Distributed Timeline

Large Scale Distributed Computing, Project 2

Duarte Sardão, up201905497
Henrique Sousa, up201906681
Mateus Silva, up201906232
Melissa Silva, up201905076

Table of Contents

01

Introduction

The specification of the project and our initial approach.

Specification

- The specification was simple, thus we will keep this short;
- At its core, the project should be a decentralized timeline using P2P communication and edge devices;
- Users should have an identity and be able to post short messages and subscribe to other users, forming their local timeline;
  - Remote content is available when its source user is online to forward it;
- Remote content can be made ephemeral - i.e., available for a limited amount of time;
- Choice of technology was free;

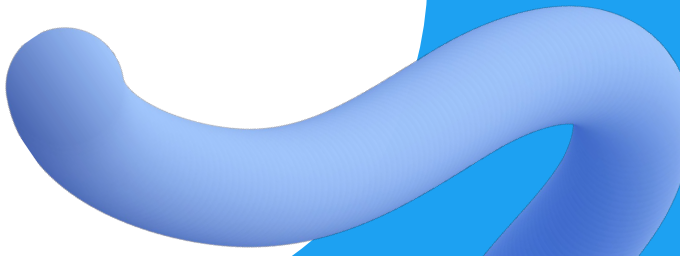
Our Approach

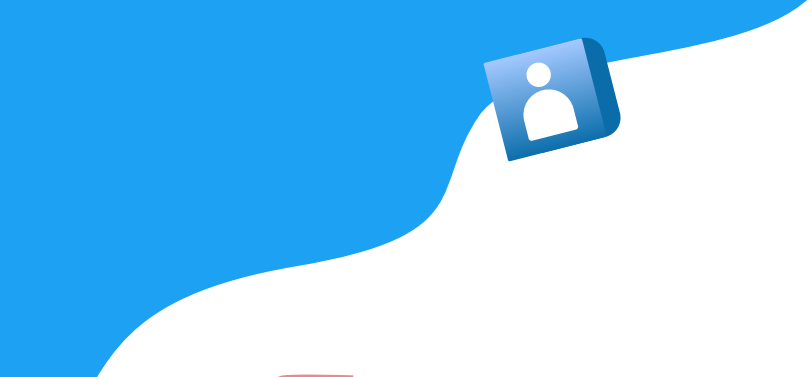
- We chose to use Python for our program, with added use of Vue (HTML, CSS + JS) for a frontend implementation;

- Our program includes 3 main components:
  - ***Frontend***: visual, user-friendly realization of the full program;
  - ***Backend*** (using FastAPI - also Python): a collection of endpoints the frontend communicates with;
  - ***Program***: the actual implementation of the program;

- We envisioned having ephemeral content, which was optional but we found interesting;

- ***Backend*** serves as *middleware* between the frontend and our program, while also enforcing our (simplified) authentication method;





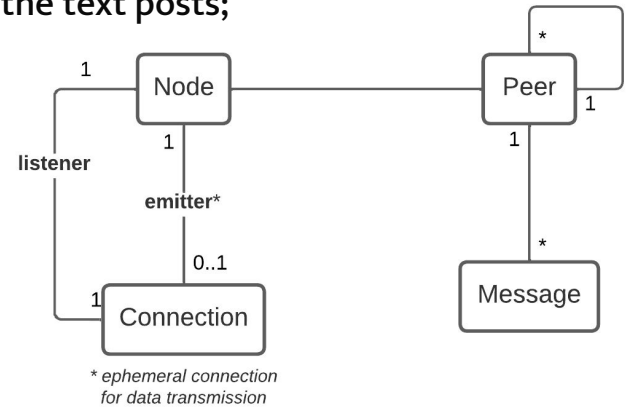
02

Implementation

How our program works and why.

Architecture

- Our program has four main classes:
  - *Node:* implementing the decentralized server functionalities;
  - *Peer:* to represent the user;
  - *Connection:* that connects users;
  - *Message:* representing a message sent through an established connection;
- Messages have a few distinct types, considering the possible actions from a user actions: subscription and unsubscription (which we dubbed *following* and *unfollowing*);
- User credentials are saved on the backend database, along with the text posts;
- Our middleware ensures *Kademlia* can access whatever it needs to send between users;
  - *Asyncio* is also used for running synchronous functions in an asynchronous manner.
  - We won't be going into detail about how the middleware works, as it's superfluous to the actual program;
    - Just know it is real-time updated;



Functioning

1 Initiation of Peers

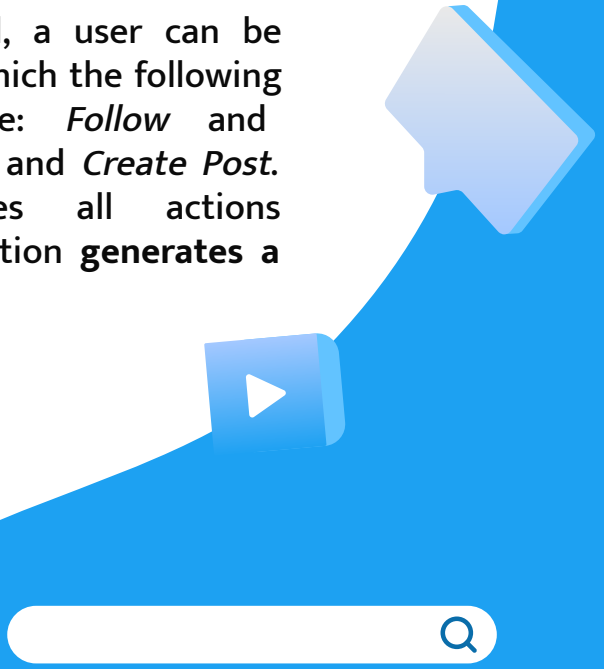
A peer is initiated with a numeric ID and associated to its **local** backend instance. This includes the creation of a *Node* object, a database and a server. From this point on, it's listening for potential messages.

2 Executable Actions

Through the frontend, a user can be authenticated, from which the following actions are available: *Follow* and *Unfollow*, *User Search* and *Create Post*. Authentication makes all actions available, and each action **generates a new message**.

3 Message Reception

A message can be received from other peers. In the lifetime of the program, these messages can be: **Follow**, **Unfollow**, **Share Feed**, **Logout**, **Get Posts**, **Mutual Follower Online**, **Follower Online** and **Following Online**.



4 Message Handling

Each *Node* includes a handler capable of dealing with all types of messages - it uses a different thread for each message correctly received and process it.

4 a) Following (a user)

Upon receiving a following message, the peer starts by checking if the user provided (within *Message*) is not itself.

Then it updates its local info and database - an entry is created for the given account in case it doesn't exist already.

4 b) Unfollowing (a user)

The behavior upon an *Unfollow* is analogous to *Follow*, with the difference that there's no need to create entries, as these should already exist.

4 c) Share Feed

Share feed sends the local feed to remote users. An event stream is used to update the frontend in real-time.

4 d) Creating a Post

Beyond saving the post on the database, the post is sent to all of a user's followers (*Share Feed*).

4 e) Follower Online

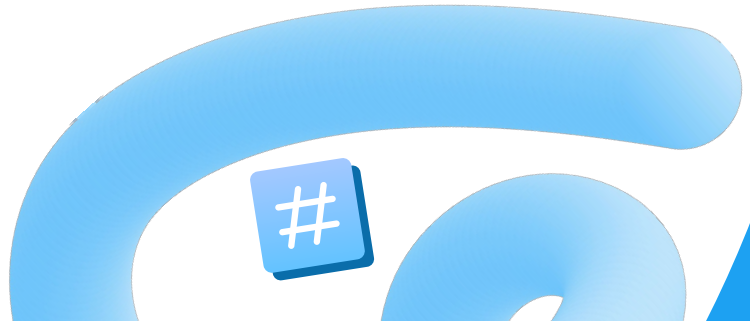
A logged in user sends a message to all the people they follow to then receive an updated version of their posts.

4 f) Following Online

A logged in user sends a message to all the peers they are followed by, letting them know they can now request post updates.

4 g) Mutual Follower Online

A logged in user, after checking in *Kademlia* that a peer they follow is *offline*, will resort to their followers as a source to receive an updated version of remote posts.



4 f) Get Posts

A message sent as a request of a user's posts - upon receiving a message, the receiver returns the posts they made after the date specified in the message.

4 h) Logout

A logged out user sends a message to *Kademlia* in order to update its information - its ***status***, from *online* to *offline*.

5 Update

All actions and messages have effects on the frontend. How this works will be shown later.

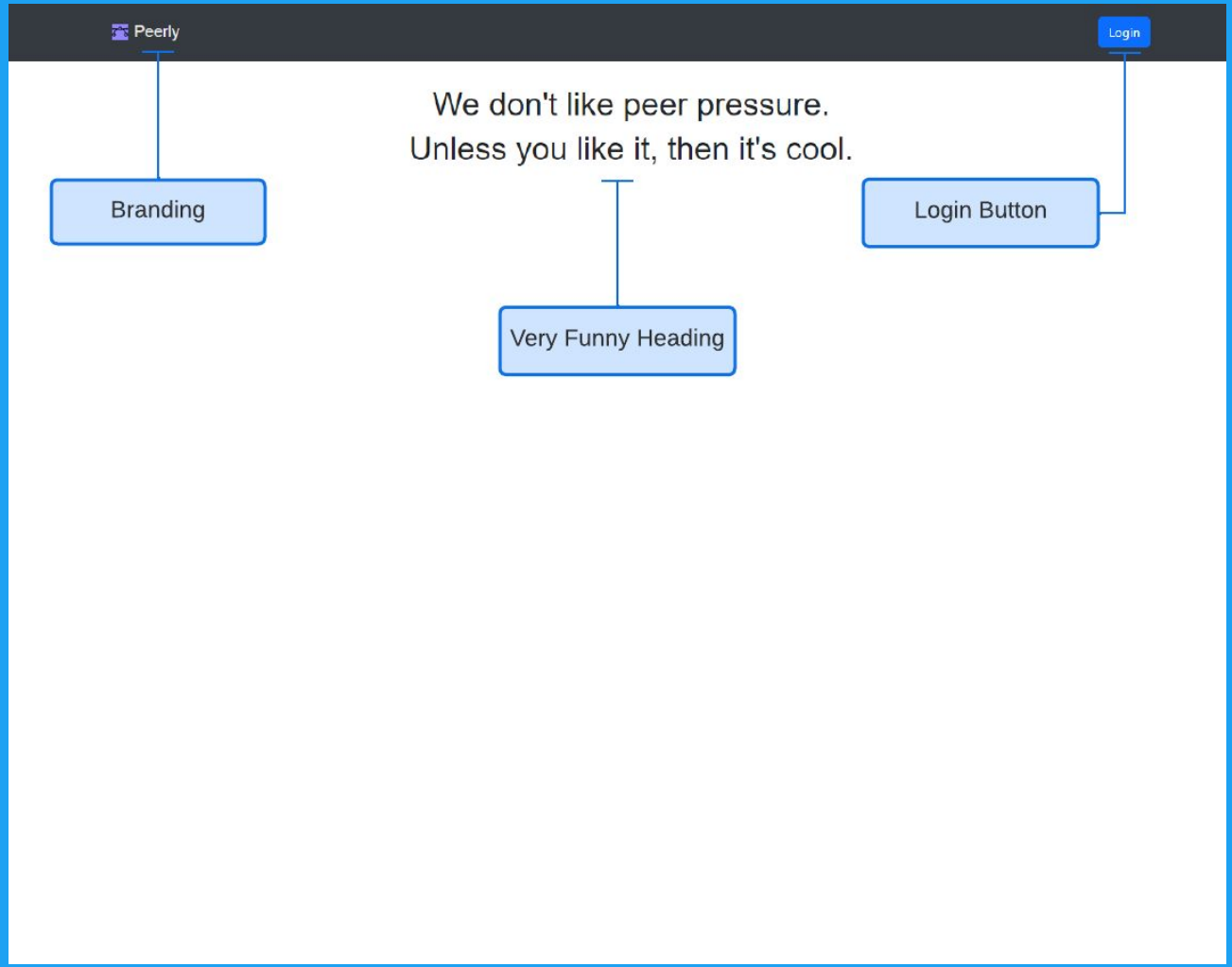
...and repeat from 2 for as long as you like!

NOTE: Remote content is deleted after 1 minute.

03

Showcase

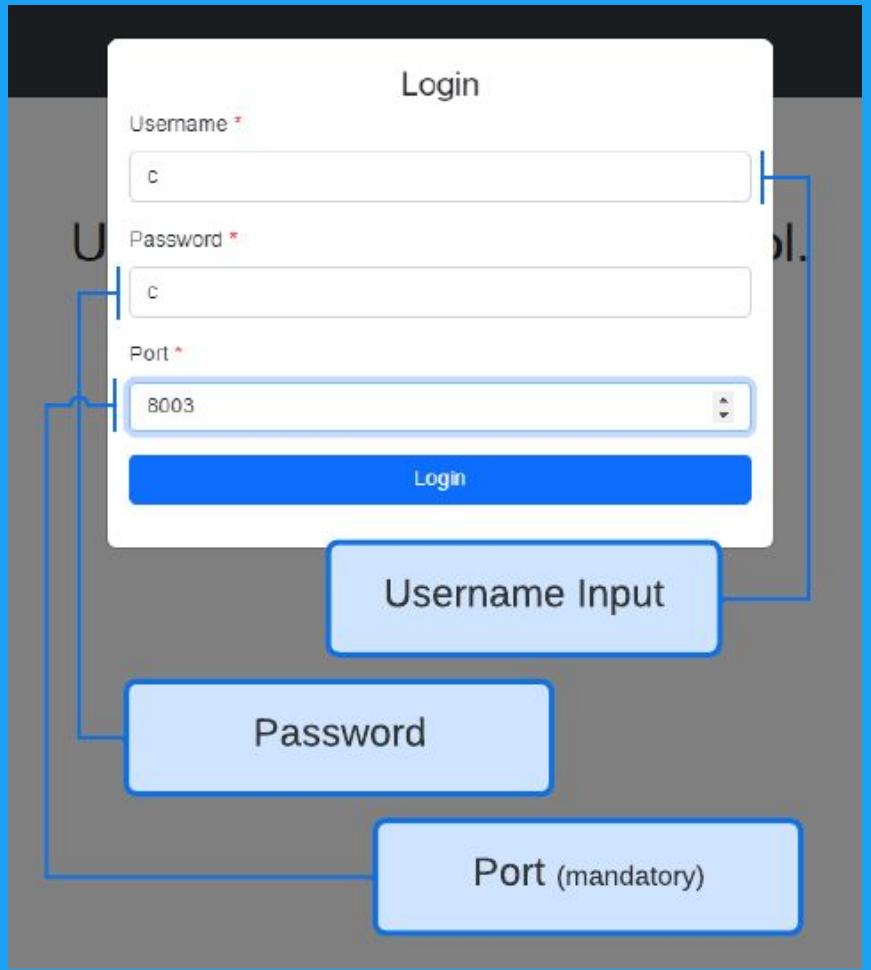
Let’s see our program working!


Peerly
Login
We don't like peer pressure.
Unless you like it, then it's cool.
Branding
Login Button
Very Funny Heading

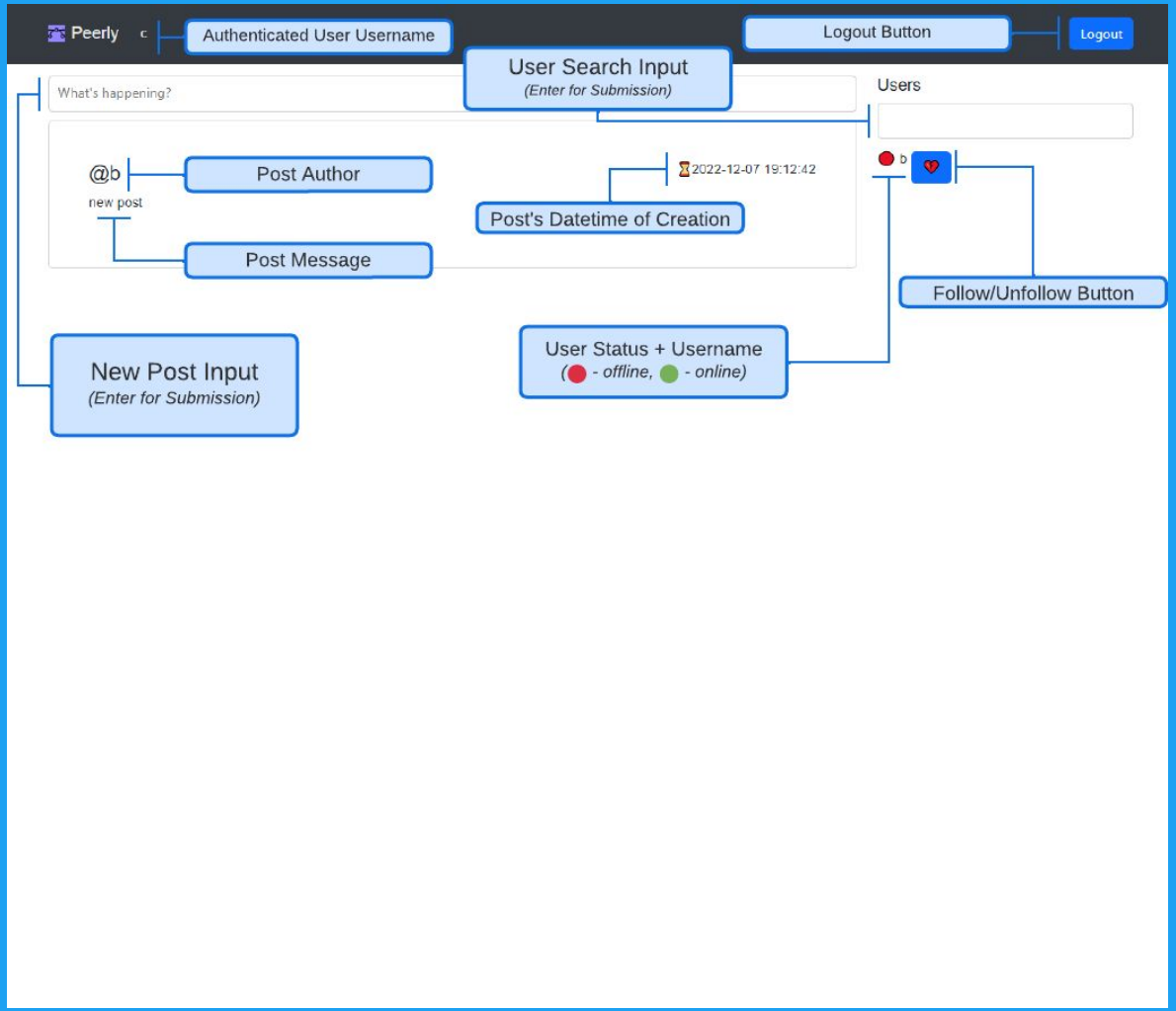
HOMEPAGE

Login

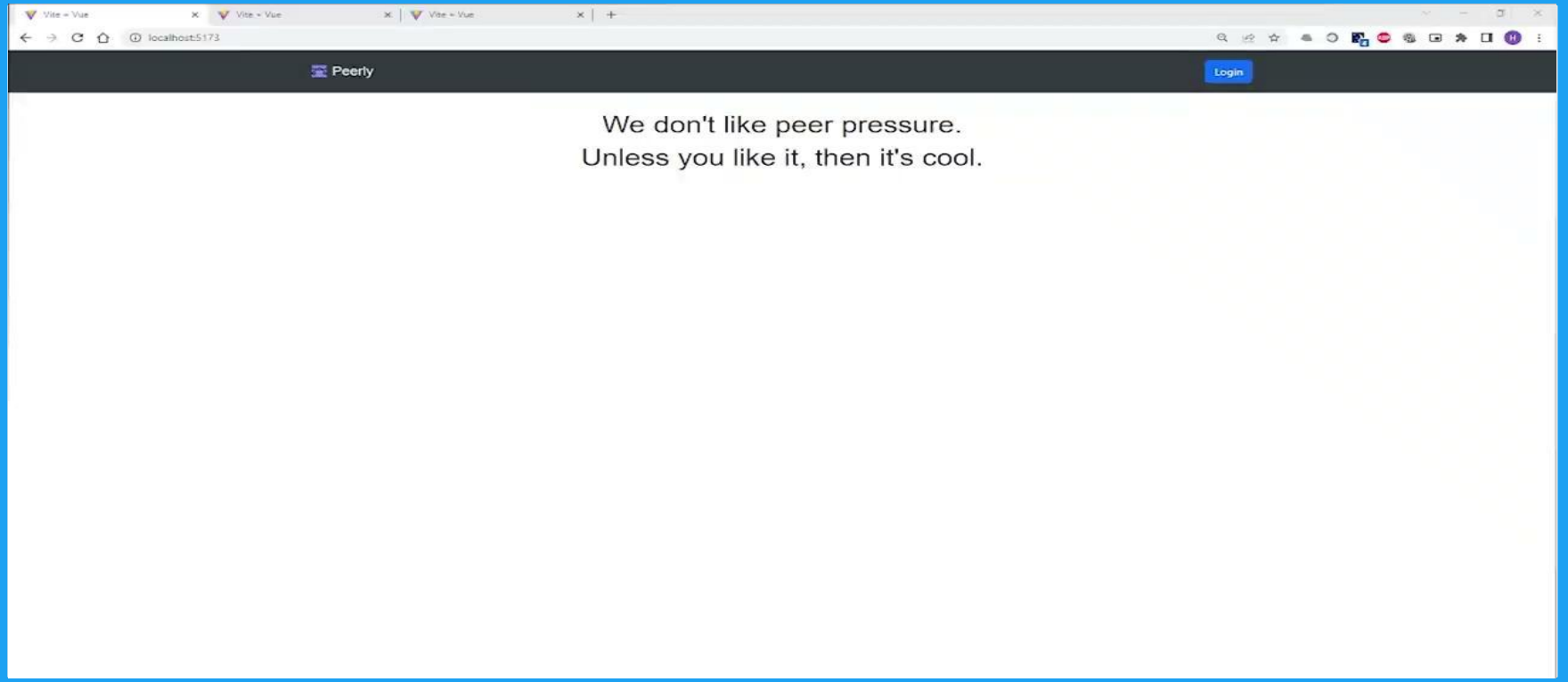
For authentication.



TIMELINE


Peerly
Authenticated User Username
Logout Button
Logout
User Search Input
(Enter for Submission)
What's happening?
Users
@b
new post
Post Author
2022-12-07 19:12:42
Post's Datetime of Creation
b
Post Message
Follow/Unfollow Button
New Post Input
(Enter for Submission)
User Status + Username
(- offline, - online)

Demonstration Video



04

Conclusions

Closing thoughts and reflections.

Conclusion

- Our approach ended up being successful in realizing the decentralized timeline;
- We were able to implement an ephemeral timeline, which added an optional (but interesting) challenge for us;
- We spent some time in doing a minimalistic frontend that hopefully allows any person to easily understand what the program is going through;
- Some difficulties were:
  - Implementing Kademlia was incredibly difficult as we found its documentation to be mostly non-extensive;
  - While superfluous to the request features, the connection between all of the components of our project took us a while, but ended up being worth it;
  - Getting real-time updates of an user's state on a local database was difficult;
- We are proud of the work we've made and believe we finished with a robust program that functions according to the specifications;