



LARGE-SCALE DISTRIBUTED SYSTEMS

Decentralized Timeline

Caio Nogueira - up201806218@fe.up.pt

Diogo Almeida - up201806630@fe.up.pt

Nuno Oliveira - up201806525@fe.up.pt

Pedro Queirós - up201806329@fe.up.pt

INTRODUCTION 01

ARCHITECTURE 02

IMPLEMENTATION 03

TABLE OF CONTENTS

04 CONCLUSION

05 REFERENCES

01

INTRODUCTION





INTRODUCTION



This presentation will focus on describing the architecture of a decentralized timeline, a project developed for the Large Scale Distributed Systems class.

In this timeline, each user, which represents a node in the peer-to-peer network, can publish messages, forming their timeline. The timeline of a user can then be obtained by following them and requesting these new messages.

In our design, the follower nodes will help store and forward the content of the followed nodes, however, since this information is ephemeral, it will only be held in these nodes for a short period of time.

Considering this, the project was developed in **Python**, a flexible programming language that provides two fundamental libraries for this implementation: **kademlia** and **asyncio**.




02

ARCHITECTURE



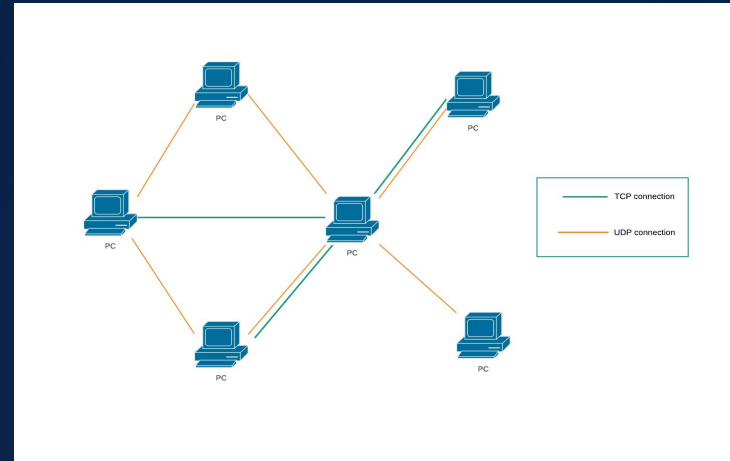
ARCHITECTURE

The chosen architecture is fully decentralized and uses a Distributed Hash Table (DHT). Kademlia was the chosen DHT since it has an implementation in Python. Furthermore, the number of nodes that need to be contacted while searching for a target node is $O(\log n)$ on average, which is extremely efficient. The details of the implementation are presented in the following sections.



NETWORK ARCHITECTURE

In the implemented network there are two types of connection between nodes: **UDP**, used in Kademlia's connections to ensure that the information is distributed among the nodes, and **TCP**, used in the connection between nodes when a *'follow'* or *'get timeline'* action takes place. TCP connections will be further explored in the implementation section.



DHT ENTRIES

In order to know the structure of each entry in the DHT, it was necessary to decide the relevant information that needs to be stored. After some discussion and analysis it was decided that the key for each entry would be the username. The respective values for each key are presented in the table below.

Followers	List of usernames that follow the user
Following	List of usernames that the user follows
Address	Address of the peer
Port	Port for the TCP connections
Online	Boolean indicating if the peer is online or not
Timeline	List of lists storing the ephemeral timelines of the follower users
Followers with timeline	List of usernames of the followers that have the user timeline stored
Followers timestamp	Dictionary having as key the followers' username and as value the timestamp of the last seen message



03

IMPLEMENTATION



LANGUAGE AND MODULES



The system was implemented with Python, heavily depending on two libraries, Kademlia and Asyncio.

Asyncio's purpose is to write concurrent code for asynchronous frameworks allowing us to do so without recurring to threads.

The Kademlia library, which itself uses Asyncio, is an asynchronous implementation of, as the name suggests, the Kademlia distributed hash table.

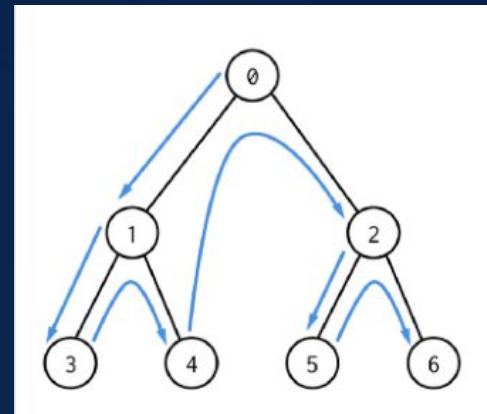
We chose to use this library in order to avoid “reinventing the wheel” when it comes to Kademlia and have more time to focus on other themes lectured in the course.

MESSAGE DELIVERY

Some of the main problems with a decentralized peer to peer network similar to the one present is how to distribute messages, establishing connections between peers and managing those connections.

In the developed system, whenever a peer wants to get the followed peers' timelines, a message is sent to those peers telling them to send their timeline. With this in mind, message delivery happens in an hierarchical way - each peer can only open X amount of connections to other peers and these peers are responsible for opening new connections and forwarding messages to the remaining peers, recursively, building a tree with X as branching factor, e.g. with $X = 2$ it would be a binary tree.

The choice of what connection to make is based on how many peers are following (the less the better in order to minimize the connections).





TYPES OF MESSAGES



For this decentralized timeline, six types of messages are used in the TCP connections:

- **Follow** - message indicating that there is a new follower;
- **Unfollow** - message indicating that a follower was removed;
- **Get** - message indicating that the timeline needs to be sent;
- **Get Stored Timeline** - message indicating that the ephemeral timeline stored needs to be sent;
- **Post** - message sent with the timelines in response to the **Get** messages;
- **Deleted Timeline** - message indicating that the ephemeral timeline was deleted and is no longer stored.



TIMELINES AND CLOCK SYNC



Timelines are obtained on demand. On startup every peer's clock is synchronized to the NTP (network time protocol) and after that every 60 seconds. When a user posts a message, it is saved along with a timestamp. This timestamp will then be used to sort messages when a timeline is obtained.

Peers that ask for a timeline will save other peers' messages for a limited amount of time. This way they are able to forward these messages in case the other peers crash.

Each peer contains a list of the peers that have their timeline. They are also notified when it is deleted so they are able to update that list. To delete messages, a peer has a separate thread that checks every 10 seconds if timelines are older than 30 seconds, if so, they are deleted.



APPLICATION START AND RECOVERY



When an already registered user logs in the application they need to recover their state. That state is recovered by performing a **GET** request to the DHT, having the respective username as key. After that, the user information stored in the network, e.g. followers and followed users, is sent as an answer to the request, allowing the user to have their data updated.

The user's timeline is locally stored in a JSON file since it always needs to be available to its owner, so it is easy to recover by only reading that file. The boolean variable indicating if the peer is online or not is also updated so the rest of the peers in the network know that the logged-in user is back online.



Fault Tolerance



When developing a decentralized service, peer crashes must be taken into account. A peer may crash while communicating with another.

Let's say a **Peer A** tries to establish a connection with a **Peer B** that has recently crashed: since no peer in the network knows about this crash, **Peer A** will try to open a TCP connection. In this case, we get a *Connection Refused* error that must be handled. It is essential that the information about the crashed peer is updated in the DHT.

Having completed this step, the recover process now depends on the operation being executed.

Fault Tolerance (cont.)

- When the crash occurs during a **follow/unfollow** operation, all we need to do is update the DHT with information about the followers/following list of both peers, as well as information about the crash.
- When the crash occurs during the **get a timeline** operation, the recovering process is a little more complex. We need to reorganize the hierarchical tree responsible for getting the messages, as well as ask the remaining peers for the messages belonging to the crashed peer. The tree is reorganized 5 times (can be changed): if all the tries are unsuccessful (all nodes in a certain hierarchy level have crashed), the retrieval process of the messages is returned. The limit of 5 retries is set to prevent the overhead of a possible “infinite” reorganization.



04

CONCLUSION



CONCLUSION



A decentralized timeline was developed using the Kademlia's DHT and TCP connections for the communication between nodes in the network. The design implement has the advantage of delivering the messages in an hierarchical way but makes it harder to implement fault tolerance in the system.

With that being said, the reorganization of the hierarchical tree improves the system's fault tolerance. Each node periodically synchronizes its clock to the NTP in order to have all the nodes in the network synchronized, despite the latency associated with this operation.

In conclusion, the objective of this project was fulfilled with the implementation of the decentralized timeline presented.



05

REFERENCES

REFERENCES

- <https://kademlia.readthedocs.io/en/latest/>
- <https://docs.python.org/3/library/asyncio.html>
- <https://docs.python.org/3/library/asyncio-stream.html>
- <https://pypi.org/project/ntplib/>
- <https://docs.python.org/3/library/datetime.html>
- <https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf>