Decentralised Hashtag Search and Subscription for Federated Social Networks

Trolli Schmittlauch Email: t.schmittlauch[at]orlives.de
Fediverse: @schmittlauch@toot.matereal.eu

Abstract— Federated social networks are a promising architecture for decentralising social networks and their underlying power structures, and achieving more independence from monopolies. So far these networks have a serious user experience drawback: They do not properly support the concept of network-global hashtags, as each server instance has its own fragmented view on the network.
This paper proposes an additional backend for federated social networks that allows users to subscribe to all posts of a hashtag and query its global post history: Instances form a Chord Distributed Hash Table (DHT) on which hashtags are distributed as keys. This maps the responsibility for relaying and storage of posts unambiguously to a certain instance. The proposed architecture features a load balancing mechanism and variable redundancy per hashtag. Its integration into the ActivityPub federation protocol is outlined, load scenarios are analysed and simulated, and common attack scenarios against DHTs are taken into account.

1 Introduction

Social networks nowadays are an ubiquitous part of many people's life: They have become the place to share experiences with friends or the public, discuss about topics of interest, get informed about news, or report from currently ongoing events.
With so many different kinds of information available through the same channel, the need for structuring information arises. A very popular mechanism for structuring posts of the same topic are **hashtags**. Proposed first by a Twitter user in 2007[1], they first got adopted by Twitter and later on by most other major social media sites like Facebook, Instagram and YouTube. Hashtags consist of the **#** symbol followed by a keyword and allow to attach tags to a post without the need of a separate tagging field. They are especially highlighted in text and clicking them retrieves all other posts with the same tag. Today, hashtags have managed to become important tools for talking about currently ongoing events like sport games (#FRAGER), TV shows (#Tatort), political topics (#SaveTheInternet) or currently on-going demonstrations (#FridaysForFuture, #WwomensMarch, #GeziPark) and are used for information discovery and exchange. Even social movements were sparked by collections of Twitter posts under a hashtag (#MeToo, #BlackLivesMatter, #MeTwo).

While all of the aforementioned major social networks are centralised services under the control of a single company, there are multiple approaches for building decentralised social networks. One approach of decentralisation are federated networks, where multiple network servers run independently from each other, operated by different entities. Each user only has an account on one of these servers, but can also subscribe to and interact with posts of users from other servers. Decentralisation promises a distribution of power and independence from a single network owner, different content policies per server, better trust structures and moderation capabilities, and diversification of features as different software implementations just have to agree on a communication protocol for being interoperable.
When it comes to hashtags, all existing federated social networks have a serious drawback: While many of them implement the concept of hashtags, only a subset of all tagged posts of the network is displayed as result. This happens as servers in federated network do not have a consistent view on all posts of the global network of servers. Only

1. https://twitter.com/chrismessina/status/223115412

the subset of posts known to a server is returned to the user, which seriously fragments discourse and limits the spread of topics within the network. Additionally, that effect provides an incentive for users to cluster on large servers with many users as these usually know of more posts and return more post results, hindering the decentralisation of the network.

With the standardisation of the ActivityPub protocol for communication between federated social networking servers, federated social networks have gained new traction with new software and a growing user base. I deem this to be a good time for approaching the long-standing issue of hashtag search and subscription in federated social networks. In this paper I propose a decentralised architecture for searching for and subscribing to posts containing a certain hashtag that extends the current federation model. Due to its growing adoption I will focus on federated networks using ActivityPub as a protocol, although the architecture should be applicable to other federated networks as well.

The paper is structured as following: After giving an overview of the basic technologies used for the system in section 2, section 3 introduces the terminology used in this paper and lays out requirements and assumptions for the envisaged system. The drawbacks of existing related work are outlined in section 4. Network architecture, storing, relaying and querying of posts are described in section 5, followed by two sections dedicated to load balancing (6) and redundancy (7) in this system. Section 8 evaluates the placement of nodes and balancing of load using real-world data, and evaluates the system performance under typical attack scenarios for DHTs. The paper ends with an overview of potential future work (9) and a conclusion (10).
A glossary of used abbreviations can be found at the end of the paper.

2 Background

2.1 Federated Social Networks

Federated systems are a common architecture concept for distributed systems. They consist of independent servers under the control of different entities. Mostly being self-governed in their own matters, they can exchange information with other servers over a common protocol to interact with content stored on other servers. For having an unambiguous ID for content or users, IDs in federated systems usually contain the domain name of the server they reside on. Well-known examples for this architecture are e-mail and instant messaging via XMPP[2].

Started in 2007, the microblogging software "Laconica" was the first social networking server built on the principles of federation. Later renamed to "StatusNet", it resides under the name "GNU social" today[3]. The initial protocol for federation of posts, called OpenMicroBlogging, was soon replaced by the OStatus protocol. It was even on a way to be standardised by the W3C, but that process stalled in the end.
Another popular and still developed federated social network is Diaspora*[4]. Started in 2010, it does not focus on microblogging but on longer posts with pictures and comments, similar to Google+ or Facebook. Its own federation protocol is incompatible with OStatus.
Friendica[5] is a software that attempts interoperability with many different social networks and protocols. It supports OStatus, the Diaspora* protocol, ActivityPub, and can additionally serve as a client for centralised networks as Twitter or Facebook.
In January 2018, the W3C Social Web Working Group released the ActivityPub standard which includes a protocol for federation between servers. It has been adopted by several new or existing projects since, the most popular ones being Mastodon[6], Pleroma[7], Friendica, and PeerTube[8].
Several more federated network services have been or are still in development, the interested reader can find a good overview of them in the Wikipedia[9].

In federated social networks, there are many terms in use for describing a social networking server. I will use the term **instance** throughout this paper for describing a logical social networking server run by some entity.
From a user perspective, federated social networks are quite similar to centralised ones: They just need an account on a single instance, their home instance, for communicating with both accounts on that instance as well as remote users on other instances. A user's client only communicates with this user's home instance. This instance stores all of the user's data and handles all necessary federated

2. https://xmpp.org/
3. https://gnu.io/social/
4. https://diasporafoundation.org/
5. https://friendi.ca/
6. https://joinmastodon.org/
7. https://pleroma.social/
8. https://joinpeertube.org/
9. https://en.wikipedia.org/wiki/Comparison_of_software_and_protocols_for_distributed_social_networking

communication with other instances. Instances being servers for multiple users with a stable power and internet connection is an important difference to other distributed social networking approaches, where clients running on user's devices directly talk to each other.
The interaction model of federated networks is account-centered: Users subscribe to the accounts of other users, e.g. by "following" them. As each full account ID consists of the instance-local user name and the instance's domain name, it becomes clear from the ID what is the home instance of the account and thus needs to be contacted about all actions affecting that account. Delivery of new posts is mostly push-based: When subscribing to the remote account alice@example.com, its home instance is notified about the new subscriber and keeps track of them. When publishing new posts, they are sent to all instances of subscribers. This also implies that instances without any subscribers to alice@example.com *do not* get the posts sent to and thus do not know about their existence. Fetching posts from a remote users account is also possible, but only used for special cases like message thread or comment resolution, as constantly polling all known instances for new posts would have far too much overhead.

When searching for posts containing a hashtag, contemporary instance implementations only return the posts found in their local database. They are missing a lot of potential posts: Posts from known instances, but without a local user having subscribed to their author[10], as well as posts from potentially unknown instances have never reached the local database and thus will not appear in the results. Subscribing to user accounts works because there is one definite instance to subscribe to for all posts of an account. That is not the case for posts containing a certain hashtag. It is not even clear which instances exist at all, as there is no central point of registration for instances but their existence is learned through user subscriptions, mentions in posts and shares/ forwards of existing posts.

2.2 ActivityPub

While most push-based federated social network protocols and architectures bear many resemblances, I choose the ActivityPub (AP) [1] federation protocol as a basis and examine how to integrate my proposed architecture into that existing protocol.

The AP specification defines two different API layers: A client-to-server protocol and a server-to-server federation protocol. Only the latter is relevant for this paper.
AP is designed according to the actor model: All protocol messages are designed as a triplet of an *activity* by an *actor* on an *object*. The vocabulary for actors, objects and activities is taken from the Activity Vocabulary [2] and is formatted according to the ActivityStreams 2.0 (AS) specification [3]. Protocol messages are serialised as JSON-LD, which is a special extensible variant of JSON designed for linked data, allowing to normalize, link and reference objects just by their IDs. IDs are dereferenceable Uniform Resource Identifiers (URIs), where the underlying object can be retrieved from. In all current implementations, these are HTTPS URIs.

Each user is represented as an actor with an ID (its URI), an *inbox* for receiving and an *outbox* for publishing messages. These boxes are endpoint URIs used in publishing and fetching of posts and other activities. When a user posts a new message, their home instance puts a "Create" activity with the new post as the object and the author as the actor into the user's outbox, from where the user's posts can be retrieved via a HTTP GET request. Additionally, the home instance sends the same "Create" activity to each inbox of the user's subscribers using an HTTP POST request. Other user actions are handled similarly, liking a post for example results in a "Like" activity with the liking account-ID as an actor and the liked post-ID as an object being HTTP POSTed to the post authors inbox. For efficient message delivery to multiple actors on the same instance, instances may provide a *sharedInbox* endpoint. Messages sent there can specify a collection of actors as recipients and are delivered to their inboxes all at once.

User actors have several more properties. One of them is the "publicKey" property, provided for verifying the integrity of signed messages. An actor's home instance also stores the corresponding private key of its actor. While the AP specification itself does not contain measures for integrity protection, there are two current best practices for message integrity verification:
HTTP signatures [4] allow the verification of posts sent directly from its originating instance by adding a signature of the message to the HTTP headers. These signatures are not relayable, so for ensuring the integrity of posts they always have to be fetched

10. ignoring shared/ forwarded posts and posts fetched due to message thread resolution, as these special cases do not change the underlying problem

from their originating instance.
Linked-data signatures [5] on the other hand are included into the message itself and are thus relayable. These kinds of signature are currently not supported by all ActivityPub server implementations.
Verification of signatures on messages always requires the public key of their actor, which has to be retrieved from their home instance.

2.3 Distributed Hash Tables

Distributed Hash Tables (DHTs) are structured peer-to-peer (P2P) networks that can provide an efficient, decentralised lookup service of unique keys to their associated values. Both keys and network nodes share the same flat identifier space, in which keys are associated to nodes by some kind of distance metric. Commonly, the normalisation of node identifiers and keys to the same identifier space is done by applying a cryptographic hash function to each, yielding constant length hash-keys and hashed IDs. The closest node to a key, according to the particular metric, is responsible for storing the key and its associated value. All other nodes take part in routing lookup requests for a key to that responsible node. [6]
One property that makes DHTs suitable for systems with a large number of nodes and many keys is its logarithmic time and space routing complexity: Routing a lookup request to the node responsible for a key typically needs at most $O(\log N)$ steps, where N is the number of nodes present in the network. For that purpose, each node maintains a routing table of just $O(\log N)$ nodes.
DHTs are self-organising and do not need any central authority. They are stable under node arrival and leave as only keys in the neighbourhood of the changing node need to be redistributed. Non-neighbour nodes remain unaffected.

3 System Model & Requirements

3.1 Terminology

Two different terms are used for participating social networking servers throughout this paper:
Servers participating in the push-federated network of ActivityPub message delivery over HTTP are called *"instances"*. Each instance consists out of one or multiple servers all reachable under the same domain name and all operated under the control of the same entity. Communication between instances is routed directly over the normal internet infrastructure. Instances serve as the home base of users, are the only server a user directly communicates with and execute all actions affecting other instances on behalf of the user.
The additional mechanisms introduced in this paper are based on the servers forming an additional server-to-server DHT network. While the participants in this network are usually the same machines as the corresponding instance, they are now called *"nodes"* in accordance to the prevalent terminology used in work on DHTs. Node communication is routed via the overlay network created by all DHT participants.

A DHT's name space is of the size $S = 2^m$ with each position within the name space being denoted by an m-bit long identifier. N denotes the number of nodes present on the DHT.

3.2 Requirements

3.2.1 Functional Requirements

The proposed architecture shall provide the following functionalities:

- **relay & subscribe**: provide a mechanism for network instances to subscribe to all public posts containing a certain hashtag
- **store & query**: interested instances can *stepwise retrieve at least 1 year of history of posts* under a hashtag in time-descending order, even when they had not been subscribed to that hashtag at that time.

3.2.2 Non-functional Requirements

Additionally, the following requirements need to be met:

- **scalability and throughput**: The whole network of federated instances shall be able to scale to a level where it can handle the same *throughput* as today's centralised social network. A reference value from the microblogging platform Twitter[11] is an average throughput of 1,620 posts per second and a peak throughput of 143,199 posts per second [7].
- **scalable to 10000s of subscriptions per instance**: Each instance shall be able to subscribe to potentially 10000s of hashtags.
- **no central authority**: The responsibility for hashtags and their posts is distributed among the participating nodes. There also shall exist no singular trusted entity serving

11. https://twitter.com/

as a centralised trust anchor, as such organisational structures are hardly achievable in such federated systems.

3.3 Assumptions

The system's architecture takes the following assumptions into account:

- **node resource requirements**: Network nodes have $5.5\times$ the storage and $2.5\times$ the bandwidth of what their own published posts require in the network. Resource usage may scale with the size of an instance.
- **high network stability**: As all network nodes are instances of a federated social network, the network structure is expected to be relatively stable and having a low churn rate compared to P2P client networks running on user's end devices. Instances are usually running on server hardware with a permanent internet connection.
- **each node belongs to an ActivityPub server instance**: It is assumed that the proposed new hashtag federation nodes run alongside with push-federated ActivityPub server instances and thus can be reached under the same domain name as the instance over HTTPS with a valid CA certificate.
- **eventual consistency is sufficient**: While the view onto the post history of a hashtag shall be consistent no matter from which instance it has been accessed, an *eventual consistency after 1 to 2 minutes* is good enough for the use case of a near real-time microblogging network.
- **size of posts**: As posts are text-only objects of the content, often having a size limit of 500–1000 characters, and some additional metadata, we assume the size of each post to be ≤ 10 KiB.
- **size of post URIs**: Each post is identified by dereferenceable URI and can be retrieved from there. Given the URIs in existing network server implementations, we assume them to have a size ≤ 1 KiB.

3.4 Security

The following security goals have to be met:

- **access to posts**: The *retractability or deniability of posts* after their initial publishing can be as unreliable as it already is in push-federated social networks, but must not become worse. Apart from that, posts do not have to be held *confidential* as only public posts are to be processed by the proposed system.
- **integrity**: The retrieved message history's integrity needs to be ensured: A single attacker should not be able to drop specific posts on their way to a subscribing or retrieving instance without that being discoverable. At the same time, possible legal obligations for responsible storage nodes to delete posts from their history need to be made transparent.
  Integrity of the received messages must be verified as well.
- **availability**: Availability of post must be ensured, e.g. by introducing redundancy of responsibility and storage. The architecture must be able to deal with overload scenarios and must be secured against abuse for Denial of Service (DoS)-amplification attacks.

For the rest of the paper I consider the following **attacker model**:
The home instances of an individual user are trusted by them. In federated social network the instance handles the user's keys, stores and receives posts created by or addressed to them.
Attackers aim at disrupting the normal operation of the system (DoS), changing the history of a certain hashtag, or abusing the system for other attacks like DDoS amplification. That means for all attacks except the infrastructure abuse, the attacker is part of the system as well and operates one or several nodes.
Attackers are assumed to be restricted in their resources: They are limited in their financial budget and their access to IP addresses, especially in the choice of their IP subnet prefix as this gets assigned to them. Furthermore, I assume that acquiring a new domain for running a node costs a fee. While there are a few top-level domains issuing free domain names, these could be put on a deny list.

4 Related Work

There is a lot of existing work on decentralised or distributed information storage, subscription and retrieval. Some of it is already focusing on social networking applications while other work is more generic, but at first look seems applicable to the problem tackled in this paper.

Peer-to-peer (P2P) networks are networks of interconnected equal clients, where all data is stored

directly at the client peers. While the federated social networks considered in this paper do not have such an equal structure and still use the client-server design pattern, a P2P network of the instance servers might be an approach for building a backend.

- Twister [8] is a P2P social network utilising the BitTorrent protocol for post distribution. Peers interested in a hashtag form a BitTorrent swarm. Unfortunately its architecture does not provide any storage guarantees, redundancy, consistency or load balancing.
- Lilliput [9] provides a redundantly distributed storage service for social networks, but has no concept of keywords. It also is not possible to re-use its concept of user profiles for the distribution of hashtags.
- Tag-indexed DHT for scalable tag search [10] proposes a DHT structure for efficiently searching social objects by a combination of their tags. This structure though is not usable for subscribing to certain tags as posts spread over the whole network in a way that counterfeits all optimisations of that structure, creating a hot-spot bottleneck at the final responsible node. The performance optimisations of the proposed structure are also problematic from a security perspective, as each of the many intermediate nodes between the responsible one and a querying node can drop messages or deny their existence, threatening the system's integrity.
- Megaphone [11] proposes a microblogging architecture on top of SCRIBE [12] over multicast-trees. While it does not have a concept of hashtags, it should be trivial to repurpose its concept of "posters" to become a node responsible for a certain hashtag. This approach also has an integrity problem: Each intermediate node in the multicast-tree can modify or drop queries and responses going through them.

Some existing federated social network protocols already have the concept of **groups**, like OStatus Groups [12] or ActivityPub group actors[13]. These groups always contain the domain name of the responsible single instance they are tied to, lacking a non-ambiguous responsibility mapping per hashtag. This leads to fragmentation of hashtag posts to multiple disjoint groups.

12. https://www.w3.org/community/ostatus/wiki/Groups
13. currently only used in https://gitlab.com/prismosuite/prismo

Relays are centralised services forwarding incoming posts to all subscribers. Instances send all their public posts to the same centralised relay. While the Mastodon Pub-Relay [14] ActivityPub relay actor or Pleroma LitePub Relay [15] re-broadcast each incoming post to all subscribed instances, the SocialRelay [13] for Diaspora* allows subscribing to posts with a certain hashtag. Such centralised relay services are not just against the spirit of federated networks, they also are potential performance bottle necks or single points of failure. Relays without the possibility of filtering subscriptions to posts with certain hashtags are also a challenge to smaller instances, as they might not have the necessary resources for receiving, processing and storing all global posts.

5 Network Architecture

This paper introduces an additional P2P backend to ActivityPub instances: Instance servers form a Chord-style [14] Distributed Hash Table (DHT) used for assigning the responsibilities for storing or relaying posts of a certain hashtag. After the responsibility for a hashtag has been looked up, all further social network specific communication is again done using ActivityPub directly between the instances.

Each Chord node n maintains pointers to its immediate successor and predecessor nodes $successor(n)$ and $predecessor(n)$.

5.1 Assigning Responsibilities

The proposed system assigns the responsibility for the storage or relaying of posts containing a certain hashtag. It is clearly defined which instance another interested instance needs to contact for publishing, querying or subscribing to posts of that hashtag.

For that purpose, look-up keys are derived from hashtags and mapped to nodes on a DHT. Each node corresponds to an instance that, after doing the lookup on the DHT, handles all further application specific communication.

More precisely, each real node joins two different DHTs, one for its role as post relay subscription point and another DHT for its role as queriable post storage. Each of these DHTs can additionally be joined using multiple identities. Both measures are part of the load balancing strategy laid out in section 6.2. But as both DHTs work in the same

14. https://source.joinmastodon.org/mastodon/pub-relay
15. https://git.pleroma.social/pleroma/relay

way and are just used to create independent name spaces, this paper often just describes how to work with a single DHT for simplicity.

The basis for the DHT used in this paper is Chord [14], a distributed lookup protocol providing the functionality of efficiently mapping keys to responsible nodes. This particular DHT has been chosen for its simplicity and deterministic mapping properties. The determinism of mapping is crucial for all nodes getting an eventual consistent view on a hashtag, as it guarantees the same key being mapped to the same nodes no matter from where a query originated. Some other popular DHTs like Kademlia cannot provide this guarantee, as they represent their namespace as a search tree. Nodes in that tree are not guaranteed to know their exact neighbour nodes, leading to a probabilistic approach of publishing data for the same key onto multiple nodes close to that key ID.
Chord's simplicity of constrained routing tables without additional optimisations also provides a basic level of defense against several routing attacks [6]. Performance optimisations based on network proximity are unnecessary for this system, as only the lookup of responsible instances is done via the DHT while heavier communication is done directly routed.

Both nodes and hashtags are spread over the same m-bit identifier space using consistent hashing by applying a cryptographic hash function h_k to the hashtag string and h_n to the node. The terms "key" or "hashtag" can refer either to the hashtag itself or its hash, depending on the context. The same is true for the usage of the term "node" in this paper.
The responsibility for a key $k = h_k(hashtag)$ is assigned to its *successor node*, with $successor(k) = \min_i(k+i) \mod 2^m$ with $(k+i) \in$ nodes $\wedge i \in \mathbb{N}$. If representing the identifier space as a circle from 0 to $2^m - 1$, $successor(k)$ is the first node clockwise from k on this so-called *Chord ring*.

The choice of m and the used cryptographic hash functions need to be chosen in a way that makes the probability of key collisions negligible. The SHA1 function chosen in the original paper on Chord has been shown not to be collision resistant in the meantime [15]. Without loss of generality this paper defines $m = 256$ and uses SHA3 as h_k. h_n is a more complex combination of different hash functions.

5.2 Generation of Node IDs

While h_k is a single cryptographic hash function yielding a key, the function h_n for deriving node IDs is a composition of different hashes of different identifiers. This is done to prevent a single attacker from introducing an arbitrary number of node identities by introducing a cost for each node through tying it to the registration of a valid domain name, and to prevent targeting of content by arbitrary positioning of nodes by tying their position largely to their IP address.
More detailed reasoning about both the considered attacks as well as the introduced countermeasures can be found in section 8.4.

The validity of a node's ID needs to be verifiable for each other node, giving the need to deterministically calculate a node's ID only from externally verifiable information. Thus a node does not just have to be reachable by its IP (included in all direct communication anyways), but also must present a valid certificate for the domain used in its ID and needs to be reachable via that domain. This certificate has to be available for the HTTPS secured communication of the node's corresponding instance anyways.

Thus the nodeID is generated by separately hashing the first 64 bits of a node's IPv6 address and the 1st and 2nd level domain name[16]. As the first 64 bits denote the smallest possible IPv6 subnet under the control of a single entity it is valid to assume all nodes within that network being under the control of the same entity. For enabling load balancing of nodes, a virtual server identifier is appended to both the domain name and the IP subnet before hashing.
The hash algorithm used is SHAKE128 with a length of 128 bits. Domain names are normalised to the Punycode format before hashing them. The byte order of all hash inputs, as well as all protocol data, is the *big endian* network byte order and virtual server identifiers are padded to full bytes. The final node ID generation function $h_n(ip, domain, vserver)$ is defined as concatenating the first 64 bits of the subnet hash to the 128 bit long hash of the 2nd level domain and the last 64 bits of the subnet hash.

```
h_n          = hash(IPv6_addr[0,63] ++
vserver)[0,63]
++ hash(domain ++ vserver)[0,127]
++ hash(IPv6_addr[0,63] ++
vserver)[64,127]
```

16. Meaning the main domain name together with its top-level domain. Composite top-level domains ("public suffix") like `.co.uk` are considered as one level.

5.3 Joining the DHT

For joining the DHT, nodes need to know at least one other existing node as a bootstrap point. In systems like ActivityPub enabled social networking servers, it can be expected that each instance already know some other instances, either through them following a local user or a local user following a remote account. As such, an instance wanting to join a node into the DHT can just query its known instances for their corresponding nodes. How instances announce their corresponding DHT nodes is left as an implementation detail, possibilities for that include a well-known port, inclusion in the NodeInfo [16] data or using an SRV DNS record.
By preferring bootstrap nodes already having a longstanding, locally initiated connection, the probability of falling victim to freshly started hijacking attacks from new nodes can be decreased. All in all though the period length of knowing another instance does not yield reliable trust information, as, at least on instances with open registration, attackers might just create an account and follow their non-trustworthy instance at any time.

After discovering a bootstrap node n_b, the joining node n_j calculates its node ID, chooses a vserver number and calls n_b`.join(`n_j`.id, `n_j`.domain, `n_j`.vserver)`. n_b must verify the validity of n_j's ID by verifying that it can actually be reached under the specified domain and has a valid CA certificate for it, and then re-calculate the ID. n_b routes the new node to its immediate successor $successor(n_j)$ which is adopted by n_j as its successor, leaving the predecessor undefined. Stoica et al. mention the possibility of maintaining a list of successors for increased resilience against node churn, for that optimisation please refer to the original paper on Chord [14].

The `join` operation does not make any other nodes of the network aware of the node having joined. For actually updating the successor and predecessor pointers of its neighbouring nodes, each node periodically runs the `n.stabilise()` function, shown in Figure 1, to learn about newly joined nodes in its neighbourhood. `n.stabilise()` asks n's successor about its predecessor and adopts that one if it is closer to n. Additionally, it notifies the successor about their existence, causing them to be adopted as the predecessor if closer than the previously pointed to node.
Nodes must always verify the node IDs before changing their pointers to them.

After being made aware of a new predecessor now having gained responsibility over parts of its keyspace, the successor of the newly joined node must let it copy the data it is now responsible for. For the role of relay nodes that includes the list of subscribers per hashtag and all still pending delivery jobs not already due in the very near future. When operating as a storage node, the whole post history of the owned hashtags is transferred.

To detect the split of the DHT into multiple disjoint rings, nodes can periodically try to look up their own ID through a randomly chosen node from their list of known instances. If the result of this is not the node itself, it needs to run `stabilise` on that potentially new successor node and notify it of its existence.

```
// called periodically by n to verify
// its immediate successor and notify
// it about existence of n
n.stabilise():
  x = successor.predecessor;
  if x ∈ (n, successor):
    verify_node_id(x);
    successor = x;
  successor.notify(n);

// n' thinks it might be
// predecessor of n
n.notify(n'):
  if !predecessor or n' ∈ (predecessor, n):
    verify_node_id(n');
    predecessor = n';
```

Figure 1. Pseudocode of periodic stabilisation function, adapted from [14].

5.4 Node Lookup

Finding the responsible node for a key means finding the successor to that key ID. The according algorithm is shown in Figure 2.

For more efficient successor search, besides the pointers to predecessor and successor, each node maintains a routing table, called *finger table* with up to m entries. These so-called *finger* entries allow the node to at least bisect the distance to the target node at each routing step. The finger table structure is shown in Fig. 3 a): The i-th table entry of n, denoted by `n.finger[i]`, is the first node that succeeds n by at least 2^{i-1}, implying `n.finger[i] = successor(`$n + 2^{i-1}$`)`. The first finger of n is its immediate successor. As the set of nodes in the

```
1  n.find_successor(key):
2    if key == n:
3      return n;
4    n' = successor;
5    // iteratively query other nodes for
6    // their closest known node
7    while key not∈ (n,n']:
8      n' = n'.closest_preceding_node(key);
9      verify_node_id(n');
10   return n';
11
12 // search local finger table for the
13 // highest predecessor of key
14 n.closest_preceding_node(key):
15   for i = m downto 1:
16     if finger[i]∈ (n,key):
17       return finger[i];
18   return n;
19
20 // called periodically to refresh
21 // finger table entries one by one
22 n.fix_fingers():
23   next = next + 1;
24   if next > m:
25     next = 1;
26   f = find_successor(n + 2^(next-1));
27   verify_node_id(f);
28   finger[next] = f;
```

Figure 2. Pseudocode for iteratively finding the responsible successor for a key, adapted from [14]

whole DHT namespace S is sparse, only $O(\log N)$ of the finger table entries are distinct, giving the system a routing complexity of $O(\log N)$ steps.

When routing a query for a key to its target node, each node returns the closest known node from its finger table. For security reasons this system uses iterative querying, allowing the querying node to verify each returned node ID itself. An example routing process is shown in Fig. 3 b).

The finger table needs to be refreshed periodically with `n.fix_fingers` to deal with node churn.

5.5 Post Relaying, Storage and Querying

When publishing, storing, querying or subscribing to posts of a hashtag, the DHT is only used for finding the responsible instance that from that point on handles all further operations out-of-band.

After being routed to the responsible node by the DHT, that node points to the domain name of the corresponding ActivityPub instance, optionally including a list of IP addresses. This instance needs to be operated by the same entity like the node under the same domain zone. The IP address of the instance needs to be cross-checked to be either the same as the node's IP or one of the explicitly returned addresses. This response can be cached locally for a short time by the querying node. Future requests then do not need to consult the DHT anymore.

When a new post containing hashtags is published, the originating instance computes the hash $h_k(hashtag)$ for each contained hashtag and looks up the responsible nodes on the DHT containing all relay-role nodes, obtaining the responsible instances. The published full post is then sent to these relay instances. The signatures of incoming posts must be verified before forwarding them to all subscribing instances of that hashtag by retrieving the post author's key from its home instance.
For privacy reasons instances only relay the dereferenceable post ID (a URI) instead of the full post content. One reason is that not all ActivityPub servers support relayable linked-data signatures on posts, but more importantly not relaying signed post content provides better deniability of posts even after publishing: If a post has been accidentally published, it can be retracted any time by the publishing instance. Only the originating instance can provide a post properly signed with an HTTP signature and can always decide not to serve that post anymore. When relaying a post URI to a large number of instances and making them fetch it at nearly the same time, care must be taken not to overload the original publishing instance: All received fetch jobs within a short random time frame have to be gathered as a batch. If fetching of posts fails, retries have to be scheduled by a randomised exponential backoff algorithm.
Additionally to the subscribing instances, the relay instance also forwards the post URI to the instance found to be responsible by looking it up on the DHT of storage-role nodes, where it is saved for later queries.

User actions leading to instances (un)subscribing to or querying posts from hashtags work similarly: The instance computes $h_k(hashtag)$ as a key, gets routed to the responsible node on the DHT pointing to the responsible instance and there (un)subscribes that hashtag or fetches the stored post URIs and then dereferences them locally.

All these application layer operations (except for the routing) are to be handled via the ActivityPub

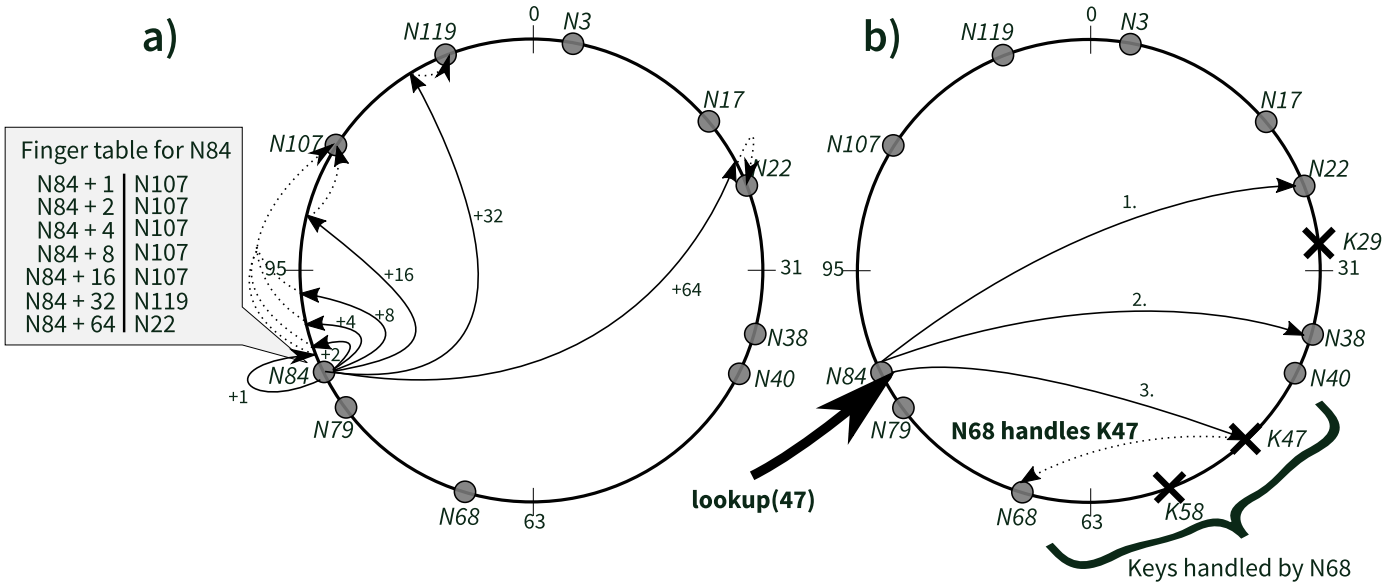

Figure 3. **a)**: finger table structure of node 84, **b)**: iterative lookup of key 47 from node 84 on a Chord ring with $S = 2^7$

protocol [1]. Being based on *JSON-LD*, it is inherently extensible. While defining an actual protocol schema is out-of-scope of this paper, here I lay out some possible ways of doing so.

Each hashtag can be represented as an actor with its outbox and inbox. When publishing posts, they are sent in `cc` to the inbox of the actor working in its relay-role. Subscribing to a hashtag can be done by sending a `Follow` activity to the hashtag actor. Querying a post's history can be done by fetching from the outbox of the storage-role actor.

For integration into existing ActivityPub server software, it might be feasible to offload large parts of the DHT-specific routing process to an application proxy. All actors (like hashtags) that need to be addressed using DHT routing can be addressed using a new URI scheme, with the application proxy transparently replacing the actor URIs with the values resolved from the DHT lookup.

5.6 Leaving the DHT

When leaving the DHT voluntarily, a node sends its stored keys to its successor. After successful transfer of all data, both successor and predecessor of the leaving node are notified to adjust their successor and predecessor pointers accordingly.

For discovering node failures, each node periodically checks whether its predecessor node is still reachable. If that has failed, the predecessor pointer is invalidated.

5.7 Cleaning Up Old Posts

To reclaim storage space, post storage instances may delete posts older than 1 year if and only if they require storage space to be freed. Stored posts are to be deleted in a chronological order.

Additionally posts may be deleted if no more storage space is available and load balancing does not allow to offload storage. To avoid displacement of posts by flooding neighbouring keys with posts, the deletion strategy has to take fairness between the hashtags into account. A possible strategy is the round-robin deletion of the oldest post per hashtag.

6 Load Balancing

For load balancing, DHTs like Chord rely on the assumption that, thanks to consistent hashing, both keys and node IDs are uniformly distributed over the key space. This is not the case for our node IDs due to the incorporation of the IP subnet into ID generation (see section 5.2). But more importantly each key represents only a hashtag and the collection of posts containing it. Given the unequal distribution of posts over different hashtags, this results in an unequal distribution of posts over nodes of the DHT with a potential to cause storage and network load issues.

Most existing work on load balancing in DHTs allows nodes to freely choose their own node IDs [17, 18, 19, 20] or assumes equally-sized storage items per key [21, 20]. But giving nodes the ability to arbitrarily choose their IDs makes NodeID- and Sybil-attacks possible (see 8.4) and gave attackers

the ability to deliberately choose a range of hashtags they are responsible for. Thus the aforementioned approaches are unsuitable for this use case.

6.1 Distribution of Hashtag Usage in Posts

The requirements of load balancing for a post relaying and storage system heavily depend on the characteristics of how hashtags are actually used in in real-world posts.
For this purpose I was able to analyse the occurence of hashtags in posts on Twitter, one Diaspora* and two Friendica instances.

6.1.1 Working Hypothesis

I expect the distribution of posts per hashtag to be very imbalanced, following a power law distribution, such that the majority of hashtags is only used in very few posts and the majority of posts including hashtags only use a small percentile of all hashtags. This should be similar no matter whether a network is centralised or federated.

Regarding the number of post per hashtag over time I expect there to be 2 different categories of hashtags:
Special-purpose hashtags are used to refer to a very precisely limited current topic like an event currently happening (demonstrations, conferences, TV shows, political events). These hashtags show a relatively short-lived peak in the number of posts over just a small time frame with the number of posts diminishing before and after.
General topic hashtags on the other hand are longer lived and have a more constant post rate over time. They are used for tagging posts to contain content of a more general topic not bound to certain events or current discussions.

These hashtag usage dynamics are expected to differ between centralised and federated micro-blogging networks: As centralised networks can efficiently provide a full-text search over all posts, only the most relevant special-purpose hashtags are included and everything else can be found via the search. With my proposed architecture, users of federated micro-blogging networks though cannot do a global full text search but can only search for or subscribe to topic-related tags, incentivising users to include more general topic hashtags into their posts.

6.1.2 Analysis of Real-World Data

For analysing the occurence of hashtags in posts I used a dump of 1% of all Twitter posts between March 03 and April 01 2019 as well as data sets about hashtag usage on https://pod.geraspora.de/ [23], one of the most popular Diaspora* [17] instances, and on the two Friendica [18] instances https://pirati.ca/ and https://squeet.me/. The Geraspora data set spans 1 year from 2nd Feb 2018 to 1th Feb 2019 and the Friendica data sets span from August 2012 (pirati.ca) or respectively October 2014 to March 2019.

The total distribution of posts per hashtags (Fig. 4) confirms the hypothesis of a strong power law distribution: The majority of hashtags is used in little posts: The proportion of tags just appearing once in the analysed post dump is 69.7% for Twitter[19], 58.9% for Geraspora and 13.6% for Friendica[20]. A small amount of hashtags has a high number of posts and thus will have to bear the highest storage and network load: For Twitter, the hashtag with the most posts is just one of 1428165 tags appearing in the post dump, but makes up already 1.4% of all posts. Only 0.17% of the hashtags cover half of the posts. Obviously, a feasible load balancing has to be able to deal with a heavily skewed load distribution.

For understanding the development of post rates per hashtag over time I analysed a one-month-long interval of posts for each network (Fig. 5). For Twitter this was the time between 4th March and 4th April 2019, for all other networks the whole January 2019 was analysed.
The result confirms the hypothesis of two different kinds of hashtag usage: All network graphs show both hashtags consistently used over longer periods as well has short-lived tags with steep peaks. Differences between the different platforms are not obviously apparent. While very large peaks are only to be found in the Twitter posts, this could be caused by the federated, non-global view of the other platforms. It is notable that Friendica also receives posts from Twitter and Diaspora* servers and that Diaspora* allows subscription to hashtags (although only yielding locally available posts) in its user interface. As this phenomenon of more topic hashtags in federated social networks is likely to become stronger with a working global hashtag federation system, it is hard to estimate from current data anyways.

This distinction is relevant for estimating the

17. https://diasporafoundation.org/

18. https://friendi.ca/

19. It has to be noted, that the dump only contains 1% of all posts during that time.

20. Due to the aggregation of 2 instances, a single-post hashtag present on both instances already appears to have two posts.

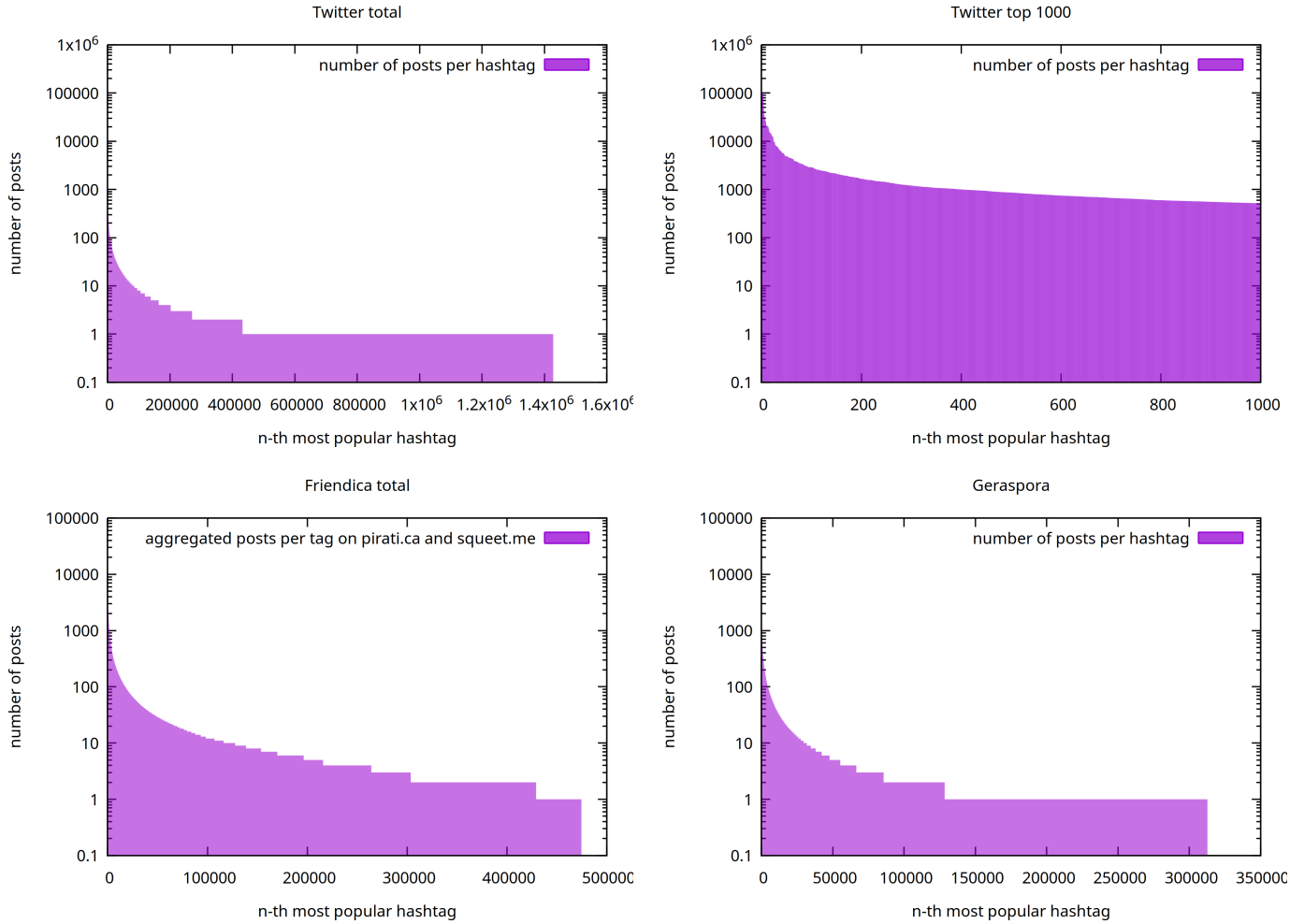

Figure 4. total number of posts per hashtag: The distribution of posts on hashtag clearly follows a very steep power law in all inspected networks. With a linear y-axis scale, the plots would look almost empty.

network load and storage requirements of nodes: The peaking post rate of special-purpose hashtags means a high network load in a short time frame. The storage requirements are defined by the number of posts during the peak time, making them almost constant after the peak has passed.
General topic hashtags though usually do not have such a high peaking post rate as special-purpose ones, making the network load more constant and the post storage requirements growing constantly over time.

The resulting load balancing problems are:

1) balancing the network load on instances of receiving and sending posts
2) balancing the required space for storing posts

This becomes evident when looking at edge cases for each of these two resources:

Storage load peaks appear at hashtags with a continuously high posting rate. The responsible storage instance has to store the post URIs for at least 12 months. Taking the tag #NowPlaying, a tag with one of the highest continuous post rates in the Twitter dump, as an example, the extrapolation of its 21,026 posts over 1 month to a full year looks the following:

$$\frac{21,026 \text{ posts} \cdot 100 \cdot 12 \text{ months} \cdot 1 \cdot 1024 \text{ Bytes}}{2^{30}} = 24.06\text{GiB}$$

Storing 24 GiB of data for a year is manageable for a single node. Still a mechanism for offloading full hashtags to other nodes is needed to avoid multiple storage-intensive hashtags falling under the responsibility of a single storage instance.

Network load peaks arise at relay instances when a large number of posts containing a certain hashtag are created: All these posts are sent to the

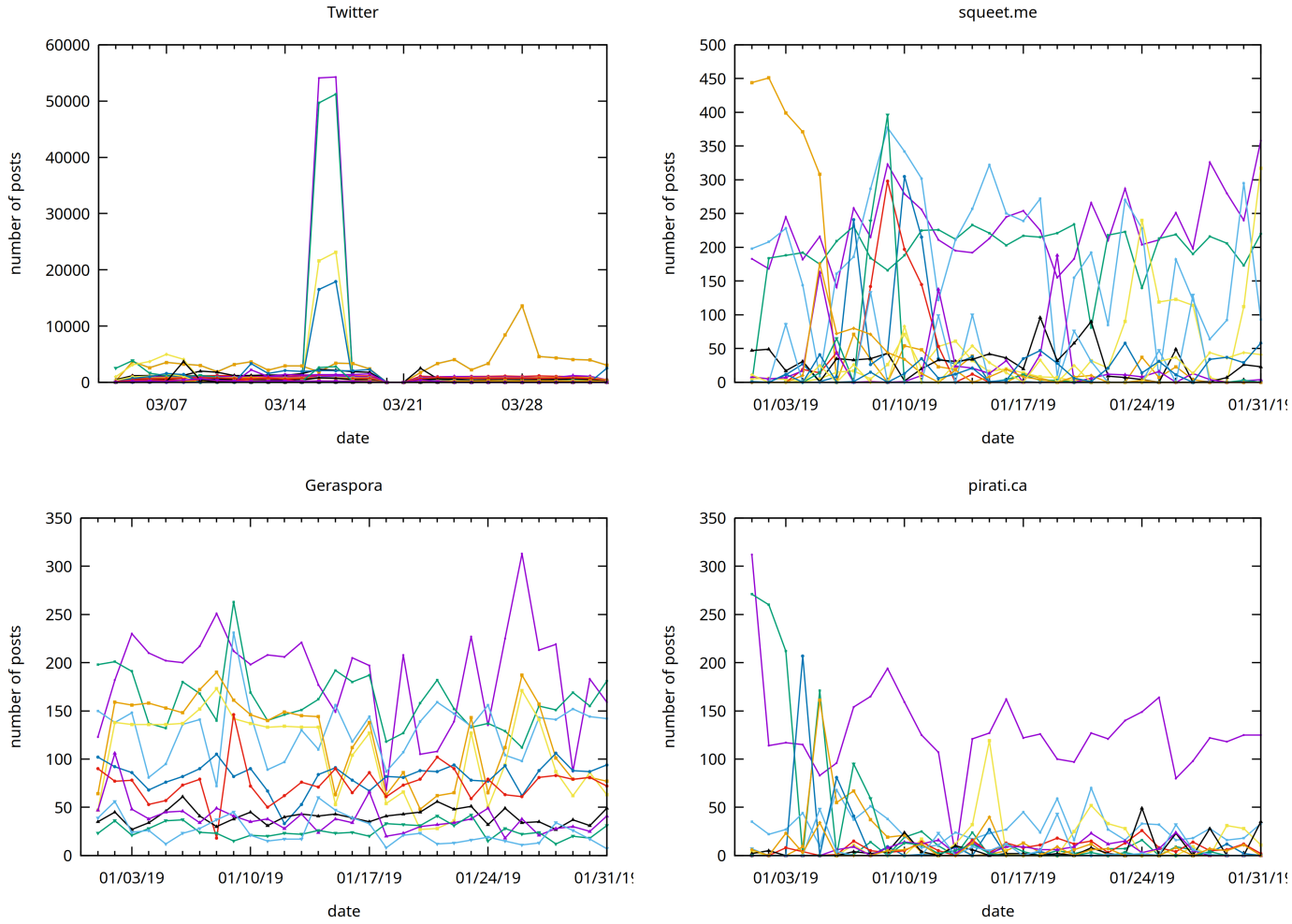

Figure 5. posts per hashtag per day: Development of the usage of several hand-picked hashtags shows clear differences between short-peaking special-purpose and constant-rate topic hashtags.

relay and then have to be distributed to a potentially large number of subscribers each. This large number of requests can cause similar effects as a Denial of Service (DoS) attack, overloading the instance and its link.

When taking the highest peaking Twitter hashtag from Figure 5 as an example, the number of incoming posts for the responsible relay instance is a $54276 \cdot 100$ post peak in 24 hours. Assuming 1000 subscriber instances, the resulting post rate is at least

$$\overbrace{\frac{5,427,600 \text{ posts}}{86,400 \text{ s}}}^{\text{incoming posts}} + \overbrace{1000 \cdot \frac{5,427,600 \text{ posts}}{86,400 \text{ s}}}^{\text{forwarding to subscribers}}$$
$$= 1001 \cdot \frac{5,427,600 \text{ posts}}{86,400 \text{ s}} \approx 62,882 \frac{\text{posts}}{\text{s}}$$

This number of requests is not trivial to handle, especially for smaller nodes, and requires an uplink bandwidth of up to $490.8 \frac{\text{MBit}}{\text{s}}$ and a downlink bandwidth of up to $4.9 \frac{\text{MBit}}{\text{s}}$. Even more, probably the real peaking rate of posts was much higher as this is just an average rate over 24 hours. For that reason, both a load balancing mechanism allowing to offload certain keys to other instances as well as splitting the load of one hashtag among multiple nodes are needed.

This shows that it is necessary to balance the DHT keys and the different amounts of load behind them between nodes. Additionally, some popular hashtags with high network load require a way of splitting up a single hashtag over multiple nodes.

6.2 Load-balancing of Different Hashtags Between Nodes

For balancing the load of different hashtags between nodes we use a variation of the *k-choices* algorithm [24] by Ledlie and Seltzer. This algorithm uses the concept of *virtual servers (VSs)* [25] where each real

server node can join the network with multiple IDs as multiple virtual nodes. For this purpose, both the domain name and the IP subnet are concatenated with a *virtual index* c before hashing. With this index being limited by a well-known limit κ, each node can choose from κ positions in the DHT and activate any subset of this set $VSset$ of identities. Figure 6 shows the $VSset$ of 3 nodes with 8 active VSs each and their distribution on the Chord ring.

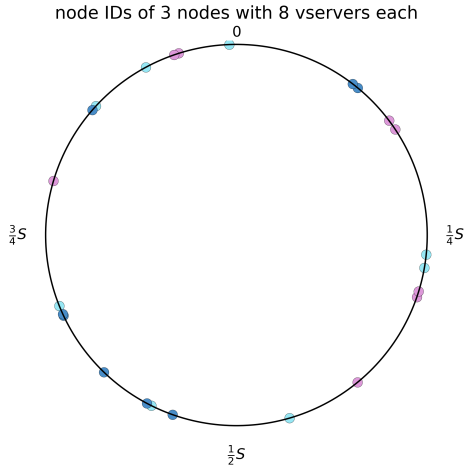

Figure 6. Plot of VS IDs on a Chord ring for 3 nodes with $\kappa = 8$. Any subset of these IDs can be activated for load balancing.

Each node has a capacity C_i. For resilience against fluctuations nodes want to work below their capacity and thus define an upper limit $U_i = C_i - U_\delta$ above which they consider themselves overloaded. Similarly, there is an underload limit L_i. Based on these two limits a node defines a target workload for itself. The k-choices algorithm attempts to minimise the mismatch between target load and actual load by guessing the expected load change when (de)activating one of its potential VSs, both for the node itself and its neighbours. Considering the target workload of both the active re-balancing node as well as its neighbour helps to avoid cascading effects, where moving load away from an overloaded node makes the receiver of that load overloaded. While the original algorithm guesses the load change just by assuming the load of the currently responsible node being split according to the percentage of changing address space, this assumption can not hold for this system due to the very imbalanced distribution of resource requirements per key. Thus I propose to query more fine-grained statistics about a node's current resource usage, at least with a resolution of smaller key intervals or even per key.

```
function node_join():                                                    1
  // set total target workload as mean                                   2
  // of over/ underload thresholds                                       3
  T_a = (U_i + L_i)/2;                                                   4
  i = κ/2;                                                               5
  while T_a > 0 and i > 0:                                               6
    T_a = T_a − vs_join(T_a);                                            7
                                                                         8
function vs_join(t_a):                                                   9
  K = {k_0 = genID(0), ..., k_{κ−1} = genID(κ − 1)};                     10
  // remove IDs already in−use                                           11
  K = K − VSset;                                                         12
  for each k in K:                                                       13
    (w_s^(n), t_s) = query_stats( succ(k) );                             14
    w_a = Σ_{i=pred(k)}^{k} w_s^(n)[i];                                  15
    w_s^(f) = Σ_{i=k}^{succ(k)} w_s^(n)[i];                              16
    c = |t_s − w_s^(f)| + |t_a − w_a| − |t_s − Σ w_s^(n)|;               17
  join at k with minimum c;                                              18
  VSset.add(new VS);                                                     19
  return w_a;                                                            20
```

Figure 7. modified *k-choices* join algorithm. Indexes: $_s$ for successor node, $^{(n)}$ for current and $^{(f)}$ for future values

After applying these modifications to k-choices, we get the algorithm in Fig. 7.
When a node joins, `node_join()` is invoked. The node's total workload target is set to the mean of the overload and underload limits. The initial number of VSs to distribute that load on is set to half the global limit κ, leaving some VS identities available for later reactive load balancing. Then VSs are joined one by one until either the workload target is reached or `i` VSs have been created (lines 6-7).
For creating a virtual server, first the set K of all possible κ IDs is generated and reduced to contain only unused IDs. For each of this possible DHT positions the successor node – the one currently responsible for the name space preceding its ID – is queried for its current workload statistics per key $w_s^{(n)}$ and workload target t_s. Based on this workload data, the joining node estimates the load distribution after joining by summing the load w_a of all keys it would then be responsible for and the remaining load $w_s^{(f)}$ for the successor node. The cost function c gives the resulting difference between target work and estimated real work by normalising all work loads by the corresponding node load targets. The potential ID with the lowest

cost is chosen to actually join the DHT using the normal join mechanism.

As this system is assumed to have only a low churn rate, active periodic re-balancing over time is necessary. For this purpose a node periodically picks the VS from its VSset with the maximum mismatch and checks whether relocating that node lowers the aggregate cost mismatch of itself and its neighbours. Relocating a VS means leaving the DHT with one VS ID and joining again with another ID from the node's VSset. While even the detailed technical report on *k-choices* [26] does not elaborate any details about calculating the maximum mismatch, its simulator code[21] reveals the approach of choosing the VS yielding the smallest departure costs $c_d = |t_s - w_s^{(f)}| - |t_a - w_a| - |t_s - \sum w_s^{(n)}|$ with $w_s^{(f)}$ denoting the combined workload of the departing node and its successor after departure.
Whether a relocation is actually done is influenced by the dampening parameter ϵ, preventing relocations yielding just a minimal cost improvement. This parameter is supposed to represent the application-dependent costs of relocation like transfer of state and storage between nodes. In this system ϵ generally is smaller for relay nodes than for storage nodes as the latter need to move their post archive to other nodes. For storage nodes ϵ is proportional to the size of posts needing to be moved. For relay nodes ϵ scales with the size of the subscribers list and number of remaining posts in their backlog.

Additionally, nodes relocate virtual servers when reaching their overload or underload thresholds U_i and L_i. Once a node has relocated more than $|VSset|$ times it is allowed to create additional or destroy existing VSs (still adhering to the global limit κ) instead of just moving existing virtual servers to another ID.

The two roles of a node feature different critical load requirements: For the relay node role I assume network bandwidth/ post throughput to be the critically limited resource, as the amount of data having to be stored locally is negligible. The role of being a storage node though requires the permanent storage of a large amount of posts. As network bandwidth is only consumed by receiving each post once, storage space is assumed to be the critically limited resource for this role.
Thus each role has its own DHT name space with capacities and workloads on the relay node DHT representing network throughput, while on the storage node DHT different variables representing storage capacities are used. Nodes usually join both DHTs and can bear both roles simultaneously for different key spaces.
Additionally this allows relocating these roles independently from each other with different ϵ parameters.

21. available at https://people.csail.mit.edu/ledlie/lb/

6.3 Load-balancing a hashtag over multiple nodes

Balancing the network load of a single hashtag over multiple relay nodes is part of the redundancy scheme applied to this system.

7 Redundancy

Nodes and instances of the system cannot be considered as absolutely reliable: Nodes or their uplinks might fail any time, leaving the DHT in an uncontrolled way and taking all their data stored with them. For keeping all data available even under node and instance failures, *data redundancy* is required.
Another kind of redundancy is *routing redundancy* with the aim of increasing the diversity of routes leading to responsible node, making malicious misrouting detectable and preventable.

My approach on redundancy for this system is based on the ideas of Cyrus Harvesf and Douglas M. Blough [27]: Replicas of keys are placed in equal distances from each other on the Chord ring. These equidistant nodes form a redundancy set of nodes responsible for the same key. The reasons for using this approach are its support for different redundancy set sizes (redundancy factors) per key and its focus on both storage and routing redundancy. The spread of keys over the whole DHT namespace, instead of spreading key copies over neighbouring nodes like the original Chord [14] paper proposes, also improves the variety of routes used to access key data. While the authors' main motivation behind that algorithm was providing routing redundancy, it also achieves data redundancy.

This system also uses the mechanism of dynamically managing the size of redundancy sets for distributing the network load of a single hashtag over multiple nodes.

Replicas of the key k are to be placed on the Chord ring in equal distances. For a redundancy factor R the various key locations form the redundancy set $Z = \left\{k + i \cdot \frac{2^m}{R} | i \in \mathbb{N} \wedge i < R\right\}$. For achieving stable replica locations, the factor R needs to be a power of 2: Increasing the redundancy set size by

one step j in $R = 2^j$, all existing replicas remain at their position while the same number of replicas is added. Similarly, when decreasing the set size by one step, half of the replicas are deleted while the other half remains at its position. Redundancy sets are always managed at a per-key level, so a node may have to deal with different Rs for each of its managed keys.
The default value of $R = 2$.

As this approach maximises the distance between replicas, it is unlikely that multiple replicas are mapped to the same node ID on the DHT. But as a physical node can be present on the DHT with multiple VS IDs for load balancing purposes, it cannot be ruled out that replicas of the same data are still be mapped to the same physical node via different VS IDs. In this case, each VS is supposed to act like a standalone node and send, store, and receive all data separately per VS. Implementations may still adopt optimisations like storage deduplication, but those must be transparent for the apparent external behaviour of the VSs.

After describing the general replica placement strategy, the following sections describe the application specific logic of managing redundancy.

7.1 Redundancy of Relay Nodes

Subscription and unsubscription requests for key k are always sent to all R instances of its redundancy set and are saved together with the time-stamp of the request. All instances of the set periodically synchronise their list of tag subscribers. For simplicity, a last-request-wins approach is used for conflict resolution, giving the last subscribe or unsubscribe request of an instance priority. Evaluating the usage of a more reliable synchronisation algorithm is left for future work.

When publishing new posts, publisher instances send their post to $\log_2 R$ of the responsible nodes at random, which forward the post to all subscribing instances and all active redundant storage nodes. As this causes posts to be delivered multiple time, receiving instances have to take care of deduplicating the received posts themselves. The response sent to the publisher contains the current R.

For having hot spare nodes readily available in case a *flash crowd* situation caused a sudden spike in traffic, nodes from the current Z already notify the additional nodes from the next larger redundancy set size via the DHT, making their corresponding instance periodically pull the current subscriber list alternating from at least 2 of the currently active relays. Once an instance is in danger of becoming overloaded, it returns an error code to the sending instance which in turn has to try the next instance from Z. Overload situations occur if an instance has nearly no bandwidth available anymore or their queue of pending jobs has continuously been growing over a short time frame, exceeding a threshold. While overload situations are also handled by the load balancing mechanism described in section 6.2, flash crowd situations usually cause quicker load spikes than what can be handled by the *k-choices* algorithm due to its probing overhead, data transfer times and latency before creating or deleting virtual server IDs. Additionally, this mechanism allows splitting a single hashtag with a load too large for a single instance over multiple instances.
As soon as an instance cannot publish its post to the required number of relay instances, it doubles R and starts using the hot-spare nodes, which now become active parts of Z. The new hot-spare nodes are notified about their duties by the previous hot-spare nodes, in case a further increase of the redundancy set needs to happen.
After a well-known cooldown time all relay nodes belonging to the next smaller redundancy set size contact the additional half of relay nodes if they are not overloaded anymore and synchronise their subscriber list to the latest state. Once a node has been contacted by all relay nodes of the next smaller redundancy set size, it stops acting as a relay and responds with a status code indicating the shrunken R. If shrinking does not happen due to not all overloaded nodes having recovered yet, the cooldown time until the next attempt is doubled.

As an additional protection mechanism against clients arbitrarily triggering the increase of a redundancy set, active relays may regularly sign a token, including the current time stamp, using their public key. This code is returned to publishing instances in overload situations and is presented to the hot-spare nodes as an attestation of a real overload situation.

7.2 Redundancy of Storage Nodes

Normally, storage nodes are supposed to store the full history of post links of a hashtag over the time of a year. While splitting the storage is not likely to be necessary for reasons of missing available space, storage nodes may get overloaded in flash crowd scenarios due to many requests or limited processing power.

As a response to relay instances delivering post URIs to be saved, storage instances return the num-

ber of partitions $P = \frac{R}{4}$ the key has been split into and the part $p = ((i \mod P) + 1)$ they are responsible for. At the initial $R = 4$ the number of partitions is 1 as the key is still unpartitioned. Unpartitioned keys are the desirable state for all non-overloaded cases.

Once a storage node becomes overloaded, it doubles R and starts splitting the responsibility for post (URI) storage by the hash of the received content. At each split the previously held responsibility interval of hash values is split in the middle. More formally, each node i from a redundancy set Z is responsible for all posts of a certain hashtag with $(p-1) \cdot \frac{2^m}{P} \leq \texttt{hash(content)} < p \cdot \frac{2^m}{P}$. This partitioning scheme keeps the redundancy factor of 4 for each part, while the total number of storage nodes for the whole key is doubled. The split is signalled in the response sent to incoming posts, posts under the responsibility of another node are rejected. As querying clients have to contact P storage instances anyhow to get a complete post history, already stored posts can remain on the overloaded node, but newly incoming posts are distributed according to the partitioning.

The nodes newly added to the redundancy set notify the previous set of responsible storage nodes about the doubling of R. Shrinking the redundancy set after a cooldown time works the same as for relay nodes, with the difference of merging the post store instead of a subscription list.

8 Evaluation

The proposed system design contains several measures for balancing load between nodes or protecting against attacks. But how do these measures work out in practice?

This section evaluates how the node ID generation mechanism performs with empirical node data, how well storage load is distributed between nodes, and how prone the system is to common attack scenarios.

8.1 Placement of Node IDs

Figure 8 shows how the node ID generation scheme works out in practice when applying it to all 1036 ActivityPub instances listed in the https://instances.social/ directory that were reachable over IPv6 on April 4th 2019. The left column shows the spatial distribution of nodes and their distances for a single node ID per IP address – domain name pair, while the right column shows these data for 8 virtual node IDs per instance data pair.

The ID distribution histogram shows two obvious large peaks, signalling clusters of nodes in close proximity to each other. Analysing the nodes put into these bins reveals the reason for that: The first peak stems from https://masto.host, a managed hosting provider for Mastodon instances running all its instances under the same IP address. The second peak is caused by instances of different operators who though all decided to hide their real instance IPs behind a reverse-proxy operated by https://cloudflare.com/ with all these proxies running within the same IPv6 /64 subnet. With the IP subnet determining a large extent of the node ID, such set-ups lead to clustering of node IDs. The disadvantages of node clustering need to be weighed against the security gains of hindering an attackers mobility. The load balancing evaluation has to show whether this trade-off is sensible.

8.2 Load Model

For evaluating the effectivity of the load balancing algorithm for storage load balancing, a mock system first needs to be equipped with test data. Achieving a realistic distribution of data in a distributed system is not trivial, as consistent data sets of all instances are hardly available. The test data distribution used here is just a potential possible scenario.

As the system is supposed to scale to a competitive level to Twitter, I take the one-month data set of Twitter posts as a basis. Extrapolating the full number of posts per year from a 0.1% random sample of Twitter posts, the resulting size of stored post (1 KiB per post) is placed on a simulated DHT ring with the default redundancy factor of 4.

The nodes to be placed on that DHT are taken from the https://instances.social dump already used in section 8.1. This dump also contains the announced total number of posts per instance. This number of posts is used to derive the node's capacities for the k-choices load balancing algorithm: The total number of posts cannot be taken as the number of posts per year due to different life times per instance, so it cannot be multiplied directly with the size of a single post. Instead, the total size of (non-redundant) posts stored in the DHT is multiplied by a factor of 4.5[22] (see assumption of node resources in 3.3) and then distributed among the instances in proportion of the instance's total number of posts.

22. The assumed factor of 5.5 can be reduced by 1 as the posts of the instance's users, having to be stored anyways, are not relevant for simulation of post storage by hashtag

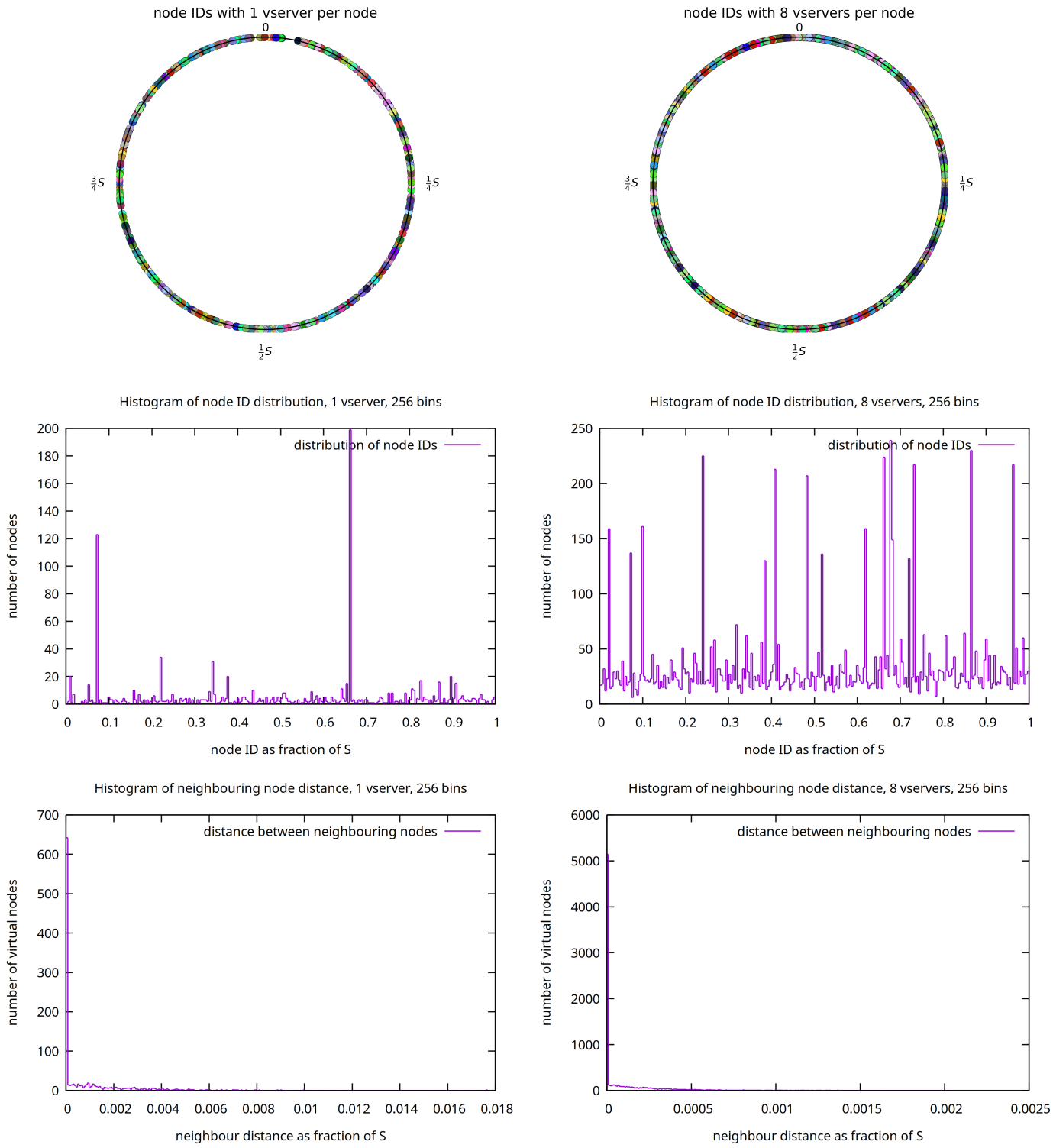

Figure 8. plot of the IDs of 1036 real-world instances on the Chord ring and their distribution over the DHT name space, with 1 or 8 vserver identities per node (see section 6)

The load balancing for relay nodes is not simulated for 2 reasons: First, there is no real-world data available regarding the subscribtions between instances or their available network bandwidths, giving assumptions no basis to start with. Second, dynamically simulating the load on instances due to message sending and receiving over a long time frame is rather complex and resource-intensive, surpassing the scope of this paper.

8.3 Load Balancing Evaluation

For evaluating the effectivity of k-choices load balancing in the system, I compare the distribution of load among nodes between two system setups: In the setup without any load balancing, each instance is only represented as a single virtual server on the DHT with a fixed position. In the other setup the k-choices load balancing algorithm is used, giving each each instance the choice between any subset of κ VS positions on the DHT.
The simulation was done on a naive Chord implementation in Python. Stored post data was assumed to be static: After populating the DHT with post data, nodes joined their VSs according to k-choices. After joining, nodes had the possibility of re-balancing their VSs choice of active VSs for 20 times. This number of load balancing runs was chosen to give nodes the ability to drop or create additional VSs, as only 1 in $|VSset|$ runs allows to do so. The order in which the nodes did their load balancing was randomised in each run.
Due to time and resource constraints, only a single simulation was done with a single set of parameters: The overload threshold was set to 95%, the underload threshold to 35% of the node's capacity and $\kappa = 8$. These values are arbitrary estimates, the choice of κ is based on the original k-choices paper [24], although that considered a different kind of load scenario. ϵ was considered to be 0 and data transfer times between nodes have been neglected.

For each node, the mismatch between the actual workload of the node n and its capacity, $w_n - C_n$, are calculated. For evaluating the total workload fit, also including a mismatch in the direction of underload, the absolute workload mismatch $|w_n - C_n|$ is calculated as well.

	unbalanced	**balanced** (best)
over capacity	525 nodes	42 nodes
average mismatch	1.59 GiB	−9.40 GiB
abs. average mismatch	57.60 GiB	49.11 GiB
median mismatch	0.00 GiB	−0.08 GiB
abs. median mismatch	4.21 GiB	0.12 GiB

Table 1
Average and median (absolute) mismatches between node work load and capacity for simulated DHT, with and without load balancing.
total size of all posts: 34332.27 GiB
total number of nodes: 1036

Table 1 shows the average and median mismatch values and the number of nodes loaded over their capacity.

Without load balancing, 40% of all nodes are overloaded. The average mismatch being positive shows that nodes are overloaded on average. Despite the median mismatch of 0, the absolute mismatches being large shows a bad general workload fit.
With load balancing enabled, the least amount of overloaded nodes was achieved after 16 re-balancing runs with just 42 overloaded nodes. After that, the number of overloaded nodes fluctuated around 50. The average and median nodes are now under their capacity (indicated by a negative mismatch), and the smaller absolute mismatches indicate a better workload fit.

While $\approx 3.2\%$ overloaded nodes are still too much for practical use and required some nodes to drop posts from their history, this result shows the effectivity of k-choices balancing. It has to be considered though that this simulation only used a single, estimated set of parameters, and did not implement splitting of keys as part of the redundancy mechanism. Furthermore, during implementation of the k-choices algorithm according to its paper [26] it turned out that some details of the algorithm are ambiguous and needed to be guessed.

The large difference in overloaded nodes clearly shows that the k-choices load balancing algorithm is efficient. Whether it is sufficient, which parameters have to be used and how it behaves in dynamic load situations with continuous change of load between load balancing runs needs to be evaluated in future work with a more sophisticated implementation.

8.4 Privacy & Security

Adding hashtag federation to federated social network using this proposed system in most cases does not raise any serious privacy concerns: Only public posts are handled by the system, and as only the URI from where a post can be fetched is relayed to subscribers the originating instance can choose to stop providing that post at any time. Instances subscribe to tags on behalf of their users, they hide the interests of particular users unless the set of active users on the particular instance is too small.

While DHT networks provide an efficient way of structuring a P2P lookup network with only knowing a small subset of its nodes and no central authority, exactly this missing global view and lack of central trust anchor poses several security challenges.

Urdaneta et al. [6] provide a broad survey of common security threats in DHT security and attempts of solving them.
Routing attacks try to alter or disrupt the routing

process within a DHT to prevent the resolution of keys, let them point to invalid nodes or even to relay the combined lookup requests on a popular ID to a third party server as a Distributed DoS. **Storage attacks** attempt to alter or forge the stored payload data returned under a key by impersonating the responsible node or taking over the relevant key space. In this system, controlling the data returned from a key allows taking over responsibility for the storage or relaying of certain hashtags. Such attackers can then deliberately drop or insert posts for a hashtag.

An important mean or precondition for many of these targeted manipulations are **Sybil attacks**: By enabling a single attacker to join the DHT with a large number of identities, posing as independent nodes, it can subvert mechanisms based on the assumption of one entity only controlling a low fraction of nodes, like majority voting systems. Additionally, introducing a large number of node identities increases the number of keys handled by the attacker and the likelihood of controlling a certain key or being included into a finger table, especially if node IDs are equally distributed.

A similar attack scenario are the **node ID attacks** (also called *node squatting*), where attackers try to control a certain keyspace or disrupt the routing by deliberately positioning a node close to or even directly at a certain ID.

Having gained the possibility of deliberately inserting multiple nodes at targeted IDs, an attacker gains the ability of isolating nodes from the network: As it is deterministic which neighbours a node tries to contact as successors or finger table members, by deliberately taking over a sufficient amount of these IDs a node can be cut off ("eclipsed") from the true view on the real network. As nodes only learn about the global network through their limited number of neighbour nodes, these being malicious can thus poison nodes' routing tables with arbitrary information. This attack is called **Eclipse attack** or **routing table poisoning** [6].

Urdaneta et al. [6] conclude that a secure DHT implementation thus requires a secure method of node ID assignment, keeping the fraction of malicious nodes low and spread, data replication and a reliable routing mechanism.

When it comes to preventing attackers from gaining to many node IDs, common approaches against Sybil attacks rely on registration and certification of nodes at a central trusted authority. This trust anchor is supposed to verify that nodes claiming to be different are actually not run by the same attacker. Such an authority does not exist in federated social networks[23] and establishing it would subvert its inherent characteristic distribution of power. Other approaches try to detect the creation of many identities from a single node by measuring network proximity. Not only does this method not fit the possibilities of creating many nodes in different physical locations in the age of cloud computing, but it is also error-prone. My system adopts the approach of tying a certain cost to the creation of new nodes: I assume that registering a domain name involves a certain registration cost. The hash of the main domain name (second and top-level domain) is used as part of the node ID.

For countering node ID attacks this is not sufficient: Attackers can compute a pre-image of the targeted node ID hash and then register the result as a domain, not needing to spend a lot of money on registering domains. For that reason, parts of the node ID are determined by something that cannot be arbitrarily chosen by an attacker: The IP address is such a property, as subnetworks are assigned by regional internet registries to operators. As with IP version 6 the number of addresses is much more generous than in previous versions of the protocol – each (local area) network operator gets at least a /64 network prefix – only the first 64bit of the IPv6 address are utilised. Thus I assume that each /64 IPv6 network is operated by the same entity. The hash of the /64 IPv6 network the node resides in is split and used to determine the most significant and least significant bits of the node ID. The hash of the node's domain name, included for Sybil protection but easier controllable by an attacker, is thus framed by bits hashed from the IPv6 subnet, so that the attacker can neither trivially control the large region the resulting node ID resides in nor the finer-grained position within such a segment. As the node ID is only derived from public information also available to each other node and appended with a vserver counter, their correctness can be verified. Unfortunately this approach does not inherently limit the number of nodes joined under a certain domain or IP address, as just changing one of those produces a different node ID. As a countermeasure nodes shall evict entries with duplicate IP subnetwork addresses or domain names from their local routing tables. I decided not to avoid these duplicates on a global scale, as such distrib-

23. ActivityPub communication over HTTPS relies on the widely-used domain certificate authorities, but these are not able to limit an attacker in its identities as only the control over a domain is needed for certificate issuance.

uted registration points like proposed in [28] are introducing additional attack vectors [6]. At least the corresponding instances to a node need to run under the same domain zone as used for generating the node's ID, this limits the variability of domains for ID generation.
Eclipse attacks are prevented by limiting the deliberate targeting of node IDs with the aforementioned ID derivation scheme. This prevents an attacker from deliberate positioning at the IDs of its victim node's neighbours. Additionally, all node IDs are validated before being inserted into routing tables. As my proposed system is just an addition to the push-federated communication of the instances, nodes do not exclusively learn about other nodes through their neighbours and can thus periodically re-sample key responsibilities through nodes learned from classical push federation. Basing the system on Chord [14] with its constrained routing table, Eclipse attacks also cannot use spoofed network distance measurements, as such optimisations are not used in Chord.

While the node ID derivation mechanism induces all these beneficial security properties, its design also poses some challenges:
A balance has to be struck between safety by limiting an attackers mobility and avoiding an uneven distribution of nodes, leading to clustering of nodes. Taking the hash of a node's full /64 subnet prefix is at the safe side of relating to a certain network operator, as /64 networks are the smallest networks assigned. But attackers having a larger subnet can generate a large number of IP addresses. Members of the RIPE NCC can obtain a /29 network [24], giving them 2^{35} options for node ID parts generated from the IP. As already a single bit change in the hash input vastly changes the produced hash and thus resulting in a vastly different node ID, attackers with large subnets have access to a large number of regions. A possible workaround for that is splitting the subnet into parts, hashing them separately and appending (parts of) the hashsums, causing equal prefixes in the input IP address to result in equal node ID prefixes. The resulting clustered ID distribution though is very likely to worsen the load balancing performance, especially with the currently sparsely assigned IPv6 address yielding many addresses with equal prefixes.
A side effect in the opposite direction was visible in Figure 8: The histogram of node ID distribution derived from actual current ActivityPub instances shows two spikes, indicating clusters of nodes. While the nodes in these clusters are probably legitimate non-malicious nodes in the eyes of their operators and users, they are placed into these clusters because of the Sybil protection properties of the node derivation: The first spike is caused by instances hosted by the managed-hosting provider masto.host. All instances by this operator are run from the exact same IPv6 address, only differing in the domain name and thus creating similar IDs with a common prefix. The second spike stems from CloudFlare who provide DDoS protection services by placing websites behind a reverse-proxy. All these reverse-proxies, serving as the entry point for all accesses from the internet, are run by the company and reside within the same IPv6 /64 subnet. This clustering behaviour cannot even be considered as a false positive, as all nodes within a cluster are under the control of the same operator, either because of being run by them or at least having manipulative power as an intermediary.
Whether my approach against Sybil IDs strikes the right balance has to turn out in practice or be shown in future extensive simulations. I deem it better to evaluate a lighter protection mechanism than prematurely introducing mechanisms like computational puzzles [29] that are known to be effective, but turn out to be hard to implement in heterogeneous systems and are an inherent huge waste of resources by design.

For countering storage and routing attacks, the measures already described for Eclipse and node squatting attacks play an important role. Additionally, the approach of replicating keys in equal distances on the Chord ring, already described in section 7, not only serves the purpose of redundancy against node leave, but also against attacker nodes serving bogus content or are routing lookups to the wrong destination. The aim is to create routing redundancy by providing d disjoint routes to a key with the placement of 2^{d-1} key replicas.
One issue with this approach is that the original paper by Harvesf and Blough [27] assumes the data behind keys to be self-validating, making just one single route to a key sufficient to receiving the correct data. This is not true for the data in this system. Given an authenticated message history synchronisation mechanism (e.g. based on Merkle trees), at least majority voting can be applied for checking the validity of returned data.
Putting together the possibility of cross-checking

24. "However, you can request up to a /29 without providing any additional justification." https://www.ripe.net/manage-ips-and-asns/ipv6/request-ipv6

returned posts from different nodes of the redundancy set and the inability of deliberately being responsible for a certain hashtag due to the node ID derivation mechanism, I consider the integrity of stored message histories to be verifiable.

The abuse of the DHT for DDoS amplification is aggravated by preventing route poisoning and other routing attacks by having secure & verifiable node ID assignment, authenticated messaging between nodes and instances, and secure routing tables with ID verification and routing redundancy.

9 Future Work

This paper explores the basic architectural decisions to make when designing a decentralised system for storing and relaying of posts in a federated system. Future work on improving the architecture itself may focus on mechanisms for consistently synchronising stored state between nodes of a redundancy set. This can improve the discoverability of manipulated post histories by allowing reliable cross-checking between nodes.
Regarding the load balancing mechanism, future work may evaluate the overall feasibility of k-choices load balancing by exploring its parameter range and examining its behaviour in a dynamic simulation.
When implementing this architecture for practical use, there are a number of open question remaining: A schema for implementing hashtag actors into ActivityPub has to be developed, exact protocol messages and error codes for communication between relay and storage instances needs to be specified, and the data serialisation format between nodes has to be defined. The integration between current instance server implementations and this new hashtag federation backend needs to be evaluated case by case.

10 Conclusion

While the decentralisation of social networks into federated systems can have many advantages, so far these networks had a mayor user experience drawback: Hashtags were not usable like in centralised social networks as different users on different server instances might have seen different sets of posts with that hashtag. This paper introduces an additional backend for federated social networks allowing to subscribe to hashtags and retrospectively query a consistent post history from every instance of the network. Hashtags and instances are placed on two Chord Distributed Hash Tables to create unambiguous responsible points for for handling posts of that hashtag's posts, one DHT instance for relaying and one for storage of posts. Instances interested in a hashtag's post can subscribe to the responsible relay or query the history from the responsible storage node.

For enabling the system to store at least 1 full year of post history without overloading some of its nodes, post data from 3 social networks was analysed to determine the need for balancing load between nodes. As a consequence, the k-choices algorithm was adopted for balancing hashtags between nodes, giving each node a choice between different sets of hashtags. This algorithm was shown to be effective in a simulation with static stored post data, but needs to be tuned further for practical usability.
Reliability of the system is improved by redundantly assigning the hashtag to multiple nodes. These nodes can also respond to flash crowd overload scenarios by splitting hashtags.

Common security threats for DHTs have been addressed in the system design to avoid manipulation of handled data or misuse of the network. A special focus was laid on preventing Sybil attacks by using a hierarchical node ID derivation function, limiting attackers in their deliberate choice of hashtags to handle.

Having outlined how to integrate this new backend into ActivityPub-based systems, this paper servers is a foundation for future specification implementation effort.

Acknowledgements

I want to thank the Language Technology Group from Universität Hamburg for providing me their collected dump of Twitter posts for analysis. Many thanks as well to Michael Vogel for providing data sets of hashtag usage on his two Friendica instances, and to Dennis Schubert for the respective data on the Geraspora Diaspora* instance.
Further acknowledgement goes to my mentors Martin Byrenheid and Clemens Deußer for their extensive feedback and support.
At last, this work would not have been possible without the work of many Free Software projects.

Glossary

AP	ActivityPub
AS	ActivityStreams 2.0

DHT	Distributed Hash Table
DoS	Denial of Service
JSON-LD	JavaScript Object Notation for Linked Data
P2P	peer-to-peer
URI	Uniform Resource Identifier
RIPE NCC	Réseaux IP Européens Network Coordination Centre – the Regional Internet Registry for Europe
VS	virtual server

REFERENCES

[1] Christopher Lemmer Webber et al. *ActivityPub*. W3C Recommendation. W3C Social Web Working Group, 23rd Jan. 2018. URL: https://www.w3.org/TR/2018/REC-activitypub-20180123/ (visited on 02/03/2019).

[2] James M Snell. *Activity Vocabulary*. W3C Recommendation. W3C Social Web Working Group. URL: https://www.w3.org/TR/activitystreams-vocabulary/ (visited on 05/11/2018).

[3] James M Snell and Evan Prodromou. *Activity Streams 2.0*. W3C Recommendation. W3C Social Web Working Group. URL: https://www.w3.org/TR/activitystreams-core/ (visited on 18/07/2019).

[4] Mark Cavage and Manu Sporny. *Signing HTTP Messages*. Internet-Draft draft-cavage-http-signatures-10. IETF Secretariat, May 2018. URL: http://www.ietf.org/internet-drafts/draft-cavage-http-signatures-10.txt.

[5] David Longley, Manu Sporny and Christopher Allen. *Linked Data Signatures 1.0*. Community Draft. W3C Digital Verification Community Group, 4th Nov. 2018. URL: https://w3c-dvcg.github.io/ld-signatures/ (visited on 04/11/2018).

[6] Guido Urdaneta, Guillaume Pierre and Maarten Van Steen. "A Survey of DHT Security Techniques". In: *ACM Computing Surveys* 43.2 (1st Jan. 2011), pp. 1–49. ISSN: 03600300. DOI: 10.1145/1883612.1883615. URL: http://portal.acm.org/citation.cfm?doid=1883612.1883615 (visited on 18/02/2019).

[7] Twitter Inc. *#numbers: Twitter Statistics from 2009*. 14th Mar. 2011. URL: https://blog.twitter.com/official/en_us/a/2011/numbers.html (visited on 24/11/2018).

[8] Miguel Freitas. "Twister - a P2P Microblogging Platform". In: (26th Dec. 2013). arXiv: 1312.7152 [cs]. URL: http://arxiv.org/abs/1312.7152 (visited on 04/11/2018).

[9] Thomas Paul et al. "Lilliput: A Storage Service for Lightweight Peer-to-Peer Online Social Networks". In: IEEE (26th International Conference on Computer Communications and Networks (ICCCN)). 2017.

[10] Alberto Mozo and Joaquín Salvachúa. "Scalable Tag Search in Social Network Applications". In: *Computer Communications* 31.3 (Feb. 2008), pp. 423–436. ISSN: 01403664. DOI: 10.1016/j.comcom.2007.08.035. URL: http://linkinghub.elsevier.com/retrieve/pii/S0140366407003222 (visited on 10/12/2018).

[11] T. Perfitt and B. Englert. "Megaphone: Fault Tolerant, Scalable, and Trustworthy P2P Microblogging". In: *2010 Fifth International Conference on Internet and Web Applications and Services*. 2010 Fifth International Conference on Internet and Web Applications and Services. May 2010, pp. 469–477. DOI: 10.1109/ICIW.2010.77.

[12] Antony Rowstron et al. "Scribe: The Design of a Large-Scale Event Notification Infrastructure". In: *Networked Group Communication, Third International COST264 Workshop (NGC'2001)*. Ed. by Jon Crowcroft and Markus Hofmann. Vol. 2233. Lecture Notes in Computer Science. UCL, London, UK., Nov. 2001, pp. 30–43.

[13] Jason Robinson. *Diaspora* Social Relay Design Concept*. URL: https://raw.githubusercontent.com/jaywink/social-relay/master/docs/relays.md (visited on 11/11/2018).

[14] I. Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications". In: *IEEE/ACM Transactions on Networking* 11.1 (Feb. 2003), pp. 17–32. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.808407.

[15] Marc Stevens et al. "The First Collision for Full SHA-1". In: *Advances in Cryptology CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. Cham: Springer International Publishing, 2017, pp. 570–596. ISBN: 978-3-319-63687-0 978-3-319-63688-7. DOI: 10.1007/978-3-319-63688-7_19. URL: http://link.springer.com/10.1007/978-3-319-63688-7_19 (visited on 21/03/2019).

[16] *NodeInfo Specification*. URL: http://nodeinfo.diaspora.software/ (visited on 22/03/2019).

[17] Y. Zhu and Y. Hu. "Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 16.4 (Apr. 2005), pp. 349–361. ISSN: 1045-9219. DOI: 10.1109/TPDS.2005.46. URL: http://ieeexplore.ieee.org/document/1401878/ (visited on 22/03/2019).

[18] B. Godfrey et al. "Load Balancing in Dynamic Structured P2P Systems". In: *IEEE INFOCOM 2004*. IEEE INFOCOM 2004. Vol. 4. Hong Kong, China: IEEE, 2004, pp. 2253–2262. ISBN: 978-0-7803-8355-5. DOI: 10.1109/INFCOM.2004.1354648. URL: http://ieeexplore.ieee.org/document/1354648/ (visited on 22/03/2019).

[19] Ananth Rao et al. "Load Balancing in Structured P2P Systems". In: *Peer-to-Peer Systems II*. Ed. by M. Frans Kaashoek and Ion Stoica. Red. by Gerhard Goos, Juris Hartmanis and Jan van Leeuwen. Vol. 2735. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 68–79. ISBN: 978-3-540-40724-9 978-3-540-45172-3. DOI: 10.1007/978-3-540-45172-3_6. URL: http://link.springer.com/10.1007/978-3-540-45172-3_6 (visited on 14/03/2019).

[20] David R Karger and Matthias Ruhl. *Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems*.

[21] John W. Byers, Jeffrey Considine and Michael Mitzenmacher. "Simple Load Balancing for Distributed Hash Tables." In: *IPTPS*. 2003, pp. 80–87.

[22] Ole Tange. *GNU Parallel 2018*. Ole Tange, Apr. 2018. DOI: 10.5281/zenodo.1146014. URL: https://doi.org/10.5281/zenodo.1146014.

[23] Dennis Schubert. *Diaspora* Tag Usage Data for Pod.Geraspora.De*. URL: https://0b101010.codes/research-things/open-data/tree/master/diaspora/tag-usage.

[24] J. Ledlie and M. Seltzer. "Distributed, Secure Load Balancing with Skew, Heterogeneity, and Churn". In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Vol. 2. Miami, FL, USA: IEEE, 2005, pp. 1419–1430. ISBN: 978-0-7803-8968-7. DOI: 10.1109/INFCOM.2005.1498366. URL: http://nrs.harvard.edu/urn-3:HUL.InstRepos:2962660 (visited on 04/04/2019).

[25] Frank Dabek et al. "Wide-Area Cooperative Storage with CFS". In: *SOSP* (2001), p. 14.

[26] J. Ledlie and M. Seltzer. *Harvard Technical Report TR-31-04: Distributed, Secure Load Balancing with Skew, Heterogeneity, and Churn*. Dec. 2004. URL: https://people.csail.mit.edu/ledlie/lb/kchoices05-tr.pdf (visited on 10/07/2019).

[27] C. Harvesf and D. M. Blough. "The Effect of Replica Placement on Routing Robustness in Distributed Hash Tables". In: *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*. Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06). Sept. 2006, pp. 57–6. DOI: 10.1109/P2P.2006.44.

[28] Jochen Dinger and Hannes Hartenstein. "Defending the Sybil Attack in P2P Networks: Taxonomy, Challenges and a Proposal for Self-Registration". In: *1st International Conference on Availability, Reliability and Security, ARES 2006; Vienna; Austria; 20 April 2006 through 22 April 2006*. IEEE Computer Society, Los Almitos (CA), 2006, pp. 756–763. ISBN: 0-7695-2567-9. DOI: 10.1109/ARES.2006.45.

[29] N. Borisov. "Computational Puzzles as Sybil Defenses". In: *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*. Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06). Cambridge, UK: IEEE, 2006, pp. 171–176. ISBN: 978-0-7695-2679-9. DOI: 10.1109/P2P.2006.10. URL: http://ieeexplore.ieee.org/document/1698607/ (visited on 06/06/2019).