Decentralized Timeline Service

Large-Scale Distributed Systems

Class 7, Group 11

- Clara Martins
- Gonçalo Pascoal
- Rafael Ribeiro

Domain Description and Requirements

- Decentralized timeline service (like Twitter, Instagram)
  - Users have an id and publish messages to their local timeline
  - Users can subscribe to other user's timelines and see their posts
  - Subscribers help store and forward the source's content when it is down (but storage can be ephemeral)
- Two main challenges:
  - **How to locate a peer by its id?**
  - **How to locate a peer's other subscribers when it is down?**

Network Architecture (PUSH vs PULL)

- The requirements do not specify the network architecture; we believe there are two main options:
  - **PUSH** architecture (Discord, Messenger…), where posts/messages are automatically sent to subscribers
  - **PULL** architecture (Twitter, Instagram…), where subscribers ask for new posts
- We chose to use a **PULL** architecture for our service

PULL Advantages

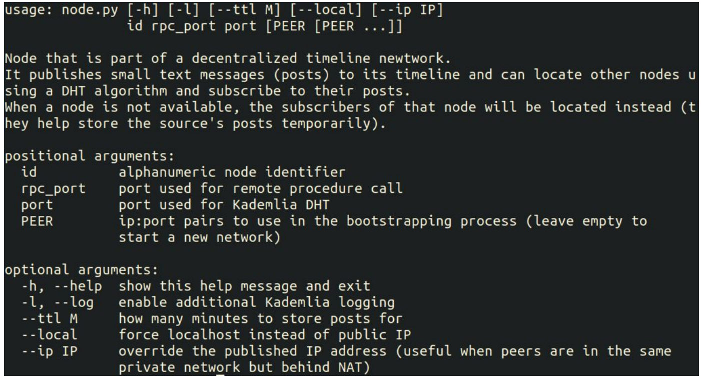
- Easier to guarantee that a subscriber gets all posts from the source (when subscriber is down, pushed posts would be lost, requires performing a request for missed posts)
- Avoids certain problems with ephemeral posts in PUSH architecture
  - Posts could get discarded before the user sees them if timer starts when peer receives the post
  - If the timer starts after the user sees the posts, an indeterminate number of unseen posts could occupy memory for an unbounded amount of time
- Requests explicitly ask for posts, simplifying the retrieval of posts from subscribers when the source is down

PULL Disadvantages

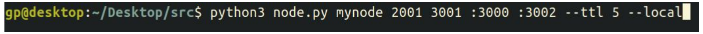
- Can introduce a delay between requests and the posts being shown to the user
- Requests can be wasted if the source has no new posts to forward

Programs and Command Line Arguments

node.py

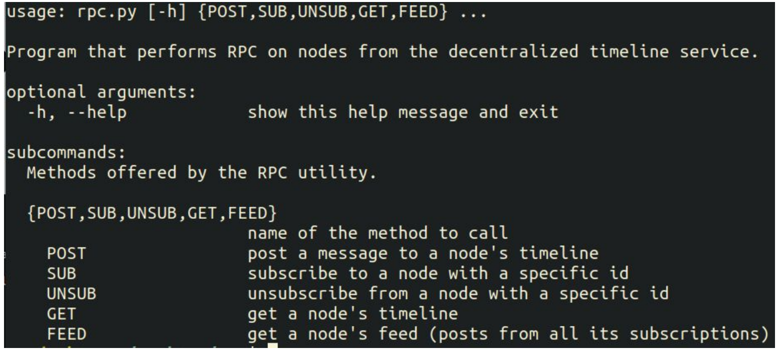


```
usage: node.py [-h] [-l] [--ttl M] [--local] [--ip IP]
               id rpc_port port [PEER [PEER ...]]

Node that is part of a decentralized timeline newtwork.
It publishes small text messages (posts) to its timeline and can locate other nodes u
sing a DHT algorithm and subscribe to their posts.
When a node is not available, the subscribers of that node will be located instead (t
hey help store the source's posts temporarily).

positional arguments:
  id          alphanumeric node identifier
  rpc_port    port used for remote procedure call
  port        port used for Kademlia DHT
  PEER        ip:port pairs to use in the bootstrapping process (leave empty to
              start a new network)

optional arguments:
  -h, --help  show this help message and exit
  -l, --log   enable additional Kademlia logging
  --ttl M     how many minutes to store posts for
  --local     force localhost instead of public IP
  --ip IP     override the published IP address (useful when peers are in the same
              private network but behind NAT)
```

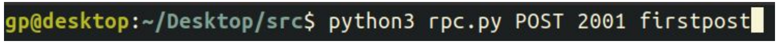


```
gp@desktop:~/Desktop/src$ python3 node.py mynode 2001 3001 :3000 :3002 --ttl 5 --local
```

rpc.py



```
usage: rpc.py [-h] {POST,SUB,UNSUB,GET,FEED} ...

Program that performs RPC on nodes from the decentralized timeline service.

optional arguments:
  -h, --help            show this help message and exit

subcommands:
  Methods offered by the RPC utility.

  {POST,SUB,UNSUB,GET,FEED}
                        name of the method to call
    POST                post a message to a node's timeline
    SUB                 subscribe to a node with a specific id
    UNSUB               unsubscribe from a node with a specific id
    GET                 get a node's timeline
    FEED                get a node's feed (posts from all its subscriptions)
```



```
gp@desktop:~/Desktop/src$ python3 rpc.py POST 2001 firstpost
```

Locating Peers by ID

- Service is written in **Python** and uses an implementation of the **Kademlia DHT**
- In the Kademlia network, each node is identified by its ID
- In the DHT, each peer's ID maps to a serialized string containing the information about the peer in JSON format
- This information contains the peer's IP address and port, as well as the ID, IP address and port of each of its subscribers
- As usual for a DHT, in a network with `n` nodes, getting the value mapped to a specific key is an `O(log n)` operation

Locating Subscribers when Source is Down

When do we consider a source is down?

- When we can’t connect to the source

What actions do we take when the source is down?

1. Obtain the source’s information through the Kademlia network
2. Extract the subscribers’ IP addresses and ports
3. Contact other subscribers to obtain their sections of the source’s timeline
4. Merge posts received from subscribers

RPC Communication with/between Peers

RPC is used to send commands to peers (`rpc.py`). Peers can also send some commands to other peers.

User – Peer

- **`POST`** – make a new post (post gets assigned a unique id and a timestamp)
- **`SUB`** – subscribe to a peer with a certain id
- **`UNSUB`** – unsubscribe from a peer with a certain id
- **`GET`** – get a peer's posts from source or other subscribers (and update the last post received)
- **`FEED`** – build a chronological subscription feed by concurrently getting posts from all subscribed peers

Peer – Peer

- **`SUB_NODE`** – ask the peer to subscribe
- **`UNSUB_NODE`** – ask the peer to unsubscribe
- **`GET_NODE`** – get posts from the source
- **`GET_SUB`** – get posts from a subscriber when the source is down

Ephemeral Posts and Requesting Only New Posts

Ephemeral Posts

- By default, posts are stored for 15 minutes
- This can be configured using the optional command line argument `--ttl <value>` (in minutes)
- Periodically, the service discards posts that are older than this limit

Requesting Only New Posts

- By default, `GET` or `FEED` commands request all available posts
- If the user only wishes to retrieve new posts when calling `GET` or `FEED`, they can use the optional argument `-n` or `--new`
    - Peer will send the identifier of the last post received in the request, response will only contain posts with a higher id.

Persisting State and Recovering from Crashes

- Peer state is saved every 3 minutes, as well as after each major operation
  - On recovery, loads information such as local timeline, list of subscribers and subscription information (stored posts and id of last received post)
- Kademlia state is saved every 5 minutes
  - On recovery, key-value pairs in local storage are loaded
  - Neighbour addresses are stored, meaning that if a neighbour is online when the peer recovers, we do not need to specify a peer for bootstrapping

More on Fault Tolerance

The **Kademlia DHT** provides useful fault tolerance functionality for our distributed system:

- **Bootstrapping:** to avoid single points of failure, we can specify any number of peers to be used in the bootstrapping process instead of a single peer (to act as fail-safes if the peer is offline)
- **Key-value storage:** up to *k* peers store a certain key-value pair, to provide redundancy in case of peer failures