CMPE 491

High Level Design Report

for

“Secure Donation Chain”

by

Burak Katı

Date: 26.05.2022

Contents

1. Introduction

1.1 Purpose of the system

In many countries including Turkey, there are many obstacles along the way when a person wants to collect donations for a cause. For instance, one cannot collect money directly as a person in Turkey unless you apply to multiple government organizations for different permits. If a person does not want to go to the painstaking process, they must choose a private organization to collect the money for them. Even if one goes with the second option, the organization may be operating fraudulently, and they may lose their money. Even worse, the cause they want to collect money can be harmed as it may perceived as a fraudulent cause. Secure Donation Chain (SDC) aims to create a donation platform that provides a secure platform for both campaign owners and donors through its rating system on the blockchain.

1.2 Design goals

The main purpose of SDC is to create a secure environment for people who wish to send and collect money on the blockchain. In order to produce a working and maintainable system, the design goals are as follows:

1.2.1 Modularity

The system will be highly modular, allowing the already-existing modules to work properly when there are new ones to be introduced to the system. It will be implemented in such a way that adding new modules will be easy enough and it won't break the rest of the system entirely.

1.2.2 Maintainability

The subsystems should be not highly coupled, so that, any change on any subsystem should not affect others. This enables to keep the quality of service as new modules are introduced to the system(s).

1.2.3 Usability

Users should be able to use SDC on any device with minimum discomfort and the learning curve of using the webapp should be low. Since the client is the same for both web and mobile views and if every module is named clearly to describe its functionality, user should clearly understand how to interact with the client.

1.3 Definitions, acronyms, and abbreviations

SDC- Secure Donation Chain [6]

HTTP- HyperText Transfer Protocol

Our server and the client sides will communicate via HTTP protocol.

REST- Representational State Transfer

RESTful API's will be used for communications between our client and server sides as well as the blockchain.

RESTful- API’s that conform the constraints of REST architecture. [3]

2. Current software architecture (if any)

Currently, the SDC website is only a front-end design and not connected to any API’s or an Ethereum node yet.

3. Proposed software architecture

3.1 Overview

The proposed software architecture gives detailed information about the structure of SDC. In the following sections, the interactions between subsystems, technical decisions made, and modules planned to be deployed are explained.

3.1.2 Client

UI: Responsible for rendering data to the client for the user to interact with the blockchain.

Handler: Responsible for making and transferring requests to both internal and external services.

3.1.3 Server

Request Handler: Handles all the requests that come from the user and relays them to the blockchain node if necessary.

Transaction Manager: Utilizes request handler module for transaction related actions.

Data Manager: Responsible for storing data related to the users and campaigns to the database. Also, it is responsible for smart contract deployment and creation utilizing Request Handler to interact with Truffle.

User Manager: Handles all the cookie and session related tasks, utilizing both Data Manager and Request Handler.

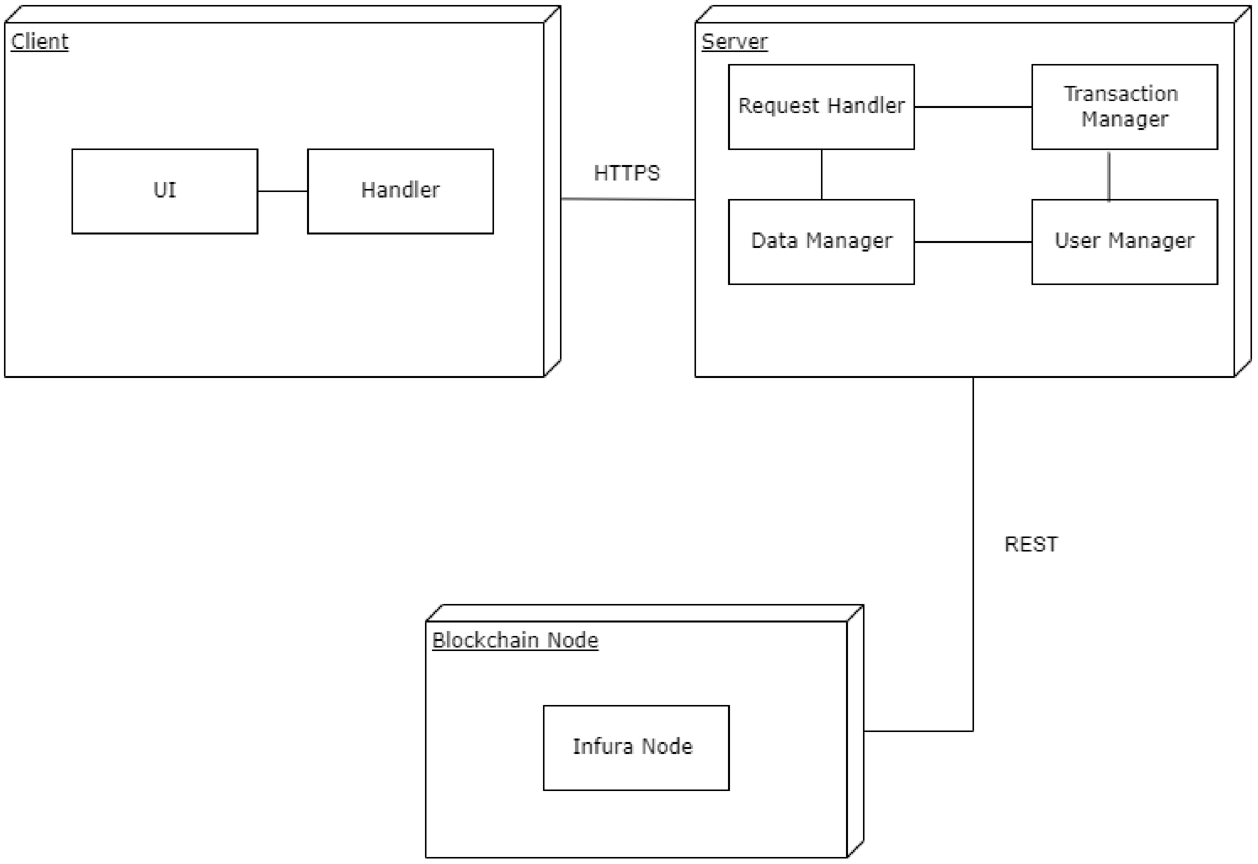


Figure 1 Component Diagram of communications between subsystems.

3.2 Subsystem decomposition

Our system will follow the client-server model on a big picture level. Our system's client-side will be a web application that acts as a bridge between the Ethereum blockchain and the user, enabling the user to add wallet(s) to our system, initiate transactions, monitor transactions on the blockchain, create campaigns, edit campaigns, rate campaigns, add comments to a campaign or another user's profile through the server-side. The server-side will handle the storage operations for user and campaign information and fetching data from both the blockchain and MongoDB to be displayed to the user. The following architecture is based on conforming our scenario and sequence diagrams. (See APPENDIX A, B)

3.2.1 Client

The client-side consists of two main subsystems as follows: the UI subsystem and the Handler subsystem.

3.2.1.1 UI

The UI subsystem is responsible for rendering data and User Interface elements to the webapp for user to interact with.

3.2.1.2 Handler

The Handler is responsible for handling requests to the backend. The Handler has different submodules for handling actions like authentication, monitoring transactions, fetching the statistics from the database.

3.2.2 Server

The server-side is also divided into several subsystems as follows: request handler, data manager, transaction manager and user manager.

3.2.2.1 Request Handler

Request handler subsystem handles both the internal and external requests. For example, when a user adds a new wallet to our system, since we use WalletConnect, our system makes a request to an external server. The response from WalletConnect comes to our server-side subsystem, and then it is forwarded as a new request to our client-side.

3.2.2.2 Data Manager

Data manager subsystem handles the database interactions such as storing and updating the campaign information and profile information.

3.2.2.3 Transaction Manager

Transaction manager utilizes the request handler subsystem and handles the interactions between our system and the Ethereum blockchain.

3.2.2.4 User Manager

The user manager subsystem handles the session information and cookie management for the security of the user accounts and enables security by secluding information about the user. For example, if a user logs in with a wallet, a session is started and until that session is finished, the user's interactions with the system is containerized.

3.3 Hardware/software mapping

Our proposed system's front-end and backend services consist of multiple submodules living inside a single server. The submodules which are responsible for storage and authentication connect to a MongoDB cluster that is stored in another server. On the other hand, our system shall create a new Blockchain node and interact with it. Both of our system modules are to be implemented with certain javascript libraries. For instance, for the client side, React framework will be used and for the server side express or nodeJS will be used and it will run on an EC2 Ubuntu machine provided by AWS. We will create and deploy smart contracts with Truffle. The node that will contain the smart contracts will be created a service called Infura.

In terms of accessibility, our system is available to any device that has a web browser, so there are no requirements on the user side to access our system.

3.4 Persistent data management

Persistent data in our system includes the wallet addresses of campaigns and donors, profile details for each user and campaign information. When a user adds a new wallet to our system, the public key of the wallet is bounded with the user at the database and used to authenticate the user for further actions such as transferring funds.

Our server will be running on an EC2 Ubuntu machine, provided by Amazon AWS. Hence, the local storage for keeping track of the session will be limited to the server's capabilities. The database system will be a MongoDB cluster which holds the user and campaign details.

3.5 Access control and security

Any user who wants to send or receive money, should log in using a cryptocurrency wallet. Each user is bounded with their wallet addresses but for better identifying themselves, they are required to pick a nickname. Any user, no matter if they are logged in or not, they can use our system's Ethereum blockchain explorer and see transaction states as they wish. Although, if they are logged in, they can see the transactions that are made by only SDC users as a privilege.

3.6 Global software control

While connecting wallets, sending money or monitoring blockchain transactions, the user's choices on the UI creates requests to our backend server. The server handles these requests and makes another requests based on them to related external API's. Most of these external API's utilize event-driven software control.

3.7 Boundary conditions

3.7.1 Connecting Wallets

Users need to login to the system using wallets provided by WalletConnect or provide their own public key if they have another wallet. The first case raises a problem: since WalletConnect is based on web3.js, which gets frequently updated, there might be maintenance breaks to update the API's and the app itself on WalletConnect's services. So, if that is the case, users must provide their own public keys and then log in to the system.

3.7.2 Crashes

In case of crashes during any action, the current action with its data is saved to local storage with a session token as frequent as possible. For example, if the user's internet connection gets interrupted while trying to add a new wallet or take any action on that wallet, the action gets cancelled before the client can send a request to the external service. However, when they get their connection back, user can select the same action and the cached information such as wallet, amount and any comments on the campaign is recovered there.

4. Subsystem services

4.1 Client

Users can access the client of the webapp through any web browser which is up to date as possible. The client is the bridge that allows the user to interact with the blockchain where the user initializes new transactions or simply monitors ongoing transactions. It uses UI and Handler subsystems to handle this interaction. The UI subsystem contains user interface elements for the user. The Handler subsystem is responsible for creating respective requests based on the events derived from the views and then communicating with the server.

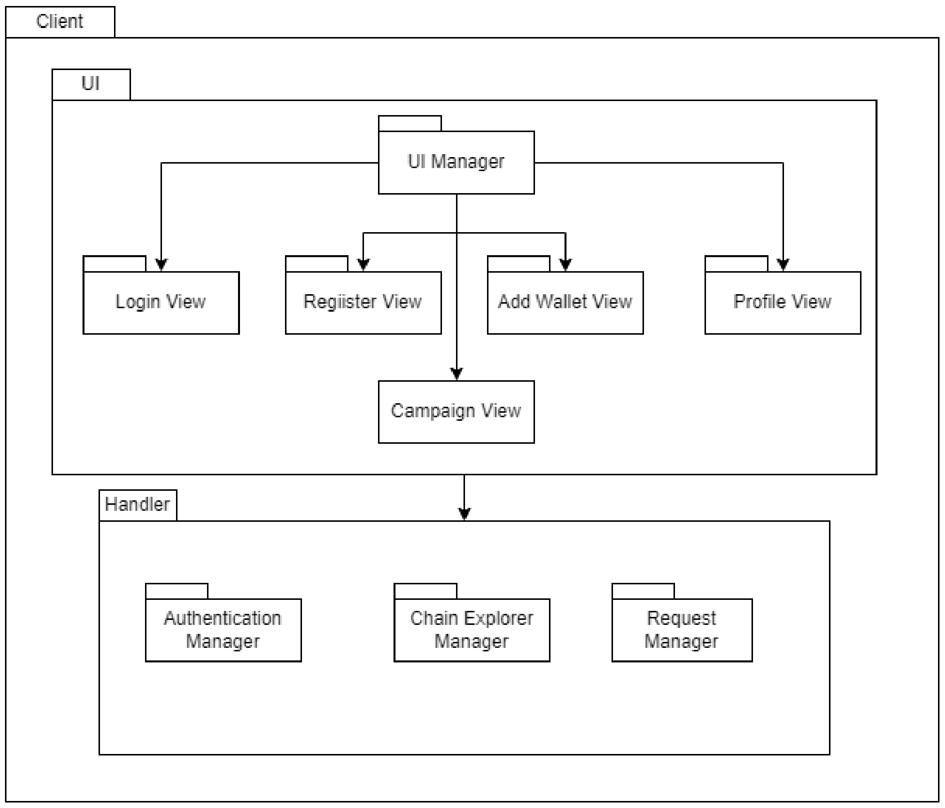


Figure 2 Hierarchy of the modules of the client

4.2 Server

The server is the main data handling point of our system. All blockchain related actions comes from the client to the server and then it is processed if it is necessary, and then new requests are created to be sent to the blockchain. The server separated into two layers as Data and Blockchain Modules. Data layer is where the database operations are done that are related to users and campaigns. The blockchain modules layer is responsible for connecting to the blockchain node and govern blockchain operations that a user makes.

4.2.1 Data

User: Handles all user information including username and public keys.

Campaign: Handles the campaigns created by users and all details about them.

4.2.2 Blockchain Modules:

Transaction Initiator: Creates a new transaction with ‘from’ and ‘to’ addresses. From and to addresses are acquired from the users’ bounded public keys.

Contract Generator: Creates smart contracts for the campaigns and deploys them to the blockchain.

Transaction Status Fetcher: Scans the Ethereum blockchain and returns requested transaction data utilizing Request Handler.

Wallet Connector: Connects a wallet to a user’s account and authorizes the user to take actions.

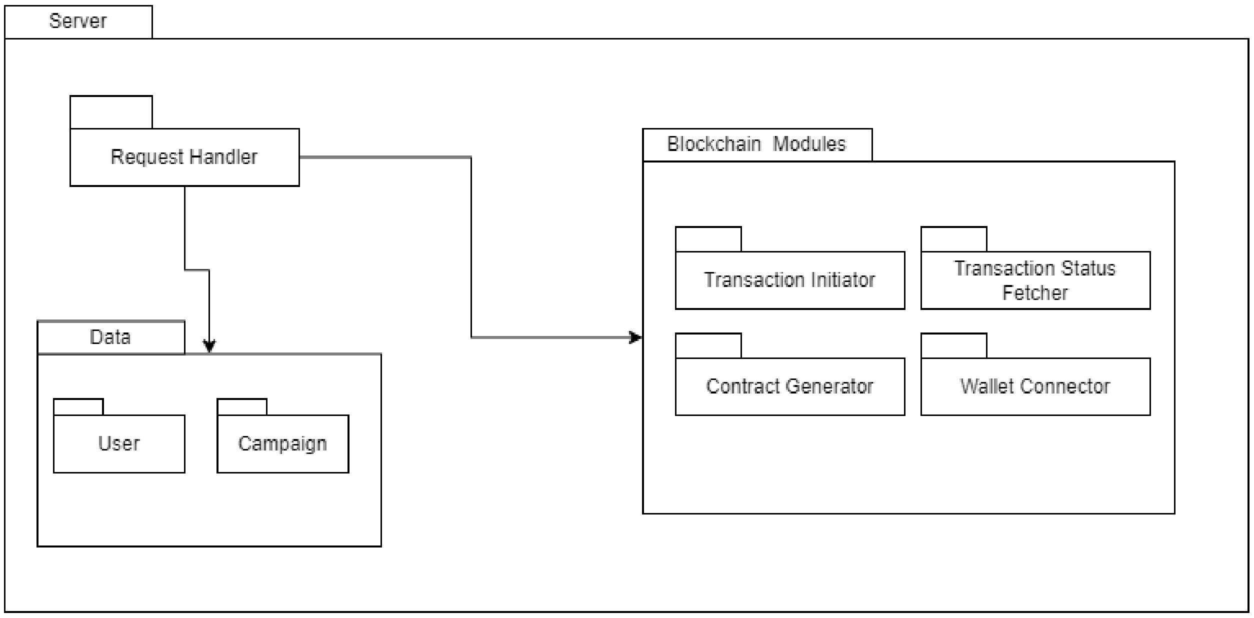


Figure 3: Hierarchy of submodules within the server

5. References

[1]: Infura https://infura.io/product/ethereum

[2]: MongoDB https://www.mongodb.com/atlas/database

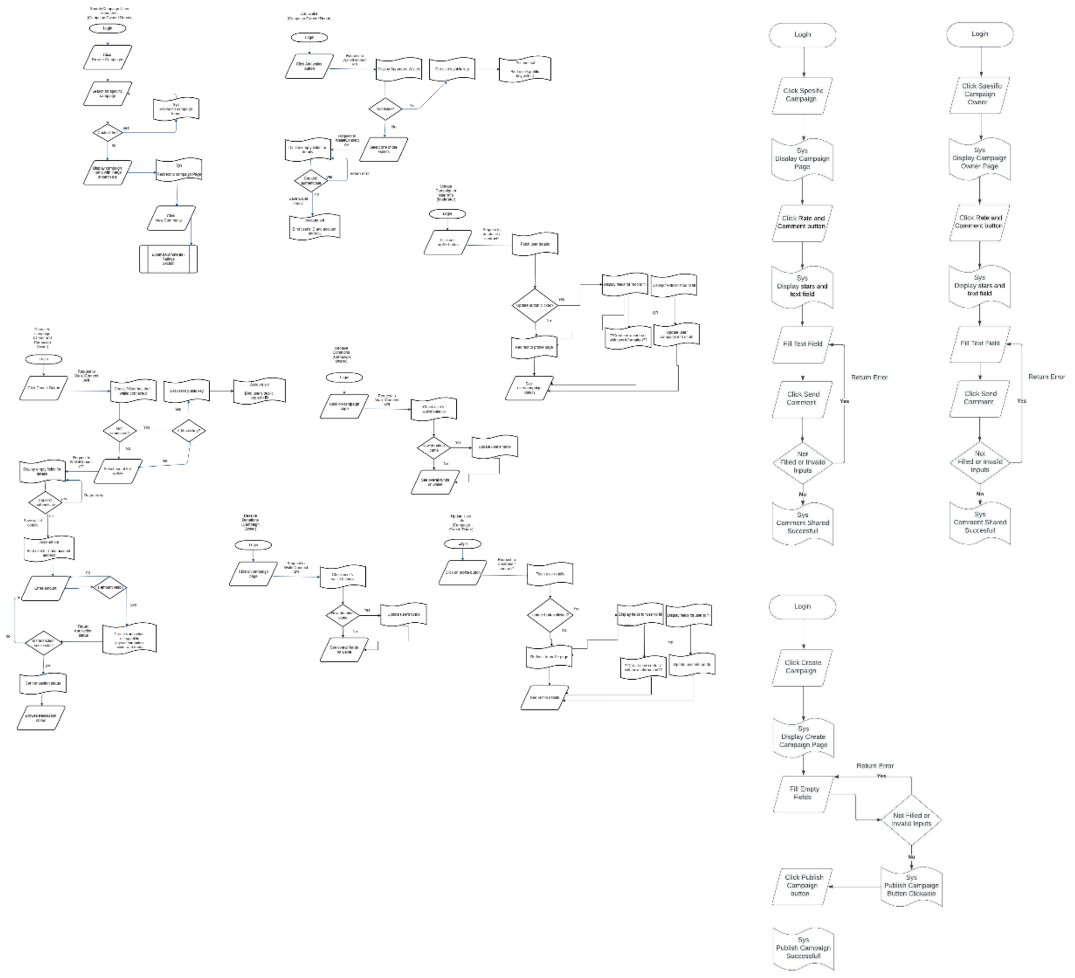
[3]: REST https://restfulapi.net/

[4]: WalletConnect https://walletconnect.com/

[5]: Truffle https://trufflesuite.com/docs/

[6]: SDC https://lelondelonmelon.github.io/bitirme/

APPENDIX A



Login
Click Specific Campaign
Sys Display Campaign Page
Click Rate and Comment button
Sys Display stars and text field
Fill Text Field
Click Send Comment
Not Filled or Invalid Inputs
Return Error
Yes
No
Sys Comment Shared Successful
Login
Click Specific Campaign Owner
Sys Display Campaign Owner Page
Click Rate and Comment button
Sys Display stars and text field
Fill Text Field
Click Send Comment
Not Filled or Invalid Inputs
Return Error
Yes
No
Sys Comment Shared Successful
Login
Click Create Campaign
Sys Display Create Campaign Page
Fill Empty Fields
Return Error
Yes
Not Filled or Invalid Inputs
No
Sys Publish Campaign Button Clickable
Click Publish Campaign button
Sys Publish Campaign Successfull

Figure 4: Scenario Diagram for our system

APPENDIX B

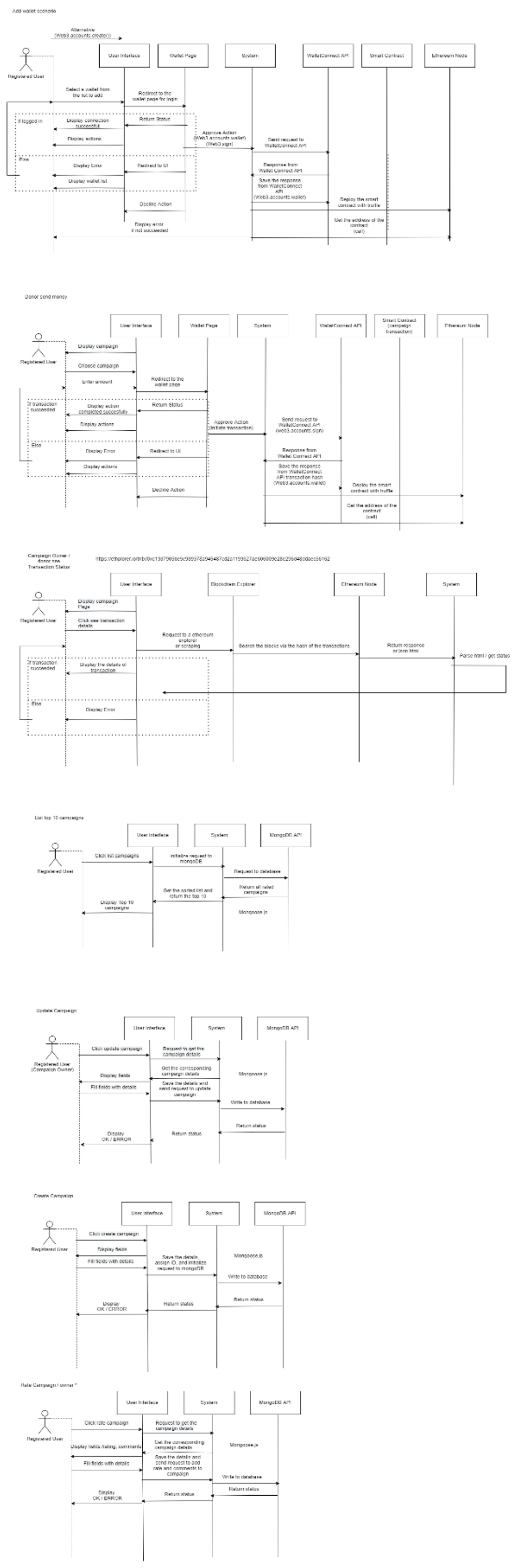


Figure 5: Sequence Diagram for our system