Honk

A Decentralized Timeline Application

SDLE Second Project
T06G14

Marcelo Couto - up201906086@up.pt
Francisco Oliveira - up201907361@up.pt
Miguel Amorim - up201907756@up.pt
André Santos - up201907879@up.pt

Problem

The problem presented is to build a distributed timeline application, aka, Twitter but with no centralized server.

Challenges:

- Protocol for the distribution of data on the P2P network
  - Needs to balance data between nodes
  - Should not have single points of failure
- Authentication/Security protocol
  - Fight repudiation (sharing of information in other’s name)
- Bootstrap - how to join the network without an obvious address

P2P and DHT Protocols

As there is **no centralized server**, communication is done via P2P. **Data must be kept distributed** through all peers. The protocol should:

1. provide a way to identify which peer has what data
2. allow for a balanced distribution of data among peers
3. not have a single point of failure

In our case, the data which is stored in a decentralized manner is the list of followers of each user.

P2P and DHT Protocols

Kademlia protocol defines a distributed hash table (DHT) for peer-to-peer systems as ours.

Kademlia uses binary trees to decide who is storing the keys (list of followers in this case), depending on its **randomly assigned id**, thus **keeping balance**. In our case, the key is of a list of followers is the hash of the user's tag name.

Kademlia has **no deletes**. As such, **every few seconds**, the **list of followers is wiped** and the **followers announce** themselves to the network. This way, the lists are more likely to contain active users and even if a node goes down, it will soon be replaced.

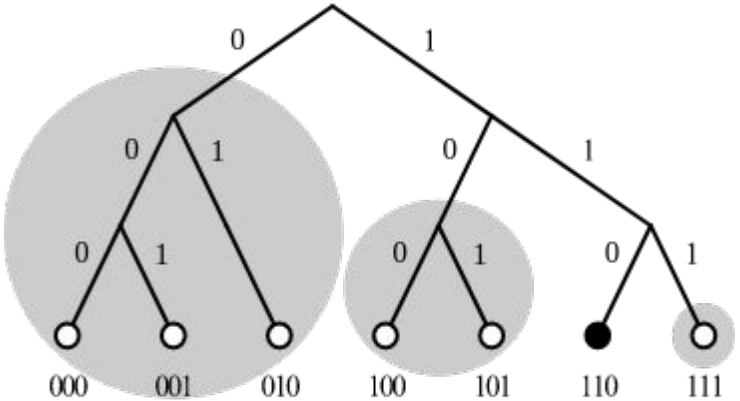


Fig. 1: Kademlia Tree

P2P and DHT Protocols

In our project, we used an implementation of Kademlia used in Bittorrent called **Bittorrent DHT**. This node library provides an **implementation of Kademlia** as well as a **simpler interface**.

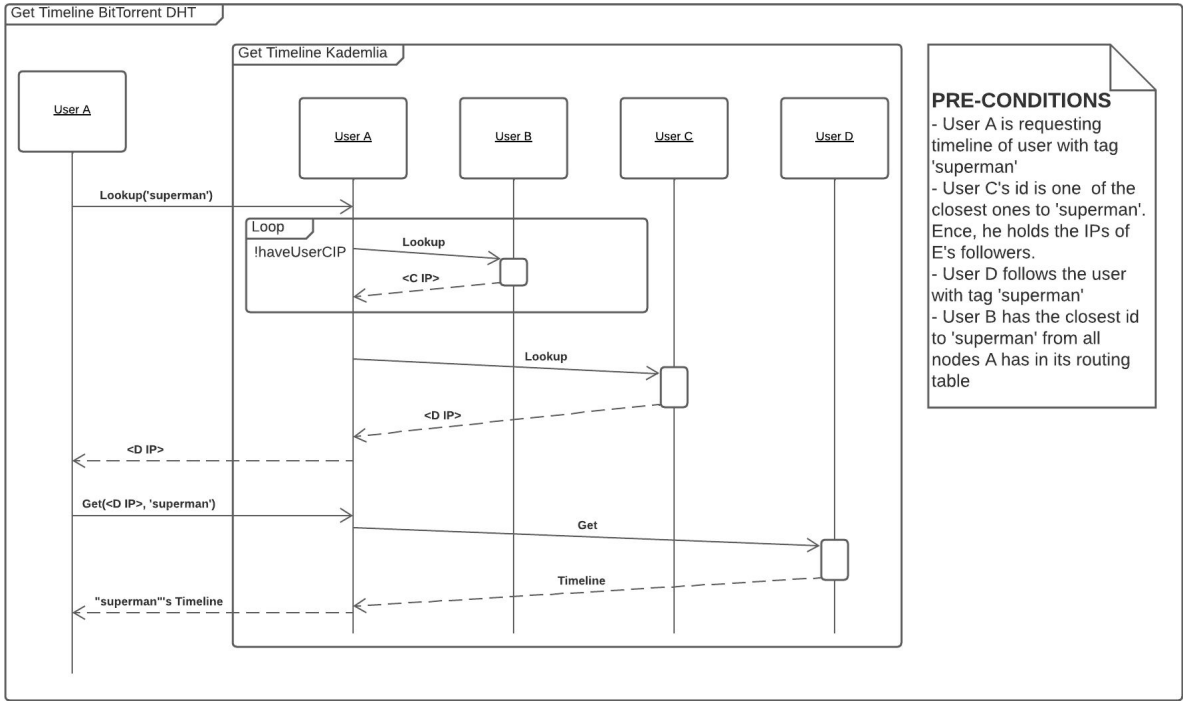


Fig. 2: Bittorrent DHT Get Timeline Sequence Diagram

Authentication

Authentication:

In order to **increase the safety** of the system and **lower** the chances of **repudiation**, we implemented a system that uses **Digital Signatures** to sign the posts.

For this, we used a node library called [node-jose](), a JavaScript **implementation of JOSE** (JSON Object Signing and Encryption). JOSE provides a way to sign JSON.

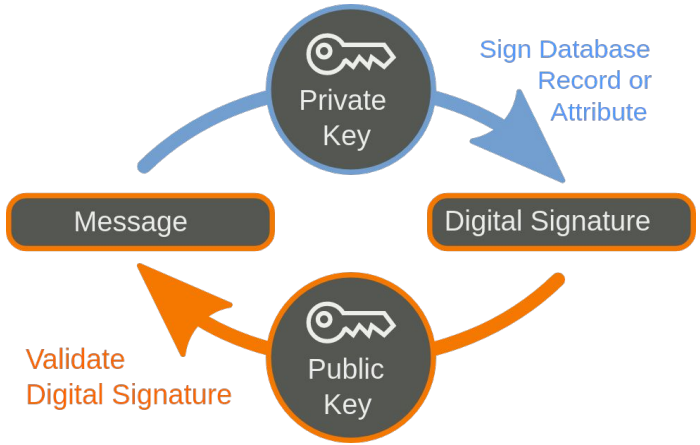


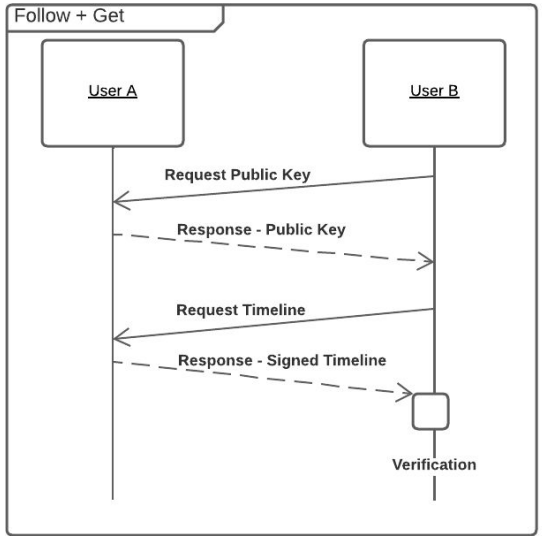
Fig. 3: Digital Signatures

Authentication

System Flow:

1. Users sign their timeline with private key
2. Users are granted public key upon its request on follow
3. Users use the public key to confirm the content comes from the whom initially gave them the key

Pitfall: the public key was initially given by an impostor - user is required to trust the first key



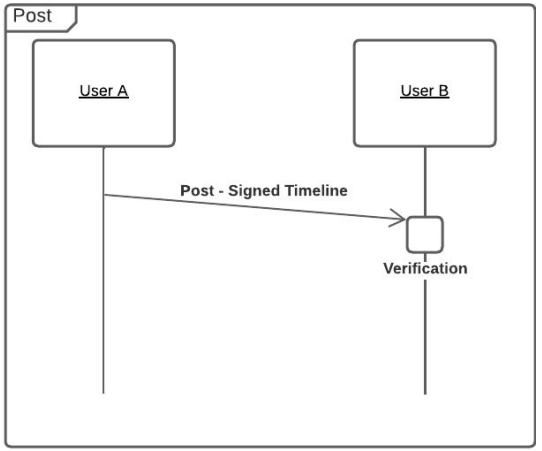


Fig. 4: Authentication Sequence Diagram

Other Tricky Scenarios

- **Bootstrap:** To solve the **bootstrap problem**, all prospective users must provide the IP address of current user to join. This will be provided via configuration file. This will make the application sort of a invite-only network.
- **Users might have old data when joining again:** syncTimeline - the user requests for the data of all the users he follows, also multicasting his timeline to his followers.

Application

Honk provides a way for users to **share** opinions, stories, experiences and other **textual information** through **posts without** the requirement of a centralized **server.**

The application comes in the format of a **self-hosted node server**. The server is divided into 3 parts:

1. **GUI**, which is comprised of compiled react files served by the Node server locally
2. **Peerfinder,** P2P Protocol (Kademlia - UDP) - resolves the IPs used for the requests to the timeline server/service
3. **Timeline Server/Service** (HTTP - TCP) - communication of the timelines between the peers (GET and PUT requests) - uses express.js for the server and axios for the service

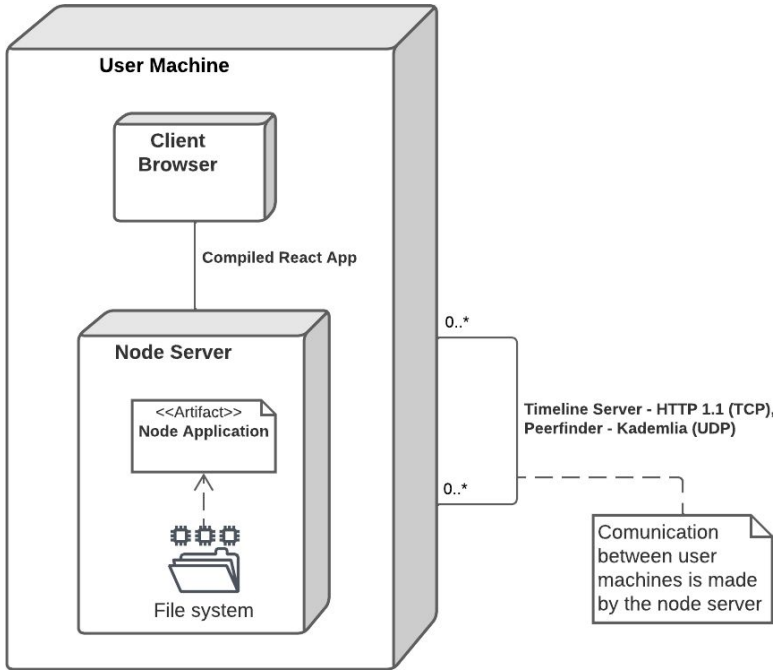


Fig. 5: Honk Deployment Diagram

Application

The **configuration** of the application is done through a **config file**, where the user defines:

- user name
- port to answer to requests for timeline
- port to answer (peerfinder) Kademlia requests
- ip of user already in the system (bootstrap)

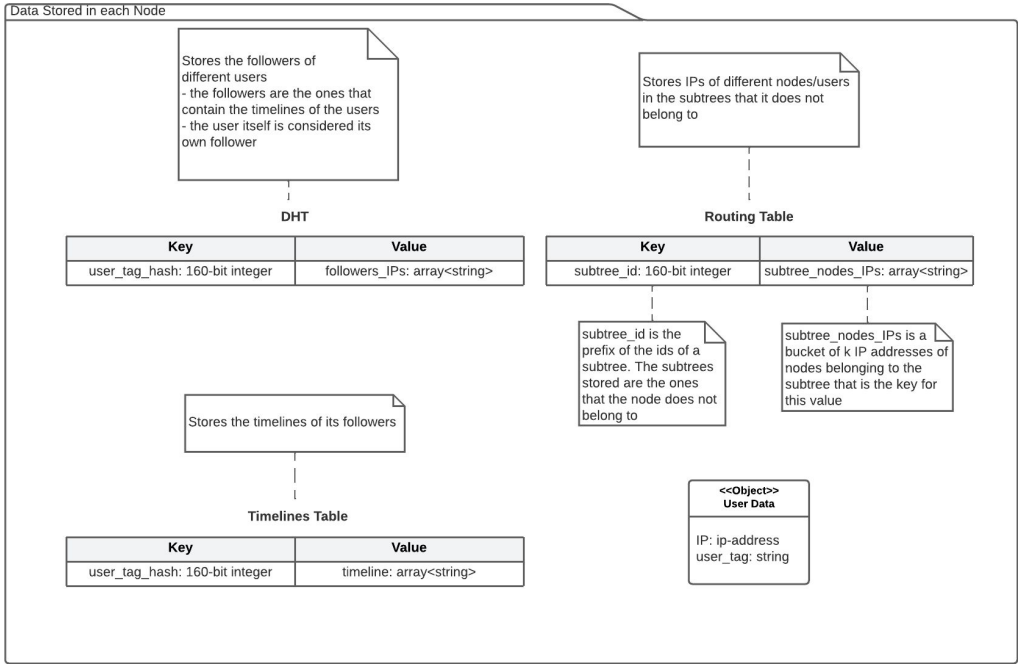


Fig. 6: Data stored in each machine for both Kademlia and the Timelines Service/Server

Conclusion

For the most part, the project was a success:

- we developed a distributed timeline application that distributes the data between peers
- the application is able to resolve the information necessities of the user with the Kademlia protocol paired with the Timeline Service/Server
- the application is scalable and with no single points of failure
- the application has a simple but intuitive GUI to pair with
- from the point a follow is made, there is a guarantee that all the posts will come from the same person

Pitfalls and Future work:

- authentication requires the user to trust the initial public key received, which can come from an impostor
- the application can be made more elaborate and more features could be added

Demo