HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

GRADUATION THESIS

A social login solution for Web3 using Shamir's secret sharing and verified DKG

NGUYEN TUAN MINH

minh.nt184294@sis.hust.edu.vn

Major: ICT Global

Specialization: Information Technology

Supervisor: Ph.D Dao Thanh Chung _____

Signature

Department: Computer Engineering

School: Information and Communications Technology

HANOI, 06/2022

Requirements for the thesis

Student information
Student name: Nguyen Tuan Minh
Tel: 0915871399 Email: minh.nt184294@sis.hust.edu.vn
Class: ICT02.K63 Program: Global ICT

This thesis is performed at: 21st floor, Charmvit Tower

Goal of the thesis

This thesis focus on addressing the challlenges associated with decentralized identity and authentication in blockchain applications, providing developers with a convenient and standardized way to implement secure and user-friendly authentication mechanism.

Main tasks

In this thesis, I will disscuss blockchain, smart contracts, and social login for Web3 Application. Next, I will describe in detail the architecture and design of the Social login system using Shamir's secret sharing and verified DKG. Lastly, i will conduct some experiments to evaluate and querying the efficacy of the solution.

Declaration of student

Nguyen Tuan Minh - hereby attests that the work and presentation in this thesis were carried out by myself under the direction of Ph.D Thanh-Chung Dao. All results presented in this thesis are authentic and have not been plagiarized. All references in this thesis, including images, tables, figures, and quotations, are cited in the bibliography in a plain and comprehensive manner. I will assume full responsibility for any copy that violates school regulations, even if it is only one.

Advisor's confirmation of the completion and defense permission

Hanoi, Ngày 31 tháng 7 năm 2023
Advisor's signature

Dr.Thanh-Chung Dao

ACKNOWLEDGMENTS

I would like to express my profound appreciation to my family, friends and significant other for their unwavering support and patince throughout the process of writing my thesis. Their love, encouragement, and confidence in my abilities have been an inexhaustible source of fortitude and inspiration for me.

Thank you for always being there for me, providing me with a nurturing enviroment, and teaching me the importance of perserverance and diligence. Your unconditional affection and encouragement have inspried me to pursue my academic objectives.

Thank you for solid friendship and for being an unending source of motivation. During the difficult times of thesis writing, your presence, laughter, and words of encouragement have brought me pleasure and helped me maintain a healthy work-life balance.

Lastly, i would like to express my sincerest gratitude to my supervisor, Dr. Dao Thanh Chung. Your direction, expertise and commitment have been indispensable to my research and academic development. Your guidance has not only increased my expertise in the field, but has also inspired me to achieve new heights of intellectual inquiry. Even when the research appeared daunting, your perseverance, encouragement, and unwavering faith in my ability propelled me forward. I am extremely appreciative of the opportunities you have afforded me and the invalueable lessons I have gained under your direction. Thank you for being an outstanding mentor and for your unwavering support throughout the process of writing my thesis.

ABSTRACT

The blockchain has emerged as a revolutionary technology with the potential to transform numerous industries by providing a decentralized and transparent platform for recording transactions and data securely. The administration of identities and authentication remains a significant challenge within the blockchain ecosystem, despite its many benefits. In order to resolve this issue, it is necessary to create software that bridges the gap between conventional web authentication methods and blockchain-based systems. This bridge software would facilitate a more user-friendly and accessible blockchain ecosystem, ensuring that users can access blockchain-based services and applications with seamless identity verification. Blockchain is renowned for its rigorous security features, and any software implementation must maintain this level of security while integrating with standard web authentication protocols. A failure to adequately resolve security concerns could undermine the trustworthiness of blockchain technology. Innovative approaches, such as Shamir's Secret Sharing[1] (SSS) and Distributed Key Generation[2] (DKG), have considerable potential for addressing these issues. SSS is a cryptographic technique that divides a secret into multiple portions before distributing them to participants. This strategy ensures that no single entity has complete access to the secret, thereby enhancing security and reducing the likelihood of unauthorized access. DKG enables the collaborative generation of cryptographic keys without requiring a singular trusted party. This distributed method adds another layer of security and decentralization to the authentication procedure. I intend to develop a social authentication solution for decentralized applications (DApps) using SSS and DKG techniques. This solution would allow users to authenticate using their social network accounts while assuring their privacy and security through the use of secure and distributed authentication protocols. I will design the system architecture, implement the required software components, and assess the solution's performance and efficacy.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

BIP	Bitcoin Improvement Proposal
CRUD	Create, Read, Update, Delete
Dapp	Decentralize application
DeFi	Decentralize finance
DKG	Distributed Key Generation
encKey	Encrypt Key
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
IaaS	Infrastructure as a Service
OAuth2	Open Authenticate 2.0
PoS	Proof of Stake
PoW	Proof of Work
SSS	Shamir's secret sharing

CHAPTER 1. INTRODUCTION

1.1 Motivation

Blockchain technology, as exemplified by Bitcoin [3] and Ethereum [4] networks, has experienced significant growth and garnered widespread attention due to its unique characteristics and prospective benefits. Blockchain has revolutionized many industries, including finance, supply chain, healthcare, and more, by providing a decentralized, transparent, and immutable platform for record-keeping and value transmission. However, conventional web technologies and systems offer their own set of benefits and advantages. Bridging the gap between blockchain and conventional web technologies can unleash a wealth of opportunities and synergies, resulting in a more robust and adaptable digital ecosystem.

One of the key benefits of blockchain technology lies in its ability to provide trust and transparency. The Bitcoin network, for instance, enables peer-to-peer transactions without the need for intermediaries, fostering trust among participants and reducing transaction costs. Ethereum, on the other hand, extends blockchain capabilities by supporting programmable smart contracts, enabling decentralized applications (DApps) with a wide range of use cases. Meanwhile, traditional web technologies offer a well-established infrastructure, user-friendly interfaces, and extensive compatibility with existing systems. By combining the benefits of both blockchain networks like Bitcoin and Ethereum and traditional web technologies, we can create a powerful hybrid solution that leverages the transparency of blockchain while maintaining the usability and familiarity of the traditional web. By facilitating self-sovereign identities and data control, blockchain technology promotes decentralization and empowers individuals. Users can have ownership and control over their digital assets and personal data, decreasing their dependence on centralized entities. This paradigm shift is facilitated by Bitcoin's decentralized network architecture and Ethereum's decentralized application platform. Traditional web technologies, on the other hand, provide users with convenience and familiarity via centralized authentication systems, social logins, and widespread standards. As demonstrated by Bitcoin and Ethereum, integrating these features into the blockchain ecosystem can improve user experience, encourage adoption, and bridge the gap between conventional web users and blockchain applications.

The growth and benefits of blockchain technology, exemplified by networks like Bitcoin and Ethereum, combined with the advantages of traditional web technologies, highlight the importance of bridging the gap between the two. By leveraging the strengths of both systems, we can create a hybrid solution that harnesses the transparency, security, and decentralization of blockchain while maintaining the usability, compatibility, and familiarity of the traditional web. This convergence unlocks new possibilities, expands the reach of blockchain applications, and paves the way for a more interconnected and inclusive digital future. Consequently, the objective of this thesis, a social login solution for Dapps using SSS and verified by DKG, is to combine the advantages of blockchain technology and conventional web authentication.

1.2 Contributions

Due to the fact that this solution is a large undertaking involving the implementation of numerous modules by numerous individuals, it is evident that I did not design and construct the system alone and that other developers participated in its creation. In addition, I was responsible for devising and implementing the mechanisms for the executors to share secrets and generate private keys for end users. I implement the majority of the project's features, with the exception of Shamir's algorithm for sharing secrets and the Distributed Key Generation protocol. In addition, I designed and implemented the majority of the data structures contained in smart contracts and the decentralized storage called Eueno. In addition, this system has a unique architecture for securing and enriching the user experience, as well as enabling developers to integrate existing Dapps seamlessly.

1.3 Thesis structure

The present thesis is organized into six distinct chapters, each of which fulfills a specific objective in

the comprehensive investigation of the research subject matter. Chapter 1 serves as the introductory section of this thesis, wherein the underlying motivation driving the study is established. Furthermore, this chapter highlights the significant contributions made by the research and provides a comprehensive outline of the overall structure of the thesis. Chapter 2 of this thesis aims to establish a comprehensive understanding of fundamental concepts that are crucial to the subject matter. These concepts include blockchain, transactions, blocks, wallets, Shamir's secret sharing, distributed key generation, smart contracts, executor for decentralized applications (DApps), and social login for Web3. By delving into these concepts, this chapter lays the groundwork for the subsequent analysis and exploration of the topic at hand. Chapter 3 of this study presents the proposed solution, which outlines an innovative approach that has been adopted to effectively tackle the challenges that have been identified. Chapter 4 delves into an in-depth analysis of the technical issues and design considerations encountered throughout the implementation process. This chapter aims to address and shed light on the various challenges that were confronted during the execution of the project. Chapter 5 of this study presents a comprehensive evaluation of the proposed solution, offering valuable insights into its performance and usability. In conclusion, Chapter 6 serves as the final segment of this thesis, wherein the findings are succinctly summarized and potential avenues for future research are deliberated upon. The inclusion of a reference section within a thesis serves the purpose of meticulously documenting all the sources that have been cited throughout the research, thereby upholding the academic integrity of the study.

CHAPTER 2. BACKGROUND

The background section of this thesis provides a comprehensive overview of the key concepts and technologies that serve as the foundation for our proposed solution. This chapter examines the fundamental characteristics of blockchain technology, what social login for Web3 is, what a smart contract is, and the fundamental comprehension and utilization scenarios of Shamir's secret sharing and distributed key generation. Understanding these concepts is crucial for appreciating our solution's motivations and its potential impact on the decentralized digital landscape.

2.1 Blockchain

Blockchain technology has emerged as a revolutionary innovation with the potential to transform industries and revolutionize how digital transactions are conducted. It provides a decentralized and transparent platform for secure and unchangeable record-keeping, eliminating the need for intermediaries and facilitating peer-to-peer interactions. This chapter provides a concise introduction to blockchain technology, highlighting its historical context, the problem it seeks to solve, and the primary contributions of its first author. In 2008, an anonymous person or group of people using the alias Satoshi Nakamoto [3] introduced the concept of blockchain for the first time. The seminal whitepaper titled "Bitcoin: A Peer-to-Peer Electronic Cash System" by Satoshi Nakamoto outlined the fundamental principles and architecture of blockchain technology as a solution to the issues of trust and decentralized digital currency. Bitcoin's introduction of blockchain represented a significant milestone in the evolution of cryptocurrencies and decentralized systems. Traditional centralized systems' lack of trust and security is the issue blockchain seeks to address. The reliance of centralized systems on a single trusted authority to validate and authenticate transactions leaves room for manipulation, deception, and censorship. Blockchain technology addresses these issues by establishing a decentralized network of nodes where consensus mechanisms guarantee the validity and integrity of transactions without requiring a central authority. The first author, Satoshi Nakamoto, introduced a secure and decentralized framework for digital currency transactions, laying the groundwork for blockchain technology. Combining existing cryptographic techniques, such as hash functions and digital signatures, with a distributed ledger system was Nakamoto's most significant innovation. This innovation facilitated the creation of a transparent and tamper-resistant ledger of transactions, ensuring the integrity and immutability of blockchain data. Since Nakamoto's original work, blockchain technology has expanded beyond cryptocurrencies such as Bitcoin. It has implications in numerous industries, including finance, supply chain management, and healthcare, among others. The blockchain's decentralized nature provides opportunities for greater transparency, efficiency, and trust in these industries, paving the way for innovative solutions and new business models.

2.1.1 Transactions

"Transactions are the most important part of the Bitcoin system. Everything else in bitcoin is designed to ensure that transactions can be created, propagated on the network, validated, and finally added to the global ledger of transactions (the blockchain). Transactions are data structures that encode the transfer of value between participants in the Bitcoin system. Each transaction is a public entry in bitcoin's blockchain, the global double-entry bookkeeping ledger" according to "Mastering Bitcoin: Unlocking Digital Cryptocurrencies" by Andreas M. Antonopoulos [5], which implies that transactions are fundamental components of blockchain technology, serving as the building blocks for the transfer and exchange of digital assets. Blockchain networks' security and trustworthiness rely heavily on transactions. They are intended to be verifiable and immutable, providing a transparent and auditable log of all blockchain activities. By recording each transaction on the distributed ledger, participants are able to trace the history and origin of digital assets, fostering accountability and preventing double spending. Multiple stages are involved in the creation of a transaction. The sender initiates the transaction by specifying the recipient's address and the desired transfer amount. The originator then signs the transaction with their private key, ensuring the

transaction's authenticity and integrity. Once the transaction has been digitally signed, it is disseminated to the network for validation and inclusion in a block. The validation procedure involves verifying the digital signature of the transaction using the sender's public key, thereby ensuring that the transaction has not been tampered with and that the originator has sufficient funds to complete the transfer. The transaction is submitted to a pool of pending transactions awaiting confirmation after validation. Miners, who are tasked with safeguarding the blockchain, select transactions from the pool and incorporate them into a new block. A consensus mechanism, such as proof-of-work or proof-of-stake, is then used to add the transaction to the blockchain.The creation of transactions on the blockchain enables participants to transmit digital assets without the need for intermediaries in a transparent and secure manner. It assures the system's integrity by employing cryptographic techniques to authenticate and authorize transactions, thereby rendering the process tamper-proof and fraud-resistant.

2.1.2 Blocks

A block in a blockchain is a fundamental element that is crucial to the network's structure and functionality. It functions as a repository for a collection of transactions and other pertinent data. Each block is comprised of a block preamble, which includes metadata such as the block's unique identifier, timestamp, and a reference to the previous block, establishing a chronological order. The block contains transactions, which represent numerous actions within the blockchain network. These transactions include sender and recipient addresses, digital signatures for authentication, and additional pertinent information. The block also contains a Merkle tree root [6], which provides an efficient method for verifying the validity of transactions contained within the block. In addition, each block is allocated a unique block hash that is generated by a cryptographic hash function. This block hash serves as a digital fingerprint for the block's content and ensures its immutability. In proof-of-work consensus algorithms, for instance, miners compete to find a nonce value that, when combined with the block header, satisfies specific criteria, thereby adding a layer of security through the solution of computational puzzles. The block functions as the fundamental unit of the blockchain, enabling secure, transparent, and efficient transaction storage and verification.

2.1.3 Wallet

2.1.3.1 Key and address

Key and address are foundational concepts pertaining to user identification and transaction security in the context of blockchain technology and cryptocurrencies. A key, also known as a cryptographic key, is a fragment of information utilized in cryptographic algorithms for a variety of purposes, including encryption, decryption, and digital signatures. Typically, in the context of blockchain, keys are used to secure access to digital assets and to authenticate transactions. There are various mathematically related key categories, including private and public keys. A private key is a secret, randomly generated number that is kept covert by the user. It is used to generate digital signatures, which verify the integrity and authenticity of transactions. The private key should be stored in a secure location and never shared with anyone. If a third party obtains access to the private key, they may be able to take control of the associated digital assets. On the other hand, an address is a cryptographic representation of a user's public key. In a blockchain network, it is a string of alphanumeric characters that functions as a unique identifier for receiving transactions or messages. The public key is used to generate addresses, but they do not disclose any information about the private key. When sending a transaction to a particular user in a blockchain network, the recipient's address is used as the destination. The address functions as a pseudonymous identifier, providing privacy and security. The recipient can then access and manage the digital assets associated with that address using their private key.

2.1.3.2 Wallet

A cryptocurrency wallet is a software application or hardware device that enables users to store, administer, and interact with their digital assets in a secure manner. Wallets play a crucial role in the adoption and use of cryptocurrencies by both consumers, enhancing the overall experience with a variety of advantages. Wallets provide a convenient and intuitive interface for managing digital assets. They provide a

secure solution for storing private keys, which are required for accessing and controlling cryptocurrencies. Wallets enable users to transfer and receive funds, track their transaction history, and monitor their account balances by storing private keys securely. Wallets typically include features such as address book administration, transaction history, and real-time market data, providing users with a comprehensive set of tools for managing their cryptocurrency holdings. One important aspect of wallet security is the implementation of industry standards, such as the BIP39 specification [7], in the generation and management of mnemonic phrases or seed phrases. The BIP39 specification ensures that wallets adhere to a standardized method for generating mnemonic phrases, which are human-readable sets of words. These phrases can be used to derive the cryptographic keys necessary to access and manage cryptocurrency funds. By adopting the BIP39 specification, wallets provide users with a consistent and reliable way to backup and restore their wallets, offering an additional layer of security and ease of use. MetaMask [8] is a prominent example of a cryptocurrency wallet. MetaMask is a wallet extension for web browsers that enables users to interact with Ethereum-based decentralized applications (DApps) directly from the browser. It provides a user-friendly and secure interface for interacting with Ethereum accounts and the blockchain. MetaMask provides a straightforward and intuitive interface that integrates seamlessly with popular web browsers such as Chrome, Firefox, and Brave. Within minutes, users can install the MetaMask extension and configure their Ethereum wallet. After configuring the wallet, users can access their Ethereum accounts, view their token balances, and conduct transactions.

2.1.4 Consensus

The concept of consensus holds paramount importance in the realm of blockchain technology as it serves to guarantee the agreement and validity of transactions throughout the network. This mechanism encompasses a process by which decentralized nodes within the network collectively establish agreement regarding the current state of the blockchain. This paper examines two prevalent consensus algorithms, namely Proof of Work (PoW) and Proof of Stake (PoS).

PoW consensus algorithm, initially pioneered by Bitcoin [3], serves as the foundational mechanism for validating transactions and maintaining the integrity of the blockchain network. In the context of PoW, the participants referred to as miners engage in a competitive process aimed at solving intricate mathematical puzzles. In the realm of blockchain technology, the initial miner who successfully unravels the intricate puzzle is duly acknowledged and bestowed with a reward, subsequently appending a novel block to the existing chain of transactions. The aforementioned process necessitates a substantial amount of computational resources and incurs a considerable level of energy expenditure. The security of a blockchain system is upheld through the utilization of PoW, which effectively deters malicious entities from tampering with previous transactions. This is achieved by imposing a significant computational burden on any attempts to modify the blockchain's historical records.

PoS [4] consensus algorithm serves as a viable alternative to the PoW mechanism, with the primary objective of mitigating the concerns pertaining to energy consumption commonly associated with PoW. In the PoS consensus mechanism, the selection of validators to generate new blocks is determined by their cryptocurrency holdings and their willingness to "stake" said holdings as collateral. The selection of validators is typically conducted through a deterministic procedure, which frequently takes into consideration factors such as the magnitude of their stake and the duration for which they have maintained it. PoS consensus mechanism is widely acknowledged for its superior energy efficiency in comparison to the PoW mechanism, primarily due to its reduced reliance on intensive computational resources.

The utilization of both PoW and PoS consensus mechanisms presents distinct benefits and limitations. PoW consensus mechanism is renowned for its robust security measures, albeit at the cost of significant resource consumption. Conversely, the PoS protocol boasts energy efficiency advantages, yet it may exhibit vulnerability to specific forms of attacks. The selection between PoW and PoS is contingent upon the distinct objectives and prerequisites of a blockchain network.

2.2 Shamir's secret sharing

Shamir's secret sharing is a fundamental cryptographic algorithm that plays a vital role in assuring the secure distribution and storage of data across a variety of applications. This ingenious algorithm, developed by Adi Shamir in 1979, employs the principles of polynomial interpolation to divide a confidential secret into multiple shares. Then, these shares are distributed to various participants, each of whom holds a unique piece of the puzzle. The genius resides in the fact that the original secret can only be reconstructed by combining a minimum number of shares. The implementation of Shamir's secret sharing begins with the selection of the secret to be protected. The secret is then used as the constant element in a polynomial of a specified degree. This polynomial is the basis for constructing the shares. The evaluation of the polynomial at specific points corresponds to the allocation of shares to each participant. The flexibility of Shamir's secret sharing is its greatest asset. The threshold required to reconstruct the secret can be modified based on the system's particular requirements. If the threshold is set to "k," for instance, any combination of "k" or more shares can be used to recover the original secret. As long as the minimum threshold is maintained, this provides a robust mechanism for ensuring resilience against loss or larceny of shares. Shamir's disclosure of a secret has a wide range of applications in various disciplines. It is frequently employed in cryptographic key management, in which a sensitive cryptographic key is divided into portions and distributed to key holders. This adds an additional layer of security to critical systems by ensuring that no single individual can access the key without the cooperation of multiple parties.// Here is a straightforward illustration of how Shamir's secret sharing works. Suppose we wish to divide a secret value of 42 into 5 portions with a threshold of 3. By evaluating a polynomial at various points, the shares are generated. Share 1: (1, 17), Share 2: (2, 23), Share 3: (3, 38), Share 4: (4, 14) Share 5: (5, 7) x represents the point on the polynomial curve, while y is the value of the polynomial at that point. Now, we need at least three shares to reconstruct the secret. Consider the shares 2, 3, and 4. We can use these shares to interpolate the polynomial and determine the value at x = 0 that corresponds to our confidential value of 42 by employing interpolation. Using the Lagrange interpolation formula [9], the secret can be calculated:

$$\text{Secret} = \frac{23 \cdot (0-3) \cdot (0-4)}{(2-3) \cdot (2-4)} + \frac{38 \cdot (0-2) \cdot (0-4)}{(3-2) \cdot (3-4)} + \frac{14 \cdot (0-2) \cdot (0-3)}{(4-2) \cdot (4-3)}$$

After simplifying the equation, the hidden value is determined to be 42.

2.3 Distributed Key Generation

The Distributed Key Generation (DKG) protocol is at the vanguard of contemporary cryptographic mechanisms, offering a revolutionary method for collaboratively generating shared secret keys among multiple parties without the need for a centralized trusted authority. By removing the need for a single point of control, the DKG protocol significantly improves security, making it an indispensable instrument for protecting sensitive data and vital systems. The fundamental principle of the DKG protocol is its capacity to decentralize the generation and distribution of the secret key among all participants. This collaborative effort ensures that no single party has complete knowledge of the confidential key, thereby mitigating the risk of a single point of failure and reducing the number of exploitable vulnerabilities. The protocol consists of distinct segments, each of which plays a vital role in generating a shared secret key, the protocol's ultimate objective. In the initial stages of the key generation phase, the parties generate the secret key jointly. The distribution of these shares to the participants signifies the distribution phase. Verification is the next stage, during which each party verifies the received shares to ensure their authenticity and integrity. The true strength of the DKG protocol rests in the reconstruction phase, where the mathematical combination of the distributed shares is used to reconstruct the final confidential key. Importantly, the success of this reconstruction requires the participation of a minimum number of trustworthy parties. This threshold ensures that even if some parties are compromised or conduct maliciously, the security of the final secret key remains uncompromised so long as the minimum number of honest participants is maintained. Numerous advantages make DKG protocols an adaptable and indispensable resource for a variety of applications. DKG protocols

enable parties to collaboratively perform computations on sensitive data without disclosing their individual inputs, ensuring privacy and confidentiality. The DKG protocol facilitates the secure generation and distribution of cryptographic keys in cryptographic key management, safeguarding against unauthorized access and key compromise. Moreover, DKG protocols benefit threshold cryptography by enabling secure operations that necessitate the participation of a predefined threshold of parties. Secure communication protocols can also utilize DKG protocols to establish secure channels and prevent data interception and surveillance.

The Pedersen DKG protocol, proposed by Torben Pedersen [10] in 1991, was used in the thesis. The Pedersen DKG protocol utilizes polynomial interpolation techniques and cryptographic primitives to achieve secure and distributed key generation. It provides a robust mechanism for establishing shared secret keys without relying on a trusted central authority. The protocol involves several steps, including key generation, sharing, verification, and reconstruction.The Pedersen DKG protocol utilizes polynomial interpolation techniques and cryptographic primitives to achieve secure and distributed key generation. It provides a robust mechanism for establishing shared secret keys without relying on a trusted central authority. The protocol involves several steps, including key generation, sharing, verification, and reconstruction.

2.4 Smart contract

2.4.1 History and defination

Smart contracts are agreements that automatically carry out their obligations because they are encoded in code. By eliminating the need for middlemen and supplying a safe and decentralized method to facilitate and enforce agreements or transactions, these contracts automatically execute and enforce themselves. The idea of smart contracts has been around since the 1990s, and computer scientist Nick Szabo [11] is credited with coining the term. However, smart contracts did not receive much attention or widespread use until the advent of blockchain technology, particularly with the launch of Ethereum [4] in 2015. A Turing-complete programming language was introduced by Ethereum, a decentralized blockchain platform, allowing for the creation and execution of sophisticated smart contracts. This innovation paved the way for the development of decentralized applications (DApps) that might use smart contracts to secure and automate a variety of activities, including voting systems, supply chain management, and financial transactions. Since then, smart contracts have become more well-known and are being investigated in a variety of sectors and industries for their potential to transform conventional corporate operations. Their immutability and transparency, along with the ability to automate processes and get rid of middlemen, have the potential to improve workflow, boost productivity, and cut costs.

2.4.2 Practical use cases

Smart contracts have become a game-changing technology with several real-world applications in a wide range of industries. These self-executing contracts, which are inscribed on a blockchain, allow for secure and automated transactions, doing away with the need for middlemen and enhancing participant trust. Smart contracts have transformed lending platforms, decentralized exchanges, and yield farming protocols in the field of DeFi [12]. Smart contracts offer a transparent and effective ecosystem for decentralized financial applications by automating financial transactions and following established regulations. Likewise, supply chain management has benefited from smart contracts. By facilitating seamless tracking and verification of commodities along the supply chain, these contracts improve traceability, lower fraud, and streamline logistical operations. Smart contracts have simplified real estate transactions by enabling property transfers, escrow services, and rental agreements. Smart contracts increase efficiency and transparency in the real estate sector by doing away with middlemen and automating repetitive operations.

In this thesis, the smart contract plays a pivotal role in the ecosystem by fulfilling a multitude of significant responsibilities. The storage and maintenance of configuration updates for the Perdesen Distributed Key Generation (DKG) protocol is a primary responsibility. The smart contract is responsible for managing a whitelist of decentralized applications (DApps) that utilize the aforementioned solution. This mechanism ensures that only authorized DApps are able to participate in the protocol. The smart contract is designed

to enable the asynchronous execution of the Perdesen DKG protocol rounds, which is a fundamental feature of its functionality. The process entails the antecedent creation of cryptographic keys through the secure retention of encrypted shares for each node involved in the operation. The implementation of a smart contract facilitates the asynchronous execution of the key generation procedure, thereby enhancing operational efficiency and scalability. The transparent management of the key generation process is regarded as a fundamental characteristic of the smart contract. The implementation of transparency in the process is paramount to guaranteeing the privacy and security of participants' private keys. The provision of transparency enables participants to authenticate the advancement and soundness of the key generation procedure while upholding the confidentiality of their private key data. The smart contract assumes a crucial function in the verification of signatures from nodes that are involved in the process. The implementation of a verification process within the Perdesen Distributed Key Generation (DKG) protocol serves to guarantee that exclusively legitimate users are allocated roles during each round of the cryptographic scheme. Through the process of signature verification, the smart contract ensures the integrity and authenticity of the nodes involved, thereby augmenting the overall security and dependability of the protocol.

2.5 Executor for DApps

In the domain of blockchain-based decentralized applications (DApps), the executor plays a crucial role in facilitating the execution of blockchain network transactions. As the backend component of a DApp, the executor functions as an intermediary between the user-facing frontend and the blockchain. It is primarily responsible for establishing a connection to the blockchain, submitting transactions on behalf of users, and managing various interactions with the smart contract. Management of the user's private key or mnemonic is a fundamental aspect of an executor's functionality. A crucial piece of cryptographic information, the private key enables the authentication of transactions and verifies the identity of blockchain users. In some implementations, the private key is stored as a mnemonic, which is a string of phrases that serves as a human-readable representation of the key. By storing the mnemonic securely in the backend, the executor can access it when required to sign transactions on behalf of the frontend, thereby ensuring a seamless and secure user experience. Executors serve as verifiers and assigners within the Perdesen Distributed Key Generation (DKG) protocol in the context of the thesis. As part of this cryptographic protocol, multiple participants generate a shared secret key without relying on a central authority that can be trusted. The executors are responsible for confirming the correctness and impartiality of the smart contract's round assignment procedure. Executors contribute to the overall security and resilience of the decentralized system by functioning as dependable parties. Their function in the DKG protocol ensures that the key generation and distribution process is carried out with diligence and impartiality, thereby protecting the shared secret key's integrity. In addition, their participation in validating the veracity of the protocol adds an additional layer of confidence to the entire system, as it ensures that the generated secret key is genuine and can be used securely in cryptographic operations.

2.6 Social login for Web3

2.6.1 Web3 and Dapps

Web3 and Decentralized Applications (DApps) have emerged as critical components of the evolution of the internet and digital ecosystems. Web3 is the vision of a more decentralized and user-centric internet, in which individuals have greater control over their data, identity, and digital interactions. It is a collection of technologies, protocols, and frameworks designed to empower users, cultivate trust, and facilitate peer-to-peer interactions. DApps, on the other hand, are applications that are created on top of decentralized networks and typically utilize blockchain technology. These applications inherit Web3's fundamental principles, including decentralization, transparency, and user ownership. From financial services and governance platforms to gaming and social media applications, they provide a variety of features. The development and adoption of Web3 and DApps have been supported by a growing body of research and innovation. Several academic papers and technical publications have contributed to the advancement of these technologies.

For instance, the paper by Wood et al. titled "Ethereum: A Secure Decentralized Generalized Transaction Ledger" provides a comprehensive overview of the Ethereum platform and its underlying principles [4]. Another significant contribution is the work by Swan, who explores the concept of "Token Economy" in his book "Token Economy: How the Web3 Reinvents Value Exchange." The book delves into the transformative potential of tokenization and its implications for various industries [13]. Moreover, the paper by Buterin et al. titled "A Next-Generation Smart Contract and Decentralized Application Platform" introduces the Ethereum platform, highlighting its unique features and use cases [14]. This paper serves as a foundational reference for understanding the capabilities and potential of DApps built on Ethereum.

2.6.2 Social login and OAuth

Social login is a prevalent method of authentication that enables users to log in to websites and applications using their existing social media accounts. Users are no longer required to establish new accounts and remember additional login credentials. Instead, users can merely click on a social media button, such as "Sign in with Facebook" or "Sign in with Google," to authenticate themselves. Social registration is supported by the OAuth2 (Open Authorization 2.0) [15] protocol, which provides a secure and standardized authentication framework. When a user logs in with a social media account, the website or application sends them to the respective social media platform for authentication. The user is then presented with a consent interface that describes the data to which the website or application requests access. Once the user grants permission, the social media platform provides the website or application with an access token that can be used to retrieve user information and authenticate the user's identity. Using OAuth2 for social authentication has multiple advantages. It increases security by removing the need for websites and applications to store user credentials. Instead, the obligation for authentication falls on the shoulders of the most reputable social media platforms. Second, social login streamlines the user experience by allowing users to log in with a few clicks and avoid the inconvenience of creating new accounts. It also allows websites and applications to utilize the extensive user profile data available on social media platforms, including user names, profile pictures, and email addresses, for personalization and customization.

2.6.3 Social login benefits DApps

The technical complexity of Web3 technology, with its underlying blockchain, cryptographic keys, and smart contracts, can be intimidating for ordinary consumers. Understanding and navigating these complexities can be a significant barrier to entry for non-technical individuals, impeding the widespread adoption and utility of Web3 platforms. The decentralized nature of Web3 can exacerbate this problem, resulting in fragmented user experiences and inconsistent user interfaces, which makes it difficult for non-technical users to interact effectively with decentralized applications. To resolve these issues, it is essential to simplify the user experience and increase Web3 technologies' accessibility. By refining complicated processes and providing user-friendly interfaces, we can enable regular users to exploit Web3's potential without being overwhelmed by its technicalities. A design that is user-friendly and intuitive can make Web3 applications more accessible to the general public, thereby promoting their widespread adoption. Using smart contracts, the Pedersen Distributed Key Generation (DKG) protocol, and Shamir secret sharing, the proposed solution presented in this thesis seeks to enhance the usability of Web3 applications. Using these cryptographic techniques, the system can generate keys in a secure and distributed manner, thereby protecting sensitive data and enhancing user privacy and security. In addition, the solution addresses the difficulties encountered by non-technical users by removing the complexities of Web3 technologies, allowing them to interact with the system without difficulty. By incorporating smart contracts, users can access decentralized applications without needing to delve into the fundamentals of blockchain operations. The Pedersen DKG protocol and Shamir secret sharing mechanisms provide a robust and secure method for generating and exchanging cryptographic keys, ensuring that users retain control over their data while still taking advantage of Web3's decentralized nature. This thesis ultimately seeks to democratize Web3 technologies by making them more inclusive and accessible to a wider audience. By lowering entry barriers and enhancing the user experi-

ence, the proposed solution aims to unleash the full potential of Web3 for regular users, allowing them to confidently and easily participate in the decentralized ecosystem. The objective is to advance a more user-friendly, secure, and decentralized digital landscape through a combination of technological innovations and user-centric design.

CHAPTER 3. SOLUTION

Chapter 3 will provide a thorough examination of the system characteristics and architecture of our novel social login system. This chapter provides a fundamental understanding of the complex mechanisms and distinguishing characteristics that set our system apart in the decentralized oracle network field. Next, the chapter will provide a detailed description of the system architecture. The social login system has been divided into three sub-processes to enhance clarity and address its inherent complexity. This section will provide a comprehensive explanation of each sub-process, emphasizing its distinct functionalities and responsibilities within the overall system flow. By breaking down the system into separate phases, readers can understand the specific steps required to process a user's request and smoothly execute the social login mechanism.

3.1 System's characteristics

This study examines the efficacy and adaptability of a system that utilizes conventional social login methods to augment the UX of DApps. The following characteristics can be more detailed:

Security - combining the potential benefits of blockchain smart-contract, SSS, and Pedersen DKG protocol. The security infrastructure of the system is established upon the utilization of blockchain smart contracts. Programmable contracts, operating on a decentralized network, guarantee the attributes of transparency, immutability, and tamper-proof execution of operations. Shamir secret sharing is used to fortify the protection of cryptographic keys and other forms of private information. This method entails breaking up a secret into smaller pieces and giving them to several parties. The Petersen DKG protocol distributes and secures cryptographic key generation, ensuring system security. This system lets a group of people generate a shared cryptographic key without anyone having the full key.

Efficacy and adaptability - By leveraging blockchain's decentralized nature, Web3Auth eliminates the reliance on centralized identity providers, reducing the potential for single points of failure and enhancing security. Additionally, blockchain-based identity systems enable instant verification of user credentials, eliminating the need for lengthy verification processes and reducing transaction times. The open solution's architecture enables seamless integration with any DApps, making it easier for developers to adopt and implement with their products.

User-friendly - This solution supports popular authentication mechanisms such as social login which are widely used in the traditional web. This familiarity allows users to leverage their existing accounts and authentication methods, reducing the learning curve and providing a seamless transition into the Web3 space. Users can easily authenticate their identities across multiple DApps using a unified and standardized protocol, without the need to remember and manage multiple usernames and passwords. This eliminates the hassle of creating and maintaining numerous accounts, making the user experience more streamlined and efficient.

Scalability - Compatible with any type of social authentication, including Facebook, Google, and Twitter. Utilizing a decentralized architecture increased the request's performance and volume by spreading it across multiple executors and parallel handling processes.

3.2 Overall of system

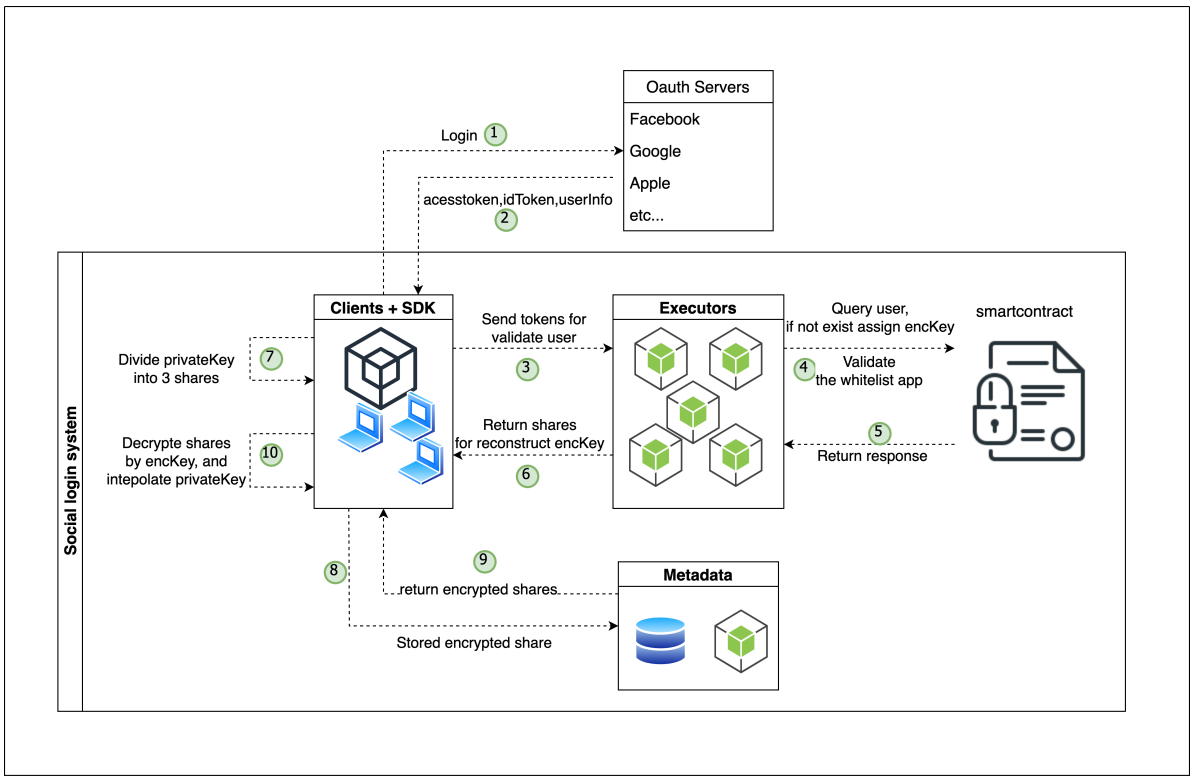

Figure 3.1: Overall the system

Figure 3.1 depicts the exhaustive and intricate request flow of a user interacting with the social login system. This intricate process has been meticulously subdivided into three distinct sub-processes, each with a specific function, to facilitate a clearer and more comprehensive comprehension of the system's overall functionality and interactions. The user's voyage begins when they initiate a request, at which point they choose the convenience and security of the social login feature. Subsequently, the initial sub-process begins, meticulously validating the user's input and ensuring the request's highest level of security. This validation phase is essential for protecting against potential vulnerabilities and maintaining the system's integrity. After the request has successfully passed the initial validation phase, the second sub-process authenticates the social logon credentials against the relevant provider's robust authentication system. This crucial phase ensures the authenticity of the user's identity and restricts system access to only authorized individuals. The rigorous authentication procedure protects against unauthorized access and fosters a secure environment for user interactions. The third sub-process assumes control upon successful authentication of the social logon credentials, orchestrating the logic for constructing the encryption key and the assignment key. These keys are essential to assuring the security and confidentiality of user information throughout their interaction with the system. The encryption key ensures that sensitive information remains confidential and protected from potential breaches, while the assignment key optimizes the user experience by facilitating the seamless assignment of roles and permissions. Throughout this complex process, the system employs smart contracts and blockchain technology, which serve as the foundation for preserving and protecting user data for the duration of the blockchain's active state. By leveraging the capabilities of smart contracts and blockchain, the system maintains a decentralized and transparent infrastructure, thereby nurturing user confidence.

3.2.1 Auth0 connection

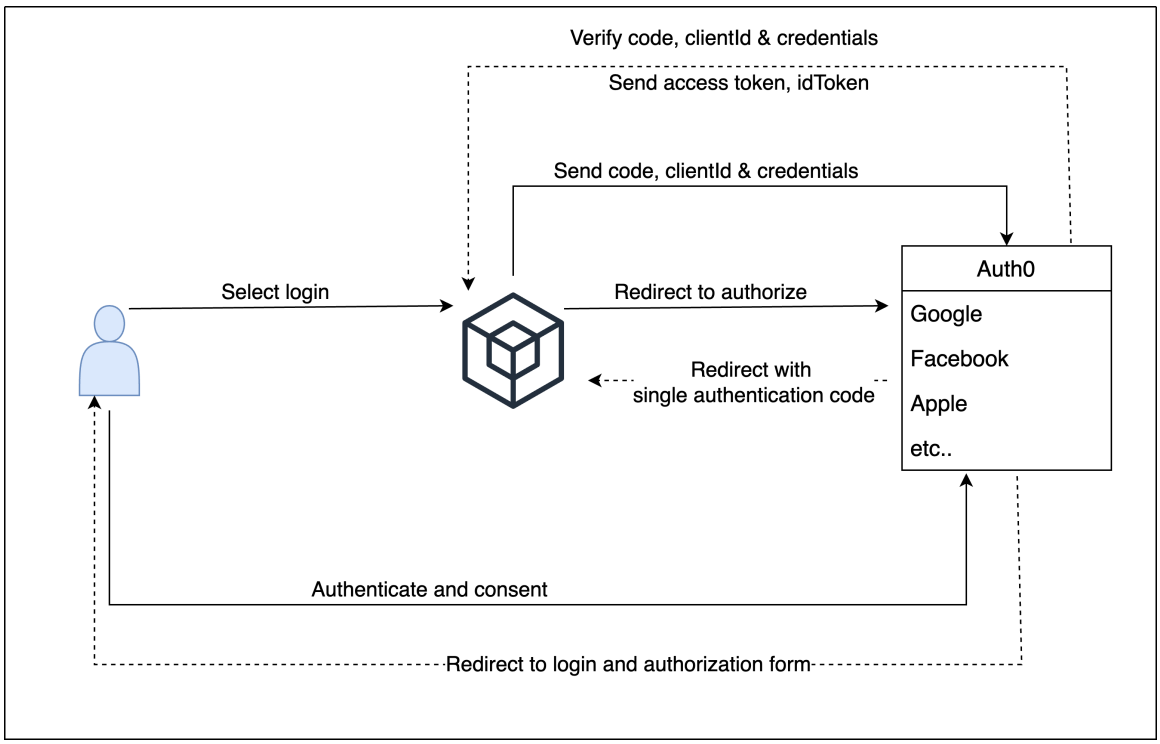

Figure 3.2: Auth0 connection

Figure 3.2 provides a comprehensive graphical representation of the Auth0 authentication and authorization solution, highlighting its critical role in facilitating secure and streamlined user interactions within the application. When a user decides to authenticate within the application, they are presented with a number of options, including traditional username and password authentication and social registration via Google or Facebook. This versatility enables users to select their preferable authentication method, thereby enhancing the overall user experience. The application initiates the authentication process by redirecting the user to Auth0's authorization endpoint once the user has selected the authentication type. This phase begins the secure process of user authentication and authorization. At the authorization endpoint, the user is presented with a login and authorization prompt, where they authenticate themselves securely and grant the required permissions. This step ensures that only authorized users have access to the application's resources and can perform particular operations, thereby protecting the application's integrity and sensitive data. The Auth0 server redirects the user back to the application with an authorization code following successful authentication and assent. This authorization code is a crucial component of the subsequent authentication and authorization processes. The application then exchanges this authorization code along with its own credentials and client ID with the token endpoint of Auth0. The Auth0 server meticulously verifies this exchange, ensuring the authenticity and validity of the supplied data. Upon successful authentication, the Auth0 server issues the application both an access token and an ID token. During the authentication and authorization procedure, these tokens serve unique functions. The access token functions as proof of authorization, allowing the application to authenticate future requests on behalf of the user. It grants access to protected resources and API endpoints, facilitating secure and seamless application-to-server communication. The access token is typically included in API request parameters by the application, facilitating secure data exchange and interactions. The ID token, on the other hand, comprises vital user information, which facilitates identification and authentication within the application. It may contain information such as the user's email address, identify, and other pertinent attributes. The ID token enables the application to recognize the user and customize the user's experience within the application, thereby creating a personalized, user-centric environment. Auth0 is a highly dependable and efficient authentication and authorization solution due to

its extensive feature set and stringent security measures. Auth0 assures the confidentiality and integrity of user data throughout the entire authentication and authorization procedure by adhering to secure protocols and industry best practices. Its seamless integration with multiple authentication methods and customizable user experiences make it a valuable asset for application developers seeking to improve the security and usability of their software.

3.2.2 Requesting encKey for Executors

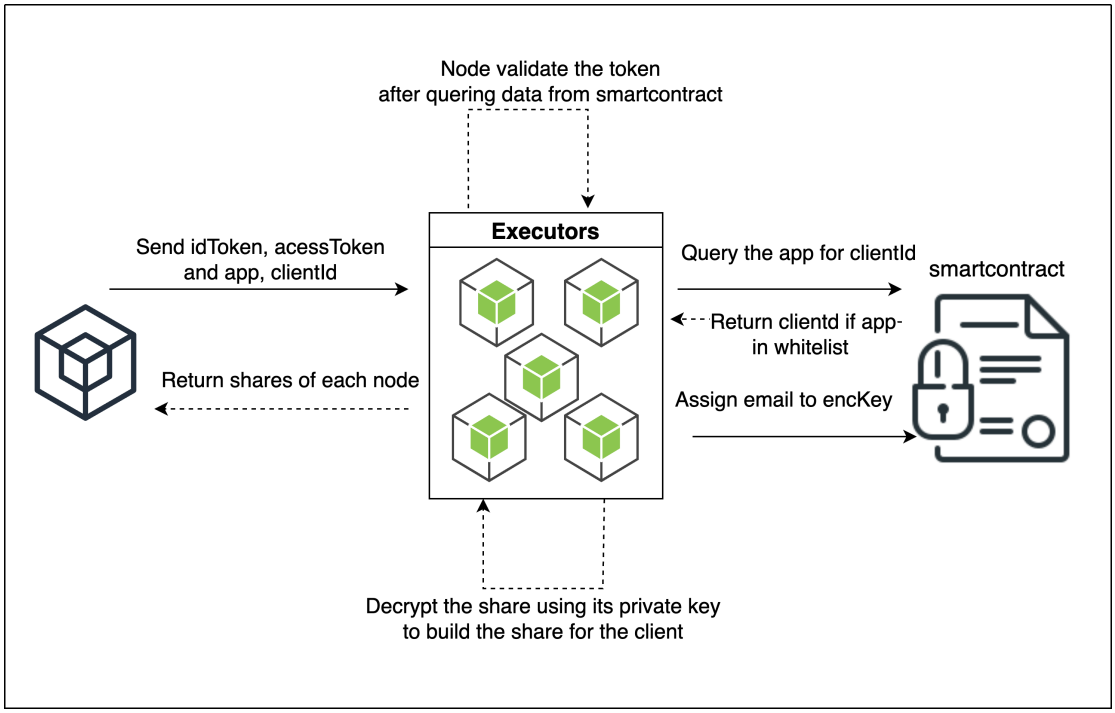

Figure 3.3: Request assign the share

The process of ensuring secure and effective authentication and authorization in the system, as depicted in Figure 3.3, consists of multiple phases working in concert to provide a seamless user experience. The SDK, which functions as the interface between the user's device and the system, is central to this process. The SDK transmits the idToken or accessToken, along with the appName and clientId, to the executors whenever the user initiates the authentication process. As essential system components, the executors play a crucial role in authenticating the user's credentials and ensuring secure smart contract interaction. They begin by querying the smart contract to retrieve the app's essential authentication and authorization parameters and configuration data. This data is essential for establishing a secure and trusted connection between the user and the application. Executors initiate the validation procedure after receiving the data from the smart contract. It is essential to validate the integrity and authenticity of the data extracted from the idToken or accessToken to ensure that the user's credentials have not been altered during transmission and are valid. When a new user attempts authentication, the executors generate a unique key for that user. This key functions as a digital identifier and is securely associated with the user's authenticated email address. This phase ensures that each user within the system has a unique and distinguishable key, allowing for secure and efficient user management. The smart contract encrypts the shares using the private keys of the nodes that participated in the protocol as part of the distributed key generation process. The encrypted shares are transmitted in a secure manner to the executors, who can decrypt them with their own private keys. This secure decryption safeguards the shares' confidentiality and integrity, preventing unauthorized access. The executors reconstruct the final share and promptly return it to the SDK after decrypting the shares. These shares play a crucial role in the subsequent phases of authentication and authorization, enabling the user to access and interact with decentralized applications (DApps) within the ecosystem with confidence and

security. By strictly adhering to this exhaustive and complex procedure, the system guarantees that user authentication and authorization are both effective and secure. The combination of smart contracts, Shamir secret sharing, and the Pedersen DKG protocol enables the system to offer a robust, trustworthy, and user-friendly Web3 authentication solution. This comprehensive strategy addresses the complexities of Web3 technologies and promotes widespread adoption and utility by improving the security, accessibility, and overall user experience of decentralized applications.

3.2.3 Generating and Storing Shares in Metadata

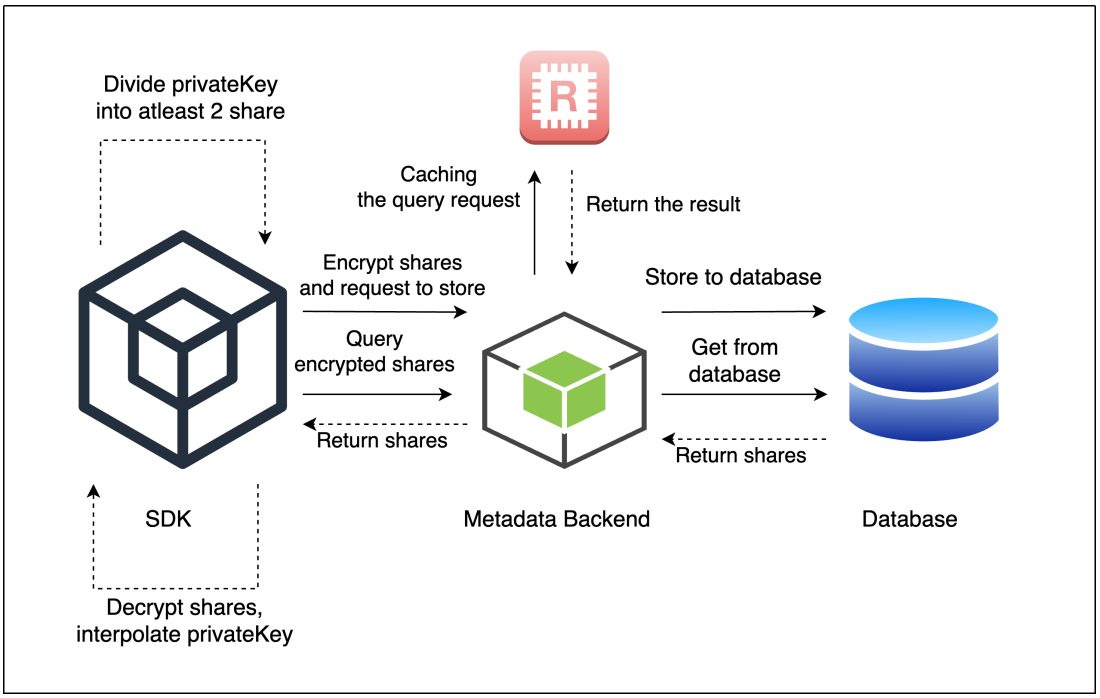

Figure 3.4: Generate Reconstruct shares

As depicted in 3.4, the process of reconstructing the private key in our system consists of multiple phases that have been meticulously designed to ensure optimum security and dependability. It commences with the implementation of Shamir's secret sharing scheme, in which the SDK divides the private key into at least two shares. Each share is then encrypted to ensure its confidentiality and integrity, preventing unauthorized access. These encrypted shares are transmitted in a secure manner to the metadata repository, where they are stored in a database. The SDK initiates a query request to the metadata repository whenever the need arises to reconstruct the private key. The backend retrieves the encrypted shares associated with the specified user and application from the database upon receiving the query. The query result, including the encrypted shares, is cached in a Redis database to boost performance and accelerate subsequent requests. After receiving the encrypted shares, the SDK initiates the reconstruction process. The SDK decrypts each share using its respective decryption key. Only authorized processes have access to these decryption keys, which are managed securely within the SDK. The SDK recovers the original confidential information contained within each share by decrypting the shares. The SDK combines the decrypted shares using interpolation algorithms, such as Lagrange interpolation, to generate the final private key. Using the available shares, this mathematical calculation interpolates the absent portions of the private key. The SDK successfully reconstructs the original private key through this procedure. Throughout the entirety of the reconstruction process, robust encryption algorithms and secure key management procedures are used to protect the privacy and integrity of the private key. The implementation of Shamir's scheme for secret sharing is essential for enhancing security because it distributes the key across multiple shares, making it resistant to single points of failure or compromise. Our system guarantees the safety and reliability of the private key's retrieval by implementing this exhaustive and secure reconstruction procedure. This feature enables users

to securely access their Web3 ecosystem accounts and confidently execute authorized operations. In addition, the comprehensive security measures ensure that the private key remains protected and inaccessible to unauthorized entities, giving users peace of mind when interacting with decentralized applications (DApps) and the broader Web3 ecosystem. This procedure's meticulous design and execution exemplify the system's dedication to providing a secure and user-friendly Web3 authentication solution.

3.3 Asynchronous create encKey process

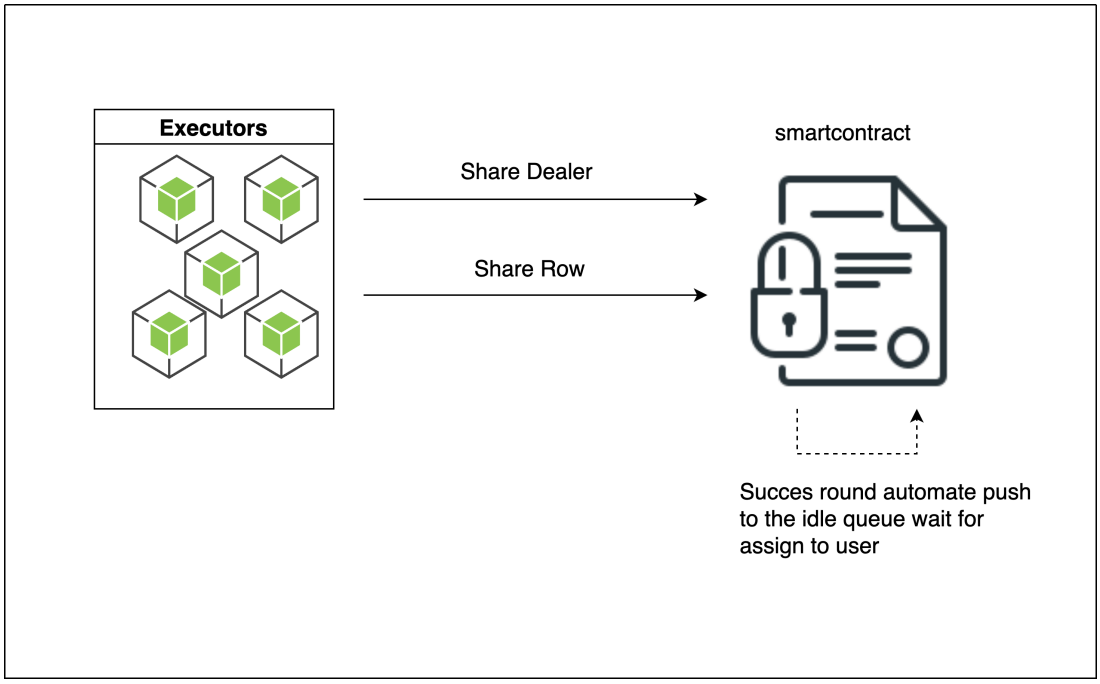

Figure 3.5: Asynchronous create encKey

The asynchronous creation of the encKey process is a crucial and ingenious element of the system's architecture, as it is intended to maximize efficiency and resource utilization. Without running the Distributed Key Generation (DKG) protocol each time, this method ensures that keys can be assigned to users quickly and seamlessly, without the need to run the protocol each time. By generating a pool of unused encKeys in advance, the system significantly reduces the time and computational overhead required for key assignment, ultimately augmenting the user experience. The executors, the pillars of the encKey generation procedure, are responsible for initiating the procedure. They constantly monitor the current round's status and the number of inactive keys. When the system detects a need for new keys, such as when the present state is null or in the "waitfordealer" phase, and the number of idle keys falls below the anticipated threshold, the executors initiate the subsequent round. This proactive strategy ensures that the system is always ready to meet user demands and minimizes any potential delays in key assignment. During the initial phase of the round, known as the "sharing dealer" phase, the executors combine and distribute their individual confidential shares to the designated sharing dealer. This strategic partnership permits each executor to contribute to the procedure without disclosing sensitive information that could compromise the overall secret key. By effectively sharing their secret shares, the executors disseminate the secret in a manner that maintains the keys' security and integrity. Once the "sharing dealer" phase is effectively completed, the protocol seamlessly transitions to the "waitforrows" phase. In this phase, the executors generate their own public key shares using the shared secrets obtained in the previous phase. These public key exchanges are essential to the conclusion of the key generation process, as they guarantee the authenticity and validity of the user-assigned keys. The system is now well-prepared for the assignment of encKeys to users, as the "waitforrows" phase has concluded. When a user requests their key, the system assigns the correct round to their assigned key and provides it, ensuring a speedy and efficient assignment. By adopting this asynchronous method for encKey generation,

the system obtains a number of substantial benefits. First, it optimizes resource usage by eliminating the needless re-execution of the DKG protocol for each key assignment. Second, it improves the scalability and performance of the system, allowing it to manage a large number of user requests without compromising response times. Lastly, it assures a secure and seamless workflow for encKey generation and assignment, reinforcing the system's dedication to user convenience and data security. In conclusion, the incorporation of the DKG protocol and the effectiveness of the asynchronous encKey creation procedure demonstrate the innovative and sophisticated nature of the system's design. The system's ability to efficiently generate and assign encKeys contributes to its overall robustness, allowing Web3 ecosystem users to enjoy a secure and user-friendly experience.

CHAPTER 4. TECHNICAL ISSUES AND DESIGN

In the fourth chapter of this dissertation, we delve into the complexities and technical aspects of our innovative social authentication system. It is a comprehensive guide for comprehending the architecture, design, and implementation of the system. This chapter provides a detailed analysis of a number of crucial components, casting light on the obstacles encountered during development and the solutions employed to overcome them.

4.1 Requirement analysis

4.1.1 General usecases diagram

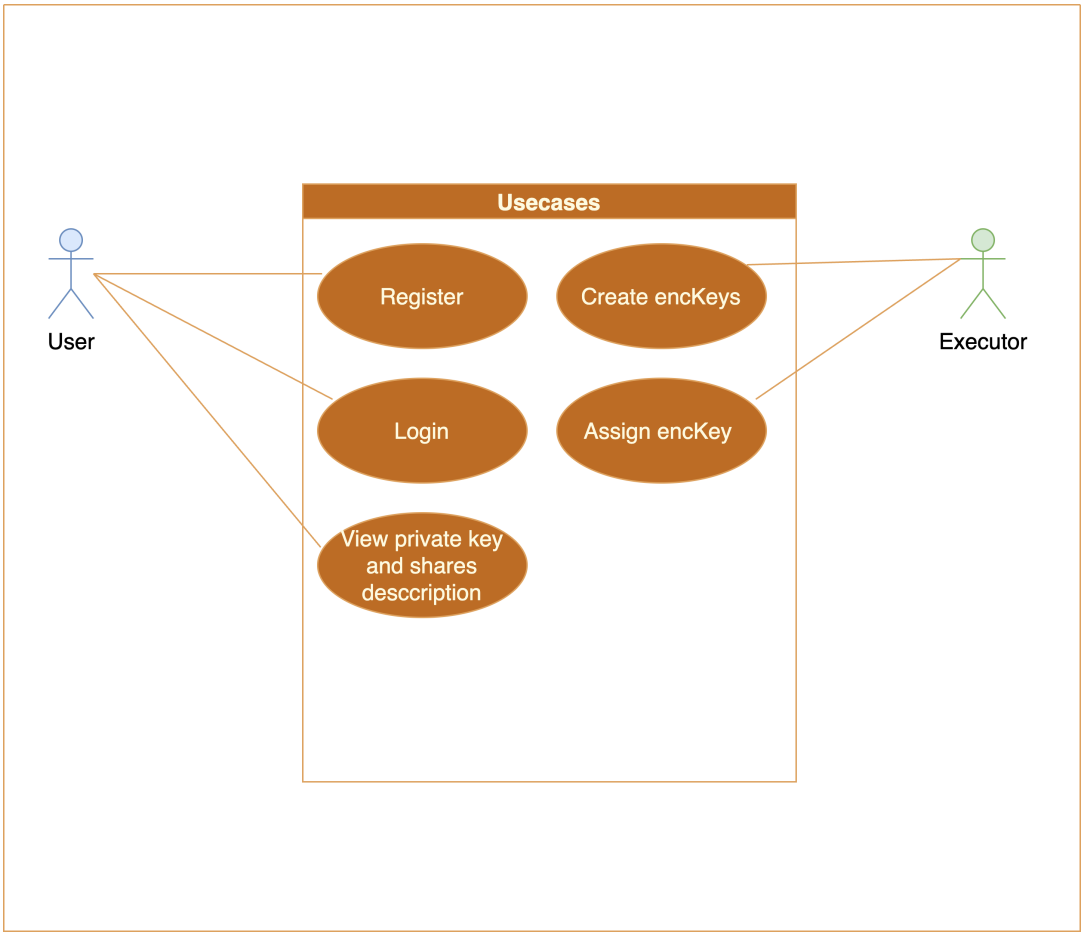

Figure 4.1: General usecases diagram

The figure 4.1 depicts the general usecases of Social login system. There are two actors included in the system. The first one is User who will use the main function social login. The table below describes more detail about functionalities of usecase:

No	Use Case	Description
1	Regiser	User using their social account such as: Google, facebook, ... for signing up the web3 spaces
2	Login	User using their social account such as: Google, facebook, ... for signing in the web3 spaces
3	View the private key and shares description	User fully controls and manages their private key and their shares.

Table 4.1: User's usecase description

The second participant is the Executor. They are special users with a crucial function who run the system's create encKey and assign encKey processes in the background.

No	Use Case	Description
1	Create encKey	Executor run the phases of Pedersen Protocol for pre-creating the encKey for user to encrypt their share.
2	Provide signature	Executors communicate with one another via signatures; if the number of valid signatures exceeds the threshold, anyone can delegate a key to a specific user via a multi-signature mechanism.

Table 4.2: Executor's usecase description

4.1.2 Usecase specifications and activity diagrams

For a better understanding of the use cases, I'd like to include the corresponding activity diagrams beneath each specification.

Usecase code	UC001	**Usecase name**	Social login	
Actor	User			
Pre-condition	At least have 1 social account			
Main flow of event	**No**	**Actor**	**Action**	
	1	User	Select social login type	
	2	System	Redirect to authorize	
	3	Auth0	Redirect to login and authorization form	
	4	User	Authenticate and consent	
	5	Auth0	Redirect with single use authorization code	
	6	System	Send code clientId and credentials to get oauth token	
	7	Auth0	Verify code, cliendId, credentials	
	8	Auth0	Send idToken, access token and refresh token	
	9	System	Authenticate the idToken or access token	
	10	System	Assign encKey for user	
	11	System	Send shares	
	12	System	Contruct the encKey	
	13	System	Construct the private Key	
Alternative flow of event	**No**	**Actor**	**Action**	
	5b	Auth0	Return error if the authenticate and consent fail or rejected	
	8b	Auth0	Response with error because fail in verification	

Table 4.3: Sign up specification

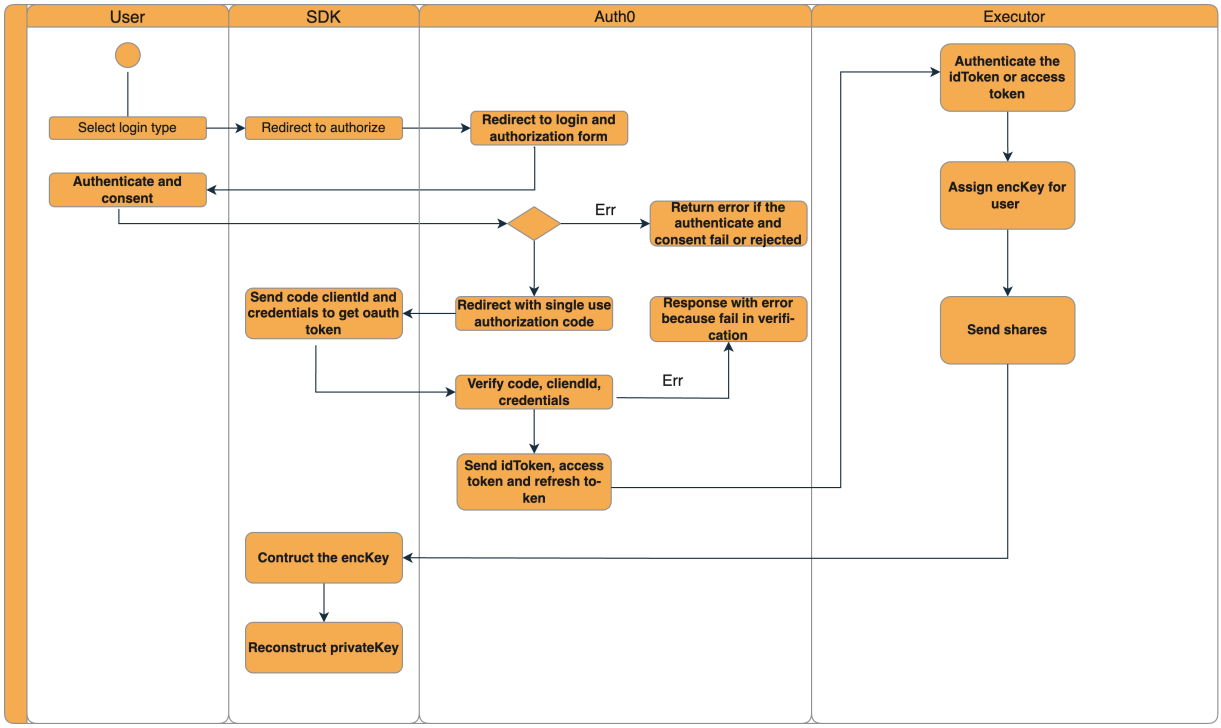

Figure 4.2: Signing up activity

Figure 4.2 is an activity diagram that illustrates the process, how users interact with the system, and the connections between the components. The SDK acts as a bridge between users and the remainder of the system, while Executors and Auth0 validate users' identities. In addition, executors assign and provide the requirements for user encKey and private key construction.

Usecase code	UC001	**Usecase name**	Social login
Actor	User		
Pre-condition	At least have 1 social account		
Main flow of event	**No**	**Actor**	**Action**
	1	User	Select social login type
	2	System	Redirect to authorize
	3	Auth0	Redirect to login and authorization form
	4	User	Authenticate and consent
	5	Auth0	Redirect with single use authorization code
	6	System	Send code clientId and credentials to get oauth token
	7	Auth0	Verify code, cliendId, credentials
	8	Auth0	Send idToken, access token and refresh token
	9	System	Authenticate the idToken or access token
	10	System	Send shares
	11	System	Contruct the encKey
	12	System	Construct the private Key
Alternative flow of event	**No**	**Actor**	**Action**
	5b	Auth0	Return error if the authenticate and consent fail or rejected
	8b	Auth0	Response with error because fail in verification

Table 4.4: Sign in specification

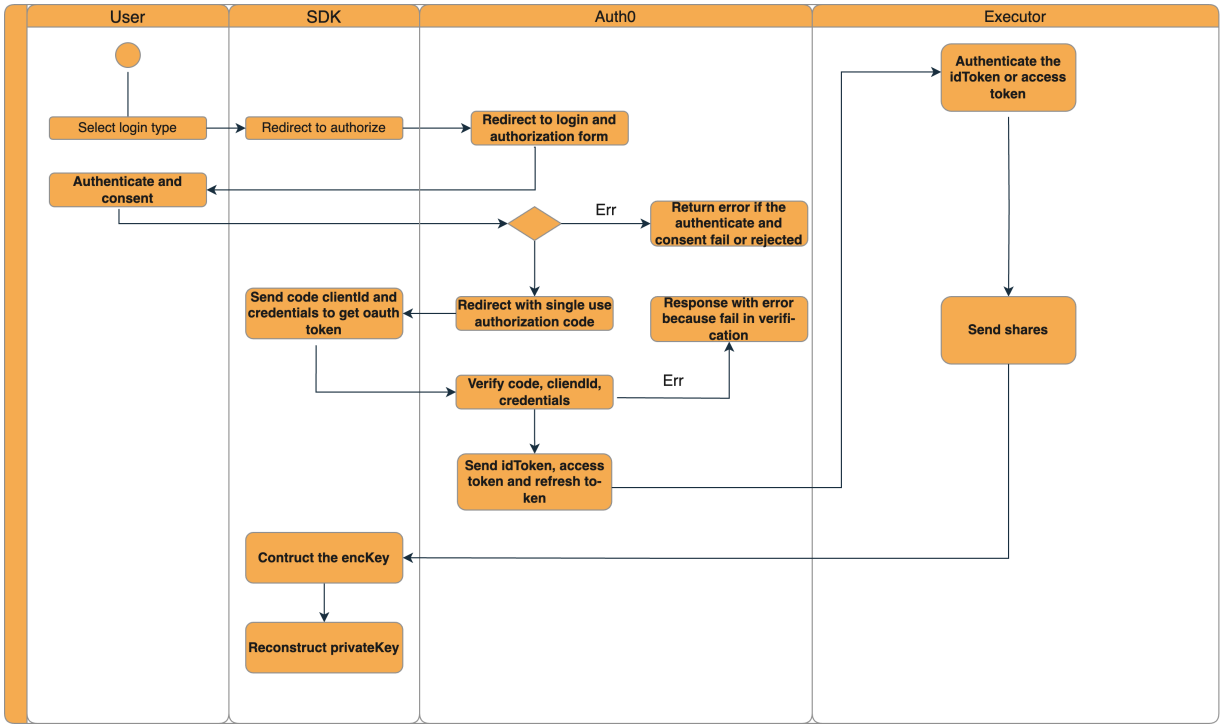

Figure 4.3: Signing in activity

Similar to figure 4.2, figure 4.3 describes in detail how users login into the system and how the system's components interact.

Usecase code	UC001	**Usecase name**	Social login
Actor	User		
Pre-condition	At least have 1 social account		

Main flow of event	**No**	**Actor**	**Action**
	1	User	Select social login type
	2	System	Redirect to authorize
	3	Auth0	Redirect to login and authorization form
	4	User	Authenticate and consent
	5	Auth0	Redirect with single use authorization code
	6	System	Send code clientId and credentials to get oauth token
	7	Auth0	Verify code, cliendId, credentials
	8	Auth0	Send idToken, access token and refresh token
	9	System	Authenticate the idToken or access token
	10	System	Send shares
	12	System	Contruct the encKey
	13	System	Query metadata
	14	System	Return shares description
	15	System	View private key and share description

Alternative flow of event	**No**	**Actor**	**Action**
	5b	Auth0	Return error if the authenticate and consent fail or rejected
	8b	Auth0	Response with error because fail in verification

Table 4.5: View private key and shares description

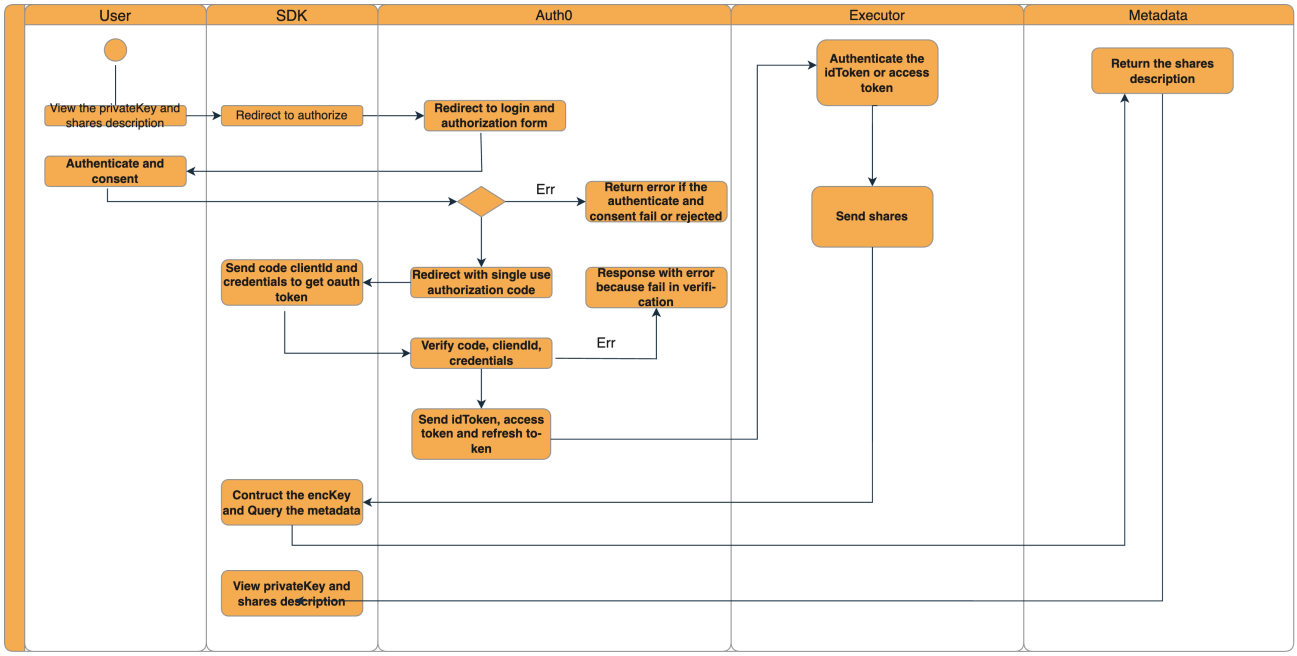

Figure 4.4: View private key and shares description activity

The last use case, showing in 4.4, for the user involves viewing their private key and their shares. This functionality allows users to have full control and management over their shares, even outside the system. By providing access to their private key and the corresponding shares, users can securely store this information in a location of their choice, such as a hardware wallet or an encrypted offline storage. This feature enhances the security and flexibility of the system, as users have the freedom to manage their shares independently and ensure the safety of their cryptographic assets. Additionally, by being able to view their private key and shares, users can also perform offline verifications and audits, ensuring the integrity and accuracy of their cryptographic operations.

Usecase code	UC001	**Usecase name**	Social login
Actor	Executor		
Pre-condition	Executor is a member of DKG		
Main flow of event	**No**	**Actor**	**Action**
	1	Executor	create encKey
	2	Executor	Query config and number of idle keys
	3	Executor	Share dealer phase
	4	Smart contract	Store the commitments and shares
	5	Smart contract	Auto update round status
	6	Excutor	Query the current round status and round information
	7	Executor	Construct the final shares and the pubkey share from the round information
	8	Executor	Share row phase
	9	Smart contract	Store the commitments and shares
	10	Smart contract	Auto update round status
	11	Smart contract	Push the complete round to the pool
Alternative flow of event	**No**	**Actor**	**Action**
	3b	Executor	Stop and wait the next time invoking

Table 4.6: Executor contributes in the DKG protocol

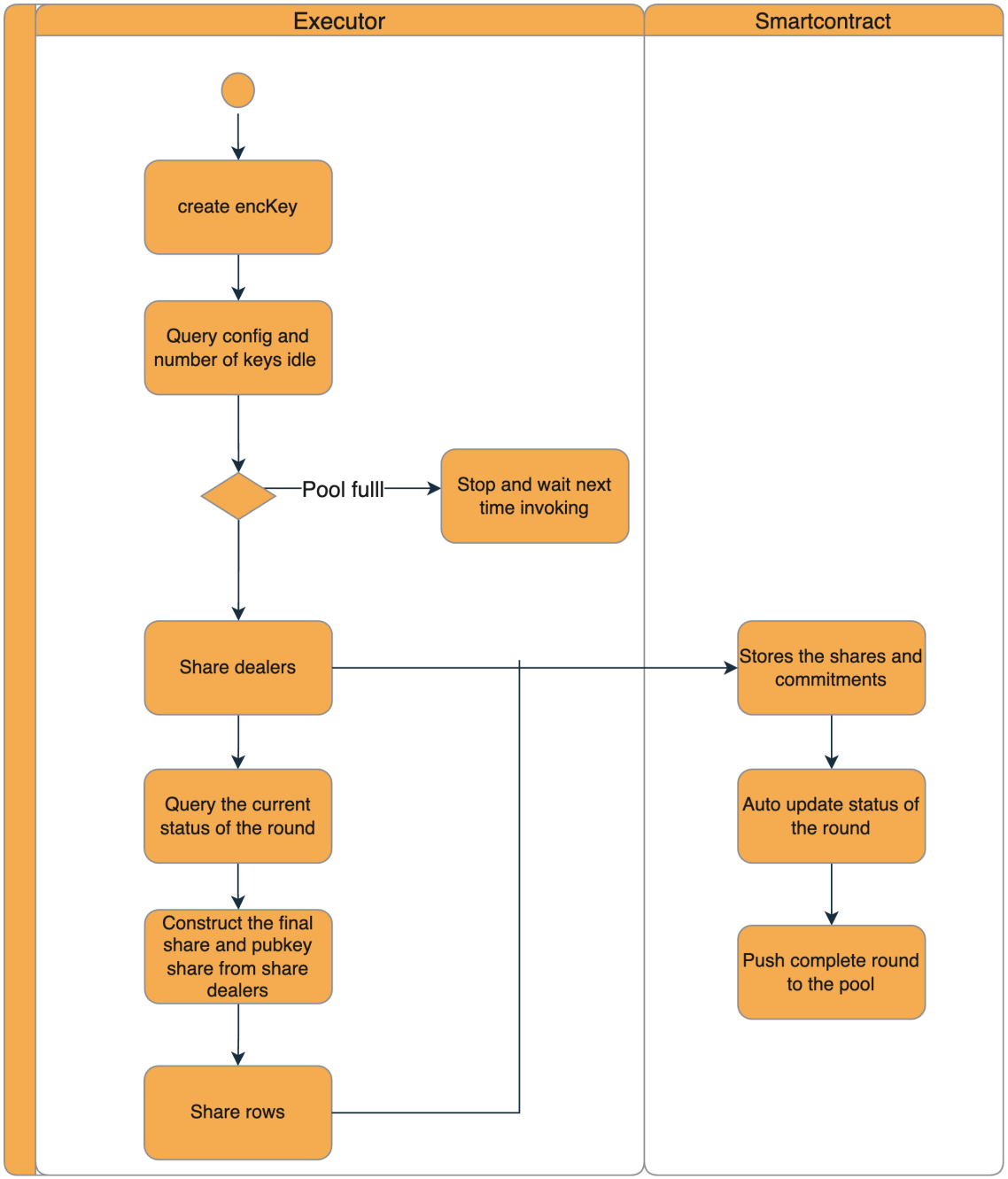

Figure 4.5: Executor contributes in the DKG protocol activity diagram

The participation of an executor in the encKey creation procedure is crucial for the successful operation of the system. The executor, as a special actor, plays an active role in contributing to the system's functioning. The process involves multiple steps and interactions between the executor and the public smart contract deployed on the blockchain.

Table 4.6 provides a comprehensive overview of the executor's involvement in the encKey creation procedure. It outlines the specific tasks and responsibilities assigned to the executor, including initiating the encKey creation round, verifying the round status, aggregating shares from other participants, generating and encrypting their own share, and finally submitting the encrypted shares to the smart contract.

Figure 4.5 complements the table by presenting an activity diagram that illustrates the sequence of actions performed by the executor during the encKey creation process. The diagram highlights the points of interaction between the executor and the public smart contract, depicting the flow of control and data between the two entities.

Usecase code	UC001	**Usecase name**	Social login
Actor	Executor		
Pre-condition	Executor is a member of DKG		

Main flow of event

No	Actor	Action
1	Executor	Assign key
2	Executor	Send idToken or accessToken to Auth0
3	Auth0	Verify token
4	Auth0	Send verify responses
5	Executor	Validate the user information
6	Executor	Send the signatures for the client for aggregate
7	Executor	Validate signatures
8	Executor	Send request assign key with signatures to smart contract
9	Smart contract	Process the request

Alternative flow of event

No	Actor	Action
6b	Executor	Reject request if invalid user information
8b	Executor	Reject request if the valid signatures is not exceed the thresholds

Table 4.7: Executor assigns key

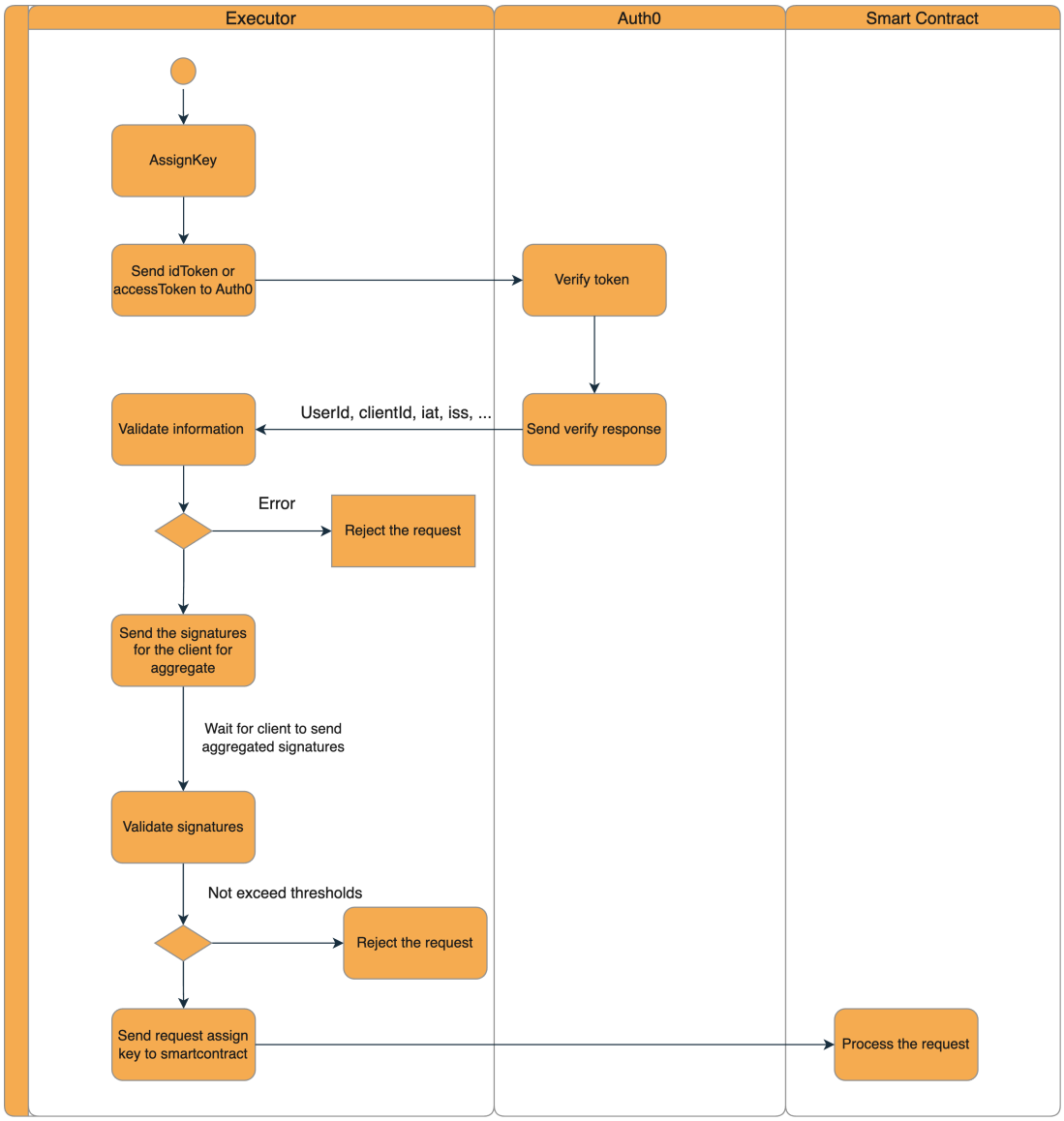

Figure 4.6: View private key and shares description activity

Figure 4.6 provides a detailed illustration of the process in which an executor assigns an encKey to a user in the social login system for Web3. This diagram showcases the interactions between three key components: Auth0, the smart contract, and the executor.

4.2 Technologies

In addition to the blockchain technology, smart contracts, and oracles described in the background section, I would like to describe other system-building tools and libraries.

4.2.1 Backend

4.2.1.1 ExpressJs

Express.js is a well-known web application framework for Node.js that facilitates the construction of scalable and robust web applications. It offers a minimalistic and adaptable approach to web development, making it simple for developers to construct server-side applications.

One of the primary benefits of Express.js is its simplicity and usability. It provides developers with a framework that is minimal and agnostic, allowing them to structure their applications according to their needs. Express.js adheres to a middleware-based architecture, enabling developers to easily incorporate middleware components to handle various application aspects, such as routing, request processing, and error handling.

Express.js [16] is widely recognized for its robust routing capabilities. It offers a concise and straightforward syntax for defining routes and managing various HTTP methods, including GET, POST, PUT, and DELETE. Express.js makes it simple to define routes for handling incoming requests and performing operations such as retrieving data from a database, processing the data, and returning the appropriate response to the client.

Within the system architecture, Express.js serves as a robust and versatile server framework that plays a vital role in handling the requests associated with the metadata of the system. Leveraging the power of Express.js, the backend of the system is equipped with the capability to receive, process, and respond to various types of requests related to the metadata. Overall, the utilization of Express.js as the server framework for handling metadata requests empowers the system to efficiently manage and process metadata-related operations. Its robust routing system, extensible middleware ecosystem, and high-performance nature make Express.js an ideal choice for building scalable and reliable backend systems that handle metadata effectively.

4.2.1.2 JSON-RPC and Jayson library

JSON-RPC[17] is a remote procedure call protocol encoded in JSON. It allows for the execution of remote procedures on a server and the exchange of structured data between the client and the server. In the system, JSON-RPC is employed to facilitate communication between different components and services, enabling the invocation of remote procedures and the transmission of data in a standardized and efficient manner. The utilization of JSON-RPC ensures interoperability and seamless integration between various system components.

The Jayson library, a JSON-RPC implementation for Node.js, is used to manage JSON-RPC communication between the client and server. Jayson is a framework for creating JSON-RPC servers and clients in Node.js that is lightweight and efficient. It simplifies the creation and management of JSON-RPC requests and responses, facilitating integration with the server and client components of the system. You can define the server-side methods that will be exposed to clients via JSON-RPC using Jayson. Jayson handles the serialization and deserialization of JSON-RPC messages. These methods can be implemented as functions or class methods. It offers a convenient API for managing the request payload, extracting the method name and parameters, and executing the appropriate server-side logic. The Jayson library integrates seamlessly with the Express.js server, allowing the server component of the system to process incoming JSON-RPC requests using the specified server-side methods. It offers an elegant and efficient solution for implementing JSON-RPC communication in Node.js, making it an excellent choice for the communication layer of

the system.

The Jayson library is incorporated in the system to process JSON-RPC requests from the client to the executor component. This decision is made because the role of the executor predominantly entails handling and processing requests unrelated to database operations. Using Jayson for JSON-RPC communication allows the executor to efficiently manage requests without requiring direct database interaction. JSON-RPC requests may include information and instructions for the executor to perform particular duties or computations, such as generating shares, decrypting data, or validating signatures. The executor processes these requests, carries out the required logic, and returns the appropriate response to the client. Implementing JSON-RPC for the executor enables a clean separation of concerns, as the executor component is exclusively focused on executing the requested tasks and does not interact directly with the databases. This design decision streamlines the executor's responsibilities and improves its capacity to respond to client requests. Using a well-established library such as Jayson also facilitates the implementation of JSON-RPC communication, resulting in a robust and dependable solution. The library manages the serialization and deserialization of JSON-RPC messages, ensuring that requests and responses are formatted correctly and are compatible between the client and executor. This simplifiees the development process and helps keep the communication protocol consistent and compatible.

4.2.2 Database

MySQL[18] is an extensively utilized open-source relational database management system (RDBMS) for storing and querying structured data. In your system, MySQL is used exclusively for storing and querying metadata in the metadata backend section. MySQL offers a dependable and effective method for storing structured data in tables, with support for a variety of data types and indexing options. It provides comprehensive transactional capabilities, ensuring data consistency and integrity during concurrent operations. MySQL's robust query language, SQL (Structured Query Language), enables efficient data retrieval and manipulation through a variety of query operations, including filtering, sorting, joining, and aggregating. The metadata backend section of the system is responsible for managing and storing metadata associated with user authentication, authorization, and other pertinent data. MySQL functions as the underlying database technology for the structured storage of this metadata. In the future, the metadata may include encrypted shares or ecrypted information of multiple system modules. By leveraging MySQL, the system can take advantage of its scalability and performance enhancements, enabling it to efficiently manage large volumes of metadata. In addition, MySQL provides security mechanisms, data backup and recovery, and replication options, all of which contribute to the dependability and accessibility of your metadata storage.

4.2.3 Frontend

The frontend of the system plays a critical role as it enables users or clients to interact with the system in a seamless and user-friendly manner. It serves as the interface through which users can access the system's features, functionalities, and data.

Tkey is a GitHub-hosted open-source library designed to provide cryptographic key management capabilities. The library seeks to simplify the handling and management of cryptographic keys, making it easier for application developers to implement robust security measures. The Tkey library provides a vast array of key generation, storage, and usage-related features and functions. It supports a variety of cryptographic algorithms, such as symmetric and asymmetric encryption, digital signatures, and hash functions. This enables developers to employ various cryptographic techniques based on their particular requirements. Tkey's emphasis on security is one of its primary features. The library employs key management best practices, such as secure key storage, protection against unauthorized access, and secure key sharing protocols. Tkey adheres to industry standards and recommendations to assure the privacy, integrity, and accessibility of cryptographic keys. In addition, Tkey provides a simple and intuitive API, easing the integration of cryptographic key management into applications. The library provides developers with comprehensive doc-

umentation and implementation examples. Based on the foundation provided by the Tkey open-source library, our system extends its functionality by implementing modules that are specifically tailored to our application's needs. These custom modules enhance Tkey's compatibility with our system's architecture by enhancing its primary features.

4.2.4 Virtualization

Docker is an open-source platform that has revolutionized the way applications are developed, shipped, and deployed. It provides developers with a powerful set of tools to automate the packaging and deployment of applications inside lightweight, portable containers. By leveraging containerization technology, Docker enables the creation of isolated environments that encapsulate an application and its dependencies. This ensures that the application runs consistently and predictably across different systems, regardless of the underlying infrastructure.

Containers are self-contained units that contain everything needed to run an application, including the code, runtime, system tools, and libraries. They offer advantages such as isolation, scalability, and reproducibility. With Docker, developers can package their applications along with all the necessary dependencies into a container image, which can then be deployed on any system that has Docker installed. This eliminates the need for complex setup processes and reduces compatibility issues between different environments.

Docker Compose is a complementary tool to Docker that simplifies the management of multi-container applications. It allows developers to define and configure multiple containers, their networks, and volumes using a simple YAML file. This makes it easier to define complex, interconnected services and orchestrate their deployment. With Docker Compose, developers can quickly spin up an entire application stack with a single command, making the development and testing process more efficient.

In the context of this thesis, Docker plays a crucial role in encapsulating the application and its dependencies, ensuring portability and consistency. By using Docker, the application can be packaged into a container image, making it highly portable and enabling easy deployment across different environments. Docker-compose is utilized during the development phase to define and manage the various services and dependencies required by the application. This allows for rapid system configuration and simplifies the process of setting up a development environment.

4.2.5 Rust, wasm and cosmwasm

Rust is a systems programming language known for its emphasis on performance, reliability, and safety. It offers a high level of control over system resources while providing memory safety guarantees through its ownership and borrowing system. Rust's combination of low-level control and high-level abstractions makes it well-suited for building efficient and secure software, including blockchain applications.

WebAssembly (Wasm) is a binary instruction format that allows for the execution of code on a variety of platforms, including web browsers. It provides a portable and efficient runtime environment for running code at near-native speeds. Rust is one of the languages that can compile to WebAssembly, allowing developers to leverage Rust's performance and safety features in web-based applications.

CosmWasm is a specific implementation of WebAssembly designed for smart contract development within the Cosmos ecosystem. It extends the capabilities of WebAssembly to support the unique requirements of blockchain-based smart contracts. With CosmWasm, developers can write smart contracts in Rust and compile them to WebAssembly, leveraging the language's safety guarantees and performance optimizations. CosmWasm provides a secure execution environment for these smart contracts, ensuring the integrity and correctness of their execution on the blockchain.

The combination of Rust, WebAssembly, and CosmWasm offers several benefits for blockchain development. Rust's memory safety features help mitigate vulnerabilities and potential exploits in smart contracts, reducing the risk of security breaches. WebAssembly enables efficient execution of the compiled Rust code across different platforms, allowing for interoperability and portability. CosmWasm, as an implementation of WebAssembly tailored for blockchain smart contracts, provides a secure and optimized environment for

executing Rust-based smart contracts within the Cosmos ecosystem.

4.3 Diagram design

4.3.1 Sequence diagram

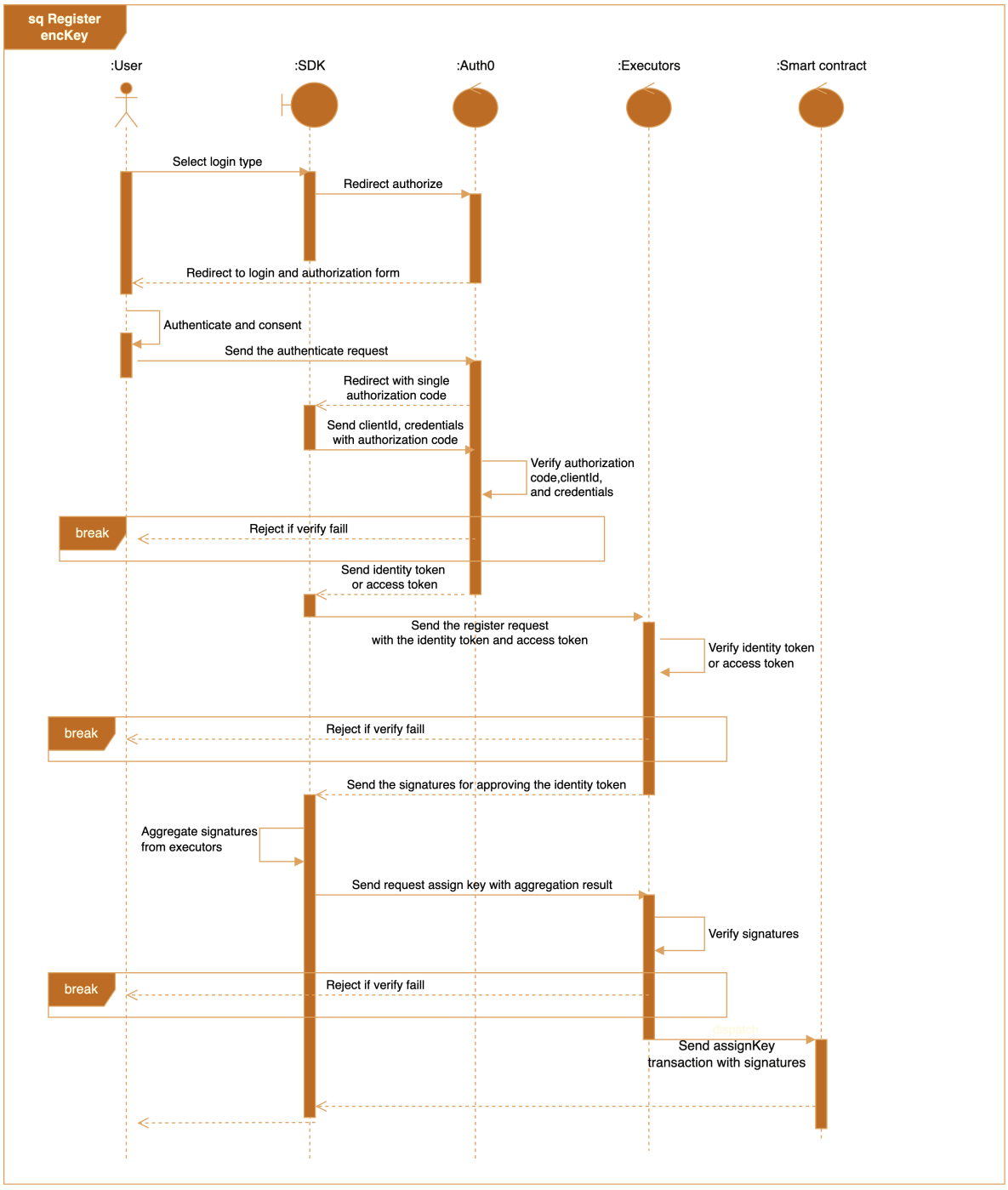

Figure 4.7: Register sequence diagram

The diagram 4.7 depicts the registration procedure for a new encKey. The user must select the social registration type and successfully complete the validation procedure.

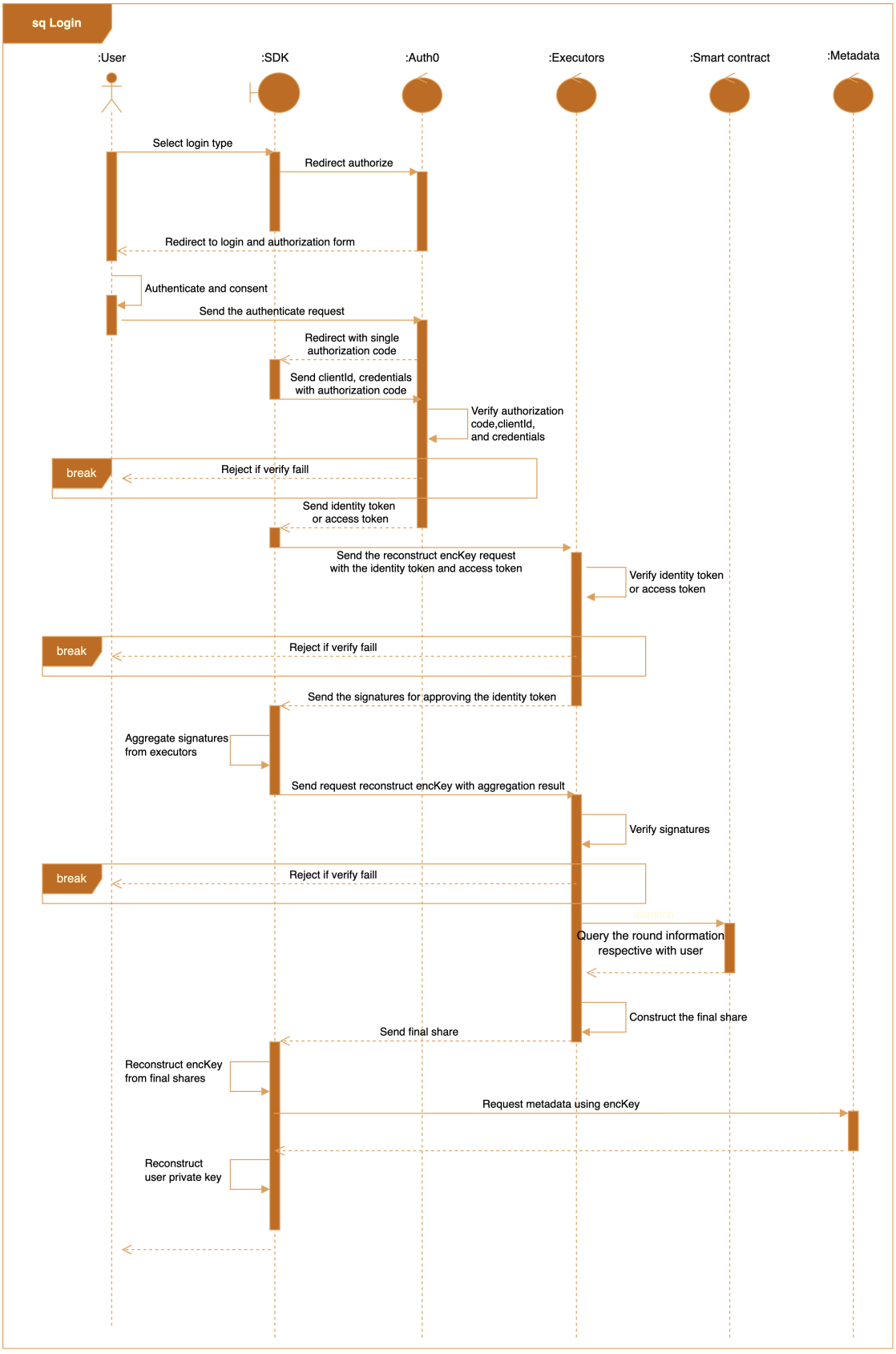

Figure 4.8: Login sequence diagram

The diagram 4.8 depicts the social authentication system's sign-in procedure. This procedure describes how the user's data pass validation and how the user reconstructs their private key.

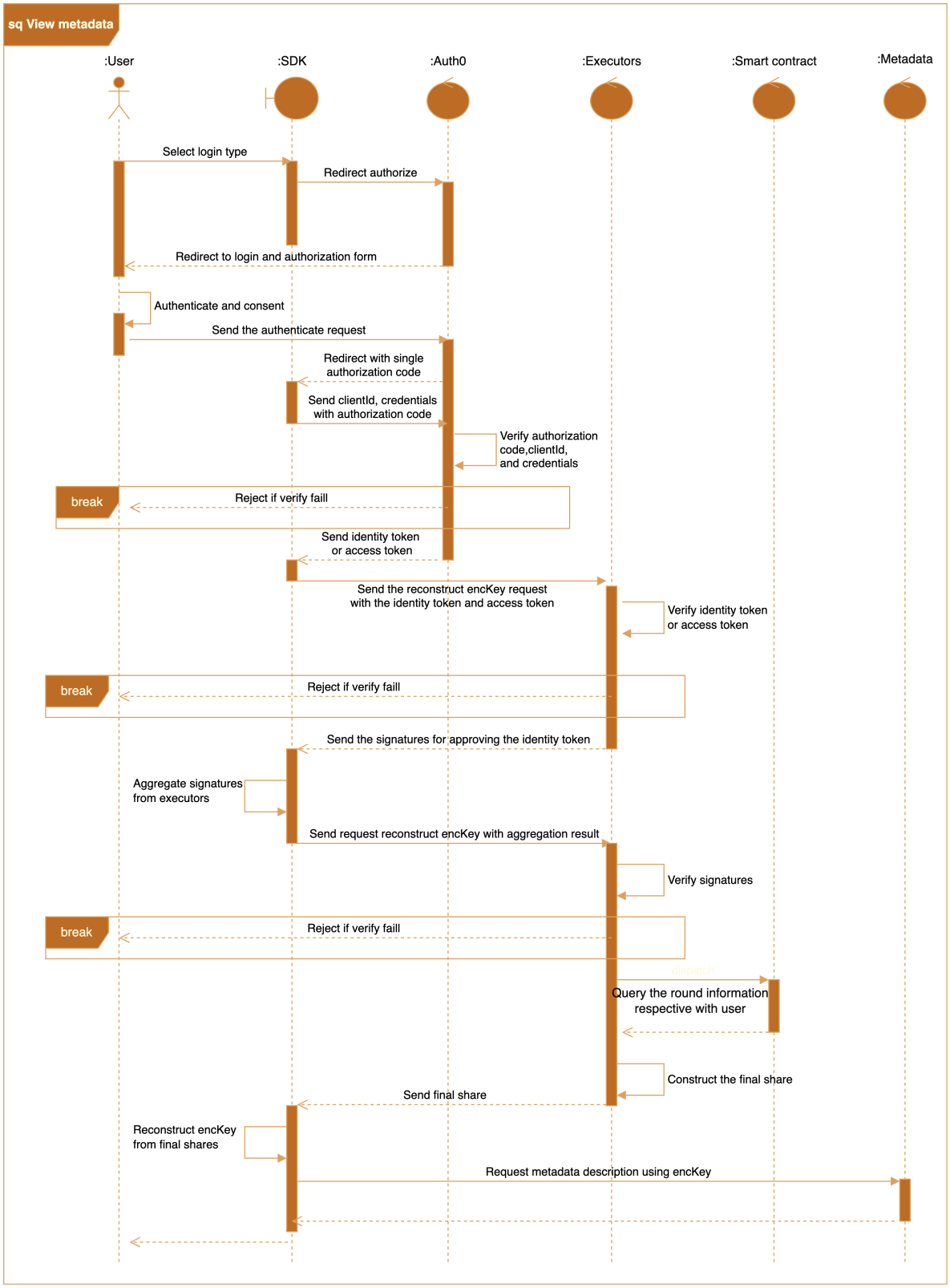

Figure 4.9: View share description sequence diagram

After logging into the system, users desire to view their share descriptions and private keys. This procedure is depicted graphically in the diagram 4.9.

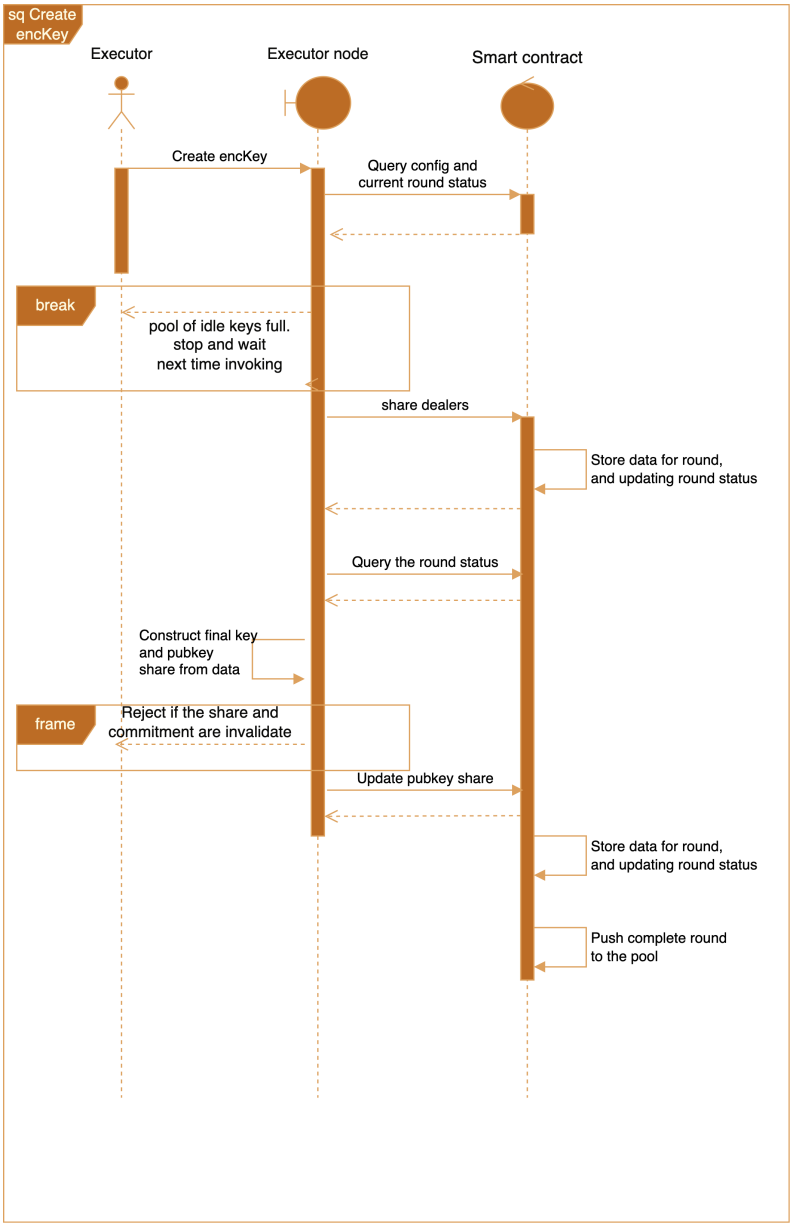

Figure 4.10: Create encKey sequence diagram

Creating a new encKey in the system is a fundamental and crucial operation that has a substantial impact on the system's overall efficacy and security. This crucial procedure is initiated when the executor, acting as a specialized actor in the system, performs the required action. To optimize the role of the executor, I meticulously designed the executor node as a cron job executor, assuring seamless management and efficient task execution. By implementing the executor node as a cron job executor, I've ensured the creation of new encKeys is optimized for efficiency and seamless administration. The cron job scheduler enables the executor to perform repetitive duties automatically and at specified intervals. This ensures that the generation of new encKeys is consistent, efficient, and always ready to satisfy the system's demands.

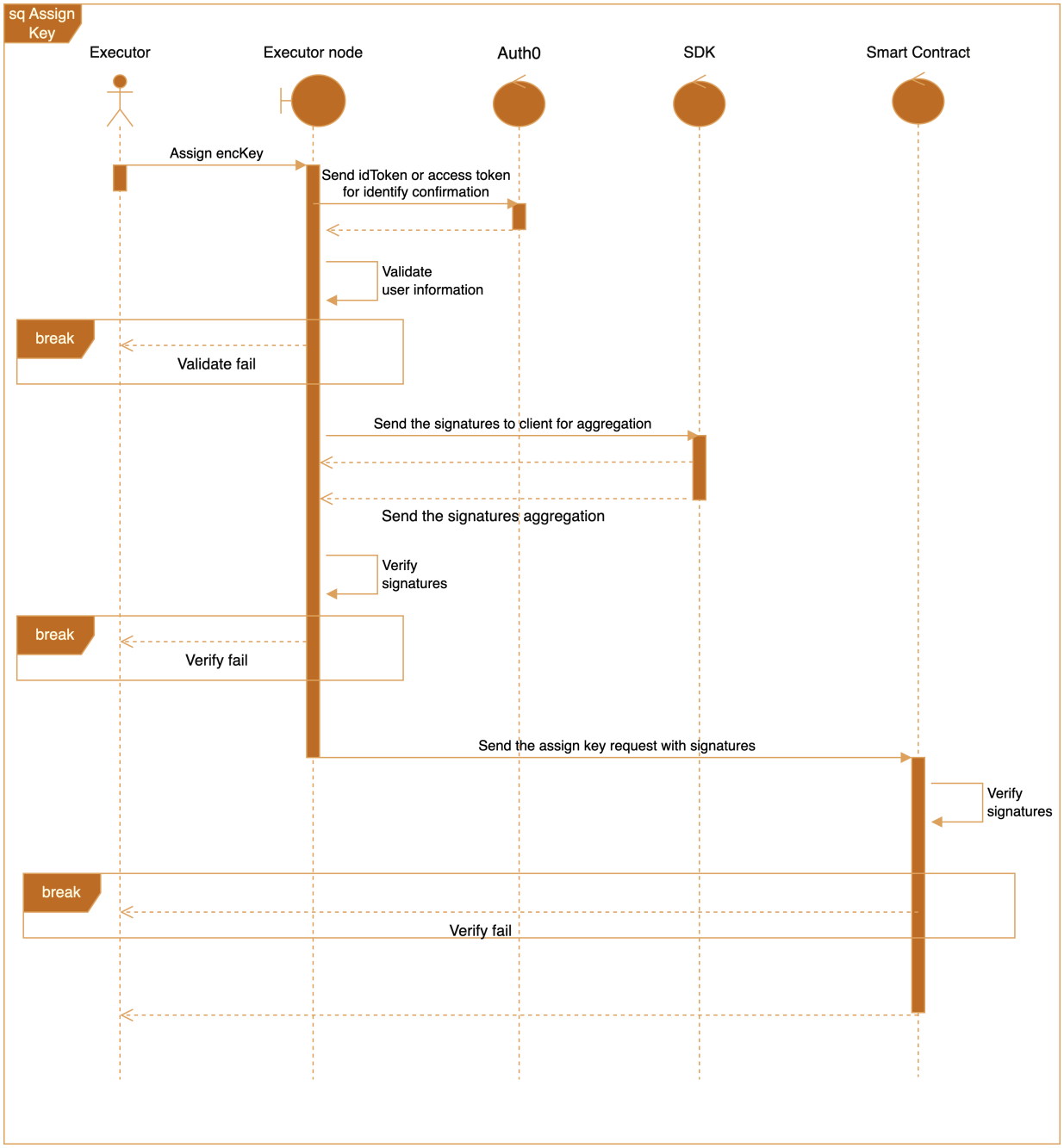

Figure 4.11: Assign encKey for user sequence diagram

The complexity and security of the executor's procedure for allocating a new encKey to a user exemplifies the robustness of the system. When a user requests their encKey, the executor receives their data, which contains crucial validation information. Through a series of exhaustive validation tests, the executor ensures that the user's data is valid, accurate, and in accordance with the system's specifications. This validation phase is a crucial line of defense against possible data inaccuracies and unauthorized access attempts. The executor verifies the signatures of other executors who participate in the assignment procedure to increase the process's security. These signature verifications are necessary to affirm the credibility and authenticity of each executor involved in the assignment process. The system ensures that only authorized and trustworthy parties contribute to the designation of the new encKey by establishing trust among all participating executors.

4.3.2 Package diagram

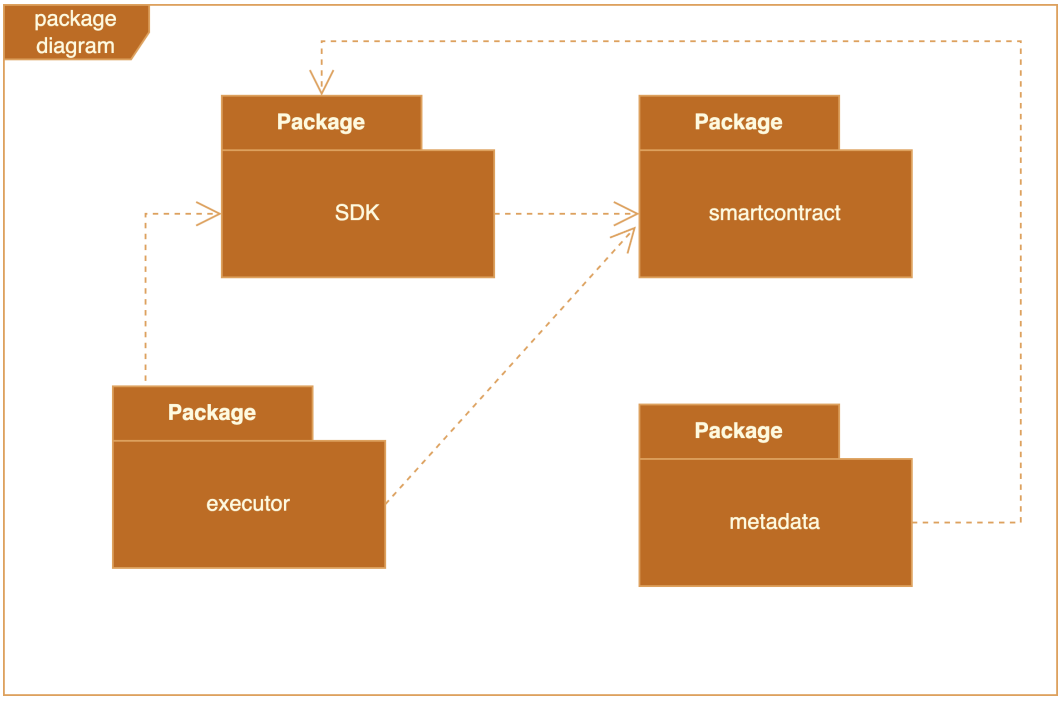

Figure 4.12: Package diagram

The system's architecture has been meticulously crafted to guarantee a secure and seamless user experience within the Web3 ecosystem. Each module is essential to the overall functionality of the system, contributing to its efficacy, security, and usability. The SDK program acts as the user interface, bridging users with the system. It enables users to effortlessly interact with the system, view their private keys and shares, and execute a variety of operations. Through the SDK, users can access and administer their cryptographic keys, which are crucial for identity verification and authorization in the Web3 environment. This program provides users with an intuitive and user-friendly interface that simplifies the complexities of Web3 technologies. By encapsulating the underlying complexities, the SDK improves accessibility, allowing a wider audience to appreciate the advantages of decentralized applications and blockchain-powered services. In the meantime, the executor package serves as a vital intermediary layer that facilitates the execution of user queries. It manages client-system interactions and processes user-initiated actions on their behalf. As a fundamental system component, the executor package is responsible for generating new encKeys, the fundamental building blocks for secure user authentication and authorization. In addition, it validates user information to ensure the validity and authenticity of performed actions. By interacting with the blockchain, the executor package can implement transactions securely in accordance with the smart contract's rules and conditions. This blockchain integration provides the system with the immutability, transparency, and trustworthiness that are inherent to blockchain technology. The metadata package functions as a dependable mechanism for data storage and querying, maintaining the essential metadata associated with user identities and encKeys. This package is essential for the secure storage of user-related data in a database, ensuring data integrity and privacy. By managing SDK package queries, the metadata package retrieves and delivers query responses efficiently. It is essential for providing seamless and secure access to user data, nurturing a high level of user confidence and trust in the system. As a secure data repository, the metadata package substantially contributes to the privacy and security of the system as a whole. The interdependencies between these products are essential to the development of a cohesive and effective system. The SDK bundle is dependent on the configuration data present in the smart contract, which provides crucial information for user interactions and authentication. To execute its actions, the executor package relies on both the data stored in the smart contract and the requests from the SDK. This harmonious cooperation enables the ex-

ecutor to execute tasks securely while adhering to the smart contract's predefined rules and conditions. The metadata package, which stores and retrieves user-related data, complements the SDK and executor products' functionality. It assures the accessibility of vital data, facilitating data access and user interactions. By carefully orchestrating the interactions between these packages, the system accomplishes a robust and user-friendly environment. The combination of blockchain technology, social authentication, and user-controlled encryption keys ushers in a new era of secure, decentralized, and user-centric Web3 applications. The interaction between these packages constitutes the system's backbone, allowing it to leverage the benefits of Web3 while mitigating the difficulties associated with technical complexities, thereby nurturing widespread adoption and utility. The seamless integration of these products empowers users, streamlines their Web3 experience, and instills confidence in the system's security and dependability.

4.3.3 Entity diagram

In the context of the centralized system described in this thesis, it is possible that the traditional Entity-Relationship (ER) diagram does not completely capture the unique characteristics of the metadata storage in the smart contract. Instead, we can depict the entities and their relationships within the system using an alternate representation. This system's smart contract serves as the central repository for metadata associated with user identities and encKeys. It operates as a decentralized database, storing and managing user-specific data on the blockchain using key-value pairings. The key-value approach is ideally suited for the metadata module because it facilitates efficient data retrieval and assures data integrity and immutability. In this context, the entities and their relationships can be described as follows:

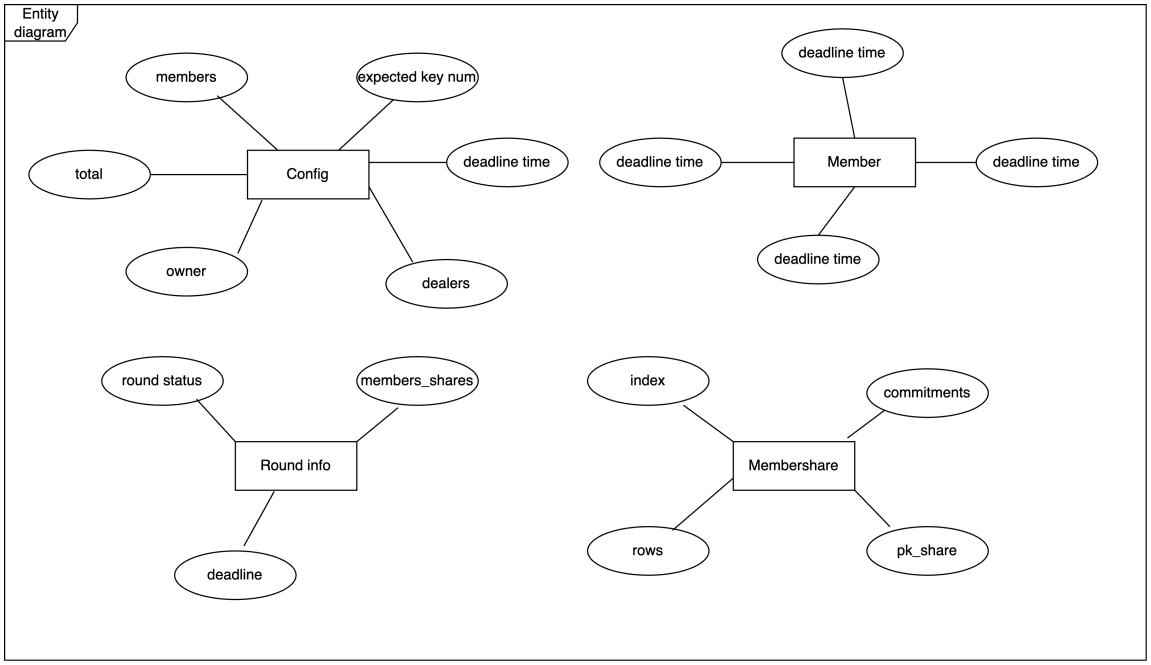

Figure 4.13: Entity diagram

All the strong entities that are stored directly as states are underlined with unique keys in the diagram. Since states are stored using key-value pairings, similar to a document-oriented database, it is unnecessary to have relationships between the entities.

The entity particulars are detailed in the tables that follow:

Attribute	Type	Required	Description
index	number	yes	The index of the member in the system plays vital role in reconstructing the encKey
pub_key	Binary	yes	The pubkey of the executor participating in the system
address	Addr	yes	The address of the executor participating in the system
end_point	string	yes	end_point is the url to executor node, which for the request in the SDK

Table 4.8: Member entity detail

Attribute	Type	Required	Description
members	Vec<Member>	yes	The member or executors operate the system
total	number	yes	Total member participate in the system
owner	Addr	yes	The address of the owner of the system
expected_key_num	number	yes	The number of expected keys are in idle pool
deadline_time	number	yes	The maxium duration of creating a key
dealers	number	yes	Thresholds for the 2 phase "Wait for dealers" and "Wait for rows"

Table 4.9: Config entity detail

Attribute	Type	Required	Description
index	number	yes	The index of executor in the system
commitments	Vec<Binary>	no	The commitments of shares of executor for the rows
rows	Vec<Binary>	no	The encrypted shares for other executors
pk_share	Binary	no	The pubkey of the final share of executor

Table 4.10: Membershare entity detail

Attribute	Type	Required	Description
round_status	number	yes	The status of the round, there are 4 status include in: WaitForDealers, WaitForRows, WaitForAssign, and Assigned
member_shares	Vec<Membershare>	no	The shares of executor for round
deadline	Vec<Binary>	no	The deadline of this round

Table 4.11: RoundInfo entity detail

4.4 Message design

4.4.1 Smart contract message

CosmWasm [19] is a smart contract platform that has been developed utilizing the Cosmos SDK[20], a blockchain framework specifically designed for the construction of DApps. CosmWasm offers a robust and optimized framework for the execution of intelligent contracts across diverse blockchain networks, ensuring both security and efficiency. The CosmWasm smart contract model is founded upon the actor model, a widely adopted paradigm utilized in the design of concurrent and distributed systems. The actor model is a computational paradigm in which each smart contract is conceptualized as an actor. An actor is an autonomous entity capable of receiving messages, performing computations on them, and transmitting messages to other actors. In the CosmWasm smart contract model, the actors are characterized by their isolation from one another, with communication occurring exclusively through the exchange of messages.

4.4.1.1 Instantiate message

The concept of instantiation messages pertains to the process of instantiating and deploying a smart contract on the blockchain.

Field	Type	Required	Description
members	Vec<Member>	yes	The information of executors participate in the system
dealers	u8	yes	Thresholds of the system for passing each phase
owner	string	yes	Owner of the smart contract
expected_key_num	uint	yes	The expected number of key in idle pool
deadline_time	uint	yes	The maxium duration of 1 round

Table 4.12: Instantiate message detail

4.4.1.2 Execute message

The execute message serves as the primary interface for processing incoming messages and executing the contract's logic in accordance with the provided input. Every execute message contained within the executor message enumeration possesses a distinct structure and data payload, thereby empowering the smart contract to effectively react to a wide array of interactions originating from users or other entities within the blockchain ecosystem.

The following tables provide a more comprehensive description of each execute message:

Field	Type	Required	Description
rows	Vec<Binary>	yes	The shares given to each participant in the DKG protocol in order to construct their final share for encKey.
commitments	Vec<Binary>	yes	The commitments are to confirm that the rows originate from the precise source

Table 4.13: ShareDealerMsg detail

Field	Type	Required	Description
pk_share	Binary	yes	pk_share is the pubkey of the final share that the executor reconstructs using rows and commitments from the share dealer.

Table 4.14: ShareRowMsg detail

Field	Type	Required	Description
members	MemberMsg	no	The member config
owner	string	no	The new owner
dealers	u8	no	The new thresholds
expected_key_num	uint	no	The new expected number of key in idle pool
deadline_time	uint	no	The new maxium duration of key creation

Table 4.15: UpdateConfigMsg detail

Field	Type	Required	Description
sigs	Vec<Binary>	yes	The signatures of the executor which approve for assigning key for user
pub_key	Vec<Binary>	yes	The pubkey of the executors that create the signature
verifier	string	yes	The name of app which is whitelisted in the system
verifier_id	string	yes	email or user_id

Table 4.16: AssignKeyMsg detail

Field	Type	Required	Description
verifier	string	yes	The name of app which is whitelisted in the system
client_id	string	yes	The client_id of the app

Table 4.17: UpdateVerifierMsg detail

4.4.1.3 Query message

QueryMsg is a specific variety of execute message within the executor message enum within the context of the CosmWasm smart contract and actor model. It is designed to manage query requests and enables the smart contract to provide read-only, non-modifiable access to its current state. When a query request is received, the smart contract processes the QueryMsg and executes the required operations to retrieve the requested information from its internal state. As a response to the inquiry, the smart contract can then construct and return the requested data.

The query messages and responses schema are desbribed in detail by the following tables:

Field	Type	Required	Description
round_id	uint	yes	The id of the round
round_status	usize	yes	The status of the round
members_shares	Vec<MemberShare>	yes	The details about the membershares of the round
deadline	uint	yes	The deadline of the round

Table 4.18: RoundInfoResponse detail

Field	Type	Required	Description
members	Vec<Member>	yes	The information of the executor participating in the system
total	uint	yes	The number of the executor
dealer	uint	yes	Thresholds of the system
owner	string	yes	Owner of the system
expected_key_num	uint	yes	The number of expected keys in idle pool

Table 4.19: ConfigResponse detail

Field	Type	Required	Description
verifier	string	yes	The app that need to query user email
start_after	string	yes	The exclude bound start of the query
limit	uint	no	Number of verifierId. Default is 10

Table 4.20: ListVerifierIdMsg detail

Field	Type	Required	Description
verifier	string	yes	The app that need to query user email
list_verifier_id	Vec<string>	yes	List of the email or user_id of the app

Table 4.21: ListVerifierIdResponse detail

Field	Type	Required	Description
msg	Binary	yes	The mesage that was signed by executor
sigs	Vec<Binary>	yes	Signatures for the message
pub_keys	Vec<Binary>	yes	The public key of executors signed the message

Table 4.22: VerifyMemberMsg detail

4.4.2 JRPC message

The executor plays a crucial role in the creation and distribution of encryption keys to users. The executor exposes four essential JSON-RPC (JRPC) methods to facilitate this functionality:

AssignKeyCommitmentRequest - The executor uses this JRPC method to request the user's commitment to a new encryption key. The user will commit to the key without disclosing its actual value. This phase verifies that the user intends to generate a valid encryption key.

AssignKeyRequest - The executor uses this JRPC method to request the actual encryption key from the user after obtaining the user's commitment. The user will provide the encrypted key in accordance with the commitment, ensuring that the key remains secret until later phases of the process.

CommitmentRequest - The executor uses this JRPC method to request the commitments made by other executors in the system. To proceed with the encryption key generation process and accomplish the desired level of security and consensus, other executors' commitments are required.

ShareRequest - Lastly, the executor uses the ShareRequest JRPC method to request shares from other executors in order to reconstruct the encryption key. As part of the distributed key generation protocol, the executor must obtain portions from multiple other executors.

The subsequent tables illustrate in depth the specifics of message and response:

Field	Type	Required	Description
jsonrpc	string	yes	The version of JRPC. Must be "2.0"
method	string	yes	The method will call in the json-rpc server
id	number	yes	An identifier established by the Client
params	Object	yes	A Structured value that holds the parameter values to be used during the invocation of the method

Table 4.23: General JPRC request detail

Field	Type	Required	Description
jsonrpc	string	yes	The version of JRPC. Must be "2.0"
result	string	no	This member is REQUIRED on success. This member MUST NOT exist if there was an error invoking the method. The value of this member is determined by the method invoked on the Server.
id	number	yes	An identifier established by the Client
error	Object	no	This member is REQUIRED on error. This member MUST NOT exist if there was no error triggered during invocation. The value for this member MUST be an Object

Table 4.24: General JPRC response detail

The subsequent tables will describe in detail the request parameters and the response value:

Field	Type	Required	Description
tokencommitment	string	yes	The hash of the idToken or access token receive from Autho0
temppubX	string	yes	The pubkey in X axis of temp key established in the reconstruct process
tempubY	string	yes	The pubkey in Y axis of temp key established in the reconstruct process

Table 4.25: CommitmentRequest parameters detail

Field	Type	Required	Description
signature	string	yes	The signature of executor sign on data
data	string	yes	The data is signed by executor
nodepubX	string	yes	The pubkey in X axis of executor node using for verify
nodepubY	string	yes	The pubkey in Y axis of executor node using for verify

Table 4.26: CommitmentRequest response value detail

Field	Type	Required	Description
verifier	string	yes	The name of the app that registered in the system
verifier_id	string	yes	The email or user_id of user
idToken	string	yes	The idtoken of the Auth0
nodesignatures	Object	yes	The result of the commitment request

Table 4.27: ShareResponseRequest parameters detail

Field	Type	Required	Description
Publickey	string	yes	The publickey of the first commitment
Share	string	yes	The share encrypted with the client's public temporary key. This portion is used to reconstruct encKey.
Metadata	Object	yes	The description of the share, which is necessary for decrypting the share

Table 4.28: ShareRespone result value detail

Field	Type	Required	Description
tokencommitment	string	yes	The hash of the idToken or access token receive from Auth0
verifier	string	yes	The app that using the system
verifier_id	string	yes	Email or useri_id respectively in the app

Table 4.29: AssignKeyCommitmentRequest parameters detail

Field	Type	Required	Description
data	Object	yes	The publickey of the first commitment
nodepubx	string	yes	The publickey in X-axis of executor
nodepuby	string	yes	The publickey in Y-axis of executor
signature	string	yes	The signature of the data in stringtify
verifierIdSignature	string	yes	The signature was signed by executor about the userid or email

Table 4.30: AssignKeyCommitmentResponse value detail

Field	Type	Required	Description
idToken	string	yes	The publickey of the first commitment
verifier	string	yes	The publickey in X-axis of executor
verifier_id	string	yes	The publickey in Y-axis of executor
nodesignatures	Object	yes	The response of the assignKeyCommitmentResponse in stringtify

Table 4.31: AssignKeyRequest parameter detail

Field	Type	Required	Description
status	bool	yes	The response of the assignKeyRequest. If the status fail, the system will retry until 5 times.

Table 4.32: AssignKeyResponse value detail

4.4.3 HTTPS request

The HTTPS request is an integral element of the system's communication between the client and the metadata server. It allows the client to perform CRUD operations on the server's database-stored metadata. The HTTPS protocol is used to ensure secure and encrypted communication between the client and the server, thereby safeguarding data and guaranteeing the privacy and integrity of transactions. When a client needs to create new metadata, it transmits an HTTPS POST request containing the required information to the server. The server processes the request, verifies the data, and adds the newly generated metadata to its database. The client sends an HTTPS GET request to the server for reading data, specifying the unique identifier of the metadata or any other criteria for retrieving specific entries. The server retrieves the requested data from the database and returns it in response to the client.

The tables below illustrate in detail about the messages and responses of the HTTPS request:

Field	Type	Required	Description
pub_key_X	string	yes	The pubkey in X-axis of encKey or the share of client
pub_key_Y	string	yes	The pubkey in Y-axis of encKey or the share of client
set_data	Object	no	The data will be store in the metadata server. If not exist, the request will be the query request
signature	string	yes	the signature of the data that will be store in the server
namespace	string	no	The namespace will store in the database. Default is "tkey"

Table 4.33: The https request detail

Field	Type	Required	Description
status	bool	yes	Status of the request to the server.
metadata	Object	no	If the request is made using the GET method, the server's response will be the metadata.

Table 4.34: The https response detail

5.1 Testing

This section will analyze the performance of the four kind of JRPC request that are essential for the login and registration use cases of the system. The requests are crucial for facilitating communication between the SDK and executors, and their performance significantly affects the overall efficiency and user experience of the system.

The four JRPC requests being analyzed are:

- **AssignKeyCommitmentRequest**-The AssignKeyCommitmentRequest is a critical step that acts as an aggregator for executor signatures, paving the way for the subsequent AssignKeyRequest
- **AssignKeyRequest**-In the AssignKeyRequest, a user initiates the process of being registered within the system. The user provides their idToken and the signatures obtained from the AssignKeyCommitmentRequest to achieve consensus across the nodes
- **CommitmentRequest**-The CommitmentRequest is a request sent by a registered user who already exists within the system. Through this request, the user seeks to obtain aggregated signatures from the nodes, facilitating the subsequent ShareRequest process
- **ShareRequest**-The ShareRequest process is initiated by users who wish to retrieve their encrypted share of the cryptographic key. This encrypted share is a crucial component that allows users to reconstruct their encKey, which is necessary for secure authentication.

To assess the performance of JRPC requests, we will conduct thorough testing and gather data on response times, request rates, and throughput. The objective is to assess the system's performance under different loads and detect any bottlenecks or areas that require enhancement. Furthermore, we will assess the influence of various factors, including network latency and system resources, on the performance of JRPC requests.

The performance analysis will offer valuable insights into the system's scalability and its capacity to handle a high volume of users and concurrent requests. Optimizing the performance of critical JRPC requests can enhance system responsiveness, reduce latency, and improve the user experience during login and registration.

To conduct tests, I utilize the autocannon[21] package, which can be executed directly in Node.js with the following hyperparameters:

- Hyperparameters for autocannon:
  - connections: 10, 100, 1000, 2000, 4000, 8000
  - durations: 10s
  - number of worker: 1
- Hardware specification:
  - Processor: Apple M1 Pro, 6 performance cores(600 - 3220 MHz) and 2 power efficiency core(600 - 2064 MHz)
  - Memory: 16GB

5.1.1 CommitmentRequest

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	0 ms	0 ms	5 ms	8 ms	0.93 ms	2.08 ms	71 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	4073	4073	6631	8359	6850.5	1134.82	4073
Bytes/Sec	5.66 MB	5.66 MB	9.22 MB	11.6 MB	9.52 MB	1.58 MB	5.66 MB

Table 5.1: 10 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	4 ms	9 ms	29 ms	36 ms	10.66 ms	8.48 ms	406 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	5391	5391	9247	10647	8964.4	1496.42	5391
Bytes/Sec	7.5 MB	7.5 MB	12.9 MB	14.8 MB	12.5 MB	2.08 MB	7.49 MB

Table 5.2: 100 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	26 ms	57 ms	134 ms	284 ms	97.63 ms	461.09 ms	9882 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	3837	3837	9607	10495	8381.9	2219.44	3836
Bytes/Sec	5.33 MB	5.33 MB	13.4 MB	14.6 MB	11.6 MB	3.09 MB	5.33 MB

Table 5.3: 1000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency (ms)	177	273	1961	2454	436.85	434.88	2551
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	361	361	1582	4531	2246.6	1550.91	361
Bytes/Sec	330 kB	330 kB	1.44 MB	4.14 MB	2.05 MB	1.42 MB	330 kB

Table 5.4: 2000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency (ms)	194	417	2923	3688	611.3	578.23	4015
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	1975	1975	3379	3779	3026.6	683.61	1975
Bytes/Sec	1.8 MB	1.8 MB	3.08 MB	3.45 MB	2.76 MB	624 kB	1.8 MB

Table 5.5: 4000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency (ms)	167	1672	5490	5668	2095.88	1205.08	5740
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	185	185	364	802	472.9	216.68	185
Bytes/Sec	169 kB	169 kB	333 kB	733 kB	432 kB	198 kB	169 kB

Table 5.6: 8000 connections performance

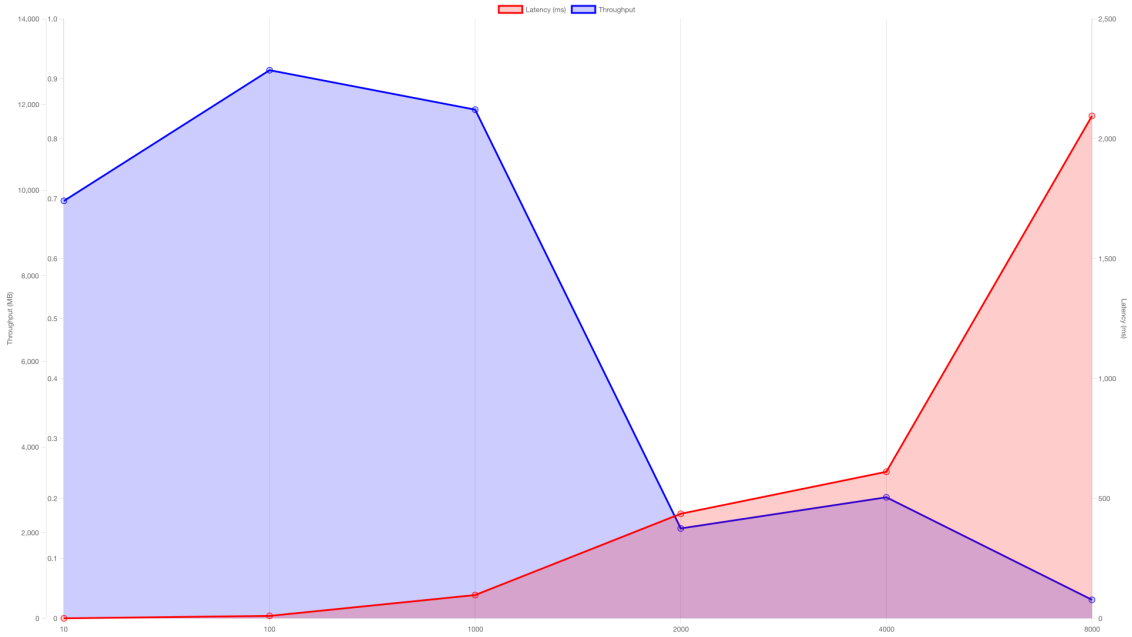

Figure 5.1: Commitment request latency and throughput line chart

The provided data provides insightful insights into the performance analysis of various connection types. The analysis includes key metrics such as latency, request per second (req/sec), and throughput, which cast light on the server's efficiency and responsiveness under different conditions. The latency, or the time it takes for a request to travel from the client to the server and return, is a crucial indicator of the responsiveness of a system. The 2.5th percentile and median latency values represent the latency encountered by the quickest 2.5% and median requests, respectively. The 97.5th and 99th percentiles, on the other hand, represent the latency for the majority of requests, with only a minor fraction experiencing higher latencies. The average latency provides a comprehensive view of the system's response time, whereas the standard deviation indicates the variation in latency values. The maximum latency represents the response time at its apex, as measured during the analysis. The request per second (req/sec) metric gauges the server's ability to process incoming requests. The values for the 1st and 2.5th percentiles represent the slowest 1% and 2.5% of observed requests, respectively.

The median value indicates the rate at which half of the requests were processed, while the 97.5th percentile represents the rate at which the majority of requests were processed. The average req/sec across all connections provides an aggregate measure of the server's performance, whereas the standard deviation illustrates the processing capacity variation of the server. Throughput, which is measured in megabytes per second (MB/s), represents the quantity of data transmitted or received per second. Similar to the latency and req/sec values, the 2.5th percentile, median, 97.5th percentile, 99th percentile, average, standard deviation, and maximum values represent various aspects of the data transmission efficacy. This data analysis helps determine the performance characteristics of the system under various connection conditions. Lower latency values indicate quicker response times and a superior user experience, with the 2.5th percentile and median latencies being especially significant for the majority of user interactions. Higher latency percentiles may indicate intermittent performance issues or congestion during peak usage periods. In the meantime, higher req/sec values indicate a server's ability to process more requests concurrently, allowing for more fluid user interactions during periods of high traffic. The standard deviation values for latency and req/sec demonstrate the performance variance. The smaller the standard deviation, the more consistent and predictable the performance. Monitoring throughput is essential for determining the efficacy of data transmission, with higher throughput values indicating superior data exchange.

5.1.2 ShareRequest

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	325 ms	465 ms	1130 ms	1289 ms	497.26 ms	177.23 ms	1307 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	10	10	19	28	19.5	5.38	10
Bytes/Sec	13 kB	13 kB	24.8 kB	36.5 kB	25.4 kB	7.01 kB	13 kB

Table 5.7: 10 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1751 ms	3888 ms	5850 ms	5937 ms	3843.44 ms	990.28 ms	6059 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	0	0	19	32	20.6	9.67	12
Bytes/Sec	0 B	0 B	24.8 kB	41.8 kB	26.9 kB	12.6 kB	15.6 kB

Table 5.8: 100 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1458 ms	5752 ms	9791 ms	9924 ms	5604.38 ms	2517.41 ms	9982 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	0	0	25	35	24.3	8.88	22
Bytes/Sec	0 B	0 B	32.6 kB	45.7 kB	31.7 kB	11.6 kB	28.7 kB

Table 5.9: 1000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	2160 ms	3892 ms	9133 ms	9575 ms	4345.31 ms	1799.95 ms	10035 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	3	3	178	247	158.81	79.46	3
Bytes/Sec	2.81 kB	2.81 kB	162 kB	225 kB	145 kB	72.3 kB	2.81 kB

Table 5.10: 2000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency (ms)	2234 ms	5475 ms	9720 ms	9995 ms	5488.23 ms	2227.17 ms	10145 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	2	2	83	177	84.6	50.73	2
Bytes/Sec	1.82 kB	1.82 kB	75.6 kB	161 kB	77 kB	46.2 kB	1.82 kB

Table 5.11: 4000 connections performance

Table 5.12: Latency, Request Rate, and Throughput (Continued)

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency (ms)	4632 ms	7469 ms	9732 ms	9867 ms	7255.9 ms	1362.87 ms	10151 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	0	0	62	234	66.7	71.33	2
Bytes/Sec	0 B	0 B	56.4 kB	215 kB	60.9 kB	65.5 kB	1.82 kB

Table 5.13: 8000 connections performance

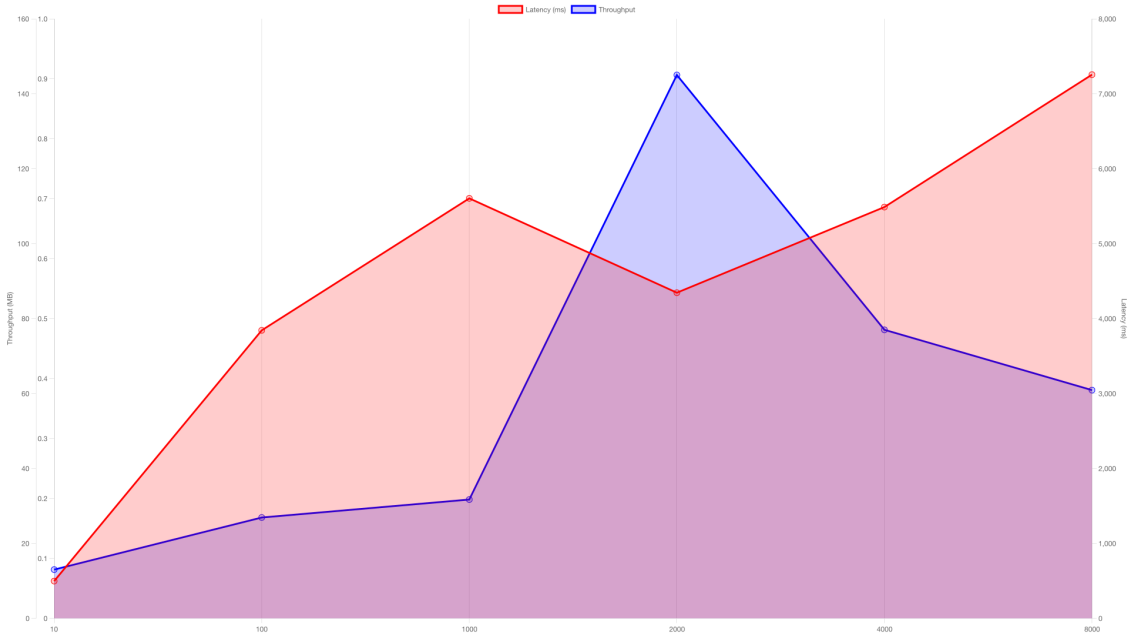

Figure 5.2: Share request latency and throughput line chart

The dataset contains performance metrics for numerous connection types, ranging from ten to eight thousand connections. These metrics provide invaluable insights into the system's performance, and they are essential for comprehending its responsiveness and capacity under varying burdens. The 2.5th percentile latency ranges from approximately 325 ms to 4632 ms, indicating that only a small percentage of queries experience extremely quick response times. The median latency at the 50th percentile ranges from 465 milliseconds to 7469 milliseconds, representing typical response times for the majority of queries. The 97.5th percentile latency, between 1130 and 9732 milliseconds, represents the response time experienced by the vast majority of requests, comprising a wider range of response times. Similarly, the 99th percentile latency ranges from 1289 ms to 9867 ms, capturing the response time for an even higher percentage of requests, including some outliers. The range of 497.26 ms to 7255.9 ms for the average latency represents the response time of the system. The standard deviation of latency ranges from 177.23 ms to 1362.87 ms, indicating the variance in response times across queries. In conclusion, the maximal latency ranges from 1307 ms to 10151 ms, representing the longest response time observed throughout the analysis. Next, the 1st and 2.5th percentile values for req/sec are typically low or even negative, indicating that only a small fraction of requests experience the slowest processing rates or may be affected by problems resulting in no response. The median req/sec values range between 19 and 62, representing the rate at which half of the queries were processed. The 97.5th percentile req/sec ranges from 28 to 234 and represents the rate at which the preponderance of requests are processed. The average req/sec ranges from 19.5 to 66.7, providing an aggregate measure of the system's ability to handle incoming requests. The range of the standard deviation of req/sec from 5.38 to 71.33 exemplifies the variability of processing rates.

The 2.5th percentile throughput values are generally low, ranging between 13 kilobytes and zero bytes, indicating slower data transmission rates for certain queries. The median throughput, which is equal to the 50th percentile, ranges from 13 kB to 0 B, representing the typical data transmission rate. The 97.5th percentile throughput ranges between 24.8 kB and 56.4 kB, representing the data transmission rate for the preponderance of requests. The 99th percentile throughput ranges between 25,4 kB and 215 kB, representing the data transmission rate for a larger proportion of requests. The average throughput ranges from 13 kilobytes to 60,9 kilobytes, representing an overall measure of the efficacy of data transmission. The throughput standard deviation ranges from 7.01 kB to 65.5 kB, illustrating the variability of data transmission rates. In conclusion, this exhaustive data analysis provides useful insights into the performance characteristics of the system under various connection types. By comprehending these metrics, system administrators can make

informed decisions, optimize performance, and ensure a smooth and effective user experience. In addition, monitoring these metrics over time can help identify performance constraints and plan for future scaling and enhancements to meet user demands effectively.

5.1.3 AssignKeyCommitmentRequest

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	285 ms	305 ms	478 ms	522 ms	323.71 ms	52.36 ms	681 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	208	208	307	339	304.7	39.09	208
Bytes/Sec	301 kB	301 kB	444 kB	491 kB	441 kB	56.6 kB	301 kB

Table 5.14: 10 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	667 ms	2259 ms	4025 ms	4232 ms	2240.57 ms	676.53 ms	4521 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	237	237	432	467	399.4	80.26	237
Bytes/Sec	343 kB	343 kB	625 kB	676 kB	578 kB	116 kB	343 kB

Table 5.15: 100 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	667 ms	2259 ms	4025 ms	4232 ms	2240.57 ms	676.53 ms	4521 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	237	237	432	467	399.4	80.26	237
Bytes/Sec	343 kB	343 kB	625 kB	676 kB	578 kB	116 kB	343 kB

Table 5.16: 1000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	919 ms	2317 ms	5220 ms	5544 ms	2803.76 ms	1186.83 ms	7750 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	160	160	403	462	375.8	91.57	160
Bytes/Sec	232 kB	232 kB	583 kB	669 kB	544 kB	132 kB	232 kB

Table 5.17: 2000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	833 ms	2289 ms	5073 ms	5429 ms	2654.79 ms	1161.33 ms	7504 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	182	182	434	460	414	78.02	182
Bytes/Sec	263 kB	263 kB	628 kB	666 kB	599 kB	113 kB	263 kB

Table 5.18: 4000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1162 ms	2633 ms	3503 ms	3567 ms	2549.27 ms	552.1 ms	5960 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	43	43	388	621	362.8	153.81	43
Bytes/Sec	62.2 kB	62.2 kB	562 kB	899 kB	525 kB	223 kB	62.2 kB

Table 5.19: 8000 connections performance

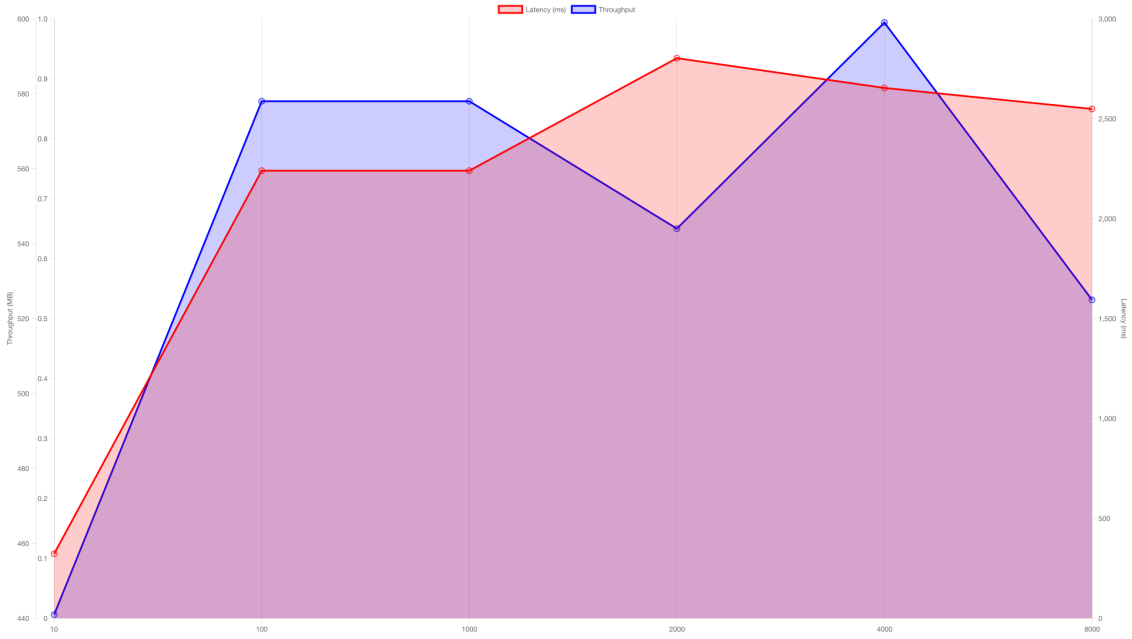

Figure 5.3: AssignKeyCommitment request latency and throughput line chart

The provided dataset comprises performance metrics for various connection types, ranging from ten to eight thousand connections. These metrics are essential for evaluating the system's efficacy and comprehending its behavior under different load conditions. The 2.5th percentile latency ranges from approximately 285 milliseconds to 1162 milliseconds, indicating that only a small percentage of requests experience extremely fast response times. The median latency (50th percentile) ranges between 305 and 2633 milliseconds, representing typical response times for the majority of queries. The 97.5th percentile latency ranges between 478 and 3503 milliseconds, representing the response times for the overwhelming majority of requests. In a similar fashion, the 99th percentile latency ranges from 522 ms to 3567 ms, encompassing response times for an even greater proportion of queries, as well as some outliers. The average latency ranges from 323.71 ms to 2549.27 ms, representing the system's response time as a whole. The range of 52.36 to 552.1 ms for the standard deviation of latency illustrates the variation in response times across queries. The maximal latency ranges between 681 and 5960 milliseconds, representing the longest response time observed throughout the analysis. The 1st and 2.5th percentile values for req/sec are generally low, ranging from 43 to 208, indicating that only a small proportion of requests are processed at the slowest rates. The median req/sec values range from 307 to 434, indicating the rate at which fifty percent of requests were processed. The 97.5th percentile req/sec ranges from 339 to 621, indicating the preponderance of requests' processing speed. The range of 304.7 to 362.8 for the average req/sec indicates the system's ability to handle incoming requests. The range of 39.09 to 153.81 for the standard deviation of req/sec demonstrates the variability of processing rates.

The 2.5th percentile throughput values are generally low, ranging from 232 kB to 62.2 kB, indicating that some queries have slower data transmission rates. The median throughput, which corresponds to the 50th percentile, ranges from 263 kilobytes to 62.2 kilobytes, representing the average data transmission rate. The 97.5th percentile throughput ranges between 583 kB and 562 kB, representing the data transfer rate for the plurality of requests. The 99th percentile throughput ranges between 669 kB and 899 kB, representing the data transmission rate for a greater proportion of requests. The range of 441 kB to 525 kB for the average throughput provides an overall measure of data transmission efficacy. The range of 56.6 kB to 223 kB for the standard deviation of throughput illustrates the variability of data transmission rates. In conclusion, this comprehensive data analysis illuminates the performance characteristics of the system across various connection types. By comprehending these metrics, system administrators are able to make informed decisions, optimize performance, and guarantee a seamless and effective user experience. In ad-

dition, continuous monitoring of these metrics enables the identification of performance constraints and the planning of future scaling and enhancements to effectively meet user demands.

5.1.4 AssignKey

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	447 ms	465 ms	2193 ms	2225 ms	547.96 ms	343.23 ms	2230 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	4	4	20	25	17.7	6.28	4
Bytes/Sec	3.44 kB	3.44 kB	17.2 kB	21.5 kB	15.2 kB	5.4 kB	3.44 kB

Table 5.20: 10 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	453 ms	500 ms	784 ms	834 ms	539.74 ms	96.42 ms	1154 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	96	96	185	219	180.8	35.31	96
Bytes/Sec	82.6 kB	82.6 kB	159 kB	188 kB	155 kB	30.4 kB	82.6 kB

Table 5.21: 100 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	837 ms	1561 ms	6822 ms	8665 ms	1842.51 ms	1257.83 ms	9971 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	68	68	277	387	265.9	91.27	68
Bytes/Sec	58.5 kB	58.5 kB	238 kB	333 kB	229 kB	78.5 kB	58.5 kB

Table 5.22: 1000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency (ms)	1644 ms	3205 ms	8045 ms	8391 ms	3589.12 ms	1408.48 ms	9823 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	4	4	360	536	347.1	142.23	4
Bytes/Sec	3.64 kB	3.64 kB	328 kB	488 kB	316 kB	129 kB	3.64 kB

Table 5.23: 2000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency (ms)	2168 ms	3930 ms	8805 ms	9295 ms	4506.56 ms	1746.61 ms	10174 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	0	0	392	628	360.8	199.87	53
Bytes/Sec	0 B	0 B	357 kB	572 kB	328 kB	182 kB	48.2 kB

Table 5.24: 4000 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency (ms)	2652 ms	5541 ms	10021 ms	10150 ms	5999.93 ms	2079.53 ms	10862 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	0	0	304	664	297.4	198.39	4
Bytes/Sec	0 B	0 B	277 kB	605 kB	271 kB	181 kB	3.64 kB

Table 5.25: 8000 connections performance

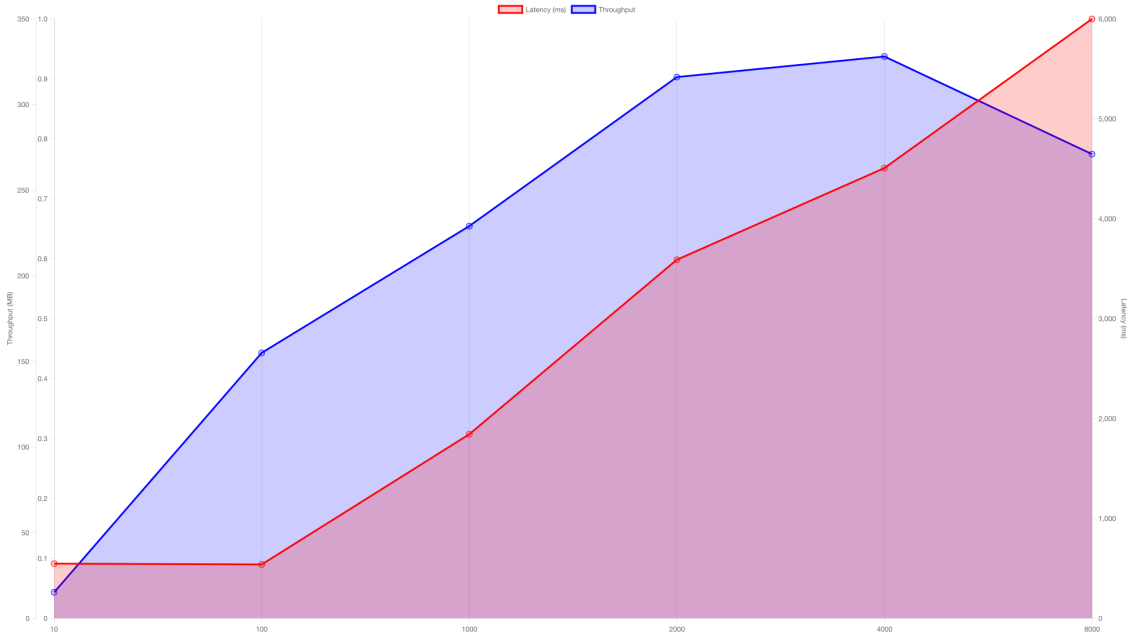

Figure 5.4: AssignKey request latency and throughput line chart

The median latency for 1000 connections is 1561 milliseconds, whereas the 2.5th percentile latency is approximately 837 milliseconds. The 97.5th and 99th percentile latencies are 6822 ms and 8665 ms, indicating response times for the overwhelming majority of requests. The mean latency is 1842.51 ms and the standard deviation is 1257.83 ms. This instance reveals a maximal latency of 9971 milliseconds.For 2000 connections, the 2.5th percentile latency is approximately 1644 milliseconds, while the 50th percentile latency is approximately 3205 milliseconds. 97.5 percentile and 99 percentile latencies are 8045 and 8391 milliseconds, respectively. Average latency for 2000 connections is 3589.12 ms, with a standard deviation of 1408.48 ms. The longest recorded latency is 9823 milliseconds. The 2.5th percentile latency for 4000 connections is approximately 2168 ms, whereas the 50th percentile latency is approximately 3930 ms. The 97.5th percentile and 99th percentile latencies are 8805 and 9295 milliseconds, respectively. The average latency for 4000 connections is 4506.56 ms, with a standard deviation of 1746.61 ms. 10174 milliseconds is the greatest observed latency in this scenario.

The median throughput for 1000 connections is 58.5 kB, with the 2.5th percentile throughput being approximately 58.5 kB. The throughputs for the 97.5th and 99th percentiles are 238 kB and 333 kB, respectively. The average throughput for 1000 connections is 229 kilobytes, with a standard deviation of 78.5 kilobytes. The highest throughput measured is 58.5 kB. For 2000 connections, the 2.5th percentile throughput is approximately 3.64 kilobytes, while the 50th percentile throughput is also 3.64 kilobytes. The throughputs for the 97.5th and 99th percentiles are 328 kB and 488 kB, respectively. The standard deviation for 2000 connections is 129 kB, while the average throughput is 316 kB. 3.64 kB is the maximum throughput observed. The median and 2.5th percentile throughputs for 4000 connections are both 0 bytes, signifying extremely low or no throughput. The throughputs for the 97.5th and 99th percentiles are 357 kB and 572 kB, respectively. The standard deviation of the average throughput for 4000 connections is 182 kB. Maximum throughput measured in this instance is 48.2 kilobytes.

5.2 Discussion

Web3Auth is a decentralized oracle network that exhibits similarities with the thesis project. The system also incorporates Shamir's Secret Sharing and DKG (Distributed Key Generation) protocol. Web3Auth provides various key features, as follows:

- **Seamless onboarding**-Web3Auth uses social login to allow users to sign up for dapps with just a few

clicks. This makes it easy for users to get started with dapps, and it also helps to improve the user experience.

- **Multi-party computation (MPC)**-Web3Auth uses MPC to provide a secure and private way for users to share their keys. This allows users to collaborate on dapps without having to reveal their private keys. MPC is a cryptographic protocol that allows multiple parties to jointly compute a function without revealing their individual inputs. This makes it a very secure way for users to share their keys, as only the final result of the computation is revealed.

Web3Auth has the following disadvantages:

- The implementation of nodes for constructing keys of Web3Authen is complex due to the presence of multiple layers within its structure.
- The persistence of user data is not guaranteed due to the storage of metadata in a database. If the organization fails to manage this properly, it can result in the loss of information, leading to significant damages.
- Web3Auth is not yet widely adopted by dapps, so users may not be able to use their Web3Auth keys on all of the dapps that they want to use. This is because the platform is still relatively new, and it is not yet as widely integrated with dapps as other platforms, such as MetaMask.
- To successfully execute projects, various infrastructure components must be operational, with particular emphasis on the node's involvement in registering and creating encryption keys. Scaling out can result in significant infrastructure costs.

The system advantages:

- By utilizing smart contracts and blockchain technology, user data can be stored and secured in a decentralized manner, guaranteeing its durability and security as long as the blockchain network remains operational.
- By transferring the responsibility of constructing encKey and assignKey to the smart contract, the need for implementing intricate nodes to handle consensus between nodes, such as Web3Auth, is eliminated.
- The system's complexity decreased, resulting in significant reductions in infrastructure and debugging costs.

CHAPTER 6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

In conclusion, the system being discussed is an innovative and efficient solution for addressing identity and authorization challenges in the Web3 domain. This system has the potential to revolutionize online authentication and access by incorporating advanced features and functionalities. This solution utilizes advanced technologies such as blockchain, social login, and user-controlled encryption keys. It also incorporates robust security measures like smart contracts, Shamir's secret sharing, and the Pedersen DKG protocol. These features provide a seamless and secure user experience, enabling individuals to confidently engage with the Web3 ecosystem.

The system's user-friendly interface facilitates convenient navigation through the authentication and authorization processes, making it a notable strength. The incorporation of social login enables a smooth and recognizable authentication process, minimizing obstacles and improving user acceptance. In addition, user-controlled encryption keys offer individuals increased control and ownership of their private data, effectively addressing the escalating concerns surrounding data privacy and security.

Moreover, the system's capacity to streamline and automate the authorization process greatly improves efficiency and convenience. Virtualization technology enables the establishment of a uniform and stable deployment environment, facilitating the configuration and scalability of applications as required. The utilization of Express.js as the backend and JSON-RPC for communication enhances the system's robustness and performance, facilitating seamless and dependable interactions among various components.

The system's capacity to tackle the urgent issues of identity and authorization in the Web3 domain renders it a valuable asset for individuals, developers, and businesses. Decentralized applications are being increasingly adopted and utilized across various industries, leading to innovation and growth within the Web3 ecosystem. The Web3 landscape is constantly changing, and the system is prepared to adapt and provide secure and efficient solutions for identity and authorization in the decentralized digital world.

6.2 Future work

The system's beta version shows promise by offering support for Google Auth0 and web browsers. The development has promising prospects for the system, as there are plans to enhance its compatibility and extend its reach. The roadmap entails incorporating support for additional reputable authentication providers, including Facebook, Apple, Reddit, and others. The system seeks to expand its user base and provide users with a greater variety of authentication options by integrating with well-known authentication services. This integration allows for seamless identity verification.

Additionally, there are plans to improve the accessibility of the system by expanding its compatibility to various platforms, such as mobile devices and iPads. This expansion will enhance user interaction by accommodating various devices, thereby fostering inclusivity and improving user experience. The adoption of mobile platforms is essential for accommodating the growing number of users accessing the Web3 ecosystem via smartphones and tablets.

In the context of cross-device support, the system currently does not offer the capability to share the private key across multiple devices. However, this feature has been identified as a critical target for future development and enhancement of the social login device. The ability to seamlessly and securely share the private key across different devices is crucial to providing users with a consistent and flexible experience.

REFERENCE

[1] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979.

[2] M. Abadi *et al.*, "Distributed key generation in the wild: A practice-oriented study," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2016, pp. 257–269.

[3] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf (visited on 06/19/2023).

[4] V. Buterin and G. Wood, *Ethereum: A next-generation smart contract and decentralized application platform*, 2014. [Online]. Available: https://ethereum.org/whitepaper/ (visited on 06/19/2023).

[5] A. M. Antonopoulos, *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, 2014.

[6] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology - CRYPTO'87*, Springer, 1987, pp. 369–378. DOI: 10.1007/3-540-48184-2_32.

[7] Bitcoin BIP-0039 Contributors, *Bip39: Mnemonic code for generating deterministic keys*, Bitcoin Improvement Proposal, Accessed on 2023-06-19, 2013. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

[8] MetaMask, *Metamask: Ethereum wallet and gateway to blockchain apps*, Website, 2023. [Online]. Available: https://metamask.io/ (visited on 06/19/2023).

[9] R. L. Burden and J. D. Faires, *Numerical Analysis*, 10th. Boston, MA: Cengage Learning, 2015, ISBN: 978-1-305-27108-9.

[10] T. P. Pedersen, "A threshold cryptosystem without a trusted party," in *Advances in Cryptology — EUROCRYPT'91*, Springer, 1991, pp. 522–526.

[11] N. Szabo, "Formalizing and securing relationships on public networks," in *First Monday*, vol. 2, University of Illinois at Chicago Library, 1997. DOI: 10.5210/fm.v2i9.548. [Online]. Available: http://firstmonday.org/ojs/index.php/fm/article/view/548/469.

[12] C. van der Veen, *DeFi and the Future of Finance: A Comprehensive Guide to Decentralized Finance*. Packt Publishing, 2021, ISBN: 978-1801073687.

[13] M. Swan, *Token Economy: How the Web3 Reinvents Value Exchange*. 2018.

[14] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," 2014.

[15] Internet Engineering Task Force, *Oauth 2.0*, https://tools.ietf.org/html/rfc6749, Accessed: June 19, 2023, 2012.

[16] *Express.js*, https://expressjs.com/, Accessed: 2023-06-19.

[17] *JSON-RPC*, https://www.jsonrpc.org/, Accessed: 2023-06-19.

[18] *MySQL*, https://www.mysql.com/, Accessed: 2023-06-19.

[19] C. GmbH, *Cosmwasm: A secure and efficient smart contract platform for cosmos*, https://www.cosmwasm.com/, Accessed: 2023-06-19, 2021.

[20] C. Developers, *Cosmos sdk: A modular framework for building blockchain applications*, https://cosmos.network/docs/tutorial/, Accessed: 2023-06-19, 2021.

[21] D. Nock, *Autocannon*, https://github.com/mcollina/autocannon, version 7.11.0, Software package, GitHub.