EMPRESTA.ME: Distributed application for goods sharing anchored in a reputation system

Technical report

Bruno Moura, João Teles, Inês Castro, Rafael Remígio, Diogo Silva, and Supervisor - André Zuquete

DETI, University of Aveiro

Tuesday 6th June, 2023

Abstract

This project focuses on the development of the EMPRESTA.ME platform, a proof-of-concept system designed to facilitate the sharing of underutilized physical assets among individuals and organizations. The platform aims to address the challenges of limited awareness and trustworthiness assessment in the lending process. EMPRESTA.ME leverages a distributed architecture and integrates a Reputation System with an algorithm to assess user trustworthiness through vouches. This enables effective connections between asset owners and individuals in need, promoting efficient asset sharing within the platform. This report provides a comprehensive overview of the development process, including requirements gathering, system design, and implementation. By adopting a proof-of-concept approach, our project aims to showcase the feasibility of a distributed asset sharing system supported by a robust Reputation System. Additionally, our work provides valuable insights for future advancements and practical applications in real-life lending scenarios.

Keywords— sharing, proof-of-concept, distributed, secure, reputation score, gift economy, decentralized

Contents

List of Figures

List of Tables

1 Introduction

Many individuals and organizations, including student groups, own physical assets that are frequently not fully utilized. These assets hold value for acquaintances within their extended social circles who lack ownership but require temporary use. Tools, appliances, musical instruments, and various other physical items fall under this category. Nonetheless, the lending process faces challenges arising from a lack of awareness regarding the circumstances of both potential lenders and borrowers. Moreover, assessing the reliability and trustworthiness of borrowers presents a common apprehension in item lending.

The main goal of this project was to develop the EMPRESTA.ME platform, a proof-of-concept system that aims to connect asset owners with individuals in need of those items. By leveraging a distributed architecture, the platform enhances the process of finding suitable connections who possess the desired items. Integral to this sharing platform is a Reputation System, built from the ground up, which incorporates a reputation algorithm that calculates user trustworthiness based on Vouches performed by users.

This report offers a comprehensive overview of the development process of the EMPRESTA.ME system, covering requirements gathering, system design and implementation. It explores the challenges and solutions encountered throughout the project and how we handled them. By presenting a proof-of-concept approach, this report demonstrates the feasibility and potential of a distributed asset sharing system, providing valuable insights for future advancements in this field and practical applications to real-life lending scenarios

2 State of the art

Our project drew inspiration from various open-source and decentralized projects. While our solution may not resemble those projects entirely, they provided valuable insights into addressing similar challenges. We thoroughly analyzed each project, focusing on their pertinent aspects that informed our approach. By examining their features and methodologies, we gained valuable knowledge to tackle our own challenges effectively.

2.1 BitTorrent

BitTorrent is a communication protocol for peer-to-peer file sharing (P2P) that allows for the distribution of files across the internet in a decentralized matter. It contrasts with the more traditional client-server communication model in that there is no central point of failure.

However, BitTorrent's architecture is not fully decentralized. It is a federated network that combines elements of the client/server and P2P models. Users download and upload files to each other as peers, which are deployed by trackers, central servers that support communication between peers.

This hybrid architecture was the inspiration for our own architecture, as we also needed to coordinate interactions between peers through a central server.

2.2 PGP

PGP (Pretty Good Privacy) employs a unique and powerful concept known as the network of trust. At its core, the network of trust is a framework that enables users to establish and verify the authenticity and integrity of public keys used for encryption and digital signatures.

Unlike traditional centralized certification authorities, PGP's network of trust relies on the endorsements and certifications made by individuals within the network. By building a web of interconnected trust relationships, users can evaluate the reliability and trustworthiness of others in the network, fostering a decentralized and community-driven approach to security and authentication.

This concept of using transitive trust between nodes in a network to assign credibility was crucial to the elaboration of EMPRESTA.ME's decentralized reputation system, VRS (Vouch Reputation System).

2.3 Bitcoin

Bitcoin's blockchain is a decentralized, peer-to-peer distributed ledger technology that powers the Bitcoin cryptocurrency. It consists of a chain of blocks, each containing data (in Bitcoin's case, transaction data). The blockchain is maintained by a network of nodes through a consensus mechanism called Proof of Work. It ensures security, immutability, and transparency by linking blocks using cryptographic hashes and requiring computational effort for validation.

EMPRESTA.ME is also decentralized, so it had to include some kind of distributed ledger, and the Bitcoin blockchain was the obvious source of inspiration. The main drawbacks we found with the Bitcoin implementation was the large ledger size (because each node requires a complete chain containing all transactions) and the resource overhead of Proof of Work.

We got around these shortcomings by relaxing some requirements that are essential for Bitcoin's needs, but not for ours. Since users do not need to store information about those they are not interested in (unlike in cryptocurrency scenarios), storing only useful information can greatly reduce the size of the ledger. Similarly, as discussed in Section 6.1.1, there was no need to use Proof of Work as a consensus mechanism, as the issue of forgery and omission can be addressed

with digital signatures and other means. Proof of Work is only used to defend against denial of service attacks.

2.4 Nano

Nano's block lattice offers a distinct approach compared to Bitcoin's blockchain. Instead of a single global blockchain, Nano utilizes a block lattice structure where each account has its own blockchain. This enables parallel processing of transactions, eliminating the need for miners and enabling faster, feeless transactions. The block lattice design enhances scalability and efficiency, making Nano an appealing alternative to traditional blockchain systems.

As such, when designing our own distributed ledger, we made it so each account has its own ledger for the reasons mentioned above.

3 System Requirements

3.1 Functional Requirements

The EMPRESTA.ME project aims to develop a distributed and secure asset sharing platform. The following functional requirements outline the key capabilities of the platform:

1. **User Registration and Login:** Users should be able to register and login to the EMPRESTA.ME platform from any device. This ensures accessibility and convenience for participants.
2. **Inventory Management:** Users should have the ability to add, remove, and edit items in their inventory. This feature allows users to maintain an up-to-date list of assets they are willing to share.
3. **Vouching System:** EMPRESTA.ME should incorporate a vouching system where users can vouch for or against other users to establish trust connections. This functionality enables participants to build a network of trusted relationships within the platform.
4. **Reputation Score:** Users should be able to evaluate the trustworthiness of other users through a reputation score. This score reflects the collective vouches received from the community and helps users make informed decisions about asset sharing. This reputation score should be calculated through the vouches.
5. **Reputation Network Exploration:** EMPRESTA.ME should provide users with the ability to explore their reputation network. Users should be able to visualize the connections they have established and understand the trust relationships within their network.
6. **Item Request:** Users should be able to announce their interest in specific items available for sharing. This feature enables users to express their desire to borrow or utilize certain assets from others.
7. **User Profile Visualization:** Users should have the capability to view the profiles of connected users. This functionality allows participants to gather information about other users, such as their inventory, reputation score, and past interactions.
8. **Community Server Hosting:** EMPRESTA.ME should provide the flexibility for communities to host their own community server. This can be accomplished by using the EMPRESTA.ME source code, a Docker Image, or by requesting EMPRESTA.ME's assistance in deploying the platform.

By fulfilling these functional requirements, the EMPRESTA.ME platform aims to create a user-friendly, decentralized, and secure environment for asset sharing.

3.2 Actors & Use Cases

The target actors of the system are **individuals** seeking to borrow or lend items within a specific community. Borrowers are individuals in need of resources, while lenders are those willing to lend out their items to help others and build trust. EMPRESTA.ME also targets **communities** that aim to create a shared inventory of items.

3.2.1 Users

Individuals who have registered with the EMPRESTA.ME system and are looking to borrow or lend items temporarily within a specific community are referred to as Users. They may be motivated by various factors such as cost savings, waste reduction, or fostering trust and cooperation within their community.

As we can see above in the Figure 4, users should be able to interact with our Mobile Application in different ways. Users should have the ability to **connect** with their acquaintances by scanning

Figure 1: Users

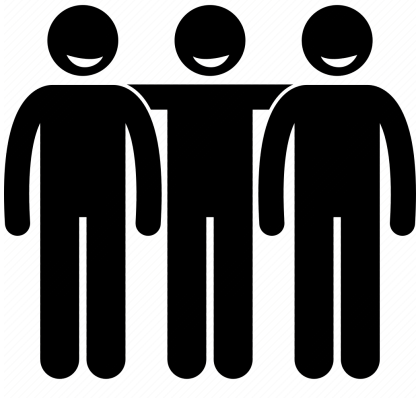


Figure 2: Community

Figure 3: Actors in this project

a QRcode. They should also be able to **join** a community by scanning a QRcode. Finally, a User should be able to **search** for available items or money to **borrow** and **announce** their own items for lending. Additionally, they can view other users' profiles and ratings to assess their trustworthiness. Users can fulfill the role of both Borrower and Lender, depending on their needs and resources.

3.2.2 Communities

Communities are groups of individuals who share a common interest or belong to organizations, and are seeking to foster trust and cooperation among their members. EMPRESTA.ME provides communities with the opportunity to create a lending and borrowing network tailored to their specific needs and interests. Communities can view the reputation of their members, encouraging a culture of responsibility and accountability within the community.

The use cases diagrams shown in Figures 4 and 5 provide a visual representation of the various interactions and functionalities available to Users and Communities within the EMPRESTA.ME system.

By facilitating lending and borrowing, the system aims to create a more resilient and sustainable model of resource sharing that benefits everyone involved.

3.3 Non-Functional Requirements

In general Non-Functional Requirements are used to define and identify system constraints and system capabilities. These requirements allow us to measure our design and implementation to objective metrics. We can then propose benchmarks for our system's performance and quality based on these requirements.

The Non-Functional Requirements we identified for our project will be grouped into the following categories:

1. Compatibility
2. Usability
3. Security

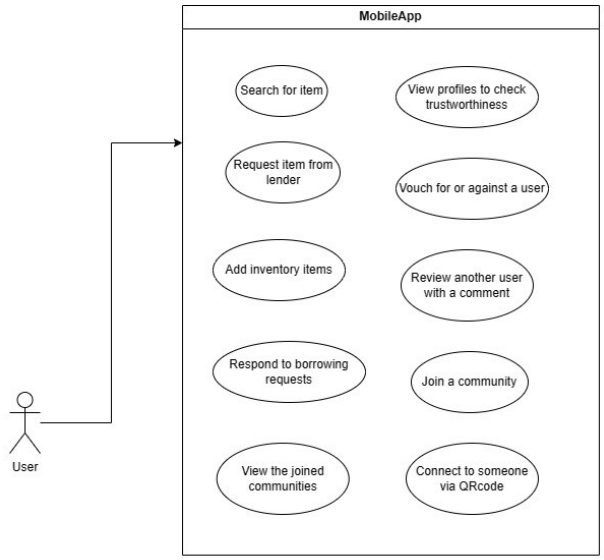


Figure 4: Use Cases Diagram - Users

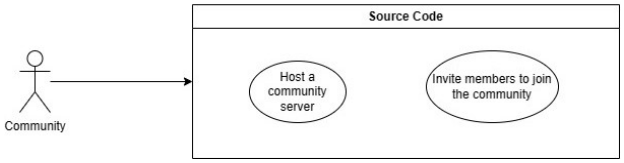


Figure 5: Use Cases Diagram - Communities

4. Maintainability
5. Fault-Tolerance

3.3.1 Compatibility

Android Compatibility
Scenario: Users of the EMPRESTA.ME service through a mobile app may have a variety of Smartphone models running a variety of different versions of the Android Operating System.
Requirements:

- EMPRESTA.ME's Mobile App must be compatible with Android Nougat [Android Nougat] (launched August 2016) all the way to Android 14 [Android 14].
- This will allow (according to Android Studio's Developer Documentation at January of 2023 [Android Documentation]) compatibility with 94.2% of Android Users.

Verification: Through the **API Level Compatibility**, the Android Operating System ensures backwards compatibility for all new API versions. This way, a application built for the Android Nougat (7.0) SDK (Software Development Kit) will be compatible with all succeeding versions all the way to version 14.

Server Operating System Compatibility
Scenario: Communities and groups of people that will create and manage a communal distribution server will be accommodating to deploy it a variety of machines. These machines may be

running different operating systems and different version of said operating systems.
Requirements:

- A server must be compatible to be deployed within the Windows operating system (from Windows 10 to Windows 11) and GNU/Linux operating system and it's subsequent distributions.

Verification: All of the communal server's functionalities will be available through a Docker Image [Docker Documentation]. This image can be used to instantiate a Docker Container, a containerized version of the application. The Docker Platform has the advantage of being highly portable and lightweight allowing for easy deployment. The Docker Architecture utilizes Linux Kernel directly being compatible with all GNU/Linux Distributions. For Windows the Windows Subsystem for Linux feature, present in all Windows Operating Systems after Windows 8, allows the installation of a virtualized Linux kernel that permits the usage of the Docker Platform.

Client/Server Compatibility

Scenario: Users of the EMPRESTA.ME's system may connect with multiple/different devices to multiple Communal Servers executing on different Operating System's.
Requirement:

- Interoperability between the End User's of the system and Communal Servers must be ensured, regardless of the platform or version of platform the User is using to access the EMPRESTA.ME system and the Operating System that is executing the Communal Server.

Verification: Compatibility and Interoperability can be ensured through Integration testing during the development process.

3.3.2 Usability

Efficiency of use, Accessibility

Scenario: The EMPRESTA.ME system is meant to be accessible and usable by all. If our system is no easy to use and accessible most users will most likely not use our product. If an interface is not efficient in executing its primary use cases, ufsers will be losing time.
Requirement

- The User Interface must be easy, efficient to use and accessible to User's who speak the English language.

Verification: To verify this requirement the User Interface will be subjected to multiple Usability Heuristic Evaluations on multiples phases of development. There will also be User Tests conducted during the development process.

3.3.3 Security

Impersonation Attacks

Scenario: One way an adversary might attack the system is by impersonating another user.
Requirement: No Entity of the system shall be able to impersonate another Entity. A User's real Identity must be mapped to one and only one Entity of the system.
Verification: This requirement shall be verified in the following ways:

- Every user should possess a asymmetric key pair as a means of authentication. In fact, the public key of this key pair is what identifies a user in the network. Every message created by a user (e.g. vouch messages) is to be signed with their private key. As such, users can verify the authorship of received messages through the means of digital signatures. Messages with missing or otherwise invalid signatures should be dismissed.

- Every user in the system must be verified when created by an Identity Provider (IdP). Each entity must be associated to one and only one identity.

Vouch Forgery
Scenario: One way an adversary might attack the System is by issuing fake Vouches in order to diminished trust in another User or increase on itself.
Requirement: No Entity of the system shall be able to issue Vouches on others behalf and in this way manipulate Reputation Scores.
Verification: Every message created by an user (e.g. vouch messages) is to be signed with their private key. As such, Entities can verify the authorship of received messages through the means of digital signatures. Messages with missing or otherwise invalid signatures should be dismissed.

Vouch Omission
Scenario: Another way an adversary might attack the System is by omitting unfavorable vouches. This can be current or previous vouches of this user.
Requirement: No user shall omit current or previous Vouches that occurred on the Reputation System Network.
Verification: This requirement shall be verified by enforcing that all vouches are shared to all users in the communal server's. Users must store current Vouches of other users, as well as the signatures of said vouches. Users must also store past Vouches of entities and said signatures. As long as at least one user within the system has the omitted message backed up the system can recover from vouch omission attacks.
When an user wants to learn another users reputation score it does not only ask the entity it wants to verify, but all entities in the system.

Server Impersonation
Scenario: An adversary may attack the system by impersonating a Communal Server in order to intercept messages and even impede Users to use the Service or impede the distribution of certain messages in the system.
Requirement: Users of the system must be able to identify if a Communal Server is an impersonation or the actual Server belonging to that Community.
No adversary must be able to perform a Man-in-the-Middle attack where the Communal Server is impersonated.
Verification: This requirement shall be verified through the employment of certification chains that validate the systems identity. If a server does not hold a valid certificate with a valid certification chain, the user who connects to this Communal Server will be notified that the server is not properly certified

3.3.4 Maintainability

Server Software Maintainability
Scenario: Continuous Software Development requires the adding of features, solutions to software flaws or faults and increases in performance or usability. These changes might be performed by Communities or Organizations at various times and at various intervals of time.
Requirement: Software updates to the Communal Server must be able to be performed without any loss of persistent data, i.e. data about latest vouches and users of the system's public keys and identities.
Software updates to the Communal Server must be able to be performed without leaving the server unreachable for periods longer than 5 minutes. The system must be able to be redeployed and be running as a previous configuration under this time.
Software updates to the Communal Server must be able to be performed without losing compatibility and interoperability to User's chosen platform.
Verification: This requirement shall be verified through:

- Integration testing of the multiple constituent components;
- Backups of all data in a persistent database that is consulted and updated at the Server's runtime;
- Evaluating the deployment time of multiple versions of the Server different machines with different hardware and software components.

Android Maintainability
Scenario: Continuous Software Development requires the adding of features, solutions to software flaws or faults and increases in performance or usability. Android User's of our system may choose to perform these changes at any time and to multiple released versions of the application.
Requirement: All updates to the Android Application used in the EMPRESTA.ME system must be able to be performed without loss of data, loss of functionality and loss compatibility and interoperability with other machines/members of the system.
Verification: This requirement shall be verified through:

- Integration testing of the multiple constituent components;
- Using the Android Application's persistent data model and choosing a language independent encoding format.

3.3.5 Fault Tolerance

Server Side Failure
Scenario: The Distribution Server deployed by a community/organization may become unreachable, unavailable or even stop executing for multiple reasons. These losses in availability may happen at any given time, and may be caused by software faults or mistakes, network issues, hardware faults of failures.
Requirement:

- In the event of loss of availability the Distribution Server must be able to learn all changes occurred in the network, i.e. must be able to learn about all new "Vouches" that occurred between Entities while it was unavailable.
- In the event of a loss of all persistent data, the Distribution Server must be able learn about all vouches that occurred between Entities.

Verification: This requirement shall be verified through:

- Periodic messages exchanged between the participants of the System and the Communal Server ensure that the server is always aware of changes in the systems;
- "What did I miss" messages, that can be exchanged by the Communal Server and the entities in the system. These messages report the changes, that the entities know of, that have occurred in the system after a given time-stamp

4 Architecture

The architecture we developed aims to provide solutions for both our functional and non-functional requirements.

As explained earlier, every user should have stored a view of the trust topology that matches their connections and interests. Additionally, they must be able do add new information to the topolgy through vouches as well as verify the authenticity and integrity of incoming changes.

Our system can be comprised of several communities consisting of servers with Flask APIs and message queues. For a user to be able to communicate with each other, they must be associated to at least one community.

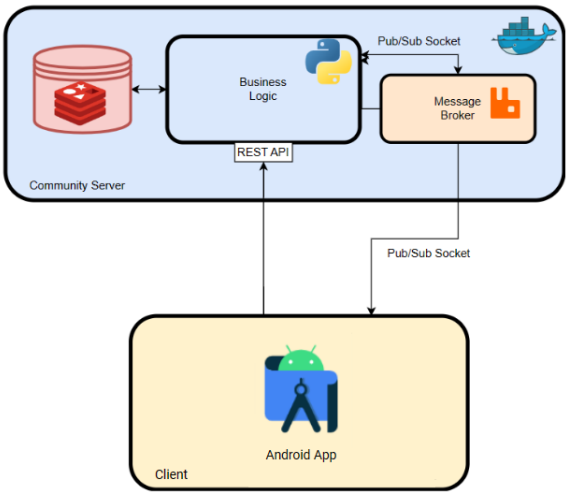


Figure 6: Example figure

4.1 Communication

All interactions between users are done through the REST API or the Pub/Sub system. The API itself is built using the Flask Framework and all business logic is done in Python. The Pub/Sub system is realized using RabbitMQ.

It is important to note that users are not restricted to communicating solely with peers who belong to the same community. In practice, communities function as intermediaries for message brokering and authentication, but they do not dictate the manner in which user interactions take place. Users have the freedom to engage with other users across communities without any inherent limitations. The role of communities primarily revolves around facilitating message exchange and providing authentication mechanisms, while the actual communication and interaction between users remain independent of community boundaries.

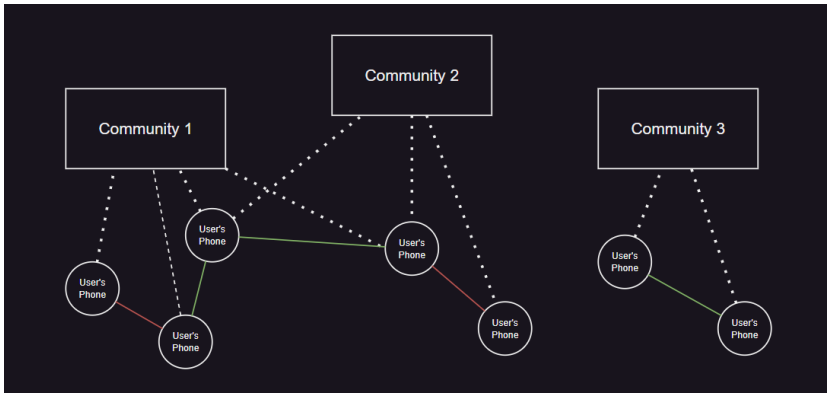


Figure 7: a trust topology, highlighting the different communites nodes are associated with

4.2 Pub/Sub System

We use RabbitMQ as a message broker, specifically leveraging the Publish/Subscribe patterns to handle message distribution between peers.

A topic is assigned to each user in the network, which holds exclusive publishing rights for messages 8. Users utilize these topics to disseminate various messages, such as vouches, to interested parties. Concurrently, users have the ability to subscribe to topics belonging to other users, enabling them to receive updates and notifications. This mechanism facilitates targeted information distribution, allowing for an effective communication and engagement among network participants.

In addition to expressing interest in specific participants, users may have a requirement to receive notifications from any user within a specified distance threshold on the trust topology. To address this, we employ a method we call Topology Spreading 9. This process involves users recursively probing the leaf nodes in their trust topology for neighboring nodes until the distance threshold is surpassed. By employing this approach, users can effectively expand their reach and ensure notifications from relevant users within the desired proximity.

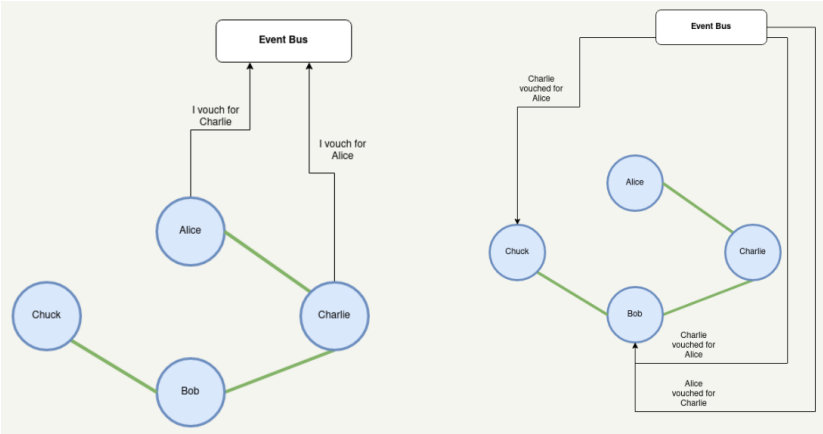


Figure 8: Topology Spreading

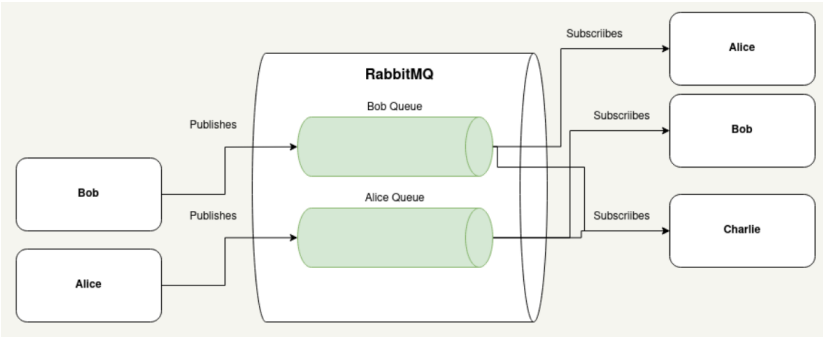


Figure 9: How RabbitMQ Pub/Sub is used

4.3 Asynchronous Communication

Asynchronous communication is achieved by integrating a publish/subscribe message broker and communication is over TCP sockets. This broker allows users to get up-to-date information

about the status of connected networks of interest. A user can subscribe to multiple channels, with each channel serving a different user on the network. In this way, relevant information about network status can be exchanged asynchronously and without causing significant overload.

4.4 Persistence

The Data Access Object (DAO) layer of the community server architecture plays a key role in handling data persistence and access to the reputation system network. It interacts with a Redis database and acts as a bridge between the application's business logic and the underlying data store.

The community server also contains its own copy of the trust topology via a Redis database. Using this database as a cache for certain operations ensures efficient access to frequently accessed data.

The model diagram in Figure 10 illustrates the entities and their relationships within the DAO layer. Essentially we are storing each currently pending authentication challenge (challenge-response authentication), a set of all the current session tokens and an hashset with each user's account info.

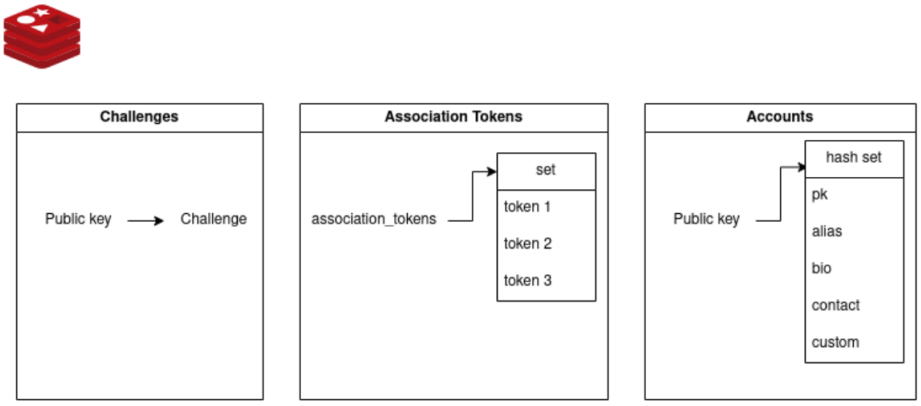


Figure 10: Model Diagram of the DAO Layer

5 Implementation

5.1 Vouch Reputation System

5.1.1 Introduction

Vouch Reputation System, or VRS for short, is a proof of concept for a decentralized reputation system for peer-to-peer networks.

The system's goal is to be a community-focused solution to the uncertainty of trust between unfamiliar peers in environments without authorities. Building upon already existing social bonds between its participants, VRS extrapolates a numerical reputation score for each node in the network in a way that reflects its standing on it.

This repository contains a small Python project for simulating a VRS network and calculating accurate reputation scores. Note that it is not a full implementation of a peer-to-peer network.

5.1.2 Vouching

In VRS, every entity (person, organization, etc.) is represented by a node. Nodes can vouch for or against any other, signalizing trust and mistrust respectively. Vouches can be retracted and altered at will. A node's vouches (both for and against) are public to all.

Vouches are stored in what is called a vouch history. It is an ordered sequence of vouches and their logical timestamp in a way that preserves their past and current vouches.

Take for example this vouch history:

Logical Clock	Node	Vouch
1	Alice	For
2	Charlie	For
3	Alice	Against

Table 1: Vouches Table

Table 3 This would mean that its owner currently vouches for Charlie and against Alice, but previously vouched for Alice.

Vouch connections are created between nodes depending on their vouches. If two nodes reciprocally vouch for the other, a vouch for connection is formed. If at least one of them vouches against the other, a vouch against connection is formed.

As such, the entities and their vouch connections can be displayed as a trust topology. When we are looking at the topology from the perspective of a particular node, we call it the observer node.

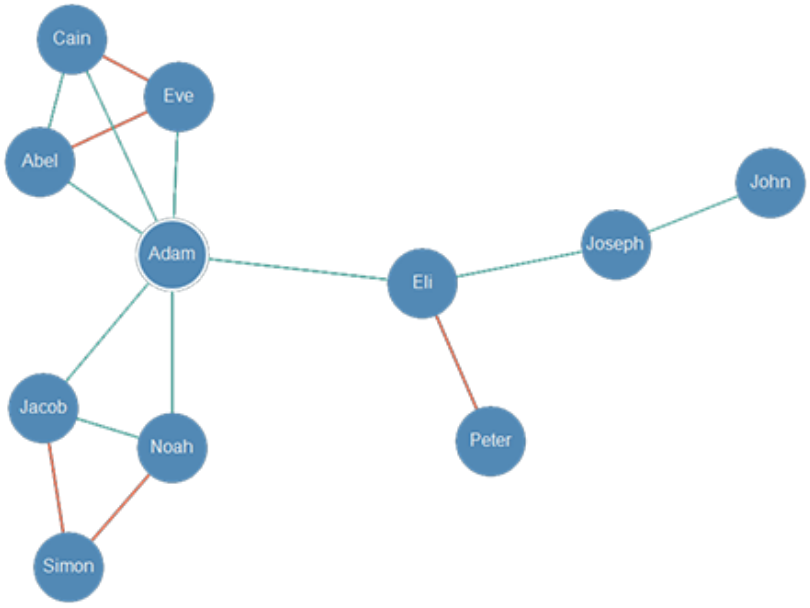


Figure 11: Adam's trust topology. Vouch for relations are in green, vouch against relations in red. The observer node is circled in white.

The reputation of nodes in the network (including those the observer does not vouch for or against) can be assessed through the vouch connections there present.

It is important to note that a node's reputation is not an intrinsic property it posesses but rather is contingent to an observer. The same node can have wildly different reputation scores when seen through the perspective of nodes with different vouch connections.

In concrete terms, let's say that Alice vouches for Bob. That means that Alice trusts him and is willing to put her own reputation on the line for him. If yourself vouches for Alice but does not know Bob, you can reasonably assume that Bob is somewhat trustworthy based on his relationship with Alice. Reversely, Bob would seem untrustworthy to you had you vouched against Alice. Things would be more complicated were you connected to multiple nodes with different opinions on Bob, but the same logic applies.

Vouching is also a way for a node to make a public statement about some other's reputation while leaving its own reputation as a collateral. That is to say, when a node vouches for some other it is essentially tying its own reputation with the other's.

For instance, let's say you and Alice vouch for each other. If Bob were to distrust Alice and not know you, your own reputation (from his point of view) would be negatively affected by your association with Alice. Reversely, if Charlie does not know you but trusts Alice, your reputation would rise from his perspective. As before, things get more complex when the reputation of a node is affected by multiple sources.

Nodes may be connected to the observer by a varying degree of proximity. Naturally, the opinions of nodes closely connected to the observer carry more weight than those of distant ones. It is more impactful to be vouched by someone closer to the observer than by a distant connection.

Not only that, but the relevance of a node's opinion is also proportional to it's own reputation score. Let's say that Alice vouches for Bob and Charlie vouches against Bob, but Alice has a higher reputation score than Charlie. As Alice has a better standing on your network, Bob's

reputation should be biased towards a good reputation score, rather than to be perfectly even between good and bad.

An observer can also apply a weight to every node. Weights serve as a multiplier for the node's opinion's relevance on the observer's network. For instance, even though you vouch for both Alice and Bob, you can make it so Bob's opinions are twice as relevant as Alice's by giving him a 2x weight while leaving Alice with the default 1x weight. Weights are not public information and can be kept private.

Finally, nodes that are deemed too loosely connected (or not connected at all) to the observer have no reputation score, as there is not enough meaningful data to estimate it. As such, they can be ommited from the network as far as the observer is concerned.

5.1.3 Reputation Score

The first step to understand how the reputation score is calculated is to convert the trust topology from a mesh to a tree.

Let's take this simple trust topology as an example:

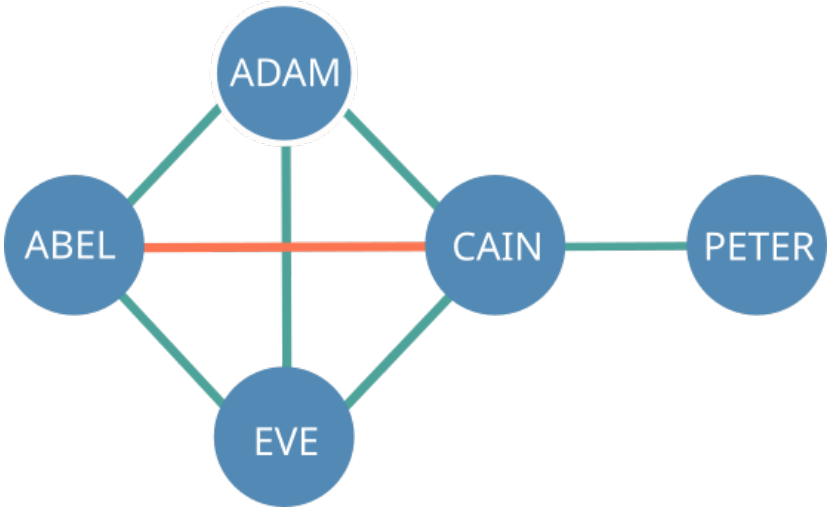


Figure 12: Adam's trust topology.

Converted into a tree, it would look as follows:

The next step is to traverse every path in the tree starting from the observer node to attribute influence to nodes. At every visit during the traversal, we keep track of two things - the distance from the observer node and the current polarity of the branch, which combined are called what is called an influence.

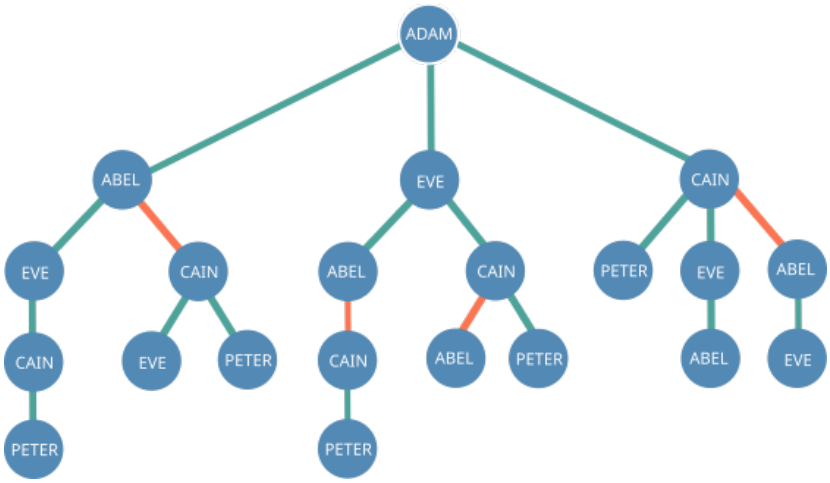


Figure 13: Adam's trust topology.

The notation used to express an influence on node is a plus or minus sign (representing polarity) followed by an integer (representing the distance). e.g. +0, -1, -5.

To illustrate, let's analyze the influences on each node of this particular branch of the tree:

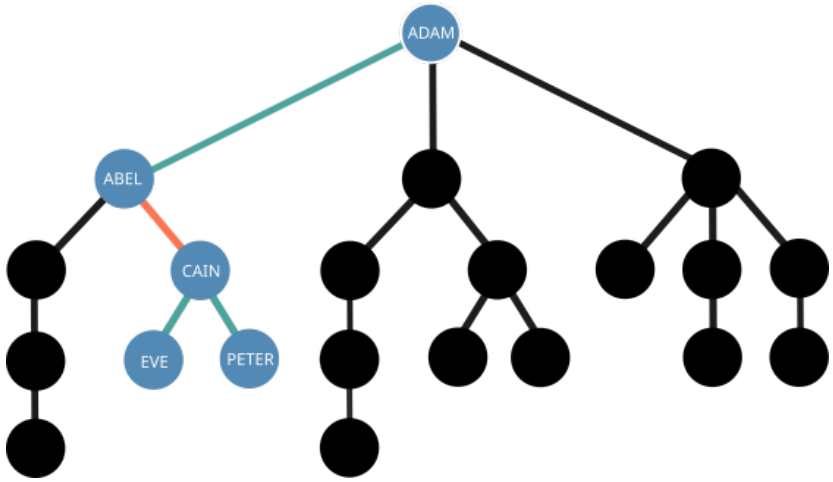


Figure 14: Adam's trust topology.

Node	Influence
Abel	+0
Cain	-1
Eve	-2
Peter	-2

Table 2: Vouches Table

Abel has an influence of +0 because it is directly vouched for by the Adam, the observer node. As they are directly connected, the distance is zero. Cain has an influence of -1 because it vouched against Abel and is one node away from being directly connected to Adam. Eve and Peter both have a -2 influence. It is easy to see that the distance is two because they are both two nodes away from being directly connected to Adam, but why is the polarity negative if Cain vouches for them both?

During the process of attributing influence, if a previously unbroken chain of positive connections is followed by a negative connection, every following positive connection is turned into a negative one. Thus, the positive connections between Cain and Eve/Peter have negative polarity because Cain was vouched against by Abel earlier in the chain.

This is the "Friend of my enemiy is my enemy" policy. This is in place to make it so users are negatively impacted by associating with known bad actors, for reasons discussed later.

A similar policy not applicable in this example is the "Enemy of my enemy might not be my friend". It states that when attributing influence to nodes, a given branch can only have one negative connection in it. From the second negative onwards, the influences are ignored. This is in place because it does not logically follow that vouching against someone considered to be disreputable is somehow evidence of good behavior, as infighting between malefactors is possible.

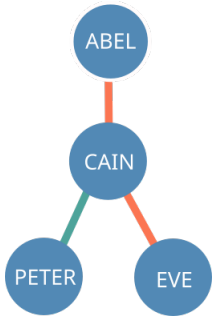


Figure 15: Example of "Enemy of my enemy might not be my friend".

Although Peter is negatively affected by his positive connection with Cain, Eve is not affected at all by her negative connection.

Back to the matter of influence. After going through all the paths in the tree, each node will have a list of influences:

Node	Influence
Abel	+0, +1, -2, -1, +2
Cain	+2, -1, -2, +1, +0
Eve	+1, -2, +0, +1, -2
Peter	+3, -2, -3, +2, +1

Table 3: Influence table. The plus and minus sign refer to positive and negative vouches, respectively. The number refers to the number of steps from between the node and the observer.

The reputation scores are calculated from the influence lists according the following procedure (not actual production code, merely illustrative):

```
def calculateReputationScore(influences : list[tuple(int, bool)]) -> float
"""Calculates the score of a node given their influence list"""
python
# weight is determined by the reverse square of the depth
def weight(depth) -> float:
    return 2**(-depth)

numerator = 0
denominator = 0

# calculate the effect of each influence in the overall score...
for dist, polarity in influences:
    # adds to the numerator either zero or one (depending on polarity)
    # times the weight as dictated by the distance to the observer
    numerator += (1 if polarity else 0) * weight(depth)

    # adds the weight to the denominator
    denominator += weight(depth)

# calculate the weighted average
score = numerator/float(denominator)
# applies distance falloff
score *= weight(dist_to_observer)
```

```
return score
```

The function iterates through each node's influence list and calculates the contribution of each influence to the overall reputation score. It uses a weighting function (by default, a exponential falloff) to assign weights to influences based on their depth. This adjustment accounts for the diminishing influence of nodes that are farther away.

The numerator keeps track of the cumulative impact of positive influences, while the denominator represents the sum of the weights of all influences. After processing all influences, the function calculates the weighted average score by dividing the numerator by the denominator. This score reflects the overall reputation of the node based on the received influences.

5.1.4 Conclusion

Through VRS, users can assess anyone's (even complete strangers) trustworthiness through their reputation score as long as someone is their personal network is connected to them without, all without resorting to authority figures.

From its set of rules emerges an entropic force that promotes network safety. Vouching in favor of a node which is generally considered to be untrustworthy will bring your own reputation score down to those that vouch against it. Reversely, vouching against a reputable account will also cause the same effect to those that vouch for it. Users that do not vouch for or against others when it becomes warranted will lose out on reputation gains, potentially new connections and a more accurate reputation estimate in their own personal network. In other words, users are incentivized to be consistent in their good behaviour and to ostracize actors that might compromise network safety, all for their own immediate self interest.

Just as well meaning users looking to maximize their own experience in the network are incentivized to be careful in regards to whom they associate with and be vigilant about troublemakers in their own personal network, malicious users are incentivized to act independently and not form connections between each other, as being associated with known bad actors will bring down their own reputation score, hampering coordination between malefactors.

5.2 Android Application

In the Mobile Application we choose to develop a Native Android Application because of the following reasons:

1. Performance: Native apps built specifically for the Android Operating System allows us to optimize the performance of the app, resulting in faster response times, smoother animations, and overall better user experience.
2. Access to Device Features: have direct access to the full range of device features and capabilities, such as camera and Trusted Execution Environments
3. User Experience: more easily provide a seamless and native user experience
4. Enhanced Security: leverage the security features and frameworks provided by the operating system. Encryption, secure storage, and other security measures protect sensitive user data more effectively.

Development was mostly carried out within Android Studio, a powerful and versatile integrated development environment (IDE). Its robust set of tools and libraries provides developers with endless possibilities to create sophisticated Android applications.

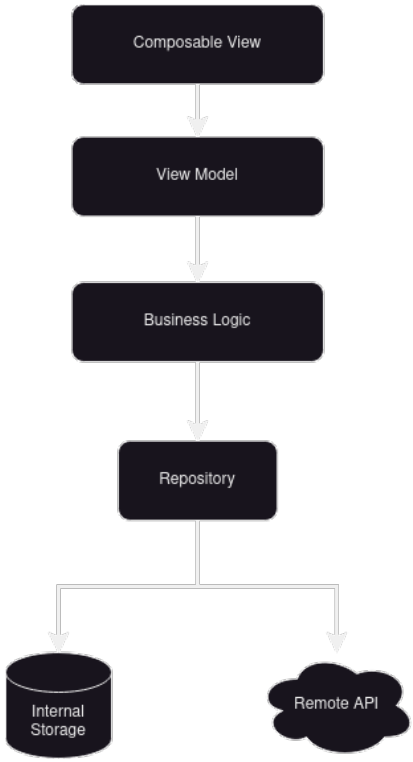
5.2.1 Android Architecture

MVC - The MVC architectural pattern, widely adopted in Android development, is designed to separate concerns and enhance code maintainability. It divides the application into three distinct components: the Model, View, and Controller. The Model encapsulates data and business logic, the View represents the UI components, and the Controller acts as the intermediary, facilitating communication between the Model and the View. By decoupling these components, developers can isolate functionalities, foster code reusability, and streamline future modifications.

Repository Pattern - The repository pattern plays a vital role in managing data access and storage within an Android application. It acts as a middle layer between the data sources, such as databases or web services, and the rest of the application's codebase. By abstracting the data access logic behind a repository interface, we can achieve several advantages. Regardless of the underlying data source, the repository interface defines a standardized set of methods to interact with the data. This abstraction shields the rest of the application from the complexities of data retrieval, storage, and caching. Developers can easily switch between different data sources, such as local databases, remote APIs, or in-memory caches, without affecting the application's overall functionality.

Jetpack Compose has revolutionized UI development in Android applications. Compose is a modern toolkit that simplifies the process of building UIs through a declarative approach. Unlike the traditional imperative method, Compose enables developers to describe the UI state and components using a reactive and concise syntax. This shift in paradigm facilitates the creation of dynamic, responsive, and highly customizable interfaces, resulting in an immersive user experience.

Combining MVC, the repository pattern, and Jetpack Compose empowers developers to create Android applications that are scalable, maintainable, and visually captivating. By adhering to the separation of concerns provided by MVC, developers can build well-organized and modular codebases. The repository pattern ensures efficient data management, enabling seamless integration of various data sources



5.2.2 Data Persistence

Data persistence is a critical aspect of Android development, as it involves storing and retrieving data to maintain its availability across different app sessions or device restarts. In this project Data Persistence is handled thought:

1. Key Storage in a Secure Storage in a Hardware Security Model (Android StrongBox)
2. SQLite Database: SQLite is a lightweight relational database management system included with Android.

To interact SQLite databases in Android applications we are using Room, an Android Jetpack library provided by Google. Room provides an abstraction layer over SQLite, offering a more robust and convenient way to interact with the database. Room incorporates the Object-Relational Mapping (ORM) approach, allowing us to define entities (representing tables) and use annotations to map them to corresponding database tables. It also provides support for defining relationships between entities, enabling the creation of more complex database structures. One of the key benefits of using Room is its ability to generate efficient and optimized database access code. It offers compile-time checks, ensuring that SQL queries are validated during the build process, reducing the chances of runtime errors. Room also provides LiveData and RxJava integration, allowing developers to observe database changes and react accordingly. By utilizing Room, we can leverage the power of a relational database in their Android applications without dealing with the complexities of raw SQL queries and database management. It promotes cleaner code architecture and simplifies common database operations, such as querying, inserting, updating, and deleting data.

Data Persistence is an important part of our distributed system since there is not a centralized entity that stores information about the Reputation System and information relevant to item announcements and item requests. All this data is shared between users of the system and stored in their local storage.

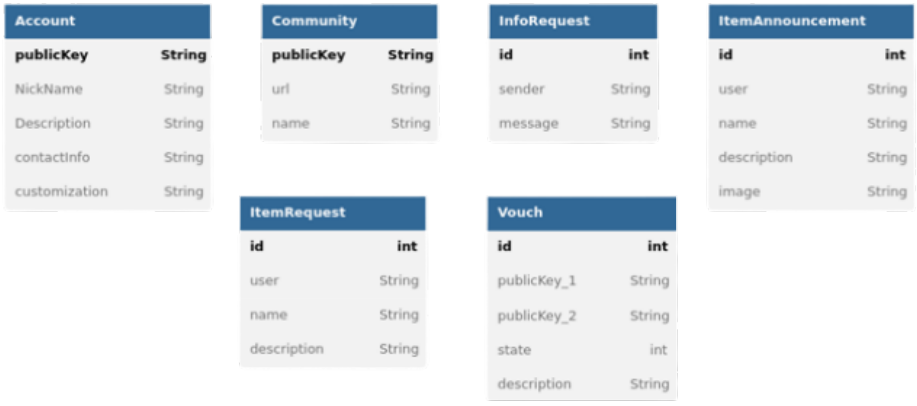


Figure 16: Model diagram of DAO layer in the android application

5.2.3 Remote API

To handle communication between Android Application and the Community's API we took advantage of Square's Retrofit Open Source Library. It simplifies the process of creating and managing network requests, handling responses, and processing data. Retrofit allowed us to define a Java interface that represents your API endpoints. Retrofit also allows integration with

various serialization libraries, the one used in this project is Google's Gson library. Retrofit also integrates well with other libraries in the Android ecosystem, such as JetPack Compose and Kotlin Coroutines, to handle asynchronous programming.

5.3 Distributed Ledger Technology

A distributed ledger is a database architecture that spans multiple interconnected computers or nodes. It offers a decentralized and transparent system for documenting and authenticating transactions or any digital data. In contrast to conventional centralized databases, which grant exclusive control to a single entity, a distributed ledger permits numerous participants to possess identical copies of the database and attain consensus regarding its contents.

To ensure unanimous agreement among system users regarding both the reputation system and shared network items, we drew inspiration from Distributed Ledger Technologies. Every network node assumes the role of preserving the system's state and storing a replicated ledger. This ledger functions as a comprehensive record of all transactions, including vouchers, "lending announcements", and user-generated "request messages" exchanged among participants. In order to counter impersonation attacks, a protective measure is implemented where the sender signs the hashes of all messages and includes them alongside the messages. Any signatures that are found to be invalid are subsequently discarded. The selected cryptographic hashing algorithm employed is SHA-256, while the signing process utilizes ECDSA with prime256v1 elliptic curve keys.

5.4 Android Security (Android Key Store and TEE)

In order to provide a secure system we need to protect sensitive data, prevent unauthorized access, and mitigate security risks.

The Key Store and Trusted Execution Environments (TEE) are important security features in the Android operating system that help protect sensitive data and ensure the integrity of critical operations.

The Android Keystore system provides a secure container for storing cryptographic keys, making them highly resistant to extraction from the device. Once keys are stored in the keystore, they can be utilized for cryptographic operations while ensuring that the key material remains non-exportable. Additionally, the keystore system offers the flexibility to impose restrictions on key usage, such as requiring user authentication for key access. For example, a user is required to authenticate within the last 2 minutes in order to be able to perform cryptographic functions.

The Android KeyStore provides the following security measures:

1. **Extraction prevention** - The application process does not handle any key material. Whenever an Android Keystore key is used for cryptographic operations within an app, the plaintext, ciphertext, and messages intended for signing or verification are sent to a system process responsible for executing the cryptographic operations. In the event of a compromise to the app's process, the attacker main be able to perform cryptographic operations but never extract the key material.
   The key material has the capability to be securely bound to the hardware components of an Android device, such as a Trusted Execution Environment (TEE)
2. **Hardware security module** - Android Devices (depending on the API Level) can have a StrongBox Keymaster or the Trusty TEE. The StrongBox hardware security module contains:
   (a) CPU
   (b) it's own Linux Kernel
   (c) Secure storage
   (d) mechanisms to resist package tampering and unauthorized

(e) random-number generator

(f) secure timers

This Hardware Module supports a subset of common algorithms and key sizes. In this project for example we are using the 256-bit prime field Weierstrass curve for Elliptic Curve Keys (also known as prime256v1).

In this project we took all the Security Recommendations and Best Practices defined in the Android Documentation such has securely storing and handling cryptographic keys and cryptographic operations.

5.5 Network Visualization Tool

One of our application's base foundation has always been lack of trust between users. We tackled this concern from the beginning with the use of our Vouch Reputation System (VRS) which lets us to create a network of trust for each user.

To take full advantage of these networks, it would be crucial to have a way displaying each of them to their respective user, showing its topology for the user to check not only each other user's trust level, but most importantly, where this level comes from. We reasoned our users would feel safer to trust someone if they knew exactly which friend vouched for and against them. To accomplish this, we developed our Network Visualization Tool.

Built on JavaScript, it runs on any web page, allowing it to be run on an android app as well. We used D3.js - a library that allows to create dynamic and interactive data visualizations. More precisely, we used D3's function forceSimulation() to create a directed graph simulation, with a node representing each user with their profile picture, and a link between two of them if they have a connection.

Each link will be either green or red, green if both of them vouched for each other, red otherwise.

Hovering (or clicking on mobile) on a node shows a tool tip with the user's name and trust level, while on a link shows a tool tip with each user's vouch message, if they left one.

Additionally, on a desktop, the Network Visualization Tool will also present a table with every user making part of the network, along with its trust value.

EMPRESTA.ME

Network Stats

- Alice - 90 %
- Bob - 82 %
- Chen - 94 %
- Dawg - 91 %
- Ethan - 87 %
- Frank - 93 %
- George - 85 %
- Hanes - 75 %
- Ivan - 99 %
- Juan - 96 %

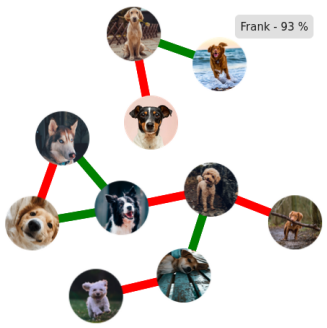


Figure 17: Example of the Network Visualization Tool

5.6 Authentication

Individuals that utilize our system and become members of a community must undergo an authentication process. The distributed nature of our system entails that many users may be connected interact with many different Community Servers. It is also important to note that some of these communities may be intended for members of an institution or organization, and for these communities it maybe be important to integrate different methods of authentication.

In the development of the authentication process in our system we took these issues into concern and tried to implement a system compatible with multiple authentication processes and allow for interoperability between Users and Community Server with different authentication methods.

Communication in the authentication process is all handled with API calls to the Community's endpoints. The process consists of 2 challenge–response authentication protocols and an association token/passphrase.

Challenge-Response: The challenge-response method is an authentication technique wherein a party presents a challenge to another party who must respond to the challenge in order to demonstrate their identity. The first party initiates the process by creating a random challenge and transmitting it to the other. The entity that received the challenge then utilizes their private key to sign the challenge and sends the signed challenge back to the issuer. The verification occurs when the first party validates the signature of the response with the seconds party's public key. This protocol validates the ownership of the public key pair for the challenged party.

Before the process begins the user must have access to the Server's Public Key. The user can get this public key either by ready the QR-Code issued and shared by the community or it request this information from a public endpoint in the community's server.

1. First Challenge-Response is used for the user to validate the Community Server's Public Key.
2. Association Process. In this method, servers with different authentication methods can validate that a user belongs to a community in different ways. This can be, for example, direct authentication by password or it can be by transmitting an authorization code corresponding to an Identity Provider. This process yields a association session token required by some privileged processes.
3. The Second Challenge-Response is used for the server to validate the ownership of a public key.

Only when all the steps in this process are completed can the user be register and authenticated in the community. For the purpose of demonstrating the versatility and interoperability of this authentication approaches we implemented two Community Servers with two different authentication methods. One uses direct approximation with a password or passphrase and the other uses the University of Aveiro Identity Provider.

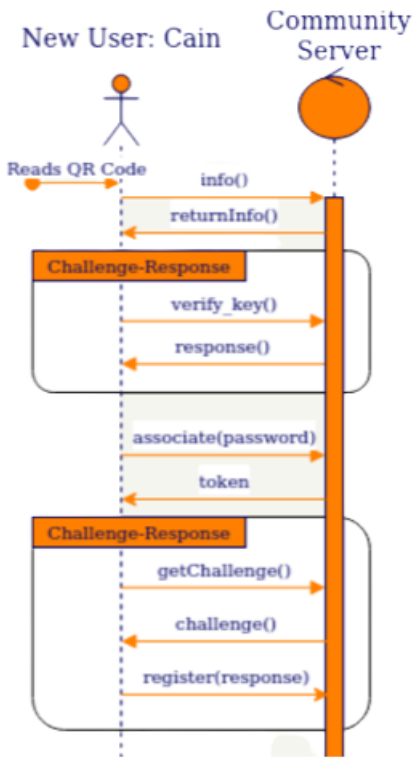


Figure 18: The authentication process

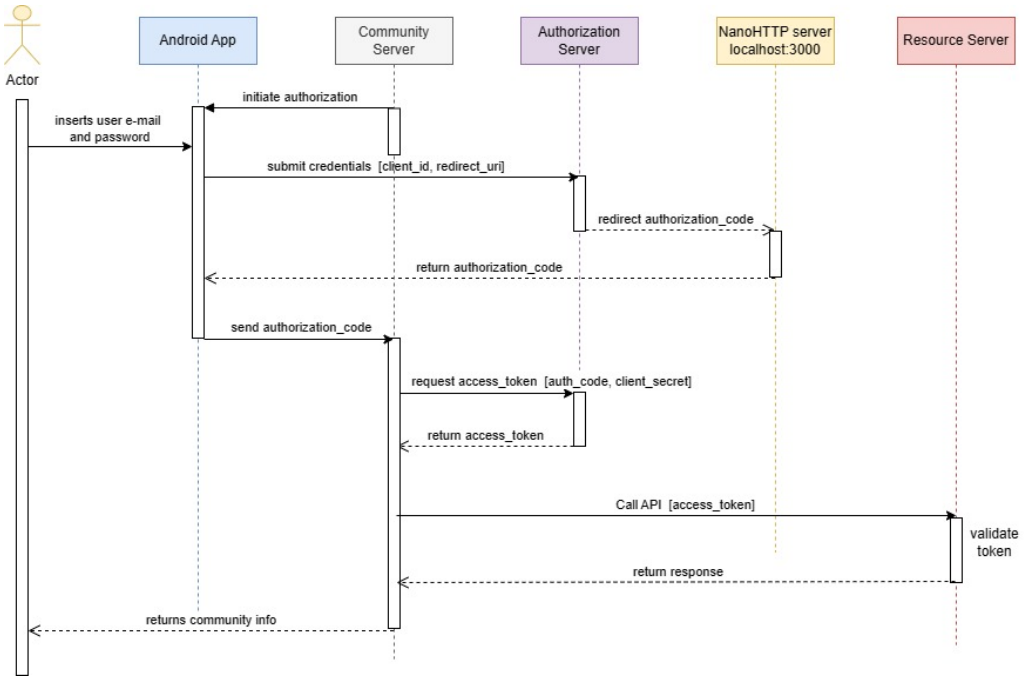
5.6.1 Authentication through the Aveiro University IdP

We have integrated an IdP into the EMPRESTA.ME application, utilizing a federated login method known as Single Sign-On (SSO). The IdP serves the purpose of enabling users to join a community, engage with fellow members, and actively participate in discussions and the sharing of physical assets. By implementing the IdP, specifically designed for the Aveiro University, we ensure that users can authenticate their identity and establish their affiliation with the university, thereby fostering a trusted and secure environment within the platform. By integrating an IdP associated with the university, the EMPRESTA.ME application can leverage the existing identity infrastructure already in place.

In addition to the integration of the IdP, as advised by our supervisor, our project also leverages the OAuth 2.0 framework to enhance the security and authentication process within the EMPRESTA.ME application. OAuth 2.0 is an industry-standard protocol that enables secure authorization and delegation of access to user resources.

In the Figure 5.6.1, we see the different phases of the authentication. When joining a community, the user must scan a QRcode published by the Community (for example "DETI"). After a successful scan, the user will be redirected to the UA service provider's authorization endpoint. There, they will be asked input their e-mail and password.

After that, the authorization server generates an authorization grant, such as an authorization



IdP Authentication Flow with OAuth 2.0 to allow users to join a community

code. This authorization grant is obtained through a GET request with the following parameters:

- **URL Authorization**: Authorization URL parameter
- **response type**: `code`
- **client id**: Client ID parameter
- **state**: Unique value used by the application to prevent Cross-Site Request Forgery (CSRF) attacks
- **scope**: Identifiers of the resources that the application is requesting access to
- **redirect uri**: Redirect URI parameter

These credentails were provided by our supervisor. After the authorization grant, the user is redirected to the URL specified **redirect uri** field. We did not ask for the STIC permission to use the IdP so we could only use "localhost:3000" as the **redirect uri**. To solve this, we created a server in this address that redirected the authorization code back to the Android App.

The next phase of the OAuth protocol is performing a handshake between the authorization code and access token. For this, the authorization code is sent to the community server and then the handshake can be performed. To be able to get the access token, we needed to specify the client secret owned by the community that is a client of the IdP provider. Finally, the access token needs to validated. When all steps are complete, users can finally see the community information and see the community inventory. By including the user's email address in the authentication and authorization process, the EMPRESTA.ME application can associate the user's identity with their community account.

5.7 Deployment

In order to facilitate widespread adoption of the community software, we have employed Docker to containerize the Flask application, as well as the Redis and RabbitMQ servers. This approach allows for streamlined deployment and enhances the scalability and portability of the

software, enabling seamless usage by a larger user base. Furthermore, Docker containers provide isolation, enhancing the security and stability of the software. Each component runs in its own isolated environment, preventing conflicts and reducing the impact of any potential vulnerabilities. Launching the Community server is a straightforward process, requiring only the presence of Docker and the execution of the designated image.

5.8 User Interface

As with any application, our goal was to provide to our users an interface as simple and intuitive as possible, hiding all the highly complex systems working behind the scenes.

Therefore, we followed Jakob's law - "Users spend most of their time on other sites" and designed our app to fall into the patterns of any modern mainstream app.

The app starts on the 'Login' page - Figure 19, where the user either registers a new account or logs in with theirs, which leads to the feed page (20), with the classical ways of finding an item: a search bar and scrollable lists of items which can be filtered by specific categories with buttons. Clicking on one of the items shows modal with the option to request the item. Requesting the item is achieved with a button that when clicked sends a notification to the lender, indicating the user's interest for this item. The notification can include an optional message that the user can write before pressing the button.

These type of notifications will be showed in the 'Notifications' page (24), which can be accessed by clicking on a bell button, located on the top right corner on the 'Feed' (20) and 'Profile' (23) pages. This page contains all the unanswered request notifications the user has received. Each notification is composed by the profile picture and username of the sender, the item being requested, the request message, if the user chose to write one, and two buttons, 'Accept' and 'Refuse' which set the response to the request.

Besides this main feed page (20), there are 3 more pages accessible, not exclusively, by clicking on the corresponding button on a navigation menu. This menu is included in every page except the login, is fixed in the bottom and has a button for the feed (19), show QR code (21), the network visualization (29) and profile pages (23), in this order.

The 'Show QR Code' page (21) fulfils the necessity of presenting the user's auto generated QR code, which can be scanned by other users on the 'Scan QR Code' page (22). The latter can be accessed by a button located conveniently below the QR code. This connection between the two pages is due to the fact that in order for two users to connect, they both need to vouch (either for or against) each other, so having a prominent link between them is a logical design choice.

The 'Scan QR Code' page (22) only has one function, using the user's phone camera (only after the user accepting giving permission) to scan QR codes. As stated before, it can scan another user's QR code, which will open the 'Vouch' page (28) and communities QR codes which will open the 'Join Community' page (25). We will explain the community related interfaces later.

The 'Vouch' page (28) is also very straight forward, it shows some information about the user about to be vouched and the vouch options. The information presented includes the username followed by the description and the public key. Bellow the user information, there is a bar for the user to, optionally, write a description justifying the vouch, followed by two buttons 'Vouch' and 'Disapprove' for vouching positively or negatively, respectively. Pressing either of these buttons registers the vouch in the system.

The 'Profile' page (23) is the last option on the navigation menu. As the name suggests, it shows some information about the user. To be more precise, the profile picture, username, description and public key, vertically presented on this order.

Going back to the community part, it all starts on the 'Join Community' page 25. This page is the entry point to any community, it is composed of the community name and description and a button to join it. Upon pressing the button, a challenge of choice of the community may be presented to the user. It can be either a password (set up by the community) or a redirect to an

IdP to login with certified credentials.

After joining a community, the user will have access to the 'Community Feed' page 26, which is just like the normal feed, but for items exclusive to that community, with the added feature of the 'Create Your Post' button. This button opens a modal 27 in which the user defines the post, writing a Title, Description and, optionally, selecting an image from the phone photos (after the user has given permission for the app to access their photos) when the 'Select an Image' button is pressed. The post will fall onto two types, either a request or a possession, which will be chosen by clicking on the 'I need a...' or 'I have a' button, respectively. After the all the necessary information is given, the user is able to submit the post, clicking on the 'submit' button.

After joining a community and vouching for someone, clicking on the third button of the navigation bar will open the Network Visualization page 29. Here the use has the view of the topology of their trust network, in which there are all the users they have vouched, and the those that these other users have vouched and so on. Each one of the users is a node connected to other users they have vouch or have been vouch by, being this connection green if both vouched between the two of them were positive, and red otherwise. The graph is interactive, so the user is free to move the nodes around, for a better visualization.

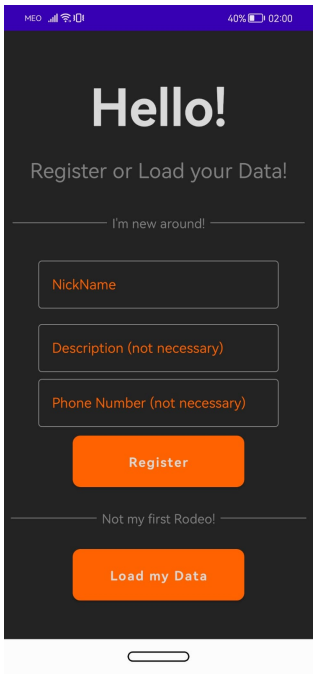


Figure 19: login

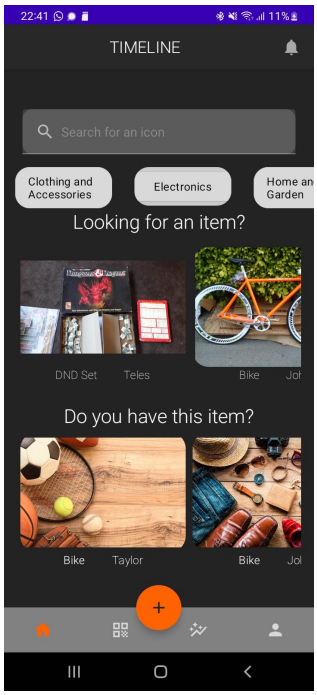


Figure 20: feed

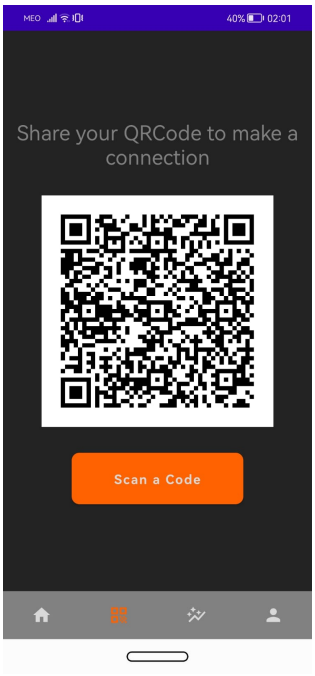


Figure 21: show QR code

[3]Note: In the figure 22, it appears as not able to parse QR Code because if we point the camera to a working QR Code, it immediately scans it and opens the 'Vouch' page. For demonstration purposes, we purposely scanned a non-functional QR Code, in order to showcase the functionality.

Figure 22: scan QR code [3]

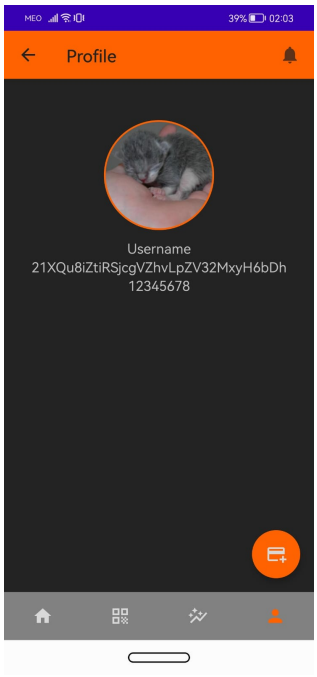


Figure 23: profile

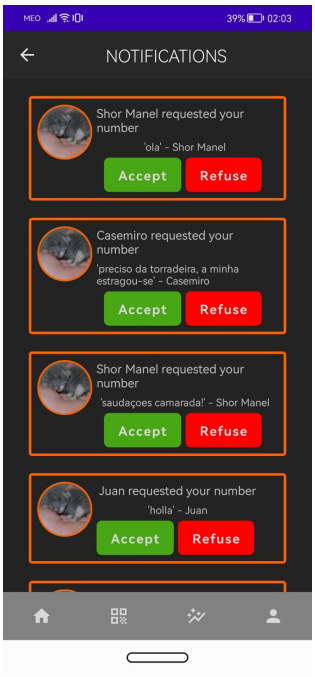


Figure 24: notifications

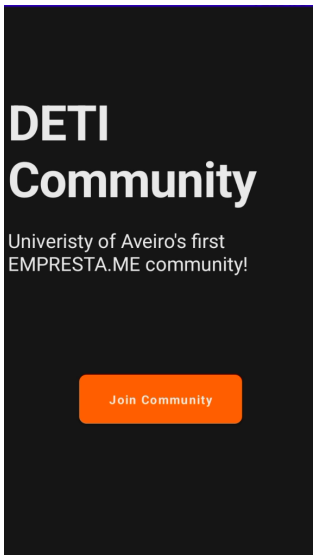


Figure 25: join community

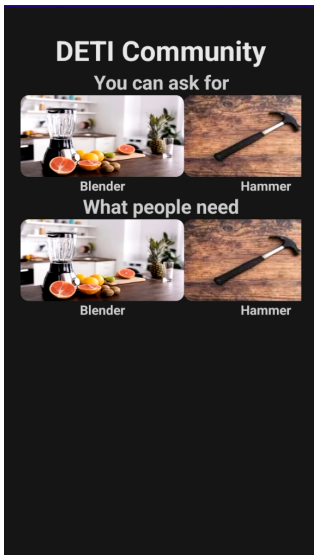


Figure 26: community feed

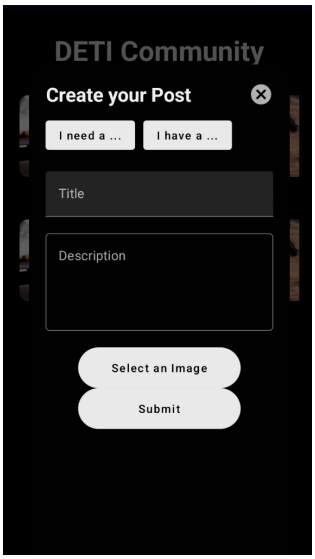


Figure 27: create post

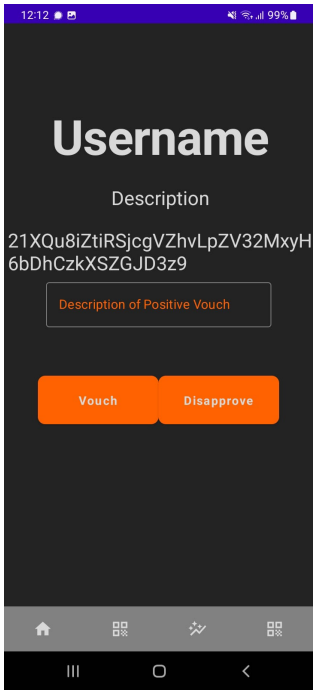


Figure 28: vouch

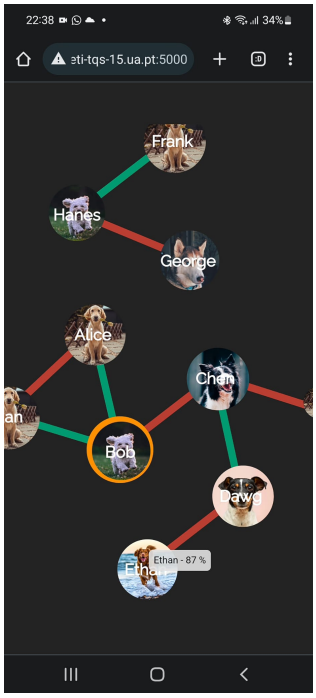


Figure 29: network

6 Security Concerns

6.1 Vouch Reputation System

6.1.1 Vouch Omission and Vouch Fraud

Say that Alice vouches for Bob. That is to say that she created a message vouching for Bob which is forwarded along the network to others. After some time, Bob behaves in some way that warrents disapproval by others in the network. Many users vouch against Bob, which significantly decreases his overall reputation. This, in turn, means that Alice is also negatively affected with their association with Bob so she chooses to retract her vouch.

For overall network health, it is important that users are held accountable not only for their current vouch relations, but their past ones as well. Otherwise, bad actors could band together without repercussion, as if one of them gets caught the rest can simply retract their vouches and be unaffected. Instead, a user's past vouches should play some role in it is reputation calculation.

Now let's say that a new user, Charlie, enters the network for the first time after the Bob fiasco. As he does not know anything yet, he needs to ask around for the vouches each node has in order to construct the trust topology. When asked to present her vouch history, Alice might be inclined to not let Charlie know about her past association with Bob as a way of preserving her reputation. If she sends a tampered version of her vouch history with that message missing, her reputation from Charlie's point of view may be artificially higher, misleading Charlie to trust Alice more than he might have had he known the full picture.

To prevent this scenario from happening, the first step is to make it so nodes do not just store the current vouch connections of the nodes they care about, but also have their previous vouches backed up. This can not be enforced, as users can always clean their cache to save disk space, but as long as at least one node within the victim's reach has the ommited message backed up the system can recover from vouch omission attacks.

Secondly, one should never ask a particular node about their vouch history. Instead, they should ask publicly about it so that anyone can chime in. This way, Charlie would ask anyone to provide for vouch messages made by Alice. Even if Alice omits her message vouching for Bob, another user that has it cached can interject and supply Charlie with the missing message. Alice can not repudiate the authenticity of the message as it contains a digital signature signed by her.

6.1.2 Impersonation Attacks

One way an adversary might attack the system is by impersonating another user. Let us explore ways this attack vector can be mitigated.

Every entity should possess a asymmetric key pair as a means of authentication. In fact, the public key of this key pair is what identifies a node in the network. Every message created by a node (e.g. vouch messages) is to be signed with their private key. As such, nodes can verify the authorship of received messages through the means of digital signatures. Messages with missing or otherwise invalid signatures should be dismissed.

Asymmetric cryptography solves the issue of impersonating an entity's in-system identity. However, it does nothing to address the matter of associating a real life identity to a in-system identity.

One way this can be solved it through certificates, perhaps in a decentralized fashion like in PGP, where nodes can certify others of their real identity and those that trust them can choose to listen. There might also be scenarios in which tying real life identities to a in-system identity is undesirable and discretion is preferred.

6.1.3 Vouch Forgery

As discussed previously, vouch messages ought to be signed by their issuer. As such, it is not possible to forge a vouch message without knowing the would be victim's private key, which of course should be kept secret.

6.1.4 Proof of Work

Proof-of-Work (PoW) technology emerged as a consensus algorithm within the Bitcoin cryptocurrency. Its primary objective is to authenticate and validate transactions, preserve the integrity of the blockchain, and establish consensus among network participants. PoW operates by imposing a requirement that computational puzzles involving hash functions must be successfully solved for each blockchain operation.

EMPRESTA.ME deviates from the use of blockchain as a distributed ledger, opting for a custom solution that carries less stringent demands. The relaxation of requirements allows for consensus among network participants to be established through digital signatures and gossip communication, rendering Proof-of-Work (PoW) unnecessary in this context. Nonetheless,1 PoW continues to find application within EMPRESTA.ME, albeit not as a consensus mechanism. Instead, it serves as an effective deterrent against Distributed Denial of Service (DDoS) attacks. PoW mandates computational effort from users as a prerequisite for accessing a service, establishing a deterrent against malicious actors. Valid users can present evidence of their computational work, whereas potential attackers are confronted with the daunting task of expending substantial computational resources to execute an attack. This PoW implementation effectively mitigates DDoS threats, safeguarding the system's availability and stability.

7 Discussion

In the beginning of this project we defined the system requirements and devised the operational framework for our reputation system. Our initial system proposition entailed creating a fully distributed and decentralized solution akin to BitTorrent, while integrating it with our reputation system. Several challenges emerged from our initial architectural design, and upon consultation with our tutor, we recognized that our proposed solution was unattainable due to the prevalent use of Network Address Translation (NAT) in most IPv4 networks. The problem arose due to the instability of computers' IP addresses within networks, leading to an inability to establish communication between computers in different private networks. Furthermore, even if it were technically feasible, implementing it would entail significant complexity.

To uphold our initial requirements and preserve the distributed nature of the system, we introduced Community Servers. These servers primarily served as communication brokers. While the overall system architecture remained largely unchanged once finalized, the approach to user communication underwent multiple re-considerations throughout the project.

Even thought our endeavors yielded a moderate level of success in the respective tasks we undertook we remain dissatisfied with the outcome. The following are notable constraints and potential areas for improvement in our system implementation:

1. The original intention was to process the visualization of the reputation system's network natively on the Android platform. However, we made the decision not to implement this approach due to the significant complexity involved in performing such extensive graphical computation with android frameworks. As an alternative approach, we implemented the community server to serve this functionality, aiming to achieve an aesthetically pleasing outcome. This situation has undesirable implications as it increases reliance on the community server. In the event of a compromise to the server, there is a risk of an attacker attempting to present an altered representation of the reputation system, aiming to exploit it for their own advantage.

2. We encountered a challenge related to the broker technology utilized in our system. The community broker implements persistent queues, which necessitates reprocessing all data in the queue upon application startup, regardless of whether it has been previously encountered. Based on our testing, we have not encountered any issues. However, in the event that the network expands to a significant scale, the computational requirements for processing these messages will experience exponential growth. One possible approach would have been to develop a broker from the ground up, or alternatively, to explore alternative technology brokers that align more effectively with our specific requirements.

Despite undergoing numerous modifications throughout the project, we successfully developed a proof-of-concept that aligns with the initial essential requirements. However, it is crucial to note that the system necessitates additional deliberation and is not currently ready for release as a marketable product. Nonetheless, the proof-of-concept offers valuable insights into the potential and benefits of incorporating compact and secure distributed systems within communal environments. The application's emphasis on security and privacy safeguards user data and transactions, instilling confidence in the system. As we move towards a more collaborative and resource-efficient future, this innovative application paves the way for the establishment of thriving sharing economy.

References

[1] Y. Holtz, “Network Graph | the D3 Graph Gallery,” d3-graph-gallery.com. https://d3-graph-gallery.com/network.html.

[2] “openid/AppAuth-Android,” GitHub, May 30, 2023. https://github.com/openid/AppAuth-Android.

[3] André Zúquete, "OpenID Connect (OAuth 2.0 + Autenticação federada UA)" Documentation (provided by our supervisor).

[4] “Desenvolvedores Android,” Android Developers. https://developer.android.com/?hl=pt-br.

[5] “Python guide,” Redis. https://redis.io/docs/clients/python/).

[6] “BitTorrent Protocol 1.0” www.bittorrent.org. https://www.bittorrent.org/beps/bep_0003.html.

[7] S. Nakamoto, “Bitcoin: a Peer-to-Peer Electronic Cash System,” Oct. 2008. Available: https://bitcoin.org/bitcoin.pdf.

[8] “Living Whitepaper - Nano Documentation,” Nano Docs. https://docs.nano.org/living-whitepaper/.

[9] H.-Y. Chien, “Dynamic Public Key Certificates with Forward Secrecy,” Electronics, vol. 10, no. 16, p. 2009, Aug. 2021, doi: https://doi.org/10.3390/electronics10162009.

[10] “bouncycastle.org,” www.bouncycastle.org. https://www.bouncycastle.org/.

[11] “Retrofit,” Github.io, 2013. https://square.github.io/retrofit/.

[12] “Hilt,” dagger.dev. https://dagger.dev/hilt/).

[13] “Save data in a local database using Room | Android Developers,” Android Developers, 2019. https://developer.android.com/training/data-storage/room.

[14] “Trusty TEE,” Android Open Source Project. https://source.android.com/docs/security/features/trusty

[15] B. R. Moura, “Vouch Reputation System,” GitHub, May 15, 2023. https://github.com/BrunoRochaDev/Vouch-Reputation-System).

[16] “Documentation: Table of Contents — RabbitMQ,” www.rabbitmq.com. https://www.rabbitmq.com/documentation.html.

A Backend

A.1 API Documentation

This section provides details about the API endpoints available in the EMPRESTA.ME platform.

B Code repositories

The code repository can be found here: https://github.com/Empresta-me.

C External Communication Website

The documentation website can be found here: https://empresta-me.netlify.app/.

C.1 Documentation

Below there is the Postman Documentation (json) for the API endpoints of the project.


```
{
   "info": {
 "_postman_id": "6f44eace-7713-4e99-8654-0a2d81d18429",
 "name": "EMPRESTA.ME Docker",
 "schema": "https://schema.getpostman.com/json/collection/v2.1.0/collection.json",
 "_exporter_id": "23776688"
  },
  "item": [
 {
     "name": "meta",
     "item": [
   {
       "name": "info",
       "request": {
     "method": "GET",
     "header": [],
     "url": {
         "raw": "http://127.0.0.1:5000/meta/info",
         "protocol": "http",
         "host": [
       "127",
       "0",
       "0",
       "1"
         ],
         "port": "5000",
         "path": [
       "meta",
       "info"
         ]
     }
       },
       "response": []
   },
```

```
  {
      "name": "verify_key",
      "request": {
    "method": "POST",
    "header": [
        {
      "key": "challenge",
      "value": "FyPXEsUKKmCkRH6YPcEYVr",
      "type": "default"
        }
    ],
    "url": {
        "raw": "http://127.0.0.1:5000/meta/verify_key",
        "protocol": "http",
        "host": [
      "127",
      "0",
      "0",
      "1"
        ],
        "port": "5000",
        "path": [
      "meta",
      "verify_key"
        ]
    }
      },
      "response": []
  }
    ]
},
{
    "name": "auth",
    "item": [
  {
      "name": "associate",
      "request": {
    "method": "POST",
    "header": [
        {
      "key": "password",
      "value": "batatinhas123",
      "type": "default"
        }
    ],
    "url": {
        "raw": "http://127.0.0.1:5000/auth/associate",
        "protocol": "http",
        "host": [
      "127",
      "0",
      "0",
      "1"
        ],
        "port": "5000",
        "path": [
```

```
        "auth",
        "associate"
          ]
      }
        },
        "response": []
    },
    {
        "name": "challenge",
        "request": {
      "method": "GET",
      "header": [
          {
        "key": "token",
        "value": "N6HpbRUxJ5ugao1krSt8o1",
        "type": "default"
          },
          {
        "key": "public_key",
        "value": "26HuBkrKLLMYTdDQ7ixsHu5PtznuecmAfZnG6PqmUm4PE",
        "type": "default"
          }
      ],
      "url": {
          "raw": "http://127.0.0.1:5000/auth/challenge",
          "protocol": "http",
          "host": [
        "127",
        "0",
        "0",
        "1"
          ],
          "port": "5000",
          "path": [
        "auth",
        "challenge"
          ]
      }
        },
        "response": []
    }
      ]
  },
  {
      "name": "acc",
      "item": [
    {
        "name": "register",
        "request": {
      "method": "POST",
      "header": [],
      "body": {
          "mode": "raw",
          "raw": "{\n    \"public_key\" : \"ndShu87QAz6cxEhFN2arSuKRmY9A848mMqwKnQYVuMwj\",\n    \"alias\"
          "options": {
        "raw": {
```

```
        "language": "json"
      }
        }
    },
    "url": {
        "raw": "http://127.0.0.1:5000/acc/register",
        "protocol": "http",
        "host": [
      "127",
      "0",
      "0",
      "1"
        ],
        "port": "5000",
        "path": [
      "acc",
      "register"
        ]
    }
      },
      "response": [amongus]
  },
  {
      "name": "login",
      "request": {
    "method": "POST",
    "header": [
        {
      "key": "public_key",
      "value": "26HuBkrKLLMYTdDQ7ixsHu5PtznuecmAfZnG6PqmUm4PE",
      "type": "default"
        },
        {
      "key": "response",
      "value": "iKx1CJMJknJWhtbAzFAYThv75YTnvFuF1iNBi3RXLfa2FMYBRWQYxMJyxNHTWwMZ8Q7eig3DGK7BpyK7XArgeKDB
      "type": "default"
        }
    ],
    "body": {
        "mode": "raw",
        "raw": "",
        "options": {
      "raw": {
          "language": "json"
      }
        }
    },
    "url": {
        "raw": "http://127.0.0.1:5000/acc/login",
        "protocol": "http",
        "host": [
      "127",
      "0",
      "0",
      "1"
        ],
```

```
          "port": "5000",
          "path": [
        "acc",
        "login"
          ]
      }
        },
        "response": []
    }
    ]
  }
    ]
}
```