DECEMBER 2022

ToEaseParrot

A Decentralized Timeline

GROUP 12
BEATRIZ SANTOS (UP201906888)
JOÃO BARRA (UP201808910)
MARIA CARNEIRO (UP201907726)
SÉRGIO DA GAMA (UP201906690)

Problem Assumptions

Besides the problem requirements, we assumed that:

- All peers are on the same network
- Users can always view other users' timelines when accessing their profiles.
- A user is responsible for saving and providing the data of the users it follows (their followers, followings, and timeline)
- Upon logout, the user session ends but its node is still connected
- Only each user has access to their own private key

Technologies

React

Used for the frontend, to help visualize our system.

NodeJs

Used for the backend, simple and compatible with the library chosen.

Libp2p

Used on the backend, to implement the decentralized system.

Peer Communication

We used TCP to ensure user communication as it guarantees reliability, flow, and congestion control.

Stream Multiplexing were used to transmit user data between multiple peers simultaneously.

Peer Discovery

We used Multicast DNS so that peers can discover each other, due to its simplicity and efficiency.

Although, it has some security issues, because the mDNS traffic is not encrypted and it can expose IP addresses and hostnames.

DHT

We used a Distributed Hash Table - Kademlia DHT - to ensure content discovery between peers and to manage authentication since it guarantees efficiency, scalability and security.

Authentication

We used RSA to manage authentication and the DHT to save registered users with their respective public keys. Upon login, users must have the correspondent private key to access the system.

The only way to fully guarantee security would be to sign all the requests with their private key and always verify it against the respective public key in the DHT.

However, as this wasn't the main focus of the work, we decided to make a simplified version, where we only do this verification in the login, opening a session for that user with no verification on further requests.

Publish/Subscribe

We used GossipSub to apply the publish and subscribe pattern to the logic of our system: any action a user performs must be known by the other users. Those actions can be POST, FOLLOW, or UNFOLLOW.

GossipSub guarantees integrity in messages and is more efficient and scalable since each node only needs to maintain connections with a small number of its neighbors in order to receive messages from the entire network.

Local Storage

We used PostgreSQL to save the system's data locally, in order to add another layer of persistence and reliability, in case any providers can't be dialed. The data is persisted when any message is received so that local data is always in the most updated state.

The system works without local storage, but it is helpful to guarantee that even if every node is down, the present one can still retrieve its state.

Timeline Flow

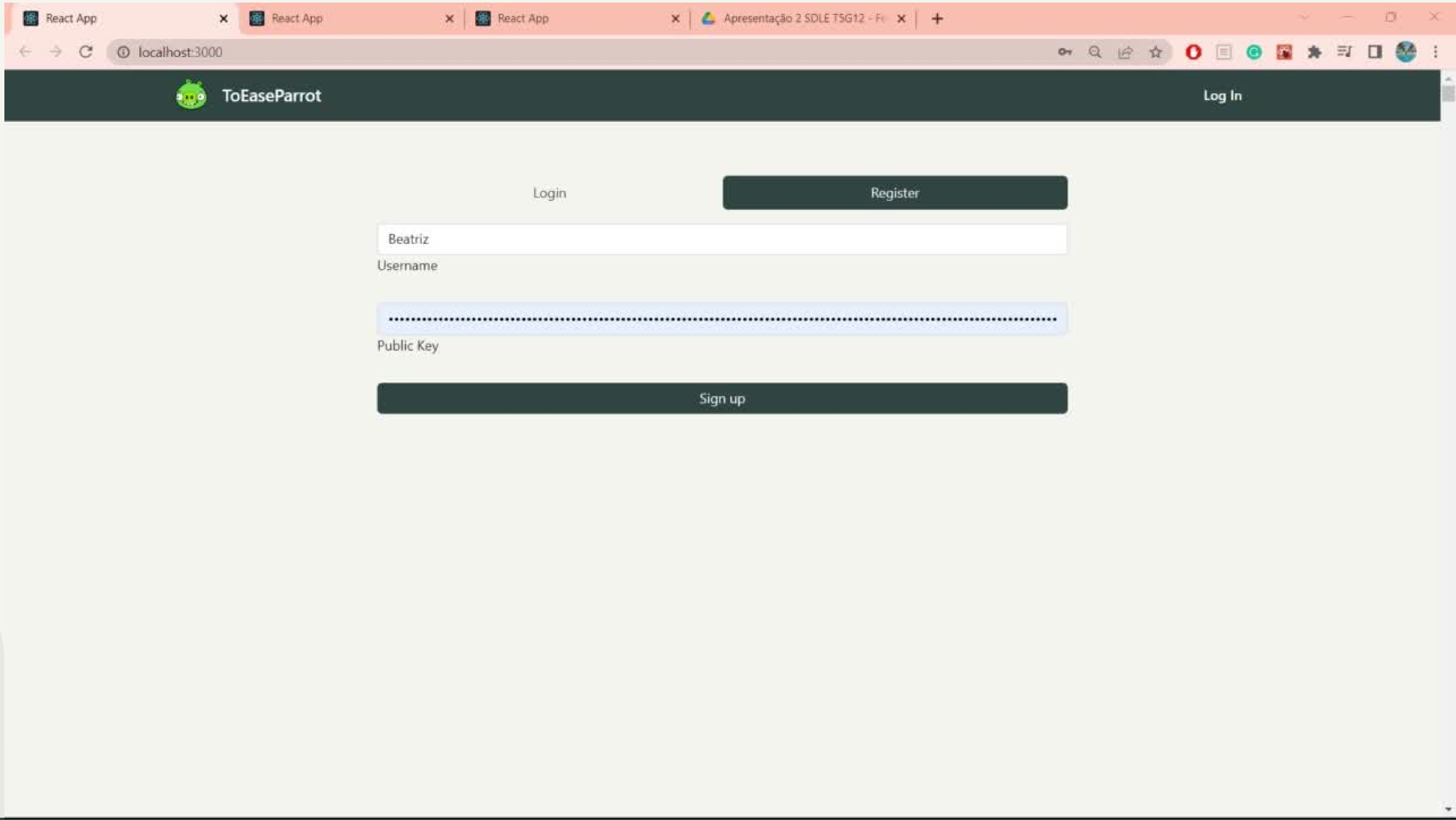
1. User X follows user Y
2. User X subscribes to the topic corresponding to user Y
3. User X loads information about user Y from its providers or the local storage
4. User X announces it now follows user Y
5. User X becomes a provider for user Y

The feed for user X is updated with the content of user Y and user X is responsible for holding user Y's data, so the system ensures persistence in case user Y goes down.

Limitations & Challenges

- Libp2p doesn't have an unprovide method, so once a user follows another, it will always be subscribed to the other user, so we have no "bad" providers
- Lack of documentation of Libp2p
- Using RSA keys to register and login guarantees security but is not practical or user friendly

Demo



Beatriz

Username

Public Key

Questions?