HyperTasks: A Blockchain Powered Request Management App

A Final Project
Presented to Professor Mahavir Jhawar
in partial fulfillment of the requirements for the course
[**CS-2361**] Blockchain and Cryptocurrencies

GitHub Link: `https://github.com/Abhimanyu0904/HyperTasks`
Demo Link: `https://drive.google.com/file/d/1w_AMA48dV9H8EYK4J-njz6hbvbLFzdGK/view?usp=share_link`

by
Abhimanyu Sharma and Paritosh Panda

December, 2023

Contents

Listings

Chapter 1

Introduction

HyperTasks is an application designed for students and faculty members of Ashoka University, offering a new approach to requesting changes and managing those requests. At its core, HyperTasks integrates the capabilities of Hyperledger Fabric, a permissioned blockchain framework, to elevate the transparency, security, and efficiency of the feature-request and implementation process.

1.1 Relevance of Blockchain

The integration of Hyperledger Fabric into HyperTasks addresses critical challenges inherent in traditional systems. By leveraging blockchain technology, the application ensures the immutability and traceability of every action. The decentralized and tamper-resistant nature of the blockchain establishes a trustless environment, mitigating concerns related to data integrity and accountability.

In a university setting, maintaining a fair and secure platform for sharing ideas and needs is paramount. Hyperledger Fabric's permissioned nature ensures that only authorized individuals, namely students and faculty members with university-issued email IDs and ID numbers, participate in the collaborative process. This not only upholds the integrity of the platform but also aligns with the privacy and security

standards expected in an academic institution.

1.1.1 Enhanced Features with Blockchain

Transparent and Immutable History

Every request, vote, and project status change is recorded as a transaction on the Hyperledger Fabric blockchain. This transparent and immutable history not only ensures data integrity but also provides an auditable trail of actions taken within the system. This imposes accountability onto the university to follow through and not backtrack on commitments they made.

Anonymous Ideation

The architecture of HyperTasks ensures that the identity of the request creator remains confidential. This feature encourages open and honest communication, fostering an environment where users can freely contribute without concerns about potential biases.

Decentralized Decision-Making

The decentralized nature of the blockchain promotes a more inclusive and democratic decision-making process. By utilizing smart contracts, HyperTasks automates the confirmation process, ensuring that only the most popular and community-supported ideas progress to the administrative dashboard. It prevents frivolous requests from being propagated to the university, and prevents important and community supported requests from being ignored.

1.2 Relevance in Academic Settings

In the academic realm, fostering change and addressing the collective needs of the university community are perpetual challenges. HyperTasks emerges as a solution that not only streamlines this process but also introduces a secure and transparent mechanism for evaluating, confirming, and managing proposed requests.

A university needs to grow with its community, and HyperTasks can help give that community a voice and the university a structured pipeline to be able to address those concerns and needs.

Chapter 2

Project Flow and Technical Details

2.1 Network Description

The application in its current state is built on the test-network provided by Fabric. It includes two peer organisations, Org1 and Org2, and an ordering organization, that uses the *Raft* ordering service. The peers manage the ledger, execute the *chaincode*, and endorse transaction proposals within the network. The ordering organization orders the transaction proposals. Hyperledger is based on deterministic consensus, meaning that any block validated by a peer is final. The Raft algorithm is based on periodically electing a leader node for each channel and then the remaining nodes follow the actions of the leader. The endorsement policy of this network required both peers to endorse a transaction for it to be successful.

2.2 Client Application Description

The client application backend is written in *Node.js* and the frontend is built using *Python* and the *Flask* web framework. The application allows two types of users to register: students and faculty. It also allows an administrator user to log in as well. This is done by accepting a pre-configured password that only the university

should know. From here on, students and faculty will be referred to as users and the university as admin.

Users can register by providing their Ashoka email ID and Ashoka ID number. After this, an admin can see pending user registration requests, validate the credentials and either reject or accept a user. Only after an acceptance is an user able to log in to the application. In both cases, an email is sent to the email ID they registered using informing them of their registration status. The application can only check if the given email ID is an Ashoka email ID, but it cannot validate the complete email ID and Ashoka ID as it doesn't have access to that information. That is why user verification is currently a manual process.

Post registration and verification, users can view the requests by other users (of the same type) and add their own. They can *confirm* (vote for) other requests to show their support. When a threshold of confirmations is reached (half of the total number of registered users of that type), that request is said to be *confirmed*. Only confirmed requests are visible to the admin, after which they can update the status of the task: from "not started" to "in progress" / "on hold" / "implemented" / "dropped".

Users can also view the history of a request, seeing all the changes it has gone through: when it was created, when it was confirmed, and all the status updates the admin has made.

2.3 Chaincode Description

2.3.1 Assets

The chaincode has two kinds of assets: user and request. The user asset has can be differentiated by three kinds: student, faculty, and university. The application automatically creates the university asset during initialization with a predefined password and email, and otherwise allows for any number of student and faculty assets.

User Asset

```
{
    "ashoka_id": str, # Ashoka ID
    "deleted": bool, # is the user deleted
    "email": str, # email id (should be Ashoka Email ID)
    "name": str, # user name
    "password": str, # hashed password of the user
    "type": str, # user type: university / student / faculty
    "verified": bool, # is the user verified by the university
}
```

Listing 2.1: User Asset Model

The `password` is hashed using the *SHA256* algorithm and stored in the user asset. Every login check is made by hashing the given input and comparing the value. This asset only contains the basic information associated with the user.

Request Asset

```
{
    "confirmations": int, # current confirmations
    "confirmed": bool, # is it confirmed
    "confirmed_by": {str: bool}, # a dictionary storing identifiers
   for who has confirmed this request to avoid double confirmation
    "created_at": str, # IST format date string
    "description": str, # description of the request
    "key": str, # key to fetch this request from the blockchain
    "required_confirmations": int, # number of required
   confirmations
    "status": str, # current status of the request: not started (
   default) / in progress / implemented / on hold / dropped
    "type": str, # can only be "request"
    "university_notes": str, # notes from the university about the
   request
```

```
    "update_type": str, # what kind of update was just performed
    "updated_at": str, # IST format date string
    "user_type": str, # user type: student / faculty
}
```

Listing 2.2: Request Asset Model

`confirmed_by` is a dictionary containing the SHA256 hash of the email of each user who has confirmed this request as the key and `true` as the value. The `update_type` key is used to query the transaction history of a request asset but filter out all the transactions where a user confirmed this asset and only show user relevant updates, such as status updates made by the university. Only a registered user can create a request asset. To prevent any biases and encourage honest requests (and their subsequent implementation), there is no directly identifiable user data associated with the request. Only the hash of the emails of those who have confirmed a request is stored.

2.3.2 Functions

Global Variables

```
facultyCounter = 0,
requestCounter = 0,
studentCounter = 0,
users = {
    faculties: {},
    students: {},
    true_faculty_count: 0,
    true_student_count: 0,
    university: false,
};
```

Listing 2.3: Global Variables

`facultyCounter`, `requestCounter`, and `studentCounter` are used to count and keep track of the number of each type of assets created. The ledger initialise

`users.faculties` and `users.students` are dictionaries that store which users have attempted to register. If the value of this key is true, that means they have been verified, else they have been deleted or not yet verified.

`users.university` stores whether the university asset has been created. It should always be true after the network is set up once.

`users.true_faculty_count` and `users.true_student_count` stores the actual number of verified and non-deleted users. This value is used to determine the threshold for the required confirmations of a request asset.

registerUser(ctx, name, email, ashoka_id, password, user_type)

This function creates the user asset of the given `user_type` and store the user details in the asset. The `password` is hashed using the SHA256 algorithm and stored in the asset. It also updates the total number of users (of each type).

addRequest(ctx, description, user_type)

This function creates the request asset. The `user_type` is stored to be allow for proper filtration when displaying requests,

queryRequests(ctx, user_type, confirmed)

To display requests in the application, this function is invoked. It can return the confirmed or unconfirmed requests of the given `user_type`. `confirmed` can be `"true"`, `"false"`, or `"all"`. In the last case, all confirmations of the given `user_type` are returned.

confirmRequest(ctx, key, email)

This function add the SHA256 hash of the email of the users who have confirmed the given request (queried by the request `key`). If the threshold of required confirmations is met, it also changes the `confirmed` key of the request asset to `true`. It also checks if the invoking user has already confirmed the request before, and if so, returns an appropriate response indicating that.

updateRequest(ctx, key, notes, status)

When the university admin updates the status of a request, this function is invoked to update the `status` and include any `notes` provided by the admin.

queryRequestHistory(ctx, key)

To show the chain of changes made to a request, this function is invoked. To prevent users from being overloaded with unneccesary changes (i.e. the confirmation updates), the result set is filtered on the basis of the `update_type` key, removing all confirmation updates from the final result. When a request reaches the `required_confirmations`, that update is included to let users see how long since being accepted by the community has the administration taken to implement a request.

validateUser(ctx, user_type, email)

After user registration, the admin has to verify the details and identity of the user. If the details are correct, this function is invoked and updates the user asset, queries by the `email` of the user. Only upon verification is a user able to log in to the website.

loginUser(ctx, user_type, email, password)

When a user logs in to the application, this function verifies the given `email` and `password` combination. It also checks if the user is validated by the admin. If both

cases pass, the user asset is returned.

queryUnverifiedUsers(ctx, user_type)

For the admin to see the user registration requests, based on the `user_type`, this function is invoked.

deleteUser(ctx, user_type, user_email)

If the admin believes the credentials to be invalid or wrong, they can reject the user registration request by invoking this function, deleting the asset from the ledger.

2.3.3 Return Value

```
{
    "message": str, # "success" or "failure"
    "error": str, # error message if failed
    "response": {} | [] | string, # if successful, and the method
   returns a value
}
```

Listing 2.4: Chaincode Return Value Model

All functions will return a `JSON` object with the indication of the success or failure of a chaincode invocation.

Chapter 3

Future Work

This project demonstrates a lot of its core principles and objectives in its current state. Yet it has some shortcomings that can be addressed by certain changes and improvements.

1. Front End: The current frontend is very basic and simplistic. A richer user experience with a more appealing design would take the application a long way.

2. User Registration: Google OAuth can be integrated to streamline the user registration and sign in process. The Ashoka ID and other student or faculty details associated with that email ID can also be integrated if given access to that data. Users need not deal with anything but entering their email ID to log in to the application. It will remove the manual verification process from the admin side as well.

3. Categories: Requests can be categorized into more specific domains, such as those related to Residence Halls, Academic Blocks, Common Facilities (such as the Gym, sports fields etc.).

4. Network: The current network needs to be upgraded to a production level network to allow hosting the app and make it usable. Such a network could

be one with 1 peer organization representing the university with an ordering service. There can be multiple peers for the different academic departments within the university as well with channels between them and the university. This can allow for student requests for interdepartmental changes (such as cross-listing a specific course between two or more departments). Such a transaction would need to be endorsed by the required departments as well.