Decentralized Timeline

Large Scale Distributed Systems, Second Assignment

Alexandre Abreu, up201800168
Diana Freitas, up201806230

Juliane Marubayashi, up201800175
Simão Lúcio, up201303845

Introduction

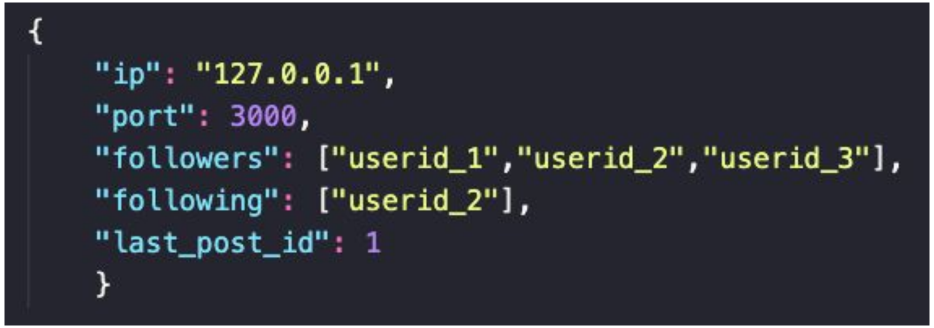
The solution was implemented using **Python 3**.

For Peer lookup in the network, a DHT was used; in this specific case, **Kademlia**'s Python 3 implementation.

In order to provide an entrypoint to the network, there is a predefined list of **bootstrap nodes** that can be started (at least one is needed).

Each Peer is represented in the DHT by the username and the information stored is related to the Peer's network address, as well as other structures that support the overall functioning of the solution. The Peers communicate using the socket implementation wrapped by the **Asyncio** library and using the TCP protocol.

In order to reduce network overload, all timeline posts are locally stored for 24h. The storage solution is a **SQL database** (SQLite3).



```
{
    "ip": "127.0.0.1",
    "port": 3000,
    "followers": ["userid_1","userid_2","userid_3"],
    "following": ["userid_2"],
    "last_post_id": 1
    }
```

Functionalities

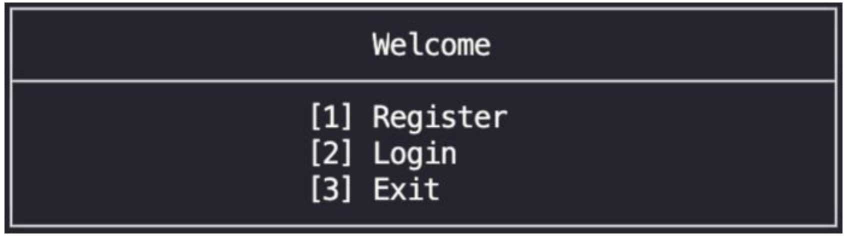
After registering and logging in, the user can perform the following actions:

Posts

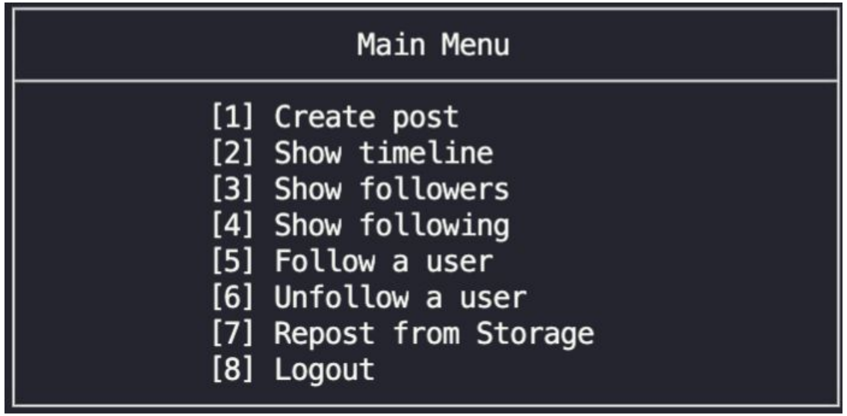
- Create Post
- Show Timeline
- Repost from Storage

Users

- Show Followers/Following
- Follow/Unfollow User

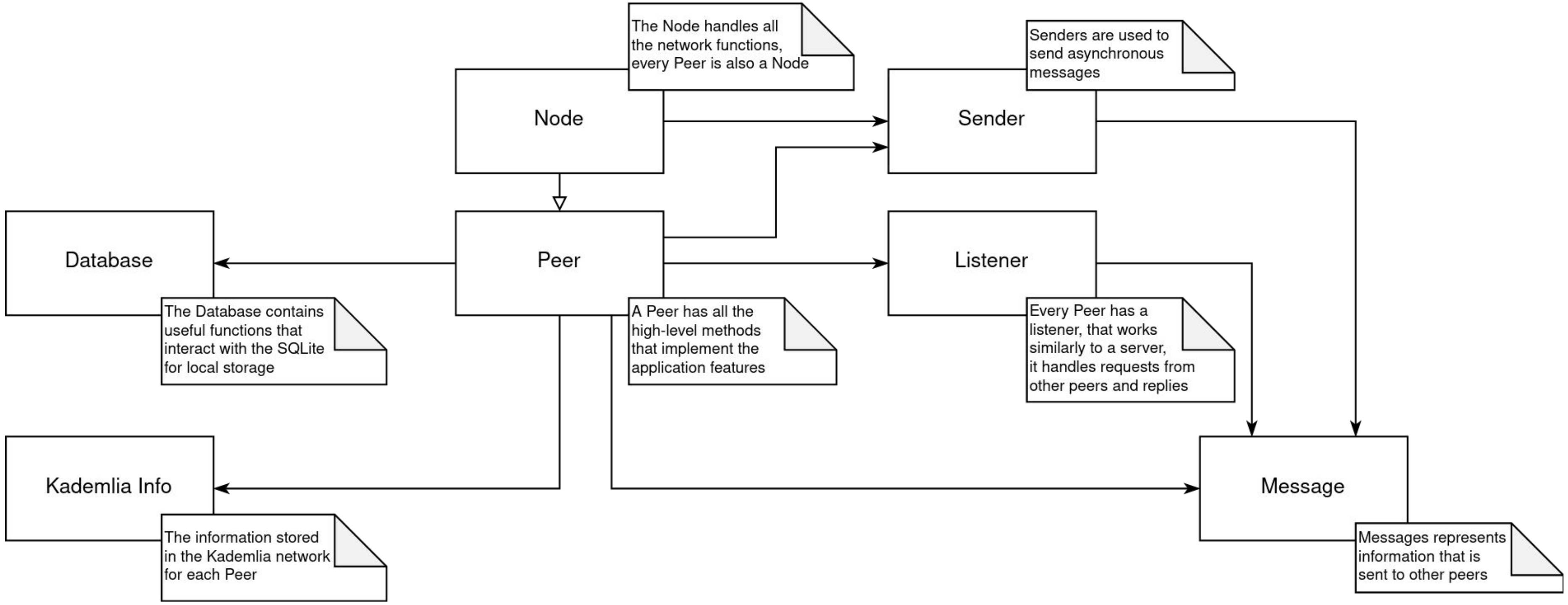


```
Welcome

[1] Register
[2] Login
[3] Exit
```



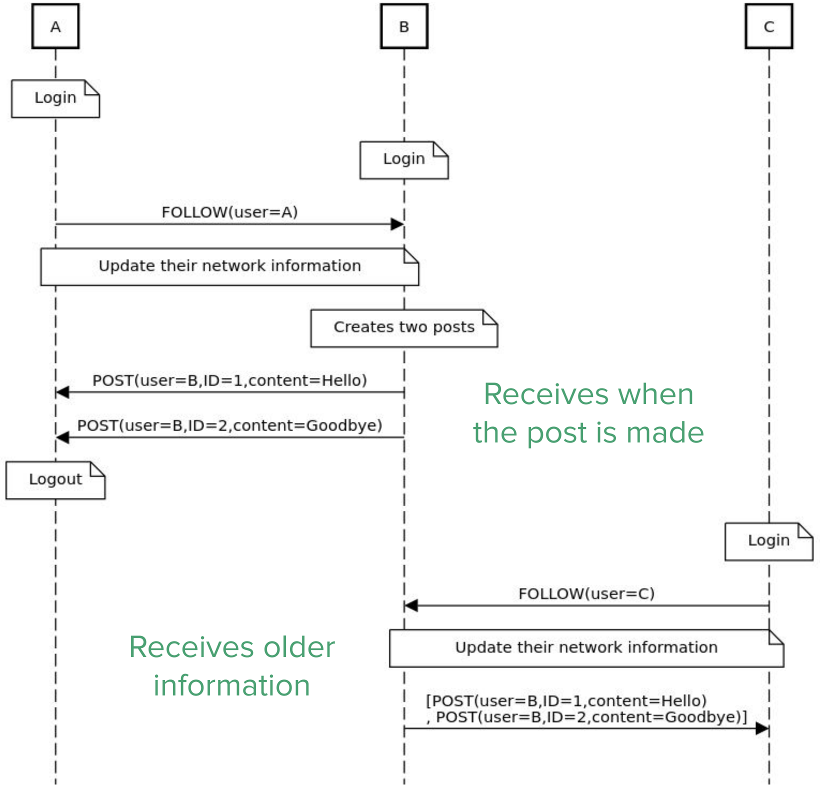
```
Main Menu

[1] Create post
[2] Show timeline
[3] Show followers
[4] Show following
[5] Follow a user
[6] Unfollow a user
[7] Repost from Storage
[8] Logout
```

System Diagram



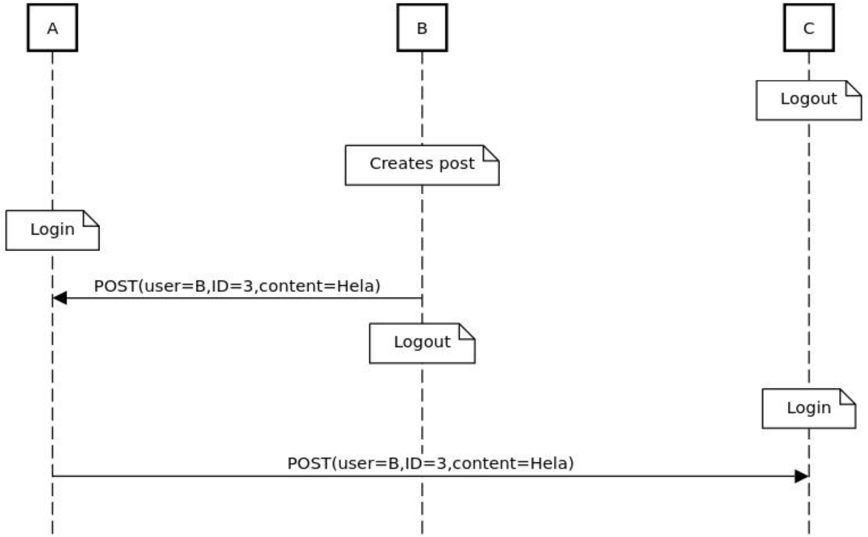
The Node handles all the network functions, every Peer is also a Node
Senders are used to send asynchronous messages
Node
Sender
Database
Peer
Listener
The Database contains useful functions that interact with the SQLite for local storage
A Peer has all the high-level methods that implement the application features
Every Peer has a listener, that works similarly to a server, it handles requests from other peers and replies
Kademlia Info
Message
The information stored in the Kademlia network for each Peer
Messages represents information that is sent to other peers

Messages Communication (follows and posts)


A
B
C
Login
Login
FOLLOW(user=A)
Update their network information
Creates two posts
POST(user=B,ID=1,content=Hello)
POST(user=B,ID=2,content=Goodbye)
Logout
Login
FOLLOW(user=C)
Update their network information
[POST(user=B,ID=1,content=Hello)
, POST(user=B,ID=2,content=Goodbye)]

Receives when the post is made

Receives older information


A
B
C
Logout
Creates post
Login
POST(user=B,ID=3,content=Hela)
Logout
Login
POST(user=B,ID=3,content=Hela)

Receives information from any other peer

NTP (timeline ordering)

- NTP (Network Time Protocol) was used for clock synchronization - ntplib module.
- Three different approaches were tested, and C was selected:

A. A request is sent to a NTP server **each time a peer made a post.** ➜ Too many requests result in NTPExceptions, which requires repeating the request many times and, consequently, delays the publication of the posts.

B. A request is **periodically** made to an NTP server - long interval between requests; and the **system clock is adjusted** with a system call. When a new post is made, the system time is used. ➜ In Linux, we were required to change the permissions in order to execute "`os.system`" to change the date.

C. A request is **periodically** made to a NTP server - long interval between requests; in order to **calculate and store the drift**. When a post is made, the **drift is used to calculate the timestamp**, without changing the system clock. ➜ At the same time, avoids sending many requests to the NTP server and does not require higher permissions.

Ephemeral Posts

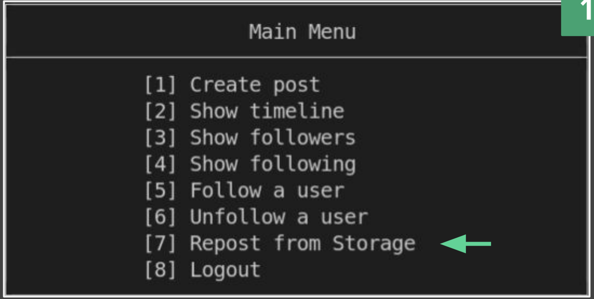
"Information from subscribed sources can be ephemeral and only stored and forwarded for a given time period."

- Posts have a **lifetime of 24 hours** – inspired by Instagram's Stories;
- Expired posts from other users are periodically removed from local storage - **garbage collector**.
- The owner of a post keeps his posts locally even after they have expired – inspired by Instagram's Storage Archive. Expired posts will be visible in the "**Repost** from Storage" option, where the user can select which post he would like to share again.

Behind the scenes: to avoid storing a similar post, the ID and the timestamp of the old post are updated. The followers will receive the post as it was a new post, avoiding the creation of new communication patterns and message types.

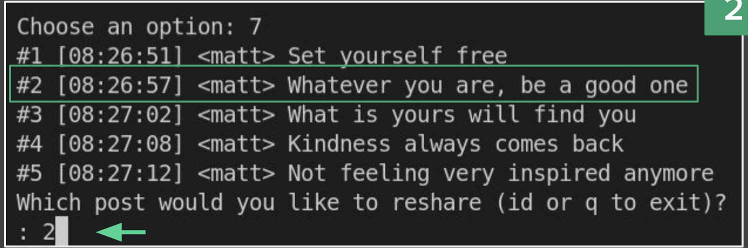
Note: to show the repost functionality, the post lifetime was temporarily set to 5 sec.

1



```
                Main Menu

        [1] Create post
        [2] Show timeline
        [3] Show followers
        [4] Show following
        [5] Follow a user
        [6] Unfollow a user
        [7] Repost from Storage
        [8] Logout
```

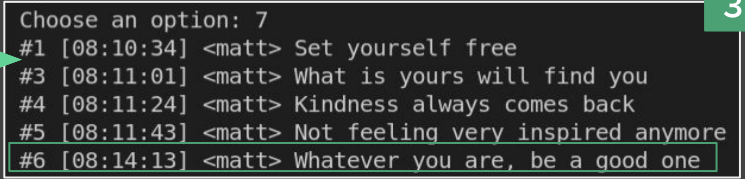
2



```
Choose an option: 7
#1 [08:26:51] <matt> Set yourself free
#2 [08:26:57] <matt> Whatever you are, be a good one
#3 [08:27:02] <matt> What is yours will find you
#4 [08:27:08] <matt> Kindness always comes back
#5 [08:27:12] <matt> Not feeling very inspired anymore
Which post would you like to reshare (id or q to exit)?
: 2
```

Post #2 will be resent to Matt's followers as a new post.

3



```
Choose an option: 7
#1 [08:10:34] <matt> Set yourself free
#3 [08:11:01] <matt> What is yours will find you
#4 [08:11:24] <matt> Kindness always comes back
#5 [08:11:43] <matt> Not feeling very inspired anymore
#6 [08:14:13] <matt> Whatever you are, be a good one
```

When the "reposted" post expires, it will appear again in the repost section, now with a new ID and timestamp.

(Un)Following offline user

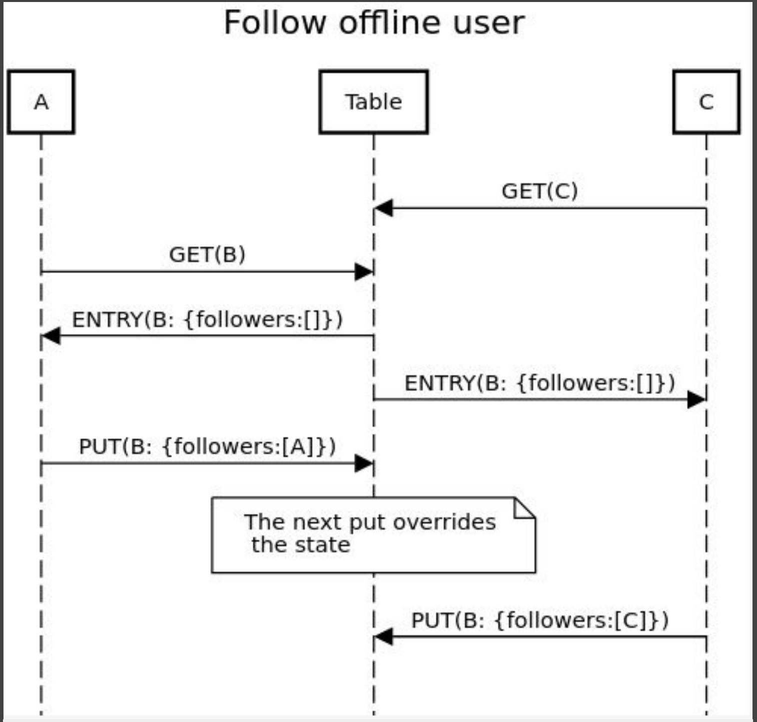
Normally, when a user **A** follows **B**, **A** sends a message to **B** announcing the event. Then **A** would update who he is following in the Kademlia Table and **B** would update his followers.

In a scenario where **B** is offline, in principle **A** could not follow him by the simple fact that **B** would never receive **A**'s follow announcement. Thus **B** will not update his followers.

As a solution, **A** could not only modify his entry in the table, but also **B**'s. Although this strategy solves the update issue, it also creates a concurrency problem as shown in the image on the right.

A similar situation would occur when unfollowing a user.

For this reason, it's not possible to follow or unfollow an offline user.



A and C try to follow B (which is offline) at the same time. Due to the non atomic nature of the operations, in the end, C overrides the state written by A, leading to a wrong state in the Kademlia table.

Table inconsistency

As we have seen in the previous slide, it's not possible for a user to modify the table of another, due to concurrency reasons.

In the image on the right, it's proved that, after a sequence of events, the entry of a specific user in the table might become inconsistent regarding the **last_post_id** variable.

In the final system state, only **C** is offline and **A, B** and **D** are online. In the Kademlia implementation used, the DHT is refreshed with some frequency and by the time the refresh is complete, if there are multiple states, it converges to the most common state of each key.

Thus, in the final scenario, the **last_post_id** variable in **C**'s DHT will converge to **0**. However, this fact does not affect the functioning of the program.

If **C** was online, he would set the table with the correct state, and, in our case, when **C** is offline, by the time he is back online he will set his state in the table. **Eventually the table will be fixed.**

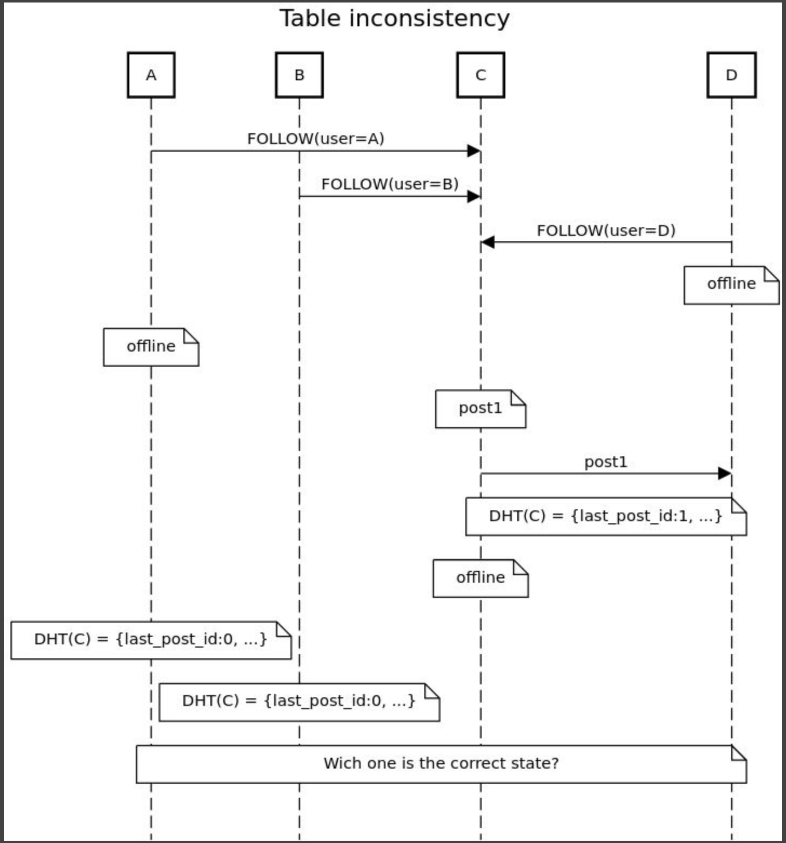
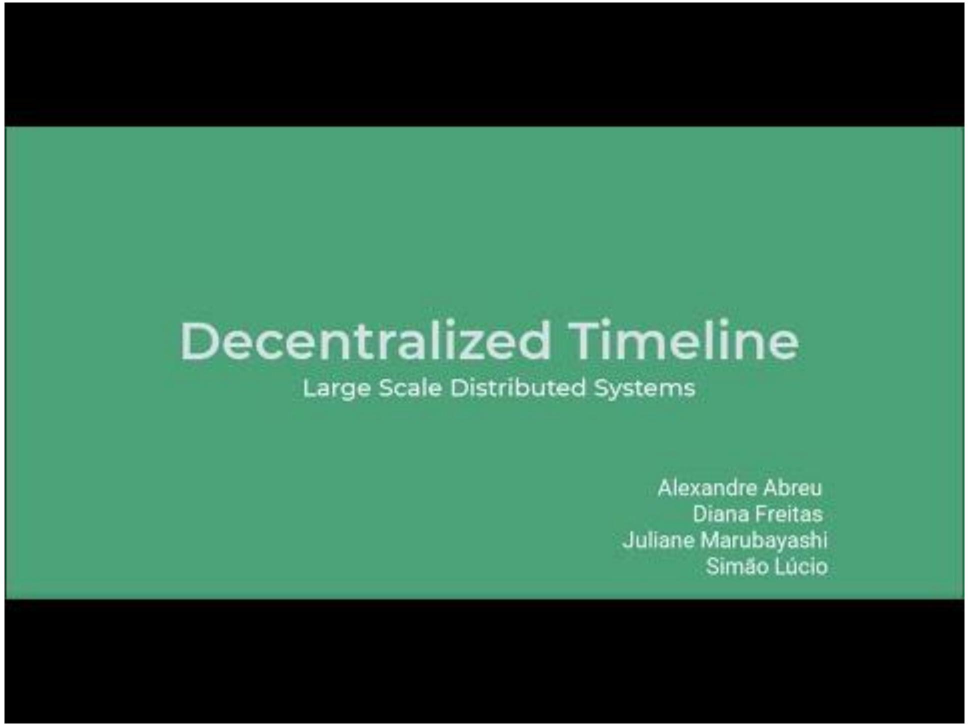


Table inconsistency
A
B
C
D
FOLLOW(user=A)
FOLLOW(user=B)
FOLLOW(user=D)
offline
offline
post1
post1
DHT(C) = {last_post_id:1, ...}
offline
DHT(C) = {last_post_id:0, ...}
DHT(C) = {last_post_id:0, ...}
Wich one is the correct state?

Demo



Decentralized Timeline

Large Scale Distributed Systems

Alexandre Abreu
Diana Freitas
Juliane Marubayashi
Simão Lúcio

Conclusion and Future Work

- The project was completed with success. Its implementation gave us practical knowledge of the topics addressed in the lectures.
- We have discussed many strategies to implement a consistent system, where the state among peers is the same. However, as the project was developed, some limitations regarding the system became clear: some operations might cause concurrency issues, due to the nature of Kademlia's implementation.
- In a bigger project, where the limitations of this system might become a problem, another implementation would be considered. Plus, some other techniques would be used to ensure the consistency, such as atomic commitment.