Decentralized Social Network

Richard Snyder
rwsnyde2@ncsu.edu
North Carolina State University
Raleigh, North Carolina

Austin Shafer
amshafe2@ncsu.edu
North Carolina State University
Raleigh, North Carolina

ABSTRACT

While social media has advanced technologically in its ability to connect users across the world, the data that is collected and stored in those networks is highly centralized, undercutting users' ownership of that data. This is good for social network corporations and digital advertisers, but not for their users. We propose a communication protocol for a distributed social network which enforces independent user profiles. We develop a system where traditional social network accounts are replaced by self hosted instances, each of which join the network and advertise a profile which other members of the network can access. Nodes discover each other through a Kademlia DHT, and implement a REST api for retrieving user content. We demonstrate that this network can handle a large number of unique clients with linear scaling and that users may disconnect at any time, thereby removing their profile and data, from the network.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; • **Networks**;

KEYWORDS

decentralization, peer-to-peer networks, distributed hash table

1 INTRODUCTION

Traditional social networks, such as Twitter, Facebook, and Instagram, are controlled by a central entity that users cannot influence. User data is recorded in massive quantities, and powers advertisement sales which keep the social network funded. All user activity, even across other websites, is recorded and analyzed in order to serve them better advertisement content. While these networks are free to join, the user has to allow a company to have full access and ownership to any usage statistics available. In addition, the images, posts, and media that is stored on those platforms may be held indefinitely by the owning group. If users want to remove their content from the platform, there are no guarantees that the platform won't keep that data in an unpublished fashion for training or other purposes. Understandably, many users have privacy concerns about this surveillance and content rights.

Some interesting technologies have emerged to solve this problem. Federated networks are a class of decentralized content network which abide by a public protocol that defines how content can be accessed across different platforms. Mastodon, an open-source, decentralized twitter-like platform, is the prime example. PeerTube uses the same federation protocol suite, but is designed as a replacement for YouTube.

The Matrix communication protocol follows a similar approach, but is designed for chat rooms and voice calls. It is a decentralized alternative to Slack or Discord. Users join a "homeserver", which serves as their designated host in a network of Matrix servers. All of these networks can be hosted individually, and join a "federation" of platforms that can freely interact with each other.

We take a much more radical approach to enforcing a decentralized network. Our solution involves transferring the ownership of social network data from companies to users. Traditionally, a user registers an account with their chosen social network, and cannot access accounts of other networks. In our system, an "account" is simply a node on a network. This concept of belonging to a network has interesting implications for data ownership. If a user wants to leave a network they simply disconnect, and all of their data is no longer visible.

In order to create such a platform, we have incorporated a distributed hash table (DHT) that maps usernames to the IP address that is currently hosting the profile with that username. When a user joins a network, they can initiate requests to other users to perform actions such as retrieving a user profile, viewing their posts, or following that user.

This paper makes the following contributions:

- A distributed social networking platform that utilizes a distributed hash table
- A news feed that traverses a list of followed users in the network

The remainder of this paper proceeds as follows. Section 2 provides a description of the System Design and the data structures that are used to maintain the system. Section 3 describes the resiliency of the system as well as the process of joining and leaving a network. Section 4 reveals the high level details of our feature set and how they are implemented. Section 5 lays out a framework for how we tested the capabilities of our network and Section 6 displays the results of those

tests. Section 7 states lessons learned throughout the implementation process. Section 8 provides previous research in the area that inspired this work. Section 9 concludes.

2 SYSTEM DESIGN OVERVIEW

There are two halves of our implementation: the DHT and the web server. The DHT uses an open source implementation of the Kademlia DHT, and the web server is built with the Django framework. The two servers are run as two separate processes which communicate through a pipe. The web server passes work tasks to the Kademlia server, which will make requests on the DHT and hand the results back.

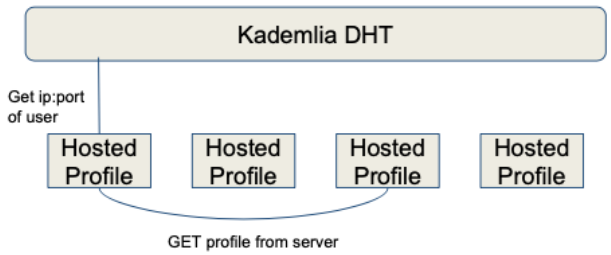

Figure 1: The DHT and the web servers combine to make a complete application

The DHT server extensively uses asynchronous io, as that is the programming interface for the open source implementation we used. Python's async is not trivial to pair with Django, which is why the Kademlia server is run in a different process. The Django server is very traditional, and it stores all of the application data in a local sqlite database.

We will now provide an example of how the two halves are interleaved. If a local user needs to grab information from a remote user, like their home user page, they first need to find the IP address of the web server hosting the profile. This is done by querying the DHT using the target remote user's username as the key. If the user exists, then the DHT will respond with the IP address of that user's profile. The initiating user can then make HTTP requests to the target user's web server.

3 JOINING AND LEAVING A NETWORK

Because our network is peer to peer, nodes can join and leave at any time without any interruption to the service of the rest of the network. This process begins with a user creating a new network, which we define as bootstrapping. The bootstrapping node will initiate their Kademlia and Django server processes, thus creating a new DHT with a single entry: their username and IP address. Then, the next new user to the network will send a join request to that node asking to be added to the DHT. Once that user has been added to the DHT, the two nodes are discoverable to each other through the web client.

After a network has been established, we can make requests to the users, and therefore nodes, in the network for data. But, because we allow users to join and leave the network as they please, there must be guarantees that disallows interruptions to the expected service. If a node is not present, any attempts to query it's information will fail and warn that this user is not present on the network. Although a bootstrap node is needed to create a new network, it can also leave the network after it has been created. Any attempts to join off of that bootstrap node will be unsuccessful, but that does not mean the network cannot be joined. The node attempting to join the network will simply have to find another node within the network to join off of and send its request to it rather than the bootstrap node. This allows for high fault tolerance and users to remove their personal data from the social network by simply disconnecting their node.

4 IMPLEMENTATION

In this section we will cover each feature one by one, and will explain at a high level how they were implemented. Each feature is exposed to the user through web pages served by Django. All REST endpoints have a similarly named 'api' endpoint which returns JSON suitable for parsing. An example of this is the user profile, which has the routes '/profile' for the html and '/api/profile' for the JSON object.

User Profile

Retrieving the profile for a user is one of the simplest interactions. The profile is ubiquitous with other social media platforms in that it has basic information like the user's full name and a text bio. In order to access the profile of another user, the initiator first looks up the URL of the target's web server by querying the DHT using the target username as the key. The initiator can then perform an HTTP GET request to the profile endpoint, or the api/profile endpoint if they would like raw JSON data. The returned data is composed of the raw content that was set by the user, the list of other users they follow, and their name and username.

Posts

Retrieving your own posts is also a relatively easy operation, as all of your posts are kept in a local database managed by Django. The posts can either be retrieved in a paginated website that users can browse, or in a JSON format intended for consumption by other servers on the network. When browsing their own posts, users can enter an input text box to create a new post. Accessing another user's posts is just like grabbing their user profile, only the posts will be displayed in the same paginated form as the local user's posts.

Following Users

All social networks provide the capability to follow another user, which for our implementation means viewing their posts in the News Feed. In order to follow a new user, you must first save their username to your own local Django database as a Following instance. Then, using the pipe to the Kademlia server, a new HTTP request is made to the remote user's server with the route /followers/<username>/addFollower. That request will be handled on the target's Django server and add a Follower instance to its database. The relevance of the Following list is apparent in the News Feed, described next.

News Feed

The news feed is the most performance intensive operation on the network. When crawling a new news feed for new content, the server requests all posts for everyone in that user's following list. Once all posts have been retrieved, they can be displayed and paginated based on their date to create a news feed. Our implementation is a chronological news feed, and does not reorder content or selectively display content based on user interests.

Profile Directory

The profile directory we implemented differs from the profile directory feature of Mastodon. In Mastodon, the profile directory is the list of all users hosted on a particular instance of Mastodon. Each instance of our network only hosts one user profile, since it is extremely decentralized. Our profile directory is the list of all known users on the network. It is essentially a list of all users observed in any prior interactions.

5 TESTING

In order to validate the scaling capabilities of our network, we created a suite of tests that request different endpoints that require access to the DHT. The independent variable is the number of nodes in the network, and as we increase the number of nodes in the network we expect to see a linear or better increase in time to make those requests.

In the first test, we created a script that makes HTTP requests to each route that is of form GET. Those routes include both the routes that return JSON (/api/*) and those that return HTML responses (/*). We did not render anything in the browser for this test, even if the response contained HTML, as we simply wanted to know how efficient accessing the DHT is as the number of nodes it contains increases. This test was run from on one to four different instances of VCL nodes running Ubuntu 18.04. These machines are all running at North Carolina State University's VCL computing lab, so the total distance the network spanned is short. To time each request, the Python time module was used.

In the second test, we wanted to test the most computationally intensive functionality we have implemented, which would be the news feed. This test requests the /feed route, and we allowed that to run in the browser and render the content that it received. Not only would this test confirm whether or not our network was scaling at desirable rates, but would also provide us with user experience quality that we did not receive in the first test, as we are now rendering the returned content. Although we expected that our implementation would not scale as efficiently as the centralized service, we still wanted to ensure reasonable performance. So, for comparison, we ran a single local instance of Mastodon, which offers the same news feed feature with pagination but with a centralized implementation. Mastodon supports decentralized federation, which we did not use in order to simulate a centralized service such as Twitter. This test was again run on VCL nodes, but we've increased the range to eight nodes in the network at intervals of powers of two. To time how long each request took, we used the Chromium developer tool's network tab.

6 RESULTS

Our results from those tests will now be presented.

The first test showed conclusively that at a low number of nodes, and therefore profiles, our implementation works consistently well across different endpoints at finding a user's IP address and requesting the relevant content from their server. Figures 2 and 3 depict the time it takes to receive a response from the different routes in HTML form (/*) and JSON form (/api/*).

For the HTML routes, there is a fairly consistent trend in how long access times take. Because the DHT does not have a large amount of keys stored, accessing an IP value from the username key is not a large task. In addition, because the nodes are not physically far apart, their responses are able to be quickly served back to the request. There is one outlier in our graph, and that would be accesing our own profile through the / route. This can be attributed to our implementation using the os module to access our local data. The I/O operations required to grab our content seem to take longer than requests that are made to other servers, but by a minuscule margin.

The same can be said for the api JSON return values. The /api routes follow a trend of receiving a response within .01 and .0125 seconds.

Now that we have shown our network scales well for small instances, we will examine our most intensive operation, the news feed. Figure 4 showcases how well our implementation compares to a centralized Mastodon version of a news feed.

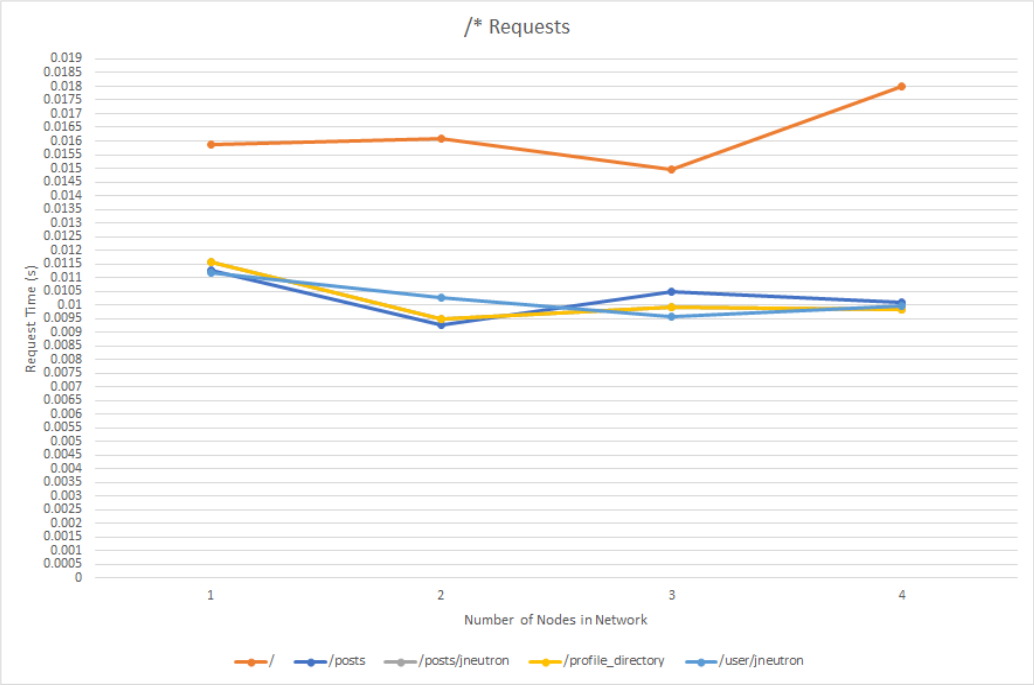

Figure 2: Time taken to reach /* endpoints

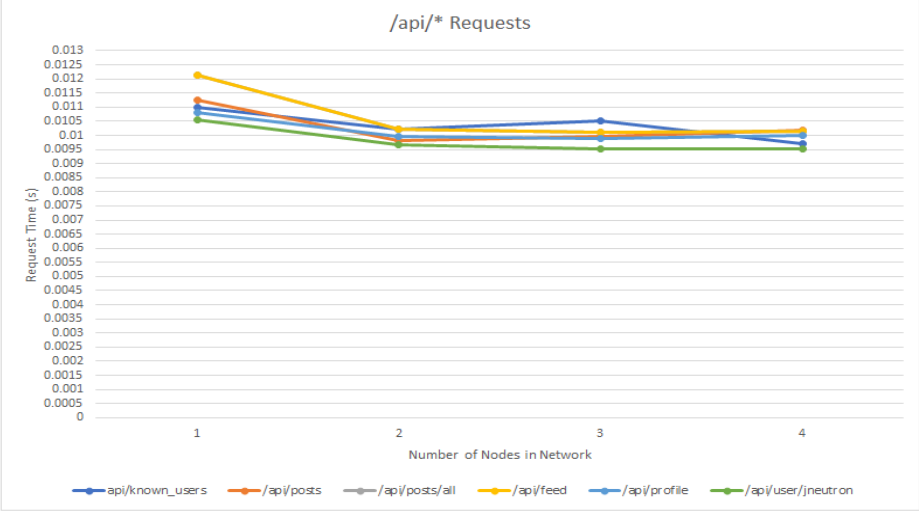

Figure 3: Time taken to reach /api/* endpoints

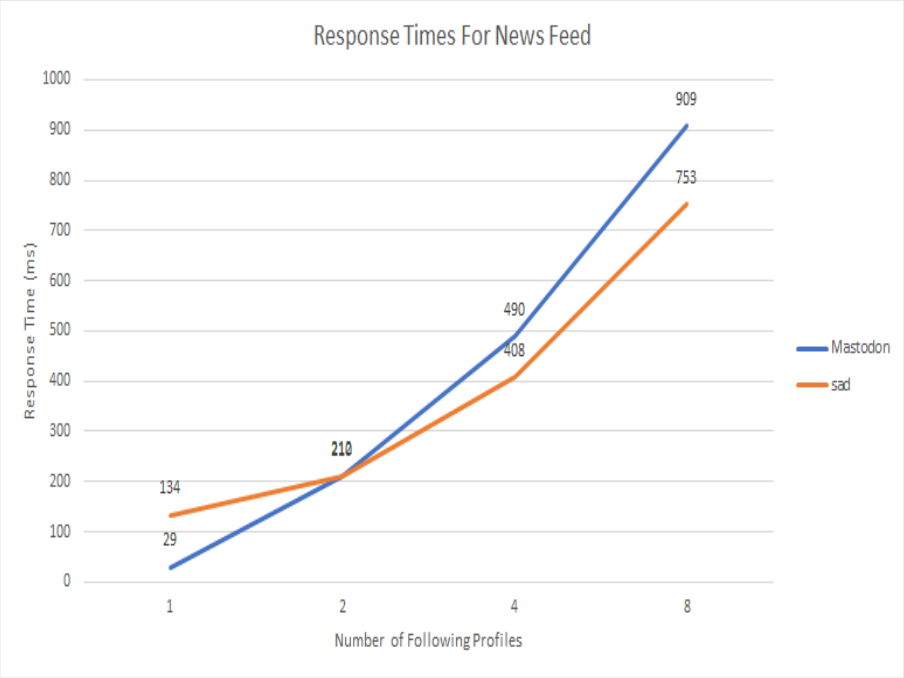

Figure 4: Time taken to load the news feed in the browser

It can be observed that both instances scale the response time at linear rates as the number of nodes increases in the network. Our implementation starts off being slower to render the page than the Mastodon instance with just one user in the following list. Then, at following two users, the response times converge, and beyond that, our implementation actually outperforms Mastodon. We believe this could be the case because the Mastodon server is compiling more metadata about the posts they are responding with than our implementation. The Mastodon server also uses a postgreSQL database for storing posts, whereas our implementation uses an sqlite database.

One note to make about these results is that Mastodon is built to be able to handle federations of hundreds to thousands of users. If linear scaling were to continue beyond eight nodes, the platform would become unusable from a user experience perspective. It is our expectation that Mastodon would scale linearly until a point, and level off. Unfortunately we were limited in the amount of VCL nodes we could test on, so it should be further investigated to find out when the linear scaling plateaus on Mastodon and if our implementation continues to scale linearly.

It is overall positive to see that in small instances we are able to prove that accessing the DHT is a consistently performant operation. In addition, our implementation provides a user experience that is within an acceptable load time for the most important features of a social network.

7 LESSONS LEARNED

One major challenge we did not anticipate was the difficulty of enumerating all keys on a DHT. Originally we needed this to implement a list of all users on a network. Due to the complexity of implementing such a DHT crawler, we settled on a simplified "known users" directory in our final implementation.

Despite that setback, we did get practical experience building applications with DHTs, and exposed ourselves to modern federation technologies such as ActivityPub[9]. We also got learned different methodologies about how to properly handle node failures and dropouts in a peer to peer network.

8 RELATED WORK

Mastodon[2]

Mastodon is a federated social network with microblogging features designed after twitter. Users create an account on an instance and can access communities on other instances through Mastodon's implementation of the *ActivityPub* federation protocol. It is arguably the most popular decentralized social network and is a large source of inspiration for this project.

Matrix[3]

Matrix is the project that gave us the idea for a distributed social network. Matrix incorporates peer-to-peer interactions by way of establishing *rooms* that users can join and leave, by which server-server and client-server interactions can take place. While Matrix is normally used for telecommunication purposes, we believe its model for entering and leaving a room is a good platform to emulate.

Maze[10]

Maze contains a centralized search engine that properly indexes files to nodes. Included in Maze is network locality consideration, which improves the efficiency of lookups for replicas in the network. In addition, Maze incorporates a socially constructed framework for its users, allowing searches to be more customized to how a user has constructed their peer list. We will attempt to greatly leverage that consideration in our own protocol.

Secure and Flexible Framework for Decentralized Social Network Services[5]

The paper presented by Aiello *et. al* discusses the necessity for attack prevention and authentication in peer-to-peer systems. This social network conducts this at the routing layer of transmission, which is backed by signed packages of the transmitted data. Another strong feature that could be useful in future implementations of our network is the blacklisting feature, which would allow users to be impervious to outside influence they do not want to encounter.

Decentralization: The Future of Online Social Networking[7]

In the paper by Yeung *et. al*, the proposed social network would incorporate a client-server relationship, such that the server of the user's choosing is one that they trust to host their data. This approach is interesting, and different to our approach, in that the user is no longer responsible for owning the hardware that hosts the data they wish to broadcast.

Efficient dissemination in decentralized social networks[8]

Another paper proposed by Mega *et. al* attempts to create a network that is efficiently formed like that of the online social network construct. They propose that, instead of a connected peer-to-peer system, a *friend-to-friend* system should be incorporated, in that communication between nodes is only enabled between the nodes that are *friends* with each other. They enact that methodology through use of gossip protocols.

9 CONCLUSION

We demonstrate a peer to peer decentralized social network, which is highly fault tolerant and performs reasonably well. We compare our implementation against a traditional, centralized application (Mastodon) to examine the tradeoffs made when transitioning to such a decentralized architecture. Finally, we document our results and postulate what could have been done better if given more time and resources.

REFERENCES

[1] 2020. *IPFS: InterPlanetary File System.* Retrieved February 3, 2020 from https://en.wikipedia.org/wiki/InterPlanetary_File_System

[2] 2020. *Mastodon: Social Networking, Back in Your Hands.* Retrieved February 3, 2020 from https://docs.joinmastodon.org/

[3] 2020. *Matrix: An Open Network for Secure, Decentralized Communication.* Retrieved February 3, 2020 from https://matrix.org/discover

[4] 2020. *NextCloud.* Retrieved February 3, 2020 from https://nextcloud.com/support/

[5] L. M. Aiello and G. Ruffo. 2010. Secure and flexible framework for decentralized social network services. In *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops).* 594–599. https://doi.org/10.1109/PERCOMW.2010.5470506

[6] Anwitaman Datta, Sonja Buchegger, Le-Hung Vu, Thorsten Strufe, and Krzysztof Rzadca. 2010. *Decentralized Online Social Networks.* Springer US, Boston, MA, 349–378. https://doi.org/10.1007/978-1-4419-7142-5_17

[7] Ching man Au Yeung, Ilaria Liccardi, Kanghao Lu, Oshani Seneviratne, and Tim Berners-Lee. [n.d.]. Decentralization: The Future of Online Social Networking. https://www.w3.org/2008/09/msnws/papers/decentralization.pdf

[8] G. Mega, A. Montresor, and G. P. Picco. 2011. Efficient dissemination in decentralized social networks. In *2011 IEEE International Conference on Peer-to-Peer Computing*. 338–347. https://doi.org/10.1109/P2P.2011.6038753

[9] Christopher Webber and Manu Sporny. [n.d.]. *ActivityPub: From Decentralized to Distributed Social Networks*. Technical Report. https://nbviewer.jupyter.org/github/WebOfTrustInfo/rebooting-the-web-of-trust-fall2017/blob/master/final-documents/activitypub-decentralized-distributed.pdf

[10] Mao Yang, Hua Chen, Ben Y. Zhao, Yafei Dai, and Zheng Zhang. [n.d.]. Deployment of a Large-scale Peer-to-Peer Social Network. https://static.usenix.org/events/worlds04/tech/full_papers/yang/yang.pdf