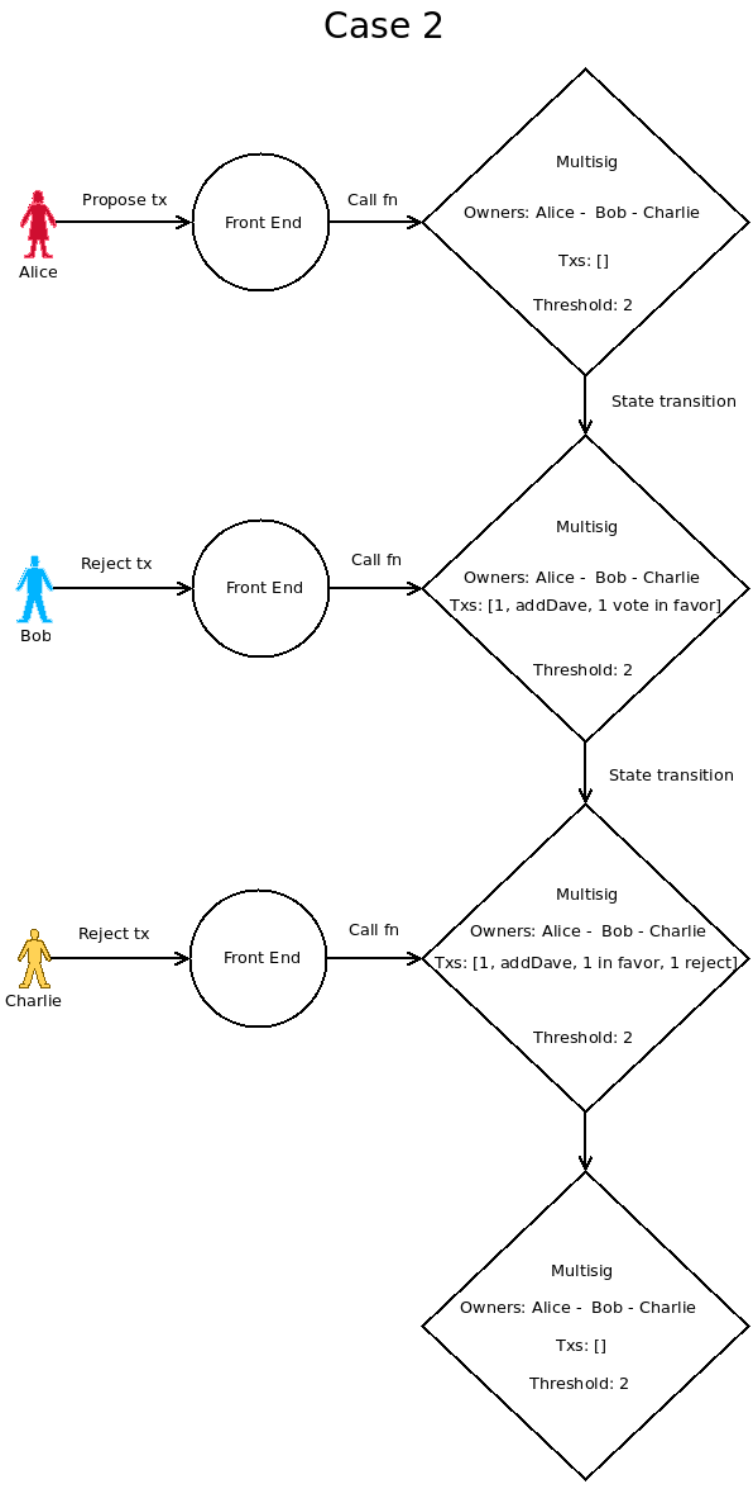
User Experience: Without a backend to centralize information, the user interface and experience may be less streamlined. Users may need to share the metadata of the deployed contract.
Increased On-Chain Costs: Without off-chain signing, the computational and financial costs associated with on-chain transactions may be higher. All transactions, successful or not, interact with the contract.
Limitations: Without a backend, you may face the limitations of the smart contract pallet, such as a maximum storage of 16k on each contract.

State transition cases

This scenario illustrates a contract that involves three owners and a threshold of two. Alice, one of the owners, suggests adding Dave as a new owner. Bob, another owner, gives his approval. Once the threshold is met with Bob's approval, Dave is successfully added as a new owner



In the following scenario, the contract is the same as the previous example. Alice suggests adding Dave as an owner. However, both Bob and Charlie reject this proposal. As a result, Dave is not added and the transaction is subsequently removed.



Trust and Metadata

Regardless of whether a backend is used, there's an inherent trust issue around metadata sharing. The metadata of third-party contracts can't be easily verified, making this a significant and risky aspect of contract pallets multisig implementation.

Conclusion

This architecture document provides a high-level view of a contract pallets multisig implementation without a backend. It highlights the key components and their interactions, as well as the advantages and disadvantages of a backendless approach. As with any architectural decision, it's crucial to weigh these factors against the specific needs and constraints of your project.