Access control and permission management on the DELA blockchain

Master Research Project in Cybersecurity

by Maxime Zammit

June 07, 2024

Supervisors: Prof. B. Ford (DEDIS), P. Borsò-Tan (DEDIS), C. Dengler (C4DT)

Decentralized Distributed Systems Laboratory & Center for Digital Trust, EPFL

Contents

Chapter 1

Introduction

1.1 Introduction

1.1.1 Overview of E-Voting

Voting is part of Switzerland culture. Every swiss citizen vote roughly four times a year on various subjects. To achieve meaningful electoral results a diverse sampling of citizen is required. Therefore, it is crucial to minimize barriers to participation. One such method is e-voting, as many Swiss citizens live abroad [7]. However, e-voting is not without challenges and risks. The École Polytechnique Fédérale de Lausanne (EPFL) has been developing its own e-voting software as an internal tool. Since 2017, the DEDIS laboratory at EPFL has been working on creating a secure and reliable e-voting platform.

Since its introduction to the campus, EPFL has used e-voting for its internal elections. Despite the controversial nature of e-voting in Switzerland, EPFL has taken the lead by implementing it in real-world scenarios, albeit with lower attack risks since these elections are conducted within the institution. Nevertheless, this provides an ideal, controlled environment to test and refine these new technologies [4].

The 2018 e-voting system was using the older blockchain, known as Cothority [3], developed by DEDIS. Since then, DEDIS has created a more advanced blockchain called DELA (DEDIS Ledger Architecture). Consequently, in 2021, it was a natural progression to enhance the existing system by rewriting it using this new technology. This initiative led to the birth of the D-Voting project.

1.1.2 Decentralized System

The D-Voting project is built following a decentralized system architecture. To ensure a common understanding, let's define a decentralized system.

Software applications can be categorized as centralized, decentralized, or distributed. According to Siraj Raval, “Centralized systems directly control the operation of the individual units and flow of information from a single center” [8, p. 17]. In simpler terms, this means there is a single point of control or authority. Most of today’s software is built following this centralized paradigm.

Distributed systems, on the other hand, spread computation across multiple nodes instead of relying on just one [8, p. 18]. However, distribution does not inherently dictates the authority structure of the software. Both centralized and decentralized systems can be distributed.

Decentralized systems are defined by the absence of a single point of control. As Raval states, “Decentralized means no node is instructing any other node as to what to do” [8, p. 18]. This implies that no single node has overarching authority. The nodes in a decentralized system may be located in different places and controlled by different individuals with varying interests. (Fig. 1.1)

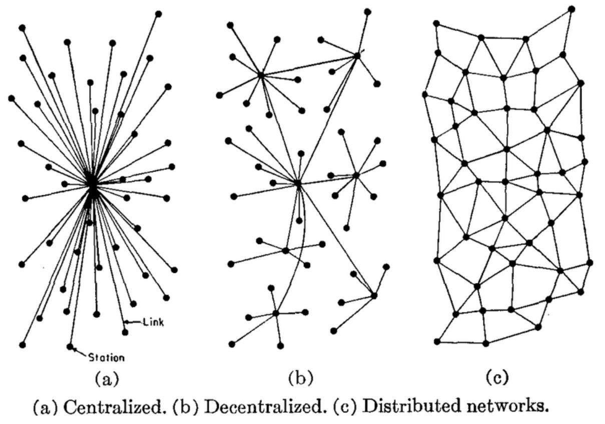

Link
Station
(a)
(b)
(c)
(a) Centralized. (b) Decentralized. (c) Distributed networks.

Figure 1.1: The different software architecture [6]

1.1.3 Single Point of Failure

One of the primary goals of using a decentralized system is to avoid single points of failure, which are defined as “non-redundant parts of a system that, if dysfunctional, would cause the entire system to fail” [5]. As stated in [9], blockchain technology helps to “eliminate any single point of failure.” Avoiding these vulnerabilities is crucial in an e-voting system, as they can be exploited to manipulate the election.

For instance, if we know when the election opens, we could cast our vote first and then perform a denial-of-service attack on one of these failure points during the remaining election period, thereby undermining the democratic process.

1.2 Goals of the Project

At the start of the semester, the D-Voting project had a major single point of failure: the permission management system. Currently, permissions are stored and managed by a backend, an additional layer between the frontend and the blockchain. This backend is a single server that signs all transactions sent to the blockchain. Consequently, all users, whether they use the web frontend or a potential alternative interface to access the blockchain, must go through this backend. Thus, this server represents a single point of failure.

To eliminate this vulnerability, we need to modify the permission management system, which can be divided into two subcategories: authentication and authorization. Authentication verifies user identity, while authorization determines whether a given user can perform a specific action.

The primary focus of our project will be on redesigning the authorization system. Although the authentication component must remain unchanged for compatibility with EPFL's Tequila authentication system and will continue to be a single point of failure, the goal is to transition the authorization system from an external database to a blockchain implementation, thereby reducing its criticality.

1.3 Corresponding Background

1.3.1 DELA

As previously mentioned, the D-Voting project is a decentralized system implemented using the DELA blockchain, which stands for DEDIS Ledger Architecture. According to the DELA project documentation [2], the primary goal of a distributed ledger is to enable a group of nodes to agree on a state without any central authority. Unlike distributed databases, which also address this problem, distributed ledgers operate on the principle that the nodes do not trust each other. One of the most famous applications of a distributed ledger is Bitcoin, which uses its ledger to store all transactions. DELA is built around six main modules, namely: Access, Execution, Ordering, Store, Transaction (Txn), Validation.

1.3.2 D-Voting before starting the project

The D-Voting project has been iteratively implemented by EPFL students under the supervision of DEDIS members and is now also co-developed by the C4DT. D-Voting replaced the old e-voting system in June 2024.

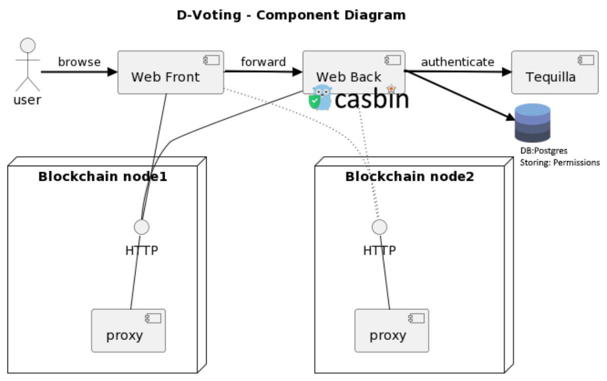


Figure 1.2: D-Voting Architecure

The project comprises four major components (Fig. 1.2):

- **Web Frontend**: A web application built with React, allowing users to cast votes and perform administrative tasks such as managing elections.
- **Web Backend**: Handles authentication and authorization, acting as a middleware that verifies requests and signs messages before forwarding them to the blockchain node, which trusts the web backend.
- **Proxy**: Creates HTTP endpoints on the blockchain, enabling direct communication between the web frontend, the backend, and the blockchain.
- **Blockchain Node**: A program running on a host that participates in the voting logic. Built on top of DELA, it includes an additional D-Voting smart contract, proxy, and two services: DKG and verifiable shuffling.

The system has been designed with the following properties in mind:

- **No single point of failure**: Ensuring that the system remains functional and uncompromised when a component is unavailable or attacked.
- **Privacy**: Ensuring that votes remain anonymous and cannot be traced back to individual voters.
- **Transparency/Verifiability/Auditability**: Ensuring that the system functions correctly, can be proven to do so, and allows for the detection of any malfunctioning parts.

As mentioned previously, permissions are currently managed by the backend which uses the Casbin library [1]. In the current state of the project, users are categorized as admins or visitors. Admins are authorized to create new voting forms and, in doing so, become the owners of those forms. Being the owner allows them to manage the form by performing various actions such as adding voters and managing its status.

Chapter 2

Design and Implementation details

2.1 Design

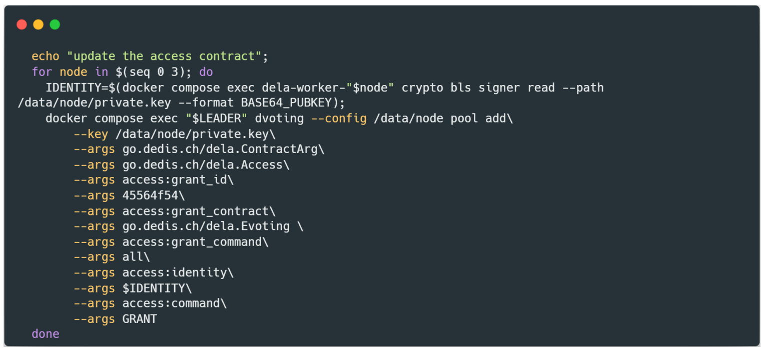
Transitioning the authorization process from the backend to the blockchain required a detailed and extensive design process. This involved making complex architectural decisions as we had to change the least possible in the way the existing API endpoint works, but at the same time transferring a process using a database to a decentralized blockchain and also accommodating the restrictions imposed by switching the programming languages.

During the initial weeks of the project, we familiarized ourselves with the codebase and explore into the DELA documentation. A key focus was understanding DARC (Distributed Access Rights Controls), a component of DELA. DARC's design, however, posed challenges for our intended use. DARC permissions are tightly coupled to specific contracts, making it difficult to represent admin permissions that need to apply across the entire system.

Additionally, DARC is currently used to verify the identities of different proxies within the blockchain. Changing this setup would mean losing a vital feature that ensures secure interactions between proxies and the blockchain (Fig. 2.1).

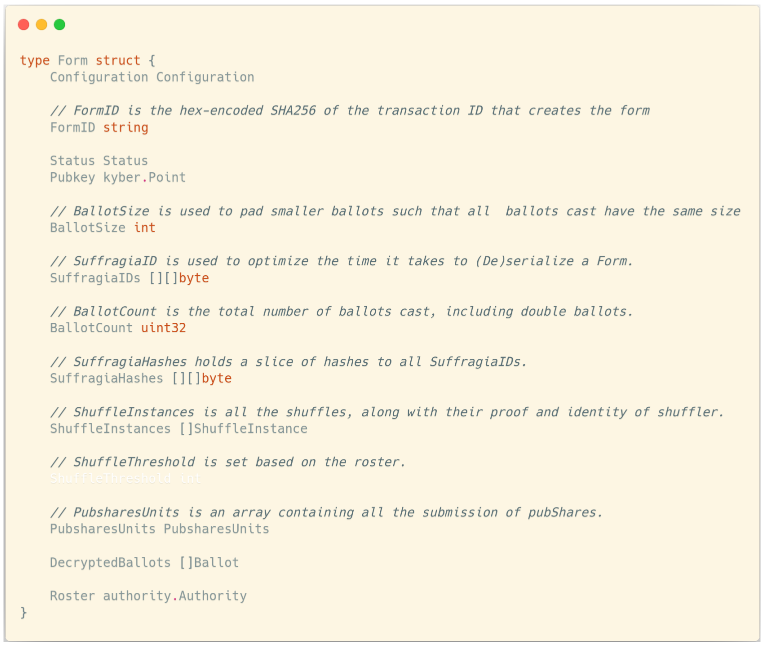
Given these challenges, we revised our initial approach. Instead of relying on DARC for managing system-wide permissions, we opted to develop a new form to handle adminship. This new form would be integrated into the system to manage administrative rights effectively. Furthermore, we modified the existing voting forms to be aware of their owners and voters, ensuring seamless management of permissions. (Fig. 2.2)

This new approach not only preserves the essential identity verification feature of DARC but also provides a more flexible and scalable solution for managing admin rights across the entire D-Voting system. By creating a dedicated admin management form and enhancing the voting forms' awareness of their administrative contexts, we have laid the groundwork for a more robust and adaptable authorization process.



```
  echo "update the access contract";
  for node in $(seq 0 3); do
    IDENTITY=$(docker compose exec dela-worker-"$node" crypto bls signer read --path
/data/node/private.key --format BASE64_PUBKEY);
    docker compose exec "$LEADER" dvoting --config /data/node pool add\
        --key /data/node/private.key\
        --args go.dedis.ch/dela.ContractArg\
        --args go.dedis.ch/dela.Access\
        --args access:grant_id\
        --args 45564f54\
        --args access:grant_contract\
        --args go.dedis.ch/dela.Evoting \
        --args access:grant_command\
        --args all\
        --args access:identity\
        --args $IDENTITY\
        --args access:command\
        --args GRANT
  done
```

Figure 2.1: Initialization of DARC from the *run_docker.sh* (docker based version of D-Voting) to accept transactions provided by the proxy

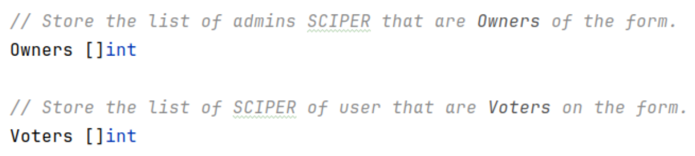


```
type Form struct {
    Configuration Configuration

    // FormID is the hex-encoded SHA256 of the transaction ID that creates the form
    FormID string

    Status Status
    Pubkey kyber.Point

    // BallotSize is used to pad smaller ballots such that all  ballots cast have the same size
    BallotSize int

    // SuffragiaID is used to optimize the time it takes to (De)serialize a Form.
    SuffragiaIDs [][]byte

    // BallotCount is the total number of ballots cast, including double ballots.
    BallotCount uint32

    // SuffragiaHashes holds a slice of hashes to all SuffragiaIDs.
    SuffragiaHashes [][]byte

    // ShuffleInstances is all the shuffles, along with their proof and identity of shuffler.
    ShuffleInstances []ShuffleInstance

    // ShuffleThreshold is set based on the roster.
    ShuffleThreshold int

    // PubsharesUnits is an array containing all the submission of pubShares.
    PubsharesUnits PubsharesUnits

    DecryptedBallots []Ballot

    Roster authority.Authority
}
```

Figure 2.2: Voting form data structure before the project

To store the list of owners and voters in the voting form, we had to make a minor modification to the form architecture. We added two fields to the form, each storing an array of SCIPER numbers. (Fig. 2.3)

In the system, currently in production, it is assumed that when a user creates a form, they will automatically become the owner of that form. We decided to preserve the status quo and to keep the same logic. An owner



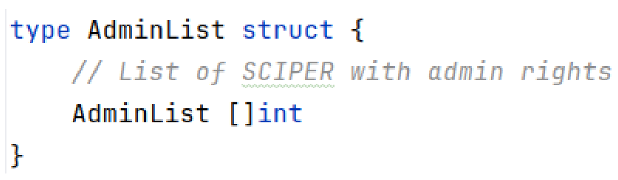
```go
// Store the list of admins SCIPER that are Owners of the form.
Owners []int

// Store the list of SCIPER of user that are Voters on the form.
Voters []int
```

Figure 2.3: Added fields to the voting form

has control over the voting process, including opening and closing the voting period and managing the list of voters. This automatic assignment ensures that every form has at least one responsible individual who can oversee its administration.

Additionally, we designed the system to ensure that a form cannot exist without an owner. This means that the last remaining owner of a form cannot be removed until they designate a new owner. This rule prevents scenarios where a form is left without any administrative oversight, thereby maintaining the integrity and manageability of the voting process. We also added verification of the user ID range that was missing in the previous version, contributing to the overall efficiency and reliability of the system.

To conclude the design section, let's discuss the design of the adminship form, which we have named AdminList. This form contains a single field: representing a list of SCIPER numbers. We chose this design to maintain consistency and simplicity in handling user IDs. Being an admin allow the creation of voting form as well as managing the AdminList. (Fig. 2.4)



```go
type AdminList struct {
    // List of SCIPER with admin rights
    AdminList []int
}
```

Figure 2.4: Field of the AdminList

The AdminList form is initialized upon receiving the first request to add an admin, following the principle of ”trust on first use.” This means that the first time an admin is added, the form is created and populated with the initial admin's SCIPER number. This approach simplifies the initialization process and ensures that the adminship management starts with a verified, trusted user. We are making the assumption that the first user is verified and trusted because it will be the sysadmin that will perform the installation and then the initialization of the system. They will be able to add the first user of their choice (themselves or someone else) as an admin.

To ensure consistency and ease of access, the AdminList form is always stored at a predetermined ID, which is the contract ID followed by AdminList. This fixed location allows for a shared form to handle adminship across the entire application.

Additionally, we designed the system to ensure that there must always be an admin. This means that the last remaining admin of the system cannot be removed until they designate a new admin. It prevents the scenarios where all data must be erased because no one can manage the system anymore.

In summary, the design of the AdminList form—with its single SCIPER list field, initialization upon first use, and consistent storage location—provides a robust and streamlined method for managing adminship throughout the application. This design ensures that the system is both efficient and secure, facilitating smooth administrative operations and maintaining the integrity of the e-voting platform.

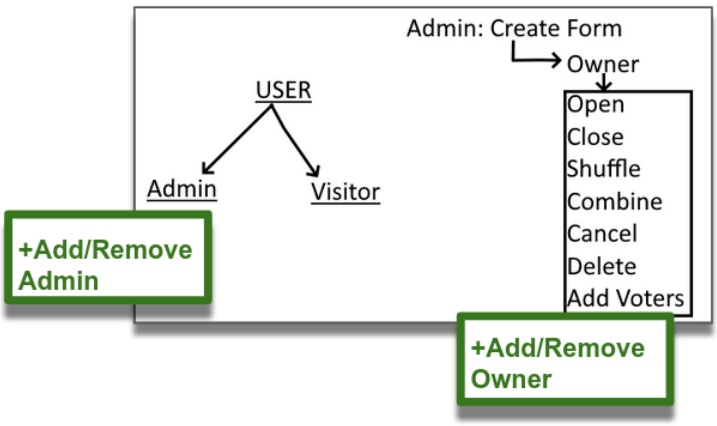


Figure 2.5: New Permission Summary

2.2 Implementation

We began the implementation process with the AdminList form. The first step was to create a new struct in the *type* package to represent the AdminList. As discussed in the design section, this struct stores the list of SCIPER numbers. We chose to store these numbers as integers, which simplifies the newly implemented validation process for SCIPER IDs.

Next, we registered the new format in the serde engine to enable JSON serialization, allowing the data to be easily stored and retrieved. To achieve this, we also defined the JSON representation of an AdminList in the *json* package, ensuring that the data format is consistent with the rest of the system. In addition to setting up the data structure and serialization, we defined several methods to interact with the AdminList. These methods are essential for managing the list of admins efficiently. We added the following methods:

- *AddAdmin*: Adds an admin to the list.
- *RemoveAdmin*: Removes an admin from the list.
- *GetAdminIndex*: Return the index of an admin in the list, if they exist.
- *AdminListFromStore*: Retrieves the AdminList from the blockchain.

By defining these interactions, we ensure that admins can be added, removed, and verified seamlessly, maintaining the integrity of the AdminList.

To initialize the AdminList, we implemented a method called *manageAdminList* in the *evoting* package. This method handles all transactions related to adding and removing admins. Here's how it works:

When adding an admin, the method first attempts to fetch the AdminList, which is always stored at a predefined location in the blockchain. If not, the method will create it and automatically add the provided admin. If the AdminList already exists, the method will verify that the user performing the action is an admin before adding the target user to the AdminList.

For removing an admin, the method includes a safety check to ensure the integrity of the AdminList. It checks that the admin to be removed is not the last remaining admin. If there are multiple admins, it will proceed to remove the requested admin. If the admin to be removed is the last one, it will raise an error to prevent the AdminList from becoming empty.

This approach ensures that the AdminList is always initialized correctly following a trust on first use design as described in the previous section and that administrative actions are performed securely. By storing the AdminList in a fixed location on the blockchain, we maintain consistency and ease of access. Additionally, the verification steps for adding and removing admins ensure that only authorized users can modify the AdminList, and the system remains robust against administrative errors or malicious actions.

Next, we moved on to modifying the form. This involved updating the existing data structure and JSON configuration. As mentioned in the design section, we added two arrays to store the SCIPER numbers, which are represented as integers.

We also implemented the *manageOwnersVotersForm* method in the *evoting* package to handle transactions for adding and removing voters and owners. This method calls specific functions in the *type* package to perform these actions. Here are the details of the functions:

- *AddVoter*: Adds a voter to the list.
- *RemoveVoter*: Removes a voter from the list.

- *GetVoterIndex*: Return the index of a voter in the list, if they exist.
- *AddOwner*: Adds an owner to the list.
- *RemoveOwner*: Removes an owner from the list.
- *GetOwnerIndex*: Return the index of an owner in the list, if they exist.

By modifying the existing data structure to include arrays for SCIPER numbers, we streamlined the process of verifying and storing user IDs.

The *manageOwnersVotersForm* method centralizes the management of voters and owners, ensuring that these lists are updated accurately and securely. By delegating the actual modifications to specific functions in the *type* package, we maintain a clear and coherent codebase.

Now, let's discuss the testing phase. To ensure the implementation functions as intended, we employed test-driven development (TDD). We drew inspiration from the existing unit tests in the *evoting* package to extend them for the newly implemented methods.

For the AdminList, we implemented the following testing scenario:

- *Initialization*: We first initialized the Contract and Form.
- *ErrorHandling*: After setting up, we conducted basic error handling tests to ensure the system responds correctly to common errors.
- *AdminAddition*: We added our first admin to initialize the AdminList. As this is the first on, it will also create the AdminList.
- *AdminRemovalAttempt*: We attempted to remove the first admin. As expected, it failed since there must always be at least one admin.
- *UnauthorizedAdditionAttempt*: Another user without any permissions tried to add themselves as an admin. This operation also failed, confirming that permission checks are working correctly.
- *AuthorizedAdditionandRemoval*: The initial admin then added the second user as an admin. After successfully adding the second admin, the first admin removed themselves, ensuring that the AdminList correctly handles changes in adminship.

We also conducted similar tests for ownership and votership rights to ensure these functions operated as expected. These tests included scenarios where:

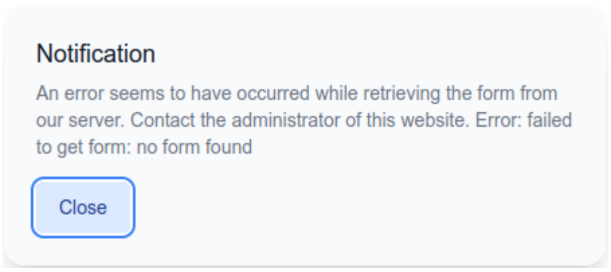
- An owner or voter is added to their respective lists.
- The removal of owners or voters is validated, ensuring that no critical permissions are lost inadvertently.

By thoroughly testing each scenario, we ensured that the system behaves correctly under various conditions and maintains its integrity and security.

To sum up, we tested various scenarios to cover new functionalities. In addition to these unit test, we modified the currently existing integration test to make them compatible with the new implementation.

However, this testing procedure is not enough to ensure our implementation work as intended and solve our single point of failure concern. As the frontend was out of the scope of the project, we need to find a way to perform end-to-end test of the different features without having the UI fully working. To do so, we use a mix of the existing UI enhanced with some cURL request. We may generate these cURL request following the API docs. In addition, as the frontend is still working with the old permission, we need to start it with two admin users to test the various features. We need to modify the *run_docker.sh local_admin*() function.

Now, if we try to create a form we will get an error telling us that no form exist. This blocking is caused by the absence of AdminList form.


Notification

An error seems to have occurred while retrieving the form from our server. Contact the administrator of this website. Error: failed to get form: no form found

Figure 2.6: Failure popup displayed after the wrongly display success is removed due to the frontend that is not updated

We may try to retrieve the current AdminList by executing a cURL request. We expect it to fail due to the absence of the admin list which need to be initialized with the first add admin transaction.

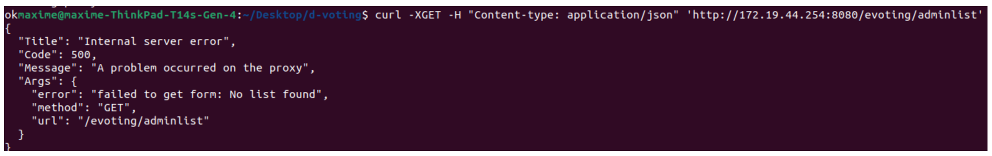
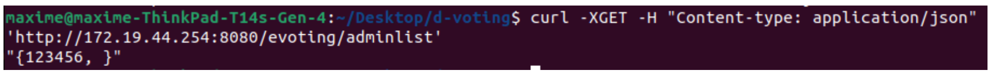

```
okmaxime@maxime-ThinkPad-T14s-Gen-4:~/Desktop/d-voting$ curl -XGET -H "Content-type: application/json" 'http://172.19.44.254:8080/evoting/adminlist'
{
  "Title": "Internal server error",
  "Code": 500,
  "Message": "A problem occurred on the proxy",
  "Args": {
    "error": "failed to get form: No list found",
    "method": "GET",
    "url": "/evoting/adminlist"
  }
}
```

Figure 2.7: Result of cURL to retrieve admin list when the list is not initialized yet

To solve this issue, we need to execute a cURL request to add an admin. We may then verify that the user has been indeed added as an admin by performing again the cURL to retrieve the current AdminList.

If we now execute following cURL command, it will attempt to remove the user from the AdminList, but it will not work as it is the only admin left.



```
maxime@maxime-ThinkPad-T14s-Gen-4:~/Desktop/d-voting$ curl -XGET -H "Content-type: application/json"
'http://172.19.44.254:8080/evoting/adminlist'
"{123456, }"
```

Figure 2.8: Result of cURL to retrieve admin list when we added the first admin

Therefore, we first need to add another admin, then we will be able to remove the first one successfully.

The same procedure works also for the add voters, with the additional manipulation that we also need to add them on the frontend to make the voting UI appear for the user. Unfortunately, it is impossible to test ownership system the same way, as the feature is not implemented yet in the frontend.

As we can notice, the implemented feature allows to verify the permission directly on the blockchain bypassing the verification from the backend. Indeed, in our test, the two user we have been using are considered as admin on the frontend. Therefore, we may consider our test mostly positive. If we want to improve our procedure, we would need to change to frontend to try the system with a real voting scenario.

2.3 Results and Evaluation

Our initial goal was to eliminate a single point of failure by transitioning the authorization process from the backend with its database to the blockchain. After a full semester of implementation, we have successfully migrated similar permissions to the blockchain. Additionally, the proxy has been updated to create new access endpoints for the API, facilitating this transition.

Since our work focused solely on the blockchain, we did not adapt the frontend. However, we created all the necessary tools within the proxy to provide the essential information for this transition. This ensures that when the frontend adaptation occurs, it will seamlessly integrate with the updated blockchain-based authorization system.

This shift to blockchain-based authorization enhances the system's security and resilience by distributing the authorization process and removing dependencies on a single backend server. The updated proxy now acts as a bridge, ensuring that the necessary data and functionalities are available for a smooth transition when frontend changes are implemented.

In summary, our efforts over the semester have resulted in the successful migration of the authorization process to the blockchain, along with the necessary proxy updates. This foundational work sets the stage for future frontend integration, ultimately enhancing the security and robustness of the overall system by reducing the number of single points of failure.

There is still a major single point of failure which is the authentication.

Chapter 3

Conclusion

3.1 Future work

A key future work to fully eliminate single points of failure in the D-Voting system is to modify the authentication process. Currently, the authentication system relies on a central point of control. To address this, we need to develop a decentralized authentication process that does not depend on any single entity, thereby enhancing the system's resilience and security.

To achieve a decentralized authentication mechanism, we can implement an asynchronous process utilizing signatures with asymmetric encryption. This approach ensures that no single point of failure exists within the system. Here's how it works:

- **Credential Verification**: The authenticator, a trusted entity within the network, will verify the user's credentials. This step ensures that the user is who they claim to be.
- **Signing the Public Key**: Once the credentials are verified, the authenticator will sign the user's public key with its private key. This step essentially acts as a stamp of approval, indicating that the user has been authenticated.
- **Blockchain Verification**: The signed public key is then submitted to the blockchain. The blockchain nodes can verify the authenticity of the signed public key by using the authenticator's public key. This verification process ensures that the signature is valid and that the public key indeed belongs to an authenticated user.

By implementing this method, we distribute the responsibility of authentication across multiple entities, thus removing the single point of failure. The system's robustness is significantly enhanced because the failure or compromise of a single authenticator does not jeopardize the entire authentication

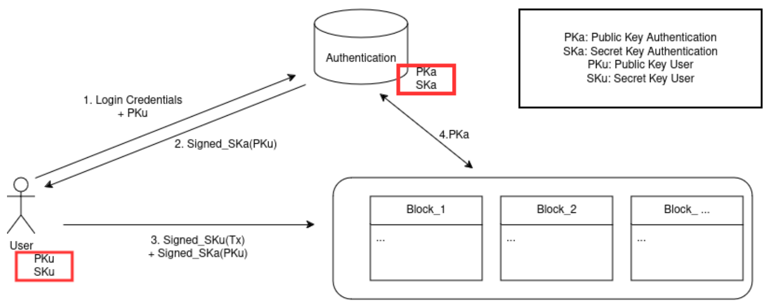


Figure 3.1: Architecture diagram of future authentication process

process. Moreover, this decentralized approach aligns with the overarching principles of blockchain technology, promoting trustlessness and security. Each authenticator operates independently, and the use of asymmetric encryption ensures that the authentication process is both secure and verifiable.

3.2 Conclusion

Throughout the entire semester, this project has been both challenging and immensely interesting. The weekly meetings were invaluable, as they allowed us to discuss major pain points and blockers, ensuring continuous progress.

At the beginning of the semester, our focus was on exploring DELA and the project's codebase, as well as discussing potential implementation strategies. Once we started implementing, we encountered several issues, such as our plan to initialize the AdminList during system start. We had to revise it because we couldn't implement it cleanly. Consequently, we adopted a "trust on first use" initialization approach, which solved our issues.

We engaged in numerous discussions about designing the permission system to retain as much of the previous logic as possible while adapting to the blockchain's limitations. A significant topic of discussion was the previously implemented rule that a form owner must be an admin. We decided to change this to: "To create a form, you need to be an admin." This change addressed the issue of someone being the owner of a form while being demoted from their admin rights, which would have required traversing the entire blockchain to identify which voting forms the demoted user owned.

This implementation helped reduce the number of single points of failure in the D-Voting system. However, the remaining single point of failure is now the authentication process, which will be challenging to address, especially given the limitations of Tequila, EPFL's authentication tool.

This project has taught us a great deal about blockchain application development and permission management. We would like to thank all our supervisors for their great assistance throughout the semester!

Appendix A

Appendix

A.1 Installations

To install the D-Voting system, please follow the detailed instructions provided in the *README.docker.md* file located at the root of the GitHub repository. This document offers comprehensive guidance on setting up the system using Docker.

We recommend using the Docker version of the system, as it employs a network virtualization setup that simplifies the development environment configuration.

You may find the Github Repository: [HERE]

The Hash commit of the README.docker is: c203ac0

You need to have a standalone Linux installation running with Docker Engine installed. [more details on Docker installation]

(By standalone we mean no VM or WSL because the network virtualization of those might mess up with the one from Docker)

There are also additional instruction in the main *README.md* on how to install the individual component locally.

Bibliography

[1] Casbin · An authorization library that supports access control models — casbin.org. https://casbin.org/fr/. [Accessed 06-06-2024].

[2] Dela - Dedis Ledger Architecture — dedis.github.io. https://dedis.github.io/dela/. [Accessed 03-06-2024].

[3] GitHub - dedis/cothority: Scalable collective authority — github.com. https://github.com/dedis/cothority. [Accessed 07-06-2024].

[4] Introducing D-Voting - C4DT — c4dt.epfl.ch. https://c4dt.epfl.ch/article/introducing-d-voting/. [Accessed 03-06-2024].

[5] What is a Single Point of Failure? Definition & FAQs — Avi Networks — avinetworks.com. https://avinetworks.com/glossary/single-point-of-failure/. [Accessed 03-06-2024].

[6] Christopher Ehmke, Florian Blum, and Volker Gruhn. Properties of decentralized consensus technology - why not every blockchain is a blockchain, 07 2019.

[7] Claude Longchamp. Swiss Abroad views: the fight to exercise political rights — swissinfo.ch. https://www.swissinfo.ch/eng/politics/swiss-abroad-views-the-fight-to-exercise-political-rights/48761518. [Accessed 03-06-2024].

[8] S. Raval. *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology*. O'Reilly, 2016.

[9] Pingcheng Ruan, Tien Tuan Anh Dinh, Dumitrel Loghin, Meihui Zhang, and Gang Chen. *Blockchains decentralized and verifiable data systems*. Synthesis lectures on data management. Springer, Cham, Switzerland, 2022.