Milestone-2: Course Project

CS - 2361 -- Blockchain and Cryptocurrencies

Prof. Mahavir Jhawar

Nandini Agrawal, Prashanthi R, Shivam Agarwal

Hyperfunds

A distributed Hyperledger Fabric application made to handle Ashoka University Faculty Research Funds in a secure way.

Find the project here (The front end is in progress.):

https://github.com/agrawalnandini/hyperfunds

Project Description

Universities and organisations across the globe promote top quality research by reserving a large amount of funds received from government, corporations and philanthropists for the purpose. The aim of this project, ***Hyperfunds***, is to ensure that the faculty at Ashoka University spend their *fixed* reserve of funds reliably with approval from the right parties - usually administrators or supervisors - using Hyperledger Fabric.

In this project, we consider three types of parties -- the faculty, the Accounts department, and the Dean of Research. An amount of funds is earmarked for each faculty member every academic year by the Dean of Research. Funds that remain unspent by the end of an academic year *do* ***not*** *carry on* to the next. If a faculty member has Rs. 1,00,000 in the beginning, every time they want to spend an amount lesser than a **fixed threshold**, say, x (let $x = 40{,}000$ here), they need to get it approved by the Accounts department. However, every time they want to spend an amount that exceeds the x, they would require an approval from the Accounts department as well as the Dean of Research. It is important to note that neither the Accounts department nor the Dean of Research is allowed to spend any of the funds earmarked for faculty. Thus, we also note that, in every spend-request either two of parties (faculty and Accounts department) or all three of the parties are involved (faculty, Accounts department, and the Dean of Research).

Project Goals

The goal of this project is to ensure that the funds are always spent by the right parties with approvals from the required parties as described in the project description above. We plan to use Hyperledger fabric to achieve the same. Hyperledger also provides immutable storage, that allows us to have a trustworthy record of all the transactions involving the funds.

Why Blockchain for this project

The Blockchain network establishes trust, accountability and transparency, which are key to our project goal. We want the spending to be as transparent as possible to make sure nobody violates the rules of this system. Having an immutable storage allows accountability of the expenditure and maintains trust between the parties.

Project Flow and Technical Details

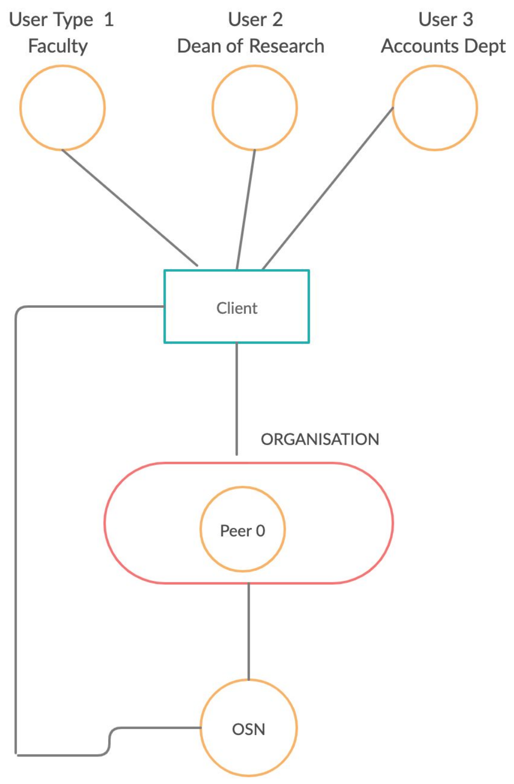

Figure 1: Network Structure for Hyperfunds

For the purpose of this project, as the Figure 1 above depicts, we plan to work with a network with one client, one organisation (with one peer) and one OSN. The three types of users -- faculty, Accounts department, and the Dean of Research -- are connected to a central client that sends out all transactions to the network. For now, we are considering only one Accounts department and one Dean of Research at Ashoka. However, the number of faculty users can be as many as we wish to have.

Each user has a dashboard that has functions based on what type of user they are. All users can sign in using their Ashoka email addresses. The Dean of Research and the Accounts department can only register once with predefined email addresses - {dean.research@ashoka.edu.in, accounts@ashoka.edu.in} to avoid any malicious activity.

Each faculty member too can register only once, but their email addresses are not predefined. On registering as a faculty member, a wallet is created. Funds are initiated by the Dean of Research from their dashboard for newly registered faculty members. The faculty dashboard has a provision to request the spending of any amount from their balance funds for research purposes. On sending any of the above two requests, a transaction will be created and sent to the rest of the network through the client. This transaction is also called the '**proposal transaction**'. The only difference between the Dean and faculty members' proposal transaction is that the Dean's proposed amount is added to the faculty members' balance and the faculty's proposed amount is subtracted from their own balance as they spend money. Additionally, at a later point, we hope to include in the transaction the hash of any document relevant to the proposal -- payment advice, bills, or other proofs of expenditure.

On the other hand, the Accounts department has a dashboard that has options to query all the proposals made until that point in time and query transactions made by faculty members by proposal transaction ID. This dashboard also has an option to approve the proposals. If this user approves any of the proposals, another transaction is created. This will be called an '**approval**

transaction'. The approval transactions are also sent to the rest of the network through the client.

The dashboard of the Dean of Research is exactly the same as the Accounts department. However, they will be able to approve only the requests that exceed the threshold. They also initiate funds to any faculty member's account on registration. They can also use this functionality to add funds to faculty members' accounts. Transactions that initiate / add funds will also require *approval by the Accounts department*.

Note: For a proposal, only when the right number of approvals is obtained from the right parties according to the rules (described in the Project Description section), will there be a change in the asset in the chainstate for that transaction.

Asset for the application:

The structure of the asset of our chainstate is as follows:

Key	Value
`{Transaction ID}`	Transaction Object: **`{UserID,Proposed amount, Approval Counter, List of Approvers, Faculty Email ID}`**

The value part of the asset is a tuple with:

UserID: This is a unique ID representing the user who creates this asset. We have designed it such that it possesses the email ID of the user as a substring.

<u>Proposed amount</u>: The amount the faculty member wants to spend from their balance / The amount the Dean wants to add to the faculty member's account. It is a positive value when the Dean makes a transaction and a negative value when the faculty members make a transaction.

<u>Approval Counter</u>: The number of approvals that the transaction has got. It should be 1 or 2 based on conditions described in the problem statement. This is set to -1 when the required number of approvals are reached, in order to prevent more approvals that might lead to confusion.

<u>List of Approvers</u>: This is a list of emails of the approvers i.e. either [Accounts department email] or [Accounts department email, Dean of Research email] based on the conditions in the chaincode.

<u>Email of faculty</u>: The email ID of the faculty member who's balance is getting updated with this transaction (if approved).

<u>Client Side Application Features</u>

There are javascript files that enable various features for the client-side application. We have described them below:

EnrollAdmin.js

This javascript file enrolls an admin to Hyperfunds. Needless to say, an admin is really important to manage any application like this.

The command to enroll an Admin is as follows:

```
node enrollAdmin.js
```

RegisterUser.js

This javascript file registers users to Hyperfunds. In our project, the Dean of Research and the Accounts department can *only register once* with pre-specified email IDs -- {dean.research@ashoka.edu.in, accounts@ashoka.edu.in}. Hence, all users are identified as Dean of Research, Accounts, or faculty (not Dean of Research or Accounts) by their email IDs. Any number of faculty members can register with their email IDs.

Note: The Dean initiates funds to the faculty members' wallet. This balance is updated on the client side after every *successful* transaction. It goes without saying that the Accounts department and the Dean will be able to view the updated balance of every faculty.

The command for registering a user is as follows:

```
node registerUser.js dor@ashoka.edu.in
node registerUser.js accounts@ashoka.edu.in
node registerUser.js fac1@ashoka.edu.in
```

Using the above commands, we can register the Dean, Accounts and a faculty member (fac1). The Dean and the Accounts are identified by the chaincode by their email IDs (since they are pre-specified).

Query.js

This javascript file provides the features to - **get balance of a faculty's account, query transactions based on transaction IDs** and **query all transactions (all/faculty wise) that have been made**. These have been described below:

1. Get Balance -

   `('getBalance', Email ID)`

   The client can use this to view the current balance of faculty members on all the three dashboards. This functionality will invoke the function **`getBalance()`** in the chaincode, which will return the balance of the faculty whose email ID is passed as argument.

   Note: faculty members will have access to only their current balance, unlike the other two types of users.

   The command to retrieve balance is as follows:

   ```
   node query.js getBalance fac1@ashoka.edu.in fac1@ashoka.edu.in
   ```

   - The first argument is **the choice of the function** in the chaincode (here, getBalance).

- The second argument is the **user ID** of the user that wants to retrieve the balance of a faculty member.
- The third argument is the **email of the faculty member** whose balance we are retrieving.

2. Query -

`('QueryTxn', Transaction ID)`

The client passes the **transaction ID** of the transaction that they want to query. The chaincode has a function **`query()`** that will return the right transaction. It is important to note that a faculty member will not be able to query a transaction that has not been made by them. If a faculty member tries to do this, an error should be thrown. However, the Dean of Research and the Accounts department can query and view any transaction by their transaction ID.

The command to query a transaction based on txnID is as follows:

```
node query.js QueryTxn dor@ashoka.edu.in 2
```

- The first argument is **the choice of the function** in the chaincode (here QueryTxn).
- The second argument is the **user ID** of the user that is querying.
- The third argument is the **transaction ID** that we want to query.

3. Query All -

`(‘QueryAllTxns’)` or **`(‘QueryAllTxns’, Email ID)`**

Using this functionality, the client will be **able to view either all transactions that have been made in the network** so far **or all the transactions made by a certain faculty member** so far. Here, it is important to note that using this functionality, a faculty member will only be able to view all the transactions made by them. They will not be able to view all the transactions made by other faculty members. However, the Dean of Research and the Accounts department can query and view all transactions.

The command to query all transactions made in the network so far are:

```
node query.js QueryAllTxn accounts@ashoka.edu.in
```

- The first argument is **the choice of the function** in the chaincode (here, QueryAllTxns).
- The second argument is the **user ID** of the user that is querying.

The command to query all transactions made by a faculty member in the network so far are:

```
node query.js QueryAllTxn dor@ashoka.edu.in fac1@ashoka.edu.in
```

- The first argument is **the choice of the function** in the chaincode (here QueryAllTxn).
- The second argument is the current **user ID** of the user that is querying.

- The third argument is the **faculty's email ID** whose transactions we are querying.

Invoke.js

This is the javascript file that provides two key features to our application - **creating a proposal transaction** request for the faculty and **creating an approval transaction** request for the other two types of users. The types of transactions have been described below --

1. Create Proposal Transaction -

   `('CreateProposalTransaction', Proposed amount, Email ID)`

   The client passes the proposed amount along with the faculty member's email ID. This method will send a request from the client side. Every proposal is added to the chainstate as a key-value pair. The transaction will not be processed until it receives enough approvals.

   The Dean of Research can create a transaction to **initiate or add funds** to any faculty member's account. **`Proposed amount`** is passed as a positive value in this case, since it has to be added to the existing balance. This transaction would require the **Accounts department's approval**.

This function is also used by faculty members to create a transaction and **spend their funds**. `Proposed amount` is converted to a negative value within the chaincode in this case, since it has to be subtracted from their existing balance. As also mentioned previously, this transaction would require approvals from the Accounts department and the Dean of Research based on conditions described in the Problem Description section.

The command to create a proposal transaction is as follows:

```
node invoke.js CreateProposalTxn 100000 dor@ashoka.edu.in
fac1@ashoka.edu.in
node invoke.js CreateProposalTxn 30000 fac1@ashoka.edu.in
fac1@ashoka.edu.in
```

- The first argument is **the choice of the function** in the chaincode (here CreateProposalTxn).
- The second argument is the **proposed amount**.
- The third argument is the **user ID** of the user that is creating the transaction.
- The fourth argument is the **faculty's email id** whose balance is getting updated.

2. Create Approval Transaction -

`('CreateApprovalTransaction', Transaction_ID)`

The second argument in this transaction is the transaction ID that has to be approved. This will send a request from the client side through the Dean / Accounts department's ID, and the chaincode will proceed to update the faculty member's balance only if the approvals meet the conditional requirements as mentioned above.

The command line to approve transactions is as follows:

```
node invoke.js CreateApprovalTxn 2 accounts@ashoka.edu.in
fac1@ashoka.edu.in
```

- The first argument is **the choice of the function** in the chaincode (here CreateApprovalTxn).
- The second argument is the **transaction ID** of the transaction that has to be approved.
- The third argument is the **user ID** of the user that is creating the approval.
- The fourth argument is the **email of the faculty** who has created this transaction.

Chaincode and Chainstate

The chaincode is the most important aspect of the application. It will possess a list of rules that Hyperfunds will work with.

Here are the global variables that we will use in the chaincode:

1. The global dictionary `balance_dict` is created for our use so that every transaction made by the faculty member uses the updated current balance. This dictionary has **faculty** `email_ID` **as key** and `Current_Balance` **as value**.
2. `txn_id` is also made global so that each transaction gets a unique key value.
3. `threshold` on which the approval criteria is based. It has been set to 40,000 by default in Hyperfunds.

There are multiple functions that the chaincode for this application will have. Below, we have defined these functions:

`CreateProposalTxn()`:

The list of arguments this method receives are: `{Proposed amount, Email ID}` of a faculty member. This method will create a proposal transaction with the given Email ID of the faculty member and the proposed amount. Here is the list of things this method will have to do:

1. If the user that initiates the transaction is a faculty member (not the Dean of Research or the accounts department), use the negative of the proposed amount as the proposed amount.

   This is because we only want to allow a faculty member to deduct an amount from their balance. Only the Dean of Research can add funds to a faculty members' account.

2. Get the balance of the faculty using the `balance_dict` and check if the proposed amount plus the balance obtained from the `balance_dict` is a positive value.
    - If no, throw the error 'Insufficient Funds! Transaction Denied.'
    - Exit the if condition.
3. Assign this transaction a `txn_id`.
4. Create an asset instance (key-value pair) with the `txn_id` as key and `{Faculty ID, Proposed amount, Approval Counter, List of Approvers, Email ID}` as value and push it to the chainstate.

`CreateApprovalTxn()`:

The arguments that this method gets are: `{Transaction_ID}`. This method will invoke a transaction whenever the Accounts department or the Dean of Research approve a transaction on their dashboard. The changes that this method makes to the transaction with `Transaction_ID` are given below:

1. Add the approver's ID to `approver_list` .
2. Increment the value of `approval_count` by 1. The default value of this variable is 0.
3. Check if the condition for the number of correct approvals is met. If yes
    - Check if the faculty has sufficient funds remaining in order to make this transaction.
        - If not then display 'Insufficient Funds! Approval Denied'
        - Set `approval_count` to a large negative value.

        - Exit out of the outer if condition.

    - Make `approval_count` = -1. This is done in order to mark that this transaction has been appropriately approved..
    - Update the global balance dictionary `balance_dict` with the value `(Current_Balance` + `Proposed amount)`.

4. Push updates to the chainstate.

`QueryTxn():`

This function allows us to query based on the Transaction ID. A faculty member will not be able to query a transaction that has not been made by them. If a faculty member tries to do this, an error should be thrown. However, the Dean of Research and the Accounts department can query and view any transaction by their transaction ID.

`QueryAllTxn():`

This function allows us to query all transactions. A faculty member will only be able to view all the transactions made by them. They will not be able to view all the transactions made by other faculty members. However, the Dean of Research and the Accounts department can query all and view all transactions.

`getBalance():`

Multiple other methods update the global dictionary `balance_dict`. This method returns the balance of a faculty based on the given email ID. We should ensure that the **faculty members should not be able to access the balance of other faculty members**. A simple pseudocode for this method would be:

```
getBalance(email_ID):
    return(balance_dict[email_ID])
```

***