Gediminas Lelešius
g@lelesius.eu

Improving Resilience of ActivityPub Services

Computer Science Tripos – Part II

St Catharine's College

May 13, 2022

Public version

Notice

Proforma, appendices and the proposal were modified before posting online: unnecessary personal data was removed and candidate's name and other contact details were included.

Declaration of Originality

I, Gediminas Lelešius of St Catharine's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed *Gediminas Lelešius*

Date *May 13, 2022*

Acknowledgements

I would like to thank my supervisor Dr Anil Madhavapeddy for introducing me to a world of federated systems, suggesting this interesting project idea, and helping throughout the course of the project.

I would also want to thank my family and my friend AJ for supporting me throughout the Tripos.

Proforma

Candidate:	**Gediminas Lelešius**
Email:	g@lelesius.eu
Project Title:	**Improving Resilience of ActivityPub Services**
Examination:	Computer Science Tripos – Part II
Year:	2022
Word Count:	**11998**[1]
Code Line Count:	**5097**[2]
Project Originator:	Dr Anil Madhavapeddy
Project Supervisor:	Dr Anil Madhavapeddy

Original Aims of the Project

The original goal was to improve the resilience of the distributed social network called ActivityPub by caching the content on multiple instances and serving them in case the origin instance goes down. I aimed to use Public-key cryptography to ensure data integrity, build a network of public key servers and verifiers and use its consensus instead of relying on individual servers to provide trustworthy data.

The core deliverable is a key server gathering and serving public keys, a verifier checking the entries of that server, and a modified Mastodon server rescuing failed ActivityPub requests using an external key server.

Work Completed

The original aims have been achieved, core deliverables – a key server and a verifier – were implemented and presented in the dissertation. Then, an ActivityPub service called Mastodon was modified to use a key server and other instances to rescue failed content queries. Finally, functional testing was conducted and the project outcome was evaluated. The system was deployed to work with the real ActivityPub network.

I also researched multiple decentralised ecosystems, learned and programmed in Ruby, designed, debugged, tested and fixed distributed systems, and found multiple bugs in ActivityPub implementations.

Special Difficulties

None.

[1]The word count was computed using TEXcount `https://app.uio.no/ifi/texcount/` with arguments `-1 -sum includes.tex 1-intro.tex 2-prep.tex 3-impl.tex 4-eval.tex 5-concl.tex`.

[2]Code lines counted using `pygount` (majority of the project, excludes comments and empty lines) and `git diff` (to count Mastodon modifications). Scripts are provided together with source code.

Contents

Chapter 1

Introduction

My work concerns distributed social networks (DSNs): networks composed of many independent servers (instances) operated by different entities but connected in a way that users from different servers can communicate seamlessly. In such networks, Public-key cryptography is (often) used to check the identity of the content author and to prevent malicious servers from injecting fake and invalid data into the network.

In this dissertation, I describe a successful attempt to improve the reliability of a DSN while preserving its security guarantees. To achieve this, I created a public-key caching network and modified an existing DSN implementation to share the data it has about other servers. That allowed me to rescue federated requests for public keys or other content when they fail due to the target instance being down by querying key-cache or DSN servers respectively. Public-key cryptography and quorums were used to check that the cached response is authentic and was not tampered with.

I designed the system such that it is incrementally deployable, requires little resources to run, can handle millions of accounts, and obeys other requirements (§2.3). I fulfilled all success criteria (§4.1) and evaluated the performance of my system (§4.2). In addition, I considered some possible attacks and showed that security guarantees of the network are not affected (§4.1.3). While analysing crawled results, I found bugs in implementations of the protocol that are deployed in production.

1.1 Motivation

Nowadays, most of our online communications, including instant messaging and social networks, are concentrated in the hands of a few companies. This centralised approach raises many issues related to privacy and censorship: how can we trust these companies to handle our sensitive data with care and not to sell it, how do we know that governments are not forcing companies to censor some content, etc. While countries are attempting to protect citizens' data by adopting privacy and data protection laws, it is hard to verify that corporations comply. Also, centralised systems are prone to blocking by governments[1], may be acquired and controlled by eccentric people or organisations[2], and their outages cause difficulties for billions of people[3].

[1]In 2022, after invading Ukraine, Russian Federation easily blocked Facebook and Twitter [1].

[2]In 2022, business magnate Mr Elon Musk bought Twitter and plans to become its chief executive [2]. Because of that, many his critics decided to move to other social networks including Mastodon.

[3]In 2021, Facebook was unavailable for several hours [3].

Distributed systems suggest an alternative (for detailed comparison see §2.1.4): instead of having a large system serving billions of users, let anyone run their own server. To preserve network externalities, define a federation protocol describing how servers should communicate together and share content. This enables a user to choose an instance based on whether they trust the owner, instead of creating monopolies due to network effects. My project considers one particular widely used protocol called ActivityPub [4] and one specific implementation named Mastodon [5].

A distributed nature gives other benefits for free. First, it provides interoperability between different services: there are many social networks written on ActivityPub, for example, Mastodon (similar to Twitter) and Pixelfed (similar to Instagram). Second, the effect of server failures is limited. Although companies running large centralised social networks put enormous effort into reducing downtime, sometimes they occur and affect all network users. On the other hand, in a distributed network, only users of the failed instance are affected. And the impact on them can be further reduced by providing (read-only) access via other instances or decentralised user accounts[4].

Sadly, the latter is not strictly true with ActivityPub. First, a user from a failed server cannot be found by searching other instances unless that server has the user in its cache. Second, if a server goes down and some other server tries to share a post created on the unreachable server, receivers reject it because they cannot find the original post This is unfortunate because a copy of that post exists in the network. See §2.2 for more details.

I implemented and present a solution that tackles both problems. Effectively, I reduce the impact of failures of ActivityPub instances. It might allow people to run their own instances with significantly lower reliability requirements because outages (even intentional server shutdowns to save power) will only affect those users but not the content they have shared. Therefore, the network should become more decentralised for the benefit of all users.

My personal hope is that one day we will have most of our data stored locally on our devices and they will act as servers, sharing data with our friends in a transparent and controllable way (see Conclusions and §5.3).

1.2 Related work

The project takes high-level inspiration from TLS Certificate Transparency (CT) [7], however, the purposes are different, hence so are the implementations. In particular, Certificate Transparency relies on Certificate Authorities to report newly issued certificates to the public log servers (my system gathers keys itself). Monitors are constantly looking for new entries, checking them and reporting unexpected or suspicious certificates to domain owners. To encourage the adoption of CT, the most popular browser Chrome and some smaller ones started marking certificates that do not use CT as untrusted [8]. In contrast, my system will be incrementally-deployable and beneficial to instances that use it: it will improve content availability.

Next, there are suggestions on how to design social networks that are reliable, enforce privacy and provide other guarantees. A particularly interesting example is the Footlights social network [9] that claims to provide a resilient platform for writing social apps that can only access users' data with explicit consent and in an observable and traceable manner. However, it describes a completely different architecture of a network that is incompatible with existing

[4]This is promising for applications requiring high availability, for instance, instant messaging. Although there are requests to support distributed user accounts in a messaging protocol Matrix [6], I am not aware of any successful implementation.

ones, hence I do not expect wide adoption of such projects in a near future. Again, my project suggests an extension to the existing ActivityPub network instead of attempting to build a new ecosystem.

While I was running my Mastodon instance during the project, I found a FediList [10] crawler in the logs. However, neither its code nor a technical description is published at the time of writing.

1.3 Report outline

Chapter 2 introduces ActivityPub protocol and Mastodon extensions, lists tools used for development and project management. Chapter 3 describes the requirements for my system and how the implementation fulfils them. Chapter 4 reports how the project was evaluated and presents performance metrics. Chapter 5 summarises what has been achieved and suggests directions for future work.

Chapter 2

Preparation

This chapter summarises the ideas needed to appreciate the work undertaken and realise the advantages of distributed social networking (§2.1 - §2.2), refines main requirements (§2.3) and highlights tools used to successfully achieve them (§2.4).

2.1 Background knowledge

To understand what the project is about, we should have some knowledge about distributed social networks. This section describes my starting point (§2.1.1), gives a brief overview of ActivityPub protocol (§2.1.2), Mastodon extensions (§2.1.3), and compares centralised and decentralised approaches (§2.1.4).

2.1.1 Starting point

The starting point is described in more detail in the project proposal (appendix C). In summary, I did not have any prior experience with ActivityPub protocol nor I have used any decentralised platforms before. I have not programmed in Ruby before, in which Mastodon is implemented.

2.1.2 ActivityPub

ActivityPub (fig. 2.1) is a decentralised social network protocol that defines both: client-to-server (between user's device and the server) and server-to-server communication (federation). The client-to-server part of the specification is not widely implemented because it would require rewriting the client and the server (many social networks were created before ActivityPub was released) or starting the implementation from scratch.

The main purpose of defining a client-to-server protocol is to allow users to choose what client to use. This is beneficial for instant messaging applications (for example, Matrix [11] defines a client-to-server protocol and there are alternative implementations of it), however, less such for social networks because different purpose platforms require not only different clients but also different servers (e.g. Instagram servers should be highly optimised to serve images while Twitter's to server short text messages).

Fortunately, the project is only concerned with the server-to-server protocol. Because of that, our system can be considered as just another type of ActivityPub service and should not be

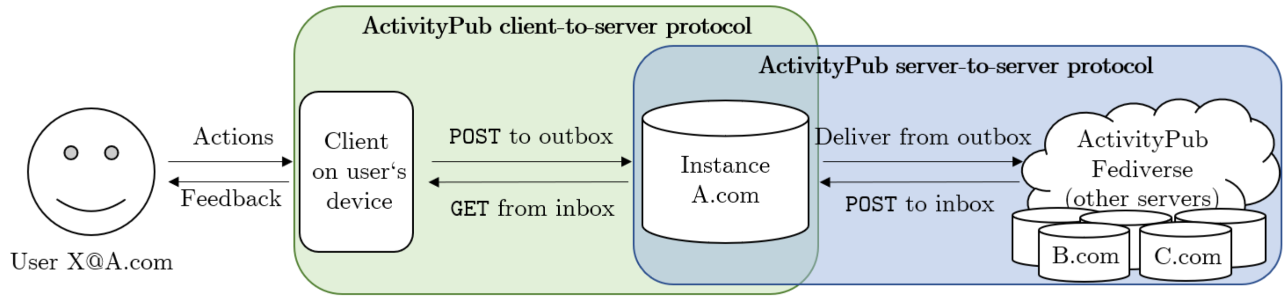


Figure 2.1: *ActivityPub protocol. HTTPS* ***`GET`*** *and* ***`POST`*** *requests are used.*
A user interacts with a Client (application). The client uses the client-to-server protocol to communicate with the user's home instance. The instance uses the server-to-server protocol to communicate with other federated instances.
Note that instances using different client-to-server protocols can still join the ActivityPub federation.

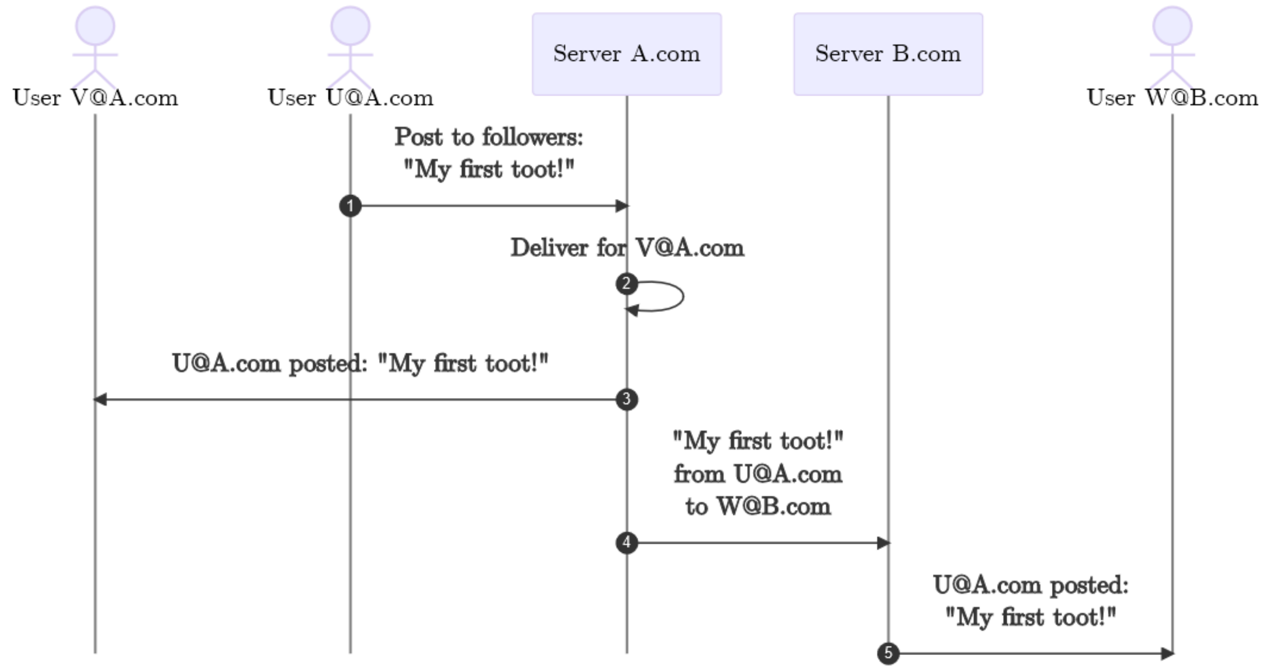

User V@A.com
User U@A.com
Server A.com
Server B.com
User W@B.com
1 Post to followers: "My first toot!"
2 Deliver for V@A.com
3 U@A.com posted: "My first toot!"
4 "My first toot!" from U@A.com to W@B.com
5 U@A.com posted: "My first toot!"

Figure 2.2: *User creating a post in ActivityPub network.*

easily distinguishable from others. That places constraints on the implementation that are further discussed in §2.3.

ActivityPub can be viewed as an API specification: users are represented as Actors, that have their Inboxes and Outboxes. The inbox is an API endpoint (URL) to which other servers can make requests to deliver content. The content (including actors) is represented by objects and serialised into JSON format [12] for exchange. Similarly, the outbox is another API endpoint that can be queried to get events related to the user.

Events are represented as Activities, serialised into JSON as well. There are activities for every action that might happen on the ActivityPub network, for example, Create, Update, Delete, etc. All activities include the actor and type and might include the object that was created or updated. All objects (including activities and actors) have their own URIs that can be queried to get the object.

A user generates events when interacting with its home server (the server to which the user's

```
{   "id": "https://example.com/users/A",
    "type": "Person", "preferredUsername": "A",
    "following": "https://example.com/users/A/following",
    "followers": "https://example.com/users/A/followers",
    "inbox": "https://example.com/users/A/inbox",
    "outbox": "https://example.com/users/A/outbox",
    "publicKey": {
        "id": "https://example.com/users/A#main-key",
        "publicKeyPem": "-----BEGIN PUBLIC KEY-----\nMIIBIjANB[...]" }
}
{   "id": "https://example.com/users/A/followers",
    "type": "OrderedCollection",
    "totalItems": 2,
    "first": "https://example.com/users/A/followers?page=1"
}
{   "id": "https://example.com/users/A/followers?page=1",
    "type": "OrderedCollectionPage",
    "partOf": "https://example.com/users/A/followers",
    "orderedItems": [
        "https://example.com/users/B",
        "https://domain.com/XYZ"
    ] }
```

Figure 2.3: *An example of an ActivityPub actor, its followers collection and first page of it. Each object can be fetched by querying URI from its* `id` *field. Some fields omitted for brevity.*

URI refers, see fig. 2.2). Activities representing those events are created and delivered to the outboxes of the audience. The audience can be a set of users (e.g. in the case of a private message), however, in most cases, it is a followers list of the user.

A list of Followers and a list of followed users[1] are properties of the user. They can be either embedded directly into the user's object or represented as a Collection object and referred to by URI. Larger collections can be split into smaller pages, each having its own URI. An example is given in fig. 2.3.

This suggests three different ways to gather data from the network: crawling using server-to-server API, following users (becoming a follower), or inserting a proxy or a relay[2] between servers. The advantages and caveats of each approach are discussed in §3.2.

2.1.3 Mastodon

Unfortunately, ActivityPub does not define how servers are authenticated. The authentication is crucial for my system because I want to cache the data on multiple untrusted servers and later verify its integrity. Therefore, I chose a particular implementation of ActivityPub – namely Mastodon – as a reference. I selected Mastodon because it is one of the most widely used ActivityPub services and is compatible with many others. That will achieve the goal of maximum compatibility.

[1]In ActivityPub, a collection of followed users is called `following`.

[2]The concept of an ActivityPub relay exists. They are used to reduce the load on servers and propagate data: they just forward all requests they receive to all subscribers.

Mastodon is a decentralised microblogging platform. It supports ActivityPub federation (communication between servers), however, it uses a custom API to communicate between the website running on the user's browser (client) and the server.

Mastodon assumes that the underlying network has an authentication mechanism that ensures that the request is answered by a server owning the requested URI. On the Internet, this is guaranteed by using HTTPS scheme URIs. Then, TLS [13] verifies that a responding server actually has the private key of the domain queried. Therefore, the requester knows that the response was not tampered with, however, it cannot prove integrity to anyone else.

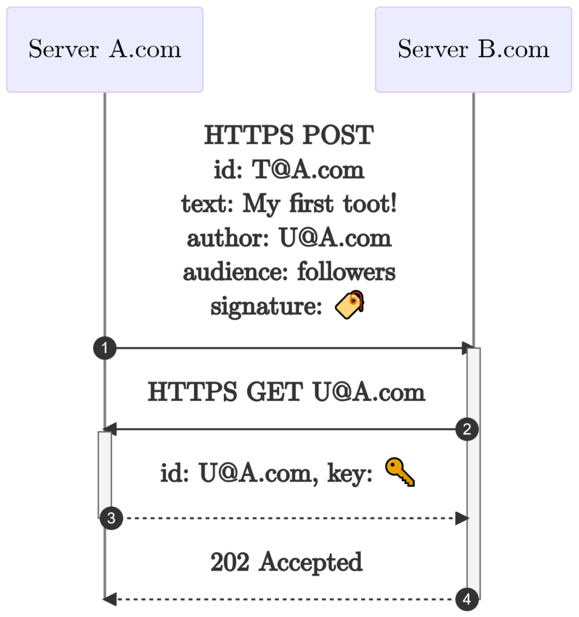


Figure 2.4: *The communication between servers when delivering a post in more detail. Note, that actor's public key may be cached for performance reasons.*

For deliveries, a server that created the activity is making the request (fig. 2.4). Because of the way TLS works, a recipient cannot check who is making a request. Therefore, Mastodon appends a signature created using the author's private key. The Mastodon then gets the public key by querying the author's object by URI and verifies the signature. Of course, a public key is cached to reduce the latency, save bandwidth and lower the load on the delivering server. They cannot be cached indefinitely because public keys can be changed (rotated).

There are two ways how a signature can be included: either by using a HTTP Signature header or adding a signature field directly to a JSON object being sent. The HTTP signature protects not only the data being sent but also request headers (including time and the target URL). This means that the receiver cannot forward the activity to others unless others expect a mismatch between HTTP header fields. On the other hand, a JSON signature facilitates further sharing and allows further receivers to check the integrity as well. This is employed in ActivityPub relays: receivers do not have to query the origin server to check originality.

In addition, Mastodon also requires WebFinger [14] protocol to resolve usernames[3] to ActivityPub object URIs. It should be straightforward to extend the system to support them.

2.1.4 Centralised and decentralised systems

Below, I give a brief comparison of centralised (e.g. Twitter[4]) and decentralised platforms: ActivityPub and Matrix (decentralised instant messaging protocol with end-to-end encryption [11]). This context is important to fully understand the motivation of the project and will allow me to formulate the requirements following the philosophy behind ActivityPub.

[3]Mastodon needs WebFinger because it was used to resolve federated usernames before ActivityPub support. Now, WebFinger names remain a human-friendly identifier but older parts of Mastodon code still rely on WebFinger.

[4]Interestingly, Twitter seems to be exploring ways to join the federation [15].

Privacy and Censorship

In centralised systems, the company behind the service has all user data, including sensitive information such as website usage patterns, content viewed, etc. Users might not be aware of what data is gathered. Moreover, owners can create and enforce rules affecting all users of the network. Similarly, governments can exercise censorship laws more easily when there is a particular responsible organisation.

Contrarily, in a decentralised system, the server only directly serves its users, while interactions with other federated servers are proxied via the users' home server. Distinguishing[5] which users requested what resources and tracking them become difficult. Moderate-size servers are optimal for the private and uncensored federation: if there are too few users, distinguishing between them becomes easy; likewise, having many users on a single instance means that information is centralised.

Typically, the actor must be revealed in some cases (for example, reacting to or sharing a post). Ideally, that should be a result of an intentional user action. Importantly, this data is shared only with instances that have users related to the acting user (e.g. followers of a user who posted a message).

The server can only censor its users, but they can move to an alternative instance while preserving the access to the federation. Furthermore, the government cannot take a network down by prosecuting a specific entity. If an instance is banned in some territory, its content might still be reachable using the resilience mechanism described in this dissertation.

Consistency model

A decentralised system must be partition-tolerant and available because some instances might be malicious and simulate failures. From the CAP conjecture [17], we deduce that it cannot be consistent (linearizable).

In Mastodon, this is evident by each server having a view of the subset of the network in which its users are interested. It is not guaranteed that views of different servers are eventually consistent.
Matrix selected a different trade-off: all servers are synchronising their state on every message. If multiple servers arrive in incompatible states, the State resolution protocol may undo changes to restore the eventually-consistent model.

A centralised system does not have such restrictions. For instance, a more reliable network between servers could mean that full partition tolerance is not required. Notice that a single (distributed) instance of the federation in isolation behaves as centralised: between its local users, it can make similar trade-offs.

Security model

Centralised systems store data in trusted databases and authenticate users before servicing requests. Assuming the code is correct, this guarantees that the network is in a valid state.

In contrast, federated systems assign all objects unique identifiers and divide them into regions owned by different servers. This is done by including the home server's identifier (e.g. domain) into objects' (e.g. users') identifiers. To trust a federated object, one verifies that it was created

[5]This resembles the Tor network [16]: privacy is created by servers executing actions on behalf of different users and not revealing who requested them.

by a server that owns the object's ID. Therefore, servers cannot alter objects belonging to others, however, they can reference them. Figure 2.5 suggests two attacks and show how they are prevented.

End-to-end encrypted objects (Matrix messages) require a two-step procedure: first, receiving server authenticates sending server; second, receiving user verifies the sender. In Matrix, the second verification fails only if one of the servers was malicious and did not authenticate the sender/receiver properly. Note that a malicious server could change actors' key pair, therefore, users should handle friend key change warnings with care.

Resilience to failures

A centralised system is a single point of failure and there are many examples of how the world's largest systems maintained by huge companies have failed. In such cases, users can either wait until the service is restored or switch to another provider and lose their data.

Decentralisation automatically limits the scope of the failure, although, the exact mechanics depend on the protocol. In ActivityPub, users of the failed instance retain a read-only view of the network via other instances (without login and other customisations). In Matrix, users of it lose access completely. Theoretically, a solution for this issue is to distribute a user account among multiple instances (multi-homing) but that might weaken the consistency model.

The work described in this dissertation is concerned with making data available on other instances and allowing to share it with others even when the origin instance (i.e. the server that created that data) is unreachable.

2.2 Main problem

As described in the §1.1, a failure of one instance restricts available actions and reachable content on other instances. In particular, I focus on two main issues (assuming relevant actors/posts are not cached, examples shown in fig. 2.6):

(a) actors from the failed instance cannot be found by search in other instances;

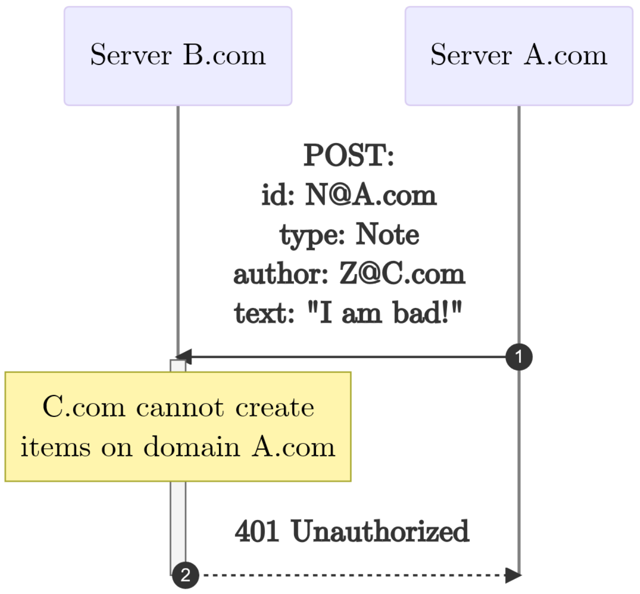
(b) posts of such actor cannot be shared from an instance that has a cached copy to other instances.

2.3 Requirements analysis

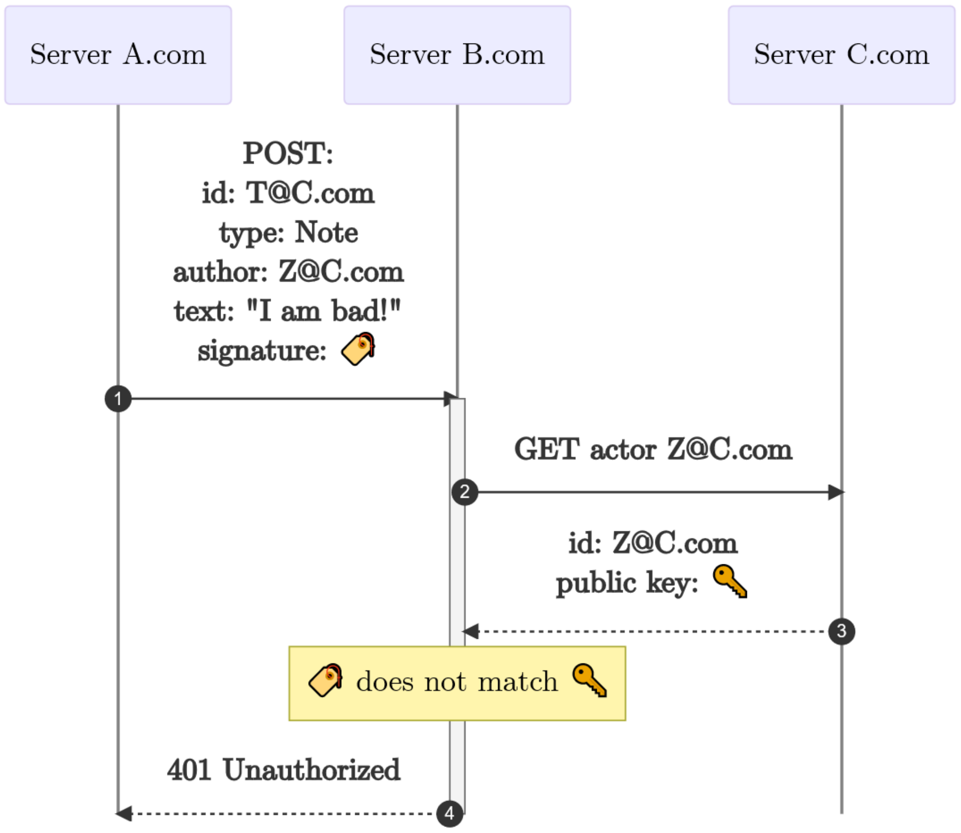
After considering what I learned about distributed systems and the philosophy behind them, I propose the following requirements:

IncrDeploy: In decentralised systems, it is impossible to deploy on all instances at the same time. Moreover, some people might be running old versions on purpose. Hence, my system has to be incrementally deployable.

LowResource: In addition, the system should not be resource-intensive and must not require money or other resources to join (except resources needed to host a server). This eliminates both, a Proof-of-Work (needs many resources) and Proof-of-Stake (needs tokens of value) based blockchain approaches.

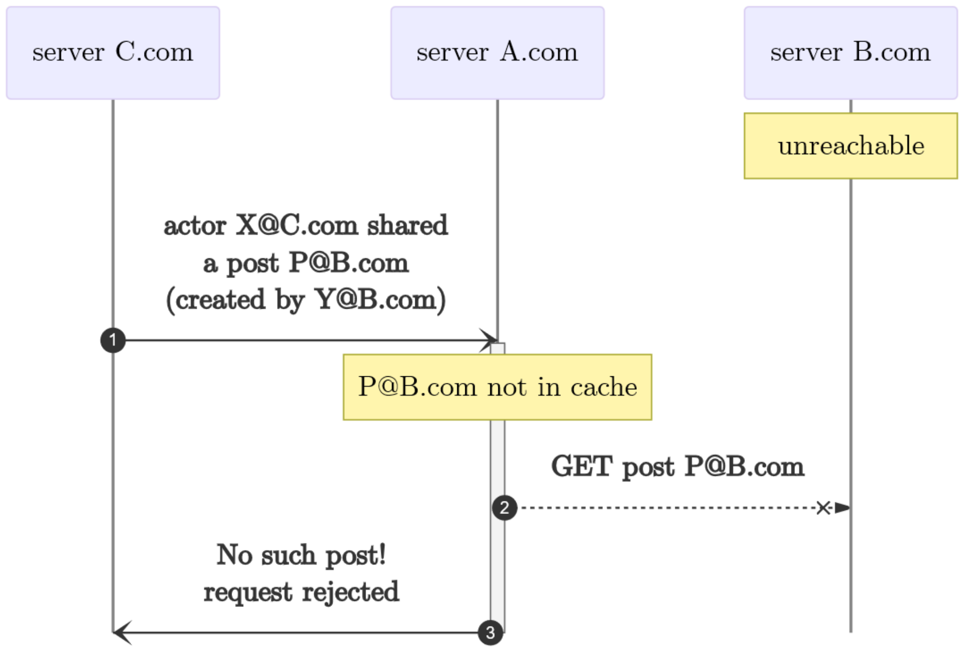


(a) Malicious server C.com tries to create a note with A.com id. The action fails because author and created object must be on the same domain.

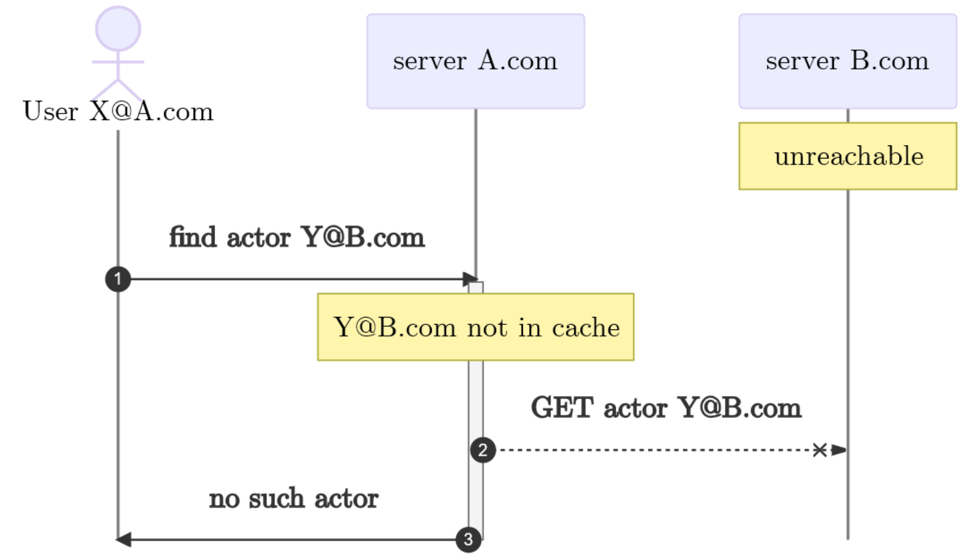


(b) Malicious server C.com tries to create a note with A.com id and author. The action fails because C.com does not have a private key of X@A.com. Note that the fetch from A.com is done using HTTPS, so C cannot tamper with it.

Figure 2.5: *How two attacks on the ActivityPub protocol are prevented.*



(a) Failed post share. Note that for A this situation is indistinguishable from post `P@B.com` not existing and being maliciously spoofed by C.



(b) Failed user lookup. Note that A cannot determine whether actor `Y@B.com` exists or X entered invalid URI/username.

Figure 2.6: *Two cases when requests fail but there might be enough information in the network to rescue them. The project focuses on how to use the information present on other servers (e.g. C in fig. 2.6a) to answer requests when their target (B) is unreachable.*

Byzantine: The system should assume the Byzantine instance model and must not inflict damage on other servers if some instances are not following the protocol. It should remain functional even if some servers behave maliciously.

Fair: The system should be as fair as possible for users from instances that have many and few (or even one) users. This means that it should not prefer caching data or rescuing requests of smaller or larger instances where practical.

Compatible: The data gathering techniques must follow ActivityPub protocol and Mastodon implementation where ActivityPub is not enough. This includes rate-limiting requests to the same domain. In addition, the system should work with all ActivityPub implementations that use the same authentication model as Mastodon.

NoInterfere: The system must not have a single point of failure, must not increase latency or otherwise slow down[6] the operation of servers unless there are failures in the ActivityPub network. The modified Mastodon instances (with my improvements enabled) must properly function when other federated servers are not faulty, even if servers forming my proposed network are unreachable or unresponsive.

Security: The system must not require users to trust a single server (for example, a centralised lookup). The trust must be distributed between multiple participants, ideally, the ActivityPub instance owner should be able to select which nodes it trusts. The modified Mastodon must preserve the security guarantees.

RealWorld: The system should be able to cope with a real-world load[7]. It should not be difficult to distribute the system and tackle a load comparable to today's centralised social networks[8].

These will be referenced in the following sections.

2.4 Software engineering

2.4.1 Programming language, libraries, and tools

The majority of the code was written in Python 3.10, the `pyenv` virtual environment was used to avoid version conflicts with packages installed on the system. Python was selected because it makes the application easy to deploy[9], `aiohttp` package provides a high-performance web server implementation, and Python natively supports asynchronous programming: important as a significant part of the system is expected to be network bound. `NumPy` and `Matplotlib` libraries were used to analyse the data and plot graphs. I also tried using the `Neo4j` graph database to explore the data but it turned out not to be that useful because of the size of the network.

[6]With one exception: the system is allowed to make rate-limited requests to the servers. Technically, this might still slow down the system, however, the limit should be low enough that the effect negligible for any reasonable server.

[7]Currently, ActivityPub has a few or a dozen million users.

[8]Facebook and YouTube declare that they have about 2–3 billion active monthly users.

[9]At the time of writing, Python 3.10 was not available on cloud machines used for testing, therefore, my project is compatible with Python 3.8+.

I used Visual Studio Code, PyCharm, and RubyMine to code and debug. LaTeX was used to prepare the dissertation and other documents. Diagrams were created using Mermaid and PowerPoint.

For testing, I used a few cloud machines and also had a self-hosted Raspberry Pi if the cloud became too expensive. I also used Vagrant to run a virtual Linux machine on a Windows computer and test the code while developing.

2.4.2 Project management

The core of the project was split into three main deliverables: a key lookup server (together with a mechanism to gather the keys), a key verifier service, and a modification of Mastodon. I used the evolutionary prototyping technique [18] to develop and refine the systems because a prototype of each part might reveal useful information and suggest an improvement for it or modifications for other parts. The prototyping method can quickly adapt to such circumstances. Also, I estimated that each part will take about two weeks to implement, thus, I tracked the progress based on what prototypes were finished. In this stage, I was preparing a report for my supervisor every two weeks, describing the work undertaken, difficulties encountered and the general state of the project. Those reports later helped me to write this dissertation.

After the core was implemented and finalised, I did testing, deployment, evaluation and extensions in an Agile-like model. I maintained a list of parts to test, bugs to fix and evaluations to do, prioritised them and worked on the most important ones. While waiting for data to be gathered, I was working on other parts.

During the second half of the Lent term, I was busy with other coursework and the project lagged behind schedule. I updated the timeline and dedicated more time to the project during Easter vacation to ensure that the project is finished on time.

2.4.3 Version control and backup

I was developing the project on my personal machine and version-controlling the source code as well as the dissertation using a Git repository hosted on GitHub. I pushed the code to the remote Git repository at least once a day, when it was in a correctly working state. That way, I was sure that I'll have all versions of my software and be able to revert changes in case something went wrong. In addition, I used continuous synchronisation to Mega cloud to avoid any data losses of uncommitted data.

2.4.4 Licences

I modified the Mastodon code which is released on GNU General Public Licence v3.0 [19]. I hosted the forked repository and changes publicly on GitHub because submitting my modified copy for the examination might be considered a distribution of source code. Under the GPL3 licence, a code distribution is only permitted if the code is made public.

All other code that was written by me is licensed under MIT licence [20] to allow use without limitation, including modifying it for further research or distributing for production use.

Chapter 3

Implementation

In this chapter, I give a high level picture of the system (§3.1), and describe the design and implementation of each part in more detail: the key lookup server (§3.2 and §3.3), the verifier (§3.4), and modifications needed for an ActivityPub service to use this network as well as how they were applied for Mastodon (§3.5). Finally, I include the repository overview (§3.6).

3.1 Overview

To fulfil the requirements (§2.3), I created an ActivityPub actor caching network consisting of key lookup servers and key verifiers. The key lookup server finds actors in the network, stores them in the database and serves them back should some instance fail. Verifiers watch key server entries, check them by querying origin servers, and, assuming entries are correct, provide signatures to the key server so it can later prove validity to requesters. I argue that this network is trustworthy because it is decentralised, and anyone can easily host their own verifier instance. Therefore, users can choose a trusted subset of verifiers and check that a quorum of them agrees on the network state. A scheme of the system is shown in fig. 3.1.

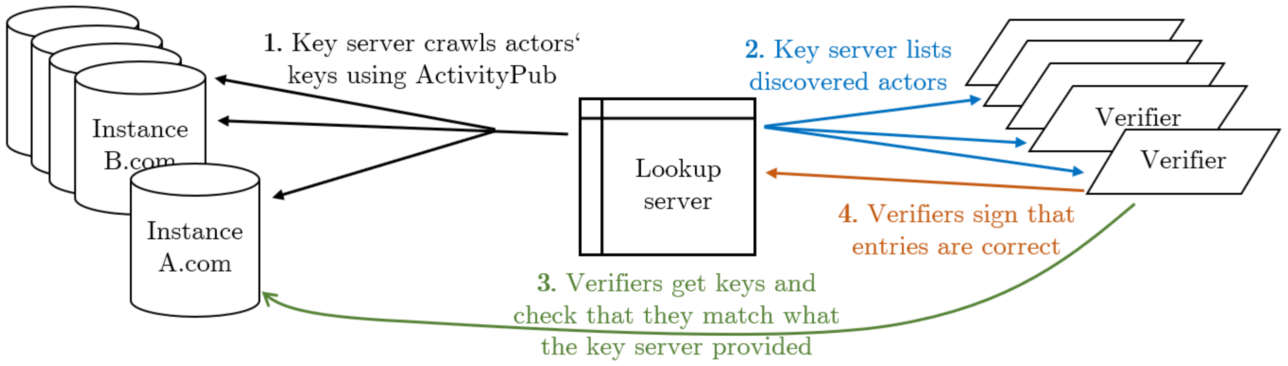

1. Key server crawls actors' keys using ActivityPub
2. Key server lists discovered actors
Instance B.com
Instance A.com
Lookup server
Verifier
Verifier
4. Verifiers sign that entries are correct
3. Verifiers get keys and check that they match what the key server provided

Figure 3.1: *A diagram of system's components and how key gathering works.*

Then, I modified Mastodon to cache not only the data but also the signatures received from other instances. This allowed me to forward a cached copy to other servers in case the origin server becomes unavailable. By checking that signatures match the authors' public keys, I can guarantee that the data is authentic. Thus, the availability of the content and social network's reliability is improved.

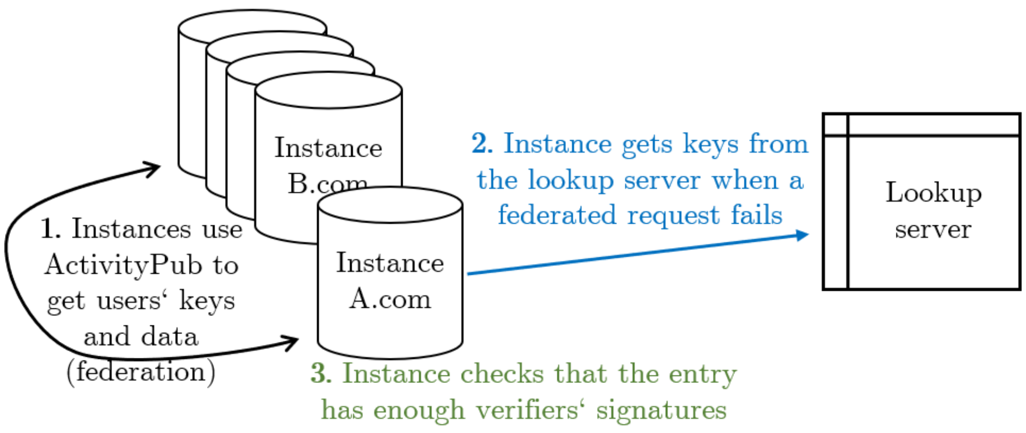

Instance B.com
Instance A.com
Lookup server
1. Instances use ActivityPub to get users' keys and data (federation)
2. Instance gets keys from the lookup server when a federated request fails
3. Instance checks that the entry has enough verifiers' signatures

Figure 3.2: *A chart depicting how the components of the system interact to share data. Note that verifiers are not involved directly: they submit their signatures to the key server beforehand and instances check those against locally stored verifiers' public keys.*

3.2 Gathering keys

For the key server to be able to return actor objects by id when their home servers are unreachable, the system must gather the data beforehand and store a copy in its own database.

There are three different approaches that I considered:

1. creating an ActivityPub relay and asking server operators to forward[1] all (public) traffic through it;
2. building a simple ActivityPub server, following some users from other servers or joining ActivityPub relays, and scanning the delivered content;
3. crawling the ActivityPub network.

In addition, for a specific service, it might be possible to use its custom API endpoints provided and gather data by querying them. Indeed, Mastodon has the "Explore users" API. However, gathering data via it has many weaknesses: it lists only the most popular users, the exact query and response formats and semantics might change in later versions, and, most importantly, it is not compatible with other implementations of ActivityPub. The first two points severely limit the usefulness of such a technique and the last violates Compatible[2]. Therefore, I decided not to use it as the main approach, although, it might be used to give the crawling approach a head start.

The ActivityPub relay approach relies on server operators adding a relay of each key server to their relays list. That would limit the effectiveness of my system and would not satisfy IncrDeploy. Furthermore, this relay would receive an enormous amount of traffic violating the LowResource constraint.

Similarly, an ActivityPub server that follows all users it learns about and/or some ActivityPub relays should have high request throughput breaking LowResource too. Moreover, it would interfere with the network by artificially increasing follower counts and server loads in the entire network, likely resulting in gatherers being banned from popular servers.

[1]This can only be done manually, by changing the server's configuration.
[2]References in small capitals refer to the requirements list.

Given all that, I implemented an ActivityPub crawler (§3.2.1). Obviously, crawling a network once is not enough: keys might change as well as new users and new links between users might appear. A crawler that quickly traverses and saves the entire network but never updates those entries would never find links between those old crawled users and newly created ones, hence, never reaching and crawling those new users. In opposition, a crawler that constantly updates old entries will eventually run out of resources for crawling new users. Updating is summarised in §3.2.2.

3.2.1 Crawling

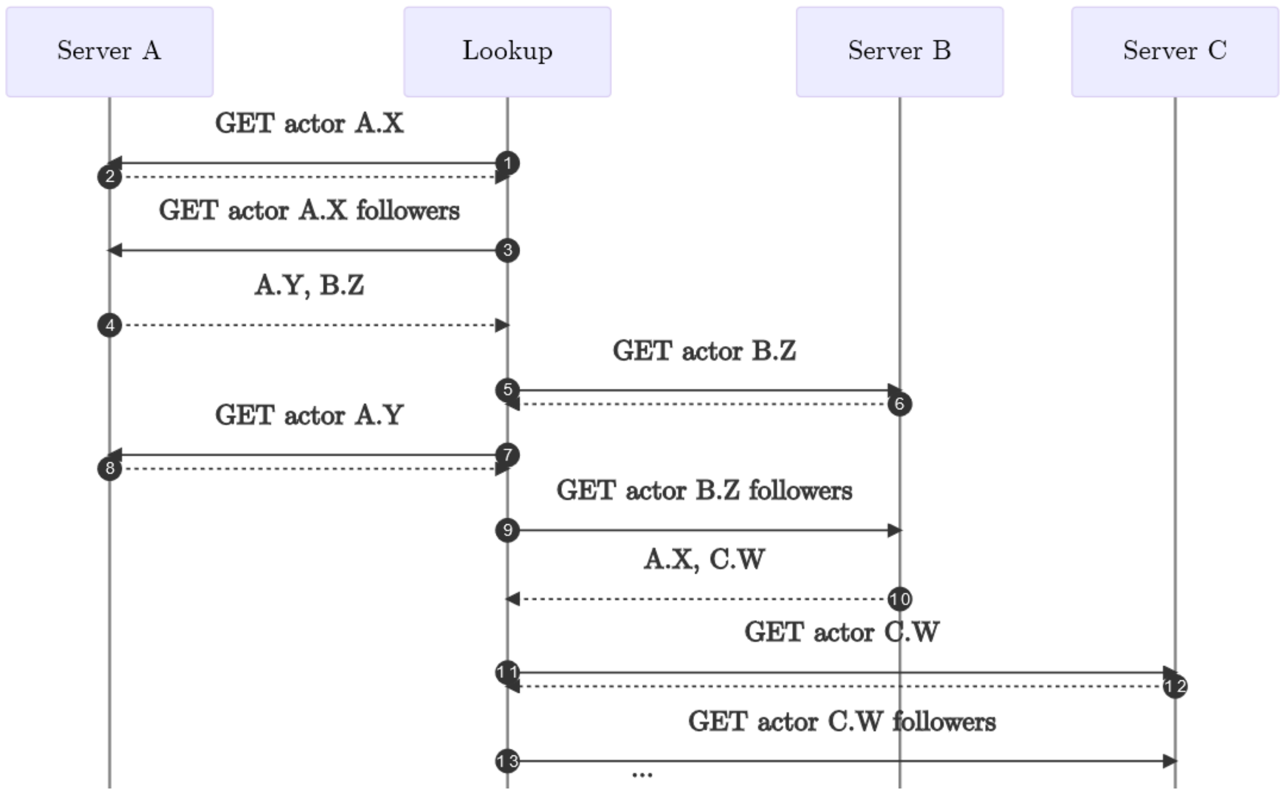


Figure 3.3: *A diagram showing how the crawler operates. The order of the queries is randomised and rate-limited, hence, this is only one of the possible execution traces. Also, the actual implementation makes multiple requests in parallel. Note, that some actors might be found again after they are already crawled, for example, `X@A`. The crawler ignores such occurrences. A separate algorithm (§3.2.2) is responsible for updating entries.*

Consider a graph of ActivityPub where nodes represent objects and an edge from object A to object B exists if some field of A references B's URI. This graph is huge, however, our crawler is only interested in a subset of it that likely contains references to new users. Empirical investigation showed that Actor's Following and Follower collections (and their pages) are such objects. An example of the crawling procedure is presented in fig. 3.3.

The crawler behaves like a graph search traversal starting from some (manually defined) initial set of actors. It keeps a queue of not yet visited users that it has reached. This queue is initialised to the starting set and stored in the database to persist between key server restarts or failures.

The URIs are chosen from the queue one by one[3]. The crawler then fetches the chosen URI and examines fields that might contain references (fields were chosen manually based on ActivityPub

[4] and ActivityStreams [21] protocols). If they contain any not yet found URIs, then a priority for those new URIs is determined based on the priority of the current object and what type they are expected to have. The new URI is inserted into the queue.

The crawler consists of multiple workers running on the same thread and working in an asynchronous fashion, i.e. a worker that is not blocked (waiting for a request result or a database query to finish) is given control.

Choosing a user to crawl

The algorithm for choosing a user to crawl is crucial not only to satisfy Fair but also to keep the crawler efficient[4] while being rate-limited per-domain basis.

Obviously, a single FIFO queue is an example of a bad selection algorithm: local references (to the same server) are more likely, hence, the crawler will often hit per domain rate limits. In addition, if some server becomes unreachable, it is not clear how to reschedule queries to it: appending them to the end will not only create batches of queries to the same domain but will also delay them breaking Fairness; the randomised insertion would not suffer from such problems but is difficult to implement in persistent storage.

I have considered two extreme alternative randomised algorithms and then derived an approach that combines the benefits of both and still can be efficiently implemented.

First alternative: have a FIFO queue for each domain and a Round-Robin for choosing a domain. Notice that this algorithm is unfair: users from small servers have much a higher probability of being fetched. Therefore, small servers will be traversed first and then the crawler will become restricted by a per domain rate limit.

Second alternative: select the next URI to fetch from the queue at random. The problem is that a crawler tends to get stuck inside a few bigger servers because there are more local references so more of them are added to the queue.

Combined approach. I schedule URIs to be fetched in chunks. For each chunk, I select one of two strategies: select a domain uniformly at random and then pick random URIs from it; or select a set of random URIs (equiprobably). Then, the chosen set is added to the work queue.

The producer-consumers model is used to distribute URIs from the work queue to workers that crawl URIs. Workers take URIs in a FIFO order (skipping rate-limited domains to comply with Compatible) and process them. The size of the work queue is much smaller than the full queue and can be adjusted based on network throughput, number of workers and other parameters. It is stored in volatile memory because recreating it randomly when the crawler is restarted will not affect fairness.

Handling network errors

All network errors that might occur when fetching an object can be divided into two groups: permanent and temporary. In case of a permanent error, the object is considered gone and never refetched or updated. Temporary errors are very different: they (might) indicate that

[3]To handle larger networks, it is possible to make the crawler distributed. I have not implemented it but in that case scheduling should be done in blocks. If a scheduler then becomes a bottleneck, then it is possible to run multiple schedulers in parallel by assigning each a subset of domains that it responsible for. This assignment might be altered dynamically to ensure even workloads and fairness.

[4]Notice that a queue might contain much more entries than the network has users.

the object cannot be reached at the moment but might become available later and, hence, the request should be retried.

Temporary errors are those:

- connection timeout (server might be overloaded),
- unreachable host (network partition between crawler and server),
- server/gateway and similar errors (5xx HTTP status codes),
- rate limit exceeded (429 HTTP status code).

All other errors are considered to be permanent, for example:

- invalid URI,
- Not found, Bad Request (4xx HTTP status codes, except 429),
- unexpected Content-Type (server does not support ActivityPub)

A domain resolution failure is an example when the crawler cannot decide if it is temporary or permanent. If the domain's DNS entry was recently updated and the crawler's cache is stale, then the error is temporary. However, it is permanent if that domain does not exist. A safe decision is to consider it a temporary error.

The crawler is also periodically checking the network connectivity and pauses if the Internet is unreachable. This avoids wasting resources and potentially marking servers as unreachable.

For temporary errors, an exponential back-off-and-retry strategy is used: all requests to that domain are paused for $\min(a * b^r \text{ seconds}, 1 \text{ day})$ where r is the number of failed requests in a row (a and b are constants, I chose $a = 10$ and $b = 5$). If a server fails R times in a row ($R = 56$ in my case resulting in around 50 days for the server to become available since the first failed request), then it is marked as Unavailable and all later requests to it are skipped.

In case of a permanent error, the URI is marked as Unreachable and never refetched.

WebFinger

As mentioned before, Mastodon uses the WebFinger protocol to resolve usernames into ActivityPub URIs, therefore, a lookup server should be able to serve those as well. However, WebFinger usernames are not a part of the ActivityPub protocol, hence they should be considered as an optional extension to the crawler and lookup server. The system must support servers that do not implement WebFinger (Compatible).

This is achieved by constructing a WebFinger username and resolving it for each crawled actor. If the resolution fails at any step, it is considered to be unsupported. However, if the resolution was successful, the WebFinger username is stored along with the actor. Verifiers will verify the username using the same resolution protocol and only sign the entry if the username is correct.

Security

Security issues may arise if the crawler encounters a malicious server: it might return an object with many references to non-existing objects on some target URL. Then crawler will waste time fetching those objects and failing and it may also increase the load on that target server.

A malicious server could also give a list of URIs that do not exist yet. The crawler will mark them as fetched and failed. Later, when an object with one of those URIs is created, the crawler will believe that it is unreachable and will ignore it.

Currently, I use a list of blocked domains – badly behaving servers can be added here manually. This is the way that Mastodon instances deal with the spam problem.

Protocol violations

Some implementations of ActivityPub violate the protocol and that leads to some issues. For example, Friendica [22] does not assign unique ids to each collection page[5]. This does not affect the functionality of the system, however, it causes the crawler to insert incorrect aliases into its database. In this case, the lookup server still serves queries for actors from such servers correctly. Similarly, PeerTube [24] used the incorrect `OrderedCollectionPage` type for follower collections[6] instead of the expected `OrderedCollection`, however, that did not break the crawler.

On the other hand, some implementations (e.g. Write Freely [26]) have infinite size collections: contents are followed by empty pages. This unnecessarily slows the crawler down, thus I decided to modify my code to handle such cases.

It is important that such inaccuracies do not affect other servers in any way, fulfilling Byzantine. However, they might result in actors from this misbehaving server being unreachable by the crawler or unavailable in the lookup server.

3.2.2 Updating

Updates are essential in the long term: once everything is crawled I will not be able to find newly joined actors unless follower lists of old ones are refetched. However, the updater must not overwhelm the crawler with old inactive accounts.

To optimally exploit the given bandwidth while ensuring maximum entry freshness, I decided to use incremental updates. Each object has an update period adjusted multiplicatively on every refetch: decreased if the object changed, otherwise increased. Minimum and maximum update periods are used to prevent wasting resources on volatile objects (updating such objects less frequently increases overall expected freshness [27]). Limit values can be adjusted based on network properties (e.g. number of users).

When the update period passes, I add an object to be updated to the crawler's queue. In other words, the same rate-limiting and scheduling algorithms are used. I determine whether an object changed by storing its hash in the database and comparing the old hash with the new one. Storing a hash uses significantly less space than the entire object (LowResource). To avoid cases when the server returns arbitrarily formatted JSONs, I normalise them before hashing.

3.3 Key server

The key server consists of two main modules: the crawler and a web server. The web server provides endpoints to get a cached copy (together with all signatures for that object received from verifiers) of an actor by URI or by a WebFinger username as well as other methods needed for verifiers (discussed in §3.4).

[5] I have reported this bug [23] and it was quickly fixed, however, many instances remain not updated.
[6] I reported it [25] and it was fixed.

3.3.1 Distributed server

To handle more users, the web server can be distributed among multiple machines. The server is inherently stateless, thus distribution is trivial: start multiple web servers (without crawlers) on multiple machines with the same underlying database and place a load balancer or assign unique URIs. However, this approach turns the database into a bottleneck.

Note, that the majority of the requests to the lookup will be to get actors because their home server failed. Hence, to avoid the bottleneck, one might set up read-only replicas of the database. They do not have to be updated immediately because the chance that a server failed quickly after being crawled is very low.

3.4 Verifier

The purpose of verifiers is to move the trust from the key server and distribute it among multiple servers, i.e. verifiers, that constantly watch key server entries and report any inaccuracies found. Those can be later reviewed by the operator and discussed via other authenticated communication channels. If a key server exhibits strange behaviour, it will lose its reputation and server operators will migrate to other key servers.

When the data is checked, the verifier uses its private key (§3.4.1) to sign each correct entry individually and sends signatures (§3.4.3) to the key server. The key server stores signatures in its database (§3.4.2) and serves them together with the data.

A verifier must obey the rate limit requirement (Compatible). This is done by having a queue (in RAM) of entries to check and choosing the oldest one whose domain is not rate-limited. To ensure efficiency while keeping the resource usage low, verifiers check entries in the same order the key server fetched them. It is already a good schedule because the crawler worked under the same constraints.

Note that the job of a verifier is very different from the crawler's: it already knows the URIs of actors and only requires one (at most three if WebFinger has to be checked too) request per actor to verify it. The majority of the actors were recently reachable (otherwise the crawler would not have fetched them), hence verifiers waste fewer resources on actors of terminated instances. In addition, verifiers do not need to store all entries they signed, meaning that the disk usage remains constant over the lifespan of the verifier. Also, that means that entries can be checked quicker than the crawler can create them, allowing verifiers to catch up even if were started later or were down for some period of time.

Each server operator can choose a subset of verifiers it trusts and specify a minimum number of signatures required to trust an entry received from the key server.

3.4.1 Keys

A verifier generates a public-private key pair when started for the first time. The public key is served on a URI that also uniquely identifies the verifier. When a verifier is added to the key server or marked as trusted in the ActivityPub service (or a proxy), it is fetched and stored in the persistent memory. This allows signatures to be checked even then the verifier that created them is unreachable.

Secrecy of the private key is crucial for the trust of the verifier. If it gets leaked, that verifier cannot be trusted and must be removed from all ActivityPub instances and proxies. The owner must create a new key pair (using a different verifier URI) and resign all entries.

Keys can be rotated on a regular basis to confine the impact of the key breaches. This should be done by assigning a new URI of the same domain for each key.

3.4.2 Signatures

There are two locations where signatures can be stored: in the individual verifiers or in the lookup server. I have chosen the latter approach. This means that a verifier does not have to be reachable when someone wants to verify its signature. An actor query becomes simpler and faster because only one network request to the key server is needed in contrast with individual requests to each verifier. Again, a verifier does not have to store all signatures in its database reducing resource requirements (LowResource).

It is also important what data is signed. The obvious choice would be to sign the entire actor object serialised into JSON. However, it includes fields that might change frequently, for example, `summary` that contains the user-created summary of the account. Therefore, I decided to sign only the fields that contain data that should not change, such as id, type, following and follower collections, inbox and outbox URIs, and, most importantly, the public key. Mastodon does not forbid public key changes, therefore, actors have to be regularly refetched and updated. The proxies should handle fields that are not included in the signature with care. Optional fields might be removed or a warning for the end-user might be displayed.

3.4.3 Communication with key servers

A key server has an API endpoint to query the entire database of actors in numbered pages. A verifier starts from the page chosen by the operator (very old entries can be ignored) and continues to go forward. When an actor is updated by the key server, the old entry is removed and a new entry appears on the last page. Therefore, it will be noticed, checked and signed by the verifiers. When an entry is removed, its page contains fewer elements but other pages are not changed (i.e. content is not shifted). This guarantees that verifiers will not accidentally miss entries.

When a verifier reaches the last page, instead of moving to the next one, it stalls and periodically refetches the page and signs new entries until a new page appears.

To implement this, I used two counters: the next page (stored in RAM) and the first unsigned page (stored in persistent memory). The next page counter tracks which page to fetch next (or refers to the last page if it was reached). The first unsigned page counter ensures that the verifier can continue working after the restart and does not skip any entries that were lost from the volatile queue. It tracks the first page that contains at least one actor that is neither signed and submitted to the key server, nor marked as unavailable and stored in the database for later retry.

Signatures from the verifier to the key server are submitted in batches to save resources. If a signature submission fails then a verifier pauses all fetching and periodically retries to submit them. This might happen if the key server crashes or if the verifier is not marked as trusted by the key server.

3.4.4 Handling errors

As described above, a verifier is expected to encounter fewer temporarily or permanently unreachable actors. However, it still has to handle cases when the origin server becomes unreachable. Due to the rareness of such errors (again, a request by a verifier is expected to happen soon after the crawler fetched it) and the non-Byzantine[7] nature of the key server that supplies actors to fetch, errors are handled in a different way than in the crawler.

A verifier tracks status of all instances that it queries: if multiple requests to the same server fail (no matter the error), the server is marked as temporarily unreachable and all requests to it go directly to the database queue for later retry. Each request is assigned a different time at which it should be made, in a way that complies with rate limiting. Therefore, no additional scheduling work will be needed when recovering from failure.

3.4.5 Revoking verifiers

Inevitably, some verifiers will become untrustworthy either due to an accidental key leak, hacker attack or some other reason. It is important to have a mechanism to eliminate such verifiers from the system.

This can be done at two levels: the key server or individual ActivityPub services. A key server operator can remove a verifier and all related signatures from its database. Alternatively, ActivityPub service operators can remove that verifier from their lists of trusted verifiers.

3.5 Mastodon modifications

An alternative to using a proxy is modifying the application directly. That involves modifying two procedures: user account fetching and resource fetching, and adding an additional API method for serving resources based on their URI.

3.5.1 Fetching user accounts

While other servers are functioning normally, the system should not interfere (NoInterfere). It is utilised when a query to get a user account from another federated server fails. In that case, a request to the lookup server is issued for that user account (either by URI or WebFinger). If the lookup server is unreachable as well or it does not contain the required entry, the entire operation fails in the same way as it would have before.

However, if the lookup server returns an actor together with signatures created by verifiers, those signatures have to be validated. I do that by iterating through the returned tuples of signature and its author, comparing author URIs with trusted verifiers, and checking their signatures. If at least one signature mismatches, I return and log an error because the key server is intended to verify the signatures before storing them in a database (that acts as an authentication of the verifier). Similarly, if there are not enough signatures from the trusted verifiers, I treat the request as failed. Finally, if enough checks succeed and none fail, I return the object and treat it as returned by the original server[8].

[7]There is nothing that protects verifiers from the malicious or misbehaving key servers. However, the purpose of verifiers is to assure that key server's entries are correct. Hence, if it provides invalid URIs then verifiers will waste resources to fetch them, instead of signing entries. Not only verifiers' operators will notice a high failure rate and report it, but the key server will have fewer signed entries leading to dissatisfied users moving to alternative servers. Affects on security are further analysed in §4.1.3.

[8]At this point, fields that are excluded from the signature can be erased or set to some default value. If needed, mandatory fields can be marked as unreliable and some warning symbol can be displayed for the user. Those will be updated on the next actor update, that are done regularly.

3.5.2 Serving resources

When new content (e.g. a post) is created, it is delivered to all federated servers that host members of its audience. As a result, the data is replicated on multiple servers. I added an API endpoint to Mastodon that given any URI (even from different domain, e.g. step 4 from fig. 3.4) returns an object and its author's signature assuming the queried server has them cached. That allows servers to share objects that they received from others, improving content availability.

To implement that, I added a new column to the objects table in Mastodon's database that contains signed JSONs as they were originally delivered. The new API endpoint only returns objects that are either public or unlisted. Because an object is queried by its URI, the requester could have fetched it from the origin server if it was reachable. Hence, the Security requirement is satisfied.

3.5.3 Fetching other resources

Finally, I use the endpoint described above to rescue unsuccessful requests for other objects (not actors). The integrity of them is verified by ensuring that they were signed by the proclaimed author, similarly to how objects delivered to the inboxes are checked.

I determine from which instance to request the object by looking at what caused the query to be issued in the first place. It can be a result of the user interaction: in that case, I can only guess a server and the chance of success is low, therefore, I treat the request as failed. However, if the failed query was caused by the delivery of some other object, (e.g. user from one server shared a post from another server), the sender presumably has already dereferenced the unreachable object before. Hence, I request that object from it. This process is summarised in fig. 3.4.

3.6 Repository overview

The project was version-controlled using two Git repositories: one containing only my code and a fork of Mastodon with my modifications. Before submitting, the Mastodon fork was copied into `mastodon` folder, hence the submission contains three folders (fig. 3.5).

```
/
├── code
│   ├── scripts
│   ├── src
│   │   ├── common
│   │   ├── lookup
│   │   ├── runners
│   │   ├── verifier
│   │   └── workarounds
│   ├── tests
│   ├── tools
│   ├── README.md
│   ├── requirements.txt
│   └── run.py
├── documents
└── mastodon
```

Figure 3.5: *Structure of the source code repository. Less important files, directories, and deep subdirectories are omitted.*

3.6.1 Source code

The majority of the source code is in the `code` directory: implementations of the key server and the verifier (`src`); tests for them (`tests`); scripts used in development (`scripts`); and all other additional scripts and tools used to gather and analyse the data and evaluate the project (`tools`).

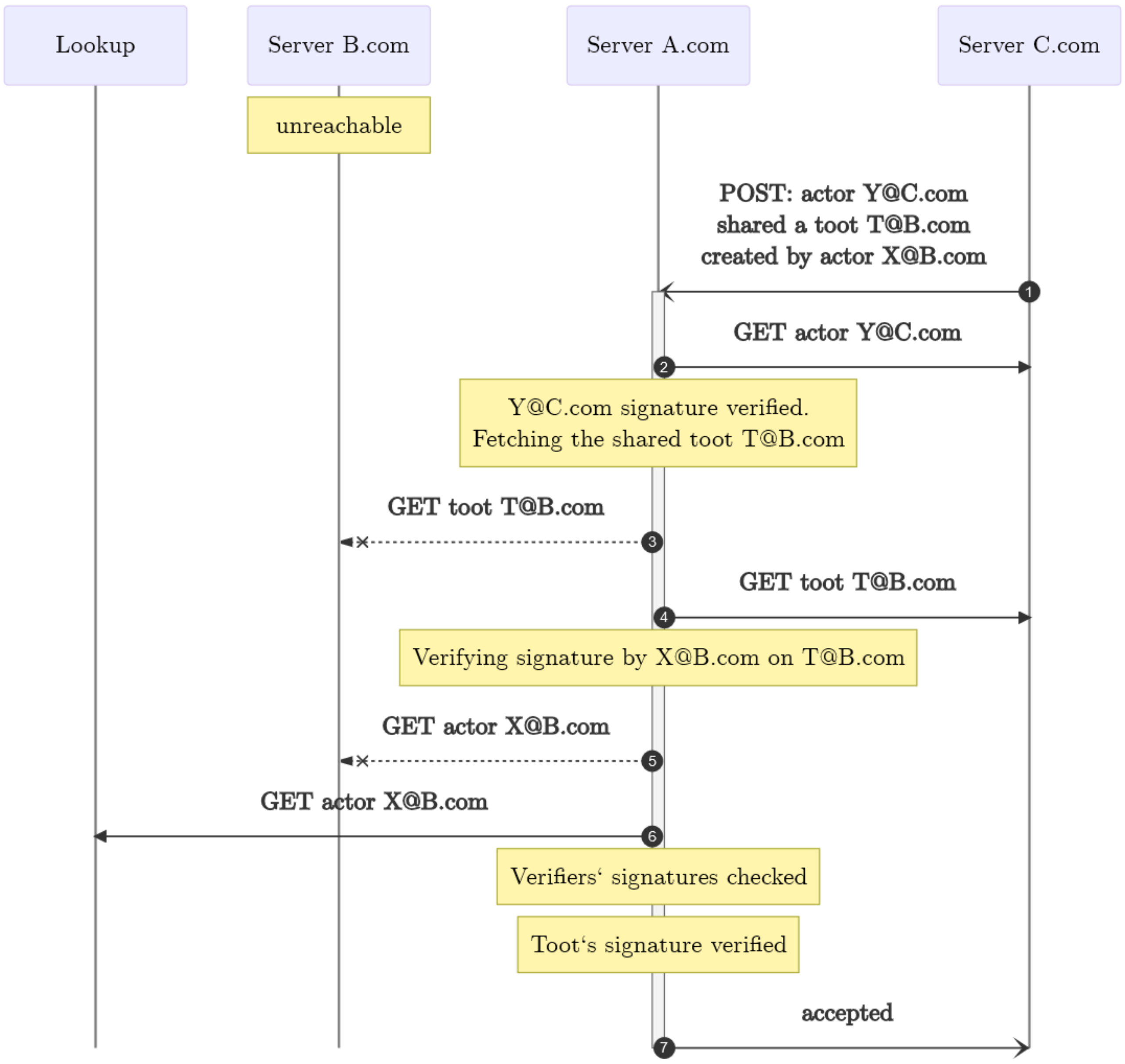

Lookup
Server B.com
Server A.com
Server C.com
unreachable
POST: actor Y@C.com shared a toot T@B.com created by actor X@B.com
1
GET actor Y@C.com
2
Y@C.com signature verified. Fetching the shared toot T@B.com
GET toot T@B.com
3
GET toot T@B.com
4
Verifying signature by X@B.com on T@B.com
GET actor X@B.com
5
GET actor X@B.com
6
Verifiers' signatures checked
Toot's signature verified
accepted
7

Figure 3.4: *A diagram depicting how I rescue a share of a toot (short message) when its origin server B is unreachable.*
Server C starts by delivering a message that its user Y shared a toot T from server B. To handle the request, A fetches actor Y, checks that message is signed correctly and attempts to fetch the shared toot. However, the request to origin server B fails, so it is rescued by asking C because C referenced the post in the first request. Then, C returns a copy together with the signature by T@B.com, which it has cached before. Again, the request to get actor X from B fails, but Lookup rescues it. First, verifiers' signatures on actor X are checked, and then its public key is used to verify the signature on the toot. Finally, the POST request from server C is accepted.

The key server, the verifier and all tools were implemented
in Python and can be executed using the `run.py` script provided. Command-line options can be used to alter the behaviour as well as JSON configurations.

Pip package manager was used to install dependencies and a list of them, required to run the project is provided in `requirements.txt`.

The `mastodon` directory contains not only my modifications (a list of modified files is given in appendix A) but also the original Mastodon code, both in Ruby.

3.6.2 Testing

In fulfilment of professional practices [28], the Python source code is covered by three levels of tests: unit tests of individual classes and functions, integration tests of components integrated together and end-to-end tests. The interaction between the Key server and modified Mastodon was tested manually.

3.6.3 Documents

Project proposal, dissertation and progress report sources, graphics and building scripts are stored in respective subdirectories of `documents`. All documents were typeset using $\LaTeX$.

Chapter 4

Evaluation

In this chapter, I evaluate the success of the project in terms of requirements (§4.1, including Correctness, Functionality, and Security) as well as performance (§4.2).

4.1 Acceptance testing

The success criteria of the project are described in the project proposal (appendix C). I have not implemented the proxy but modified Mastodon directly instead, hence the criteria changed accordingly.

4.1.1 Correctness

Ensuring the correctness of a distributed system is a difficult task. Doing a formal verification of my implementation would probably involve more work than the entire project, especially because ActivityPub and other protocols are not formally described. Arguably, that would not make the entire ecosystem more secure, because ActivityPub services have no plans to be formally checked and they are more likely to contain vulnerabilities due to large code size and frequent updates.

For that reason, I relied on extensive manual and automatic testing (§3.6.2), Python typing, and PyCharm IDE insights to catch mistakes and ensure correctness.

4.1.2 Functionality

In addition to automatic tests, I have also conducted manual testing of system functions. To do that, I deployed my system: installed and started two modified Mastodon instances in the cloud and on my computer. I created multiple users and followed users from other instances. Servers were fully functional even though a key server was not started.

Then, I ran a key server with a crawler. The crawler started from one of my cloud instance users. I waited for some time and then queried the key server with user URIs that I followed before (from both my and other instances). All queries were successful and contained no signatures as expected.

Next, I started a verifier for that key server. The verifier got the first page of actors, fetched them and tried submitting. The request failed with a 403 Forbidden response as expected because the key server did not recognise this new verifier. I then marked the verifier as authorised in my key server. After that, the verifier continued fetching and submitting entries. I queried some actors again and now entries contained one signature. Finally, I restarted the verifier and checked logs to ensure that it saves the state properly, instead of starting from the first page of actors again.

To test my Mastodon modifications, I wrote a script that delivers a message using the ActivityPub protocol. I took a private key of one of the users of the cloud instance, turned it off, and delivered a message signed by that key to my local instance. Logs showed that a query for the author's actor failed, but a lookup in the key server succeeded and I observed both the delivered message and correct user data. I then created two more modified Mastodon instances and carried out a similar test for sharing a message authored by a user whose home server is down.

To test that my crawler works with different implementations of ActivityPub, I have selected a few most popular ones (based on figures in [29]), created users on some instances of them, and started the crawler with an empty database from each user separately. After some time, I checked statistics and the database to see that it successfully discovered other servers and users. I tested Mastodon [5], PeerTube [24], Pleroma [30], and Friendica [22] which worked, as well as PixelFed [31], Writefreely [26], Lemmy [32], Birdsitelive [33], and Juick [34], which did not work because they do not disclose follower lists via ActivityPub API. However, their actors can still be crawled if links are found on other sites. I verified that by following those sites from my Mastodon account and running a crawler.

I left the key server and verifiers running for multiple days, occasionally rebooting the underlying machines forcefully to simulate power losses. During that time, they crawled more than 42 million ActivityPub objects, including more than 1.2 million actors. This acted as a smoke test for the system and the absence of crashes while running on a real ActivityPub network demonstrated that my system is reliable. Later, I did a few similar runs to measure performance statistics (see §4.2).

4.1.3 Security

Here, I analyse a number of security properties of the ActivityPub and argue that they are preserved unless a key server and a quorum of verifiers are behaving maliciously. In the latter case, only instances that trust at least N malicious servers are affected (where N is the minimum number of verifier signatures needed to trust a key server entry).

Misbehaving key server

A misbehaving key lookup server could return arbitrary entries to verifiers or federation instances. The first failure mode would mean that verifiers are refusing to sign entries and are not submitting signatures to the key server. Operators of the verifiers should notice the increased number of incorrect entries and report the key server as untrusted.

Because of how Public-key cryptography works, the malicious or misbehaving key server cannot forge signatures from verifiers because it only has their public key. Hence, in case of the second type failure, the ActivityPub servers would reject entries from the key server either because there are not enough signatures or because an entry contains incorrect signatures.

Misbehaving verifiers

A misbehaving verifier could either deliver incorrect signatures to the key server or arbitrary choose what entries to sign. In the first case, incorrect signatures would be rejected by the key server. In the second case, by assumption, the key server is functioning correctly, i.e. all entries are correct, hence all should be signed, but the verifier might reject some of them. Then, those entries will have fewer signatures than they should and fewer requests might be rescued. A key server operator should notice that a verifier is skipping too many entries and report it.

Misbehaving key server and verifiers

If both the key server and some verifiers are not behaving properly, all of the above apply, but there are new failure modes: a key server could accept incorrect verifier signatures or add incorrect entries and verifiers could sign them. In the first case, incorrect signatures will be detected by an ActivityPub instance and the operator will be notified. In the second case, the entry will not be trusted unless there are enough misbehaving verifiers to form a quorum. Recall, that the minimum quorum size and trusted verifiers (i.e. whose that can participate in a quorum) are selected by each instance operator individually.

Therefore, to inject incorrect data into an ActivityPub instance, a malicious actor must take control of a key server and enough verifiers to make a quorum on that instance. Furthermore, affected instances will not share these invalid entries with other instances because actor sharing is forbidden. Hence, the incident will be contained. Note that after injecting fake keys, the attacker will be able to push arbitrary content to hacked instances only when the claimed source server of the data is unreachable.

Misbehaving Mastodon instance

A malfunctioning instance can affect all its users. In addition, after my modifications instances are serving cached objects to others, thus a misbehaving instance might respond to these queries arbitrarily. However, a receiver (if it was not hacked) will notice that the presumed signature by the post author does not match the cached content and will report failure.

4.2 Performance

4.2.1 Crawler

Again, I started the crawler with an empty database on a real ActivityPub network and tracked a few statistics while it was running. This time, to isolate the crawler's performance, I disabled all verifiers. The crawler started from one of my accounts on ActivityPub and found almost a million users in total. It was running with a rate limit of 2 seconds per domain. I describe details of data gathering and processing in Verifier.

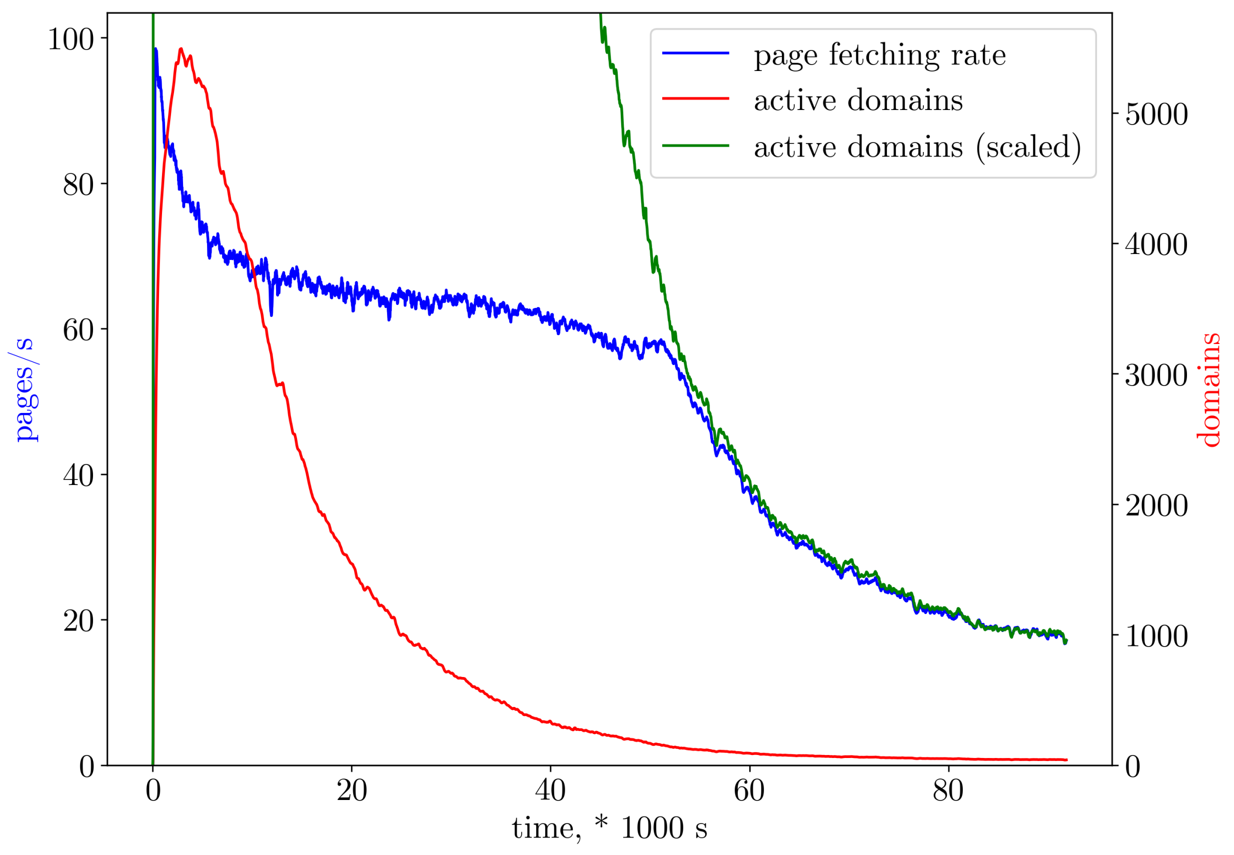


Figure 4.1: *How crawling rate (pages per second) depends on the number of active domains. Active domains here: domains on which crawler knows at least one not yet visited URI and that do not have streaks of failed requests. The green line is plotted from the same data as the red but scaled to match the blue's tail (×24).*

To produce fig. 4.1, I disabled updates as well. At first, the crawler knows only one domain but it quickly finds others and the active domains plot quickly reaches its peak. After that, it starts declining as smaller domains are fully crawled and only a few new ones are found. During that time, the fetching speed that starts high is quickly limited by the cloud provider and remains more or less constant with a slight downwards trend, possibly because slower instances take longer to crawl. The crawler is intentionally forced to wait 2 seconds (rate limit) from the last response before the next request is issued: this is to reduce the load on already slow and possibly congested domains.

After about 50 000 seconds, the crawling rate starts to decline rapidly because there are not enough different active domains, hence the crawler has to stall and wait for rate-limiting to expire. The effect becomes evident at around 130 active domains, that – at 2 s/domain/ rate limit with zero response times – could handle 65 requests per second: close to the actual rate of 59 pages/s at that time.

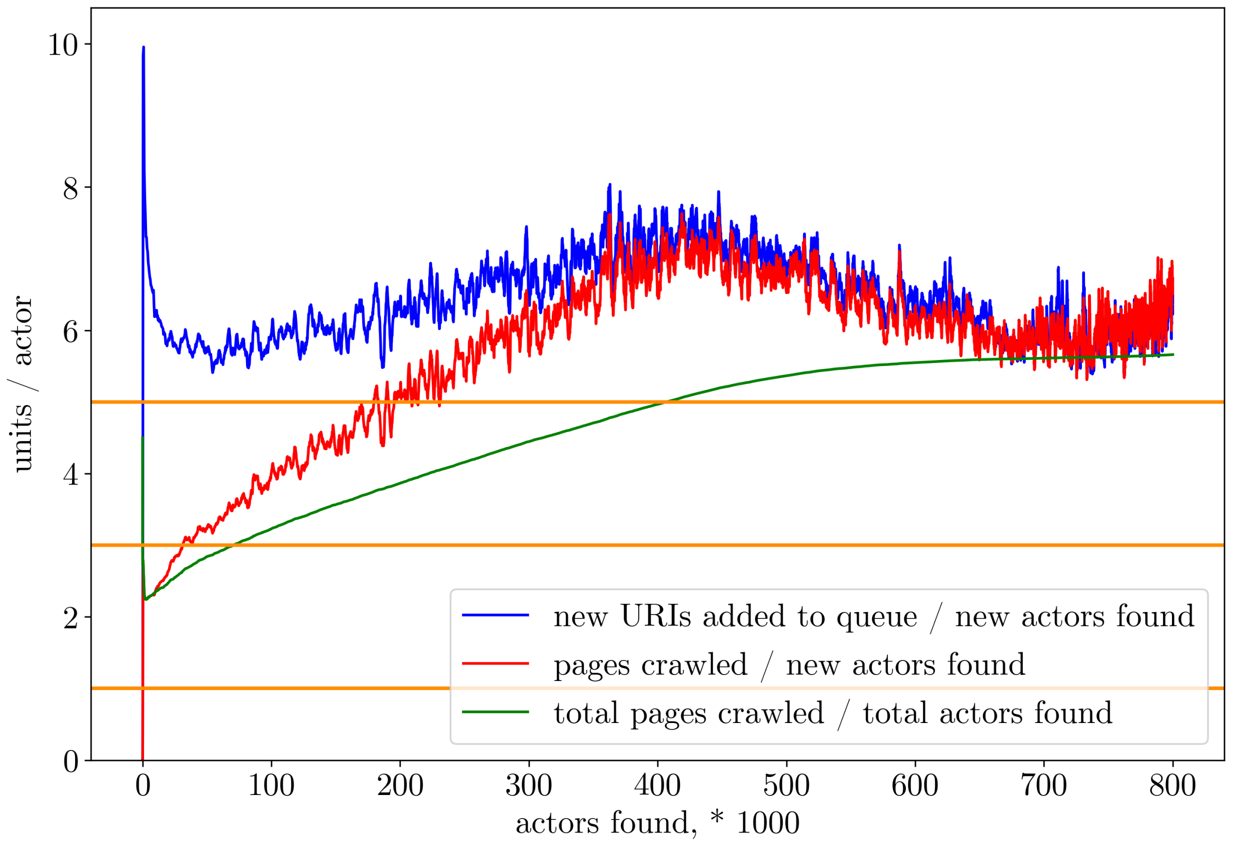


Figure 4.2: *Crawling efficiency (measured in pages crawled per new users found) and expansion of the crawled network (new URIs per user). The green line represents overall efficiency: the number of pages crawled divided by the number of actors found from start. Horizontal lines represent ideal cases: follower collections are 1) included in users' objects; 3) unpaged but not included in users; 5) not included and paged.*

Plotted from the same data, fig. 4.2 examines the efficiency of my crawler: how many requests are needed to find an unseen actor. Ideally, we want this number to be as small as possible: close to 1 request per user (lower line). However, this is not possible because almost all ActivityPub implementations do not embed follower and following collections into the user's object, resulting in two additional pages per user (middle line); most of them paginate those collections and do not embed the first page into the collection, summing up to 5 requests per user (upper line). Some users might have more than 1 page of followers, increasing the ratio even further.

Initially, efficiency is high because the database is empty: every actor and URI found are new. It starts decreasing as the database size grows because more and more actors are already known – social connections are often bidirectional, meaning that the crawler is likely to find already crawled actors. Interestingly, the trend suddenly changes in the middle of the graph. I think at that point I found a closed community but it requires further investigation.

4.2.2 Verifier

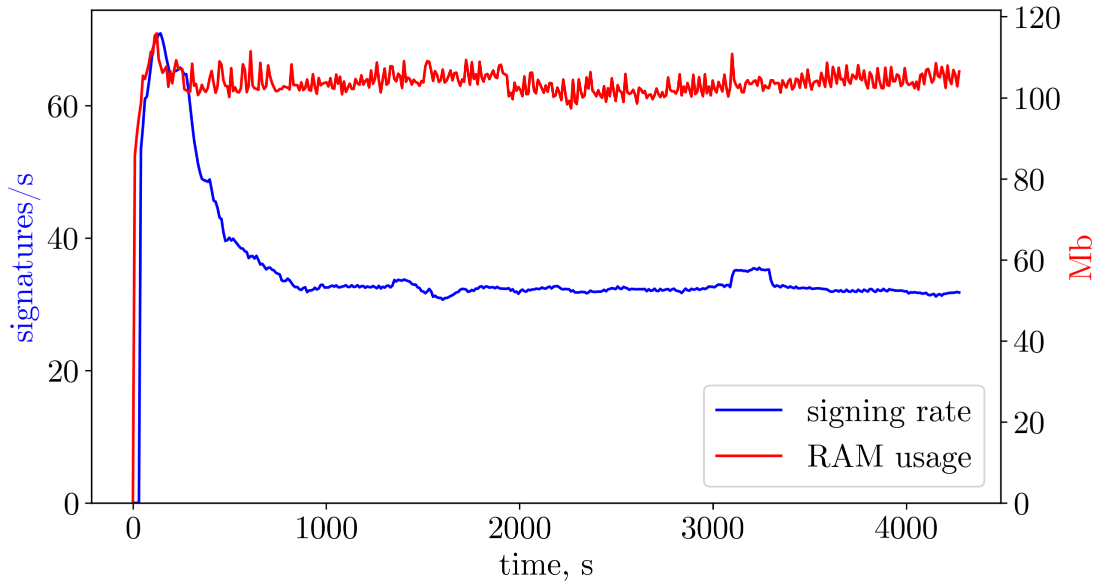


Figure 4.3: *Verifier's performance. Both, signing rate and memory usage are almost constant.*

I started a verifier to watch and sign my lookup server's entries. As expected, after initial spikes both signing rate and memory usage remained constant. The initial spike is due to the verifier looking ahead and fetching and signing entries from faster servers. Once the working queue fills up, the rate stabilises because the verifier is fetching entries in the order they are provided. The width of that spike changes accordingly to the queue size (fig. 4.4).

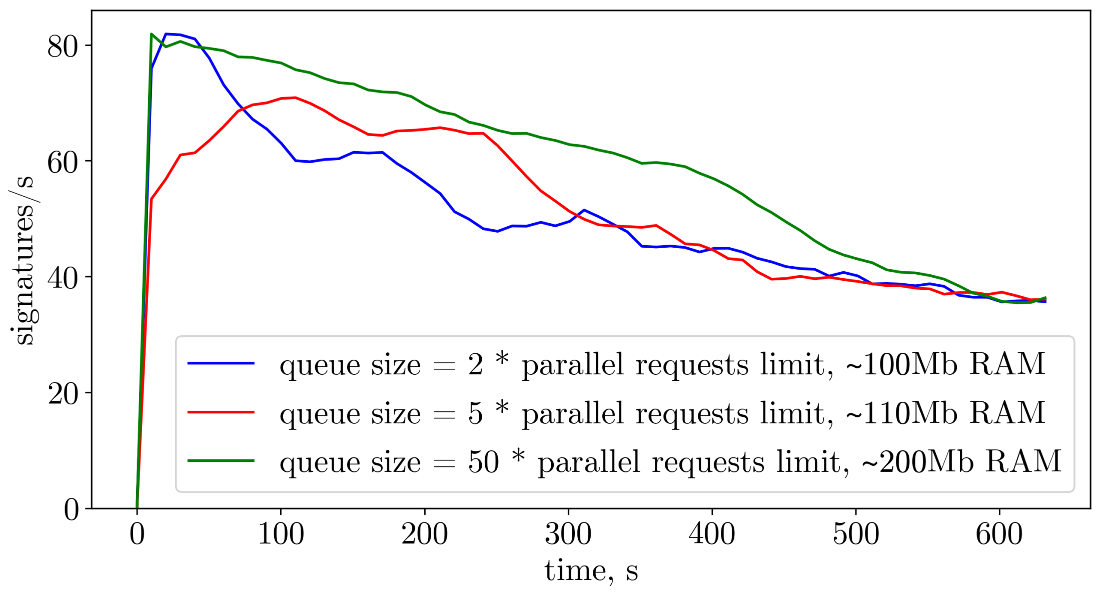


Figure 4.4: *Performance when using different verifier queue sizes. They all settle down at the same rate an remain stable.*

Chapter 5

Conclusions

The end result of this project is an incrementally-deployable network of key servers and verifiers, that is used to confine the effect of outages of ActivityPub instances. The reduced impact of failures should encourage more people to host their own servers bringing their private data under their control. I am looking forward to more private, reliable, and decentralised communication tools in the future and I believe that this project contributes towards that.

5.1 Achievements

In the Introduction chapter I explained what problems current social networks are facing and why decentralised systems might be a solution. I extended the comparison in §2.1.4, produced an overview of ActivityPub (§2.1.2) and Mastodon (§2.1.3), and based on those I composed a set of requirements (§2.3) that should be useful for anyone designing a decentralised system or extensions for it.

Next, I suggested a system that has a potential to meet those requirements (§3.1) and implemented individual components to do so. I suggested three alternative approaches to gathering data from a decentralised system, highlighted their pros and cons (§3.2), and selected the appropriate one – Crawling. I also described three strategies to crawling and implemented one that meets all requirements (§3.2.1).

Then, I addressed the problem of trust in a decentralised network and proposed a solution of having a many verifiers operated by different entities (§3.4). A user can then select a set of verifiers that it trusts and check that a quorum of them agrees on the network state, making the network resilient to malfunctioning of individual nodes (security is analysed in §4.1.3).

Finally, I added an API endpoint to Mastodon server (§3.5) that returns posts it has received and cached from other servers (§3.5.2). Then, I used this endpoint together with the key server to rescue requests to federated instances should they fail (§3.5.1 and §3.5.3).

After the core of the project was successfully implemented, I extensively tested the project using automatic and manual tests (§4.1) and measured the performance (§4.2). To understand the limits of this system, I wrote a mathematical simulation and executed it with parameters resembling the real network. It clearly showed a need for updates, and predicted 90% rescue rate for post sharing with incremental crawler, but I was unable to verify that on a real system. Despite that, the concept of using high level simulations to understand the macro behaviour of federated networks could be further researched.

5.2 Lessons learned

This project has been the most challenging that I have completed. I challenged myself to: quickly get familiar with the huge ecosystem of distributed social networks; learn a new programming language (Ruby) and extend the large code base of Mastodon written in it; and create, debug, deploy, and test a distributed system running on multiple servers.

In addition, I learned how crucial the planning is to complete the project on time and that small deviations from the schedule tend to accumulate; I improved my presentation and communication skills by preparing progress reports for my supervisor; I realised how valuable the notes, taken during the preparation and implementation stages, are when writing the dissertation.

When working on similar projects in the future, I should focus more on how modules of the system will interact to see the full picture and avoid later changes. Also, I would think about evaluation more since the preparation step and would collect and save more data while I am developing the system, in contrast to gathering it at the end.

5.3 Further work

I have identified multiple directions for further exploration of decentralised social networking. The particularly interesting topics are presented below:

- **Running on highly unreliable servers**: this project demonstrates how to limit the effects of an instance failure for the decentralised network. An obvious next step is to use this (perhaps extended) system together with unreliable servers, for example, hosted on users' mobile phones or desktop computers, to further improve privacy and security guarantees. The system should be able to cope with changing IP addresses and devices behind NAT units.
- **Decentralised accounts**: as mentioned in the Introduction Chapter, users cannot access the network while their home servers are down. This can be solved by distributing a user account between multiple instances of the federation. Trade-offs implied by the CAP theorem [17] as well as effects on security should be carefully considered.
- **Trusting home servers**: by design, ActivityPub users unconditionally trust their home servers, similar to what they would do in centralised networks. Hence, the damage – although limited to a particular instance and its users – could be caused by breaking into the server and tampering with its database. But if every user checked a small portion of all the data they receive from their home server (by querying other federated servers), someone would detect an inaccuracy. An efficient report system where a user can share and prove the misconduct of a server to others would immediately limit the scope of attacks.
- **Finding federation protocol violations**: a crawler that I wrote could be used to traverse the ActivityPub network and automatically detect protocol violations. I found three bugs by manually looking into what I crawled, so I expect there to be more.

Bibliography

[1] Dan Milmo, The Guardian. *Russia blocks access to Facebook and Twitter.* https://www.theguardian.com/world/2022/mar/04/russia-completely-blocks-access-to-facebook-and-twitter. Visited on April 15, 2022.

[2] Twitter, Inc. *Elon Musk to Acquire Twitter.* https://www.prnewswire.com/news-releases/elon-musk-to-acquire-twitter-301532245.html. Visited on May 05, 2022.

[3] Josh Taylor, The Guardian. *Facebook outage: what went wrong and why did it take so long to fix after social platform went down?* https://www.theguardian.com/technology/2021/oct/05/facebook-outage-what-went-wrong-and-why-did-it-take-so-long-to-fix. Visited on April 15, 2022.

[4] World Wide Web Consortium. *ActivityPub.* https://www.w3.org/TR/activitypub/. Visited on March 15, 2022.

[5] *Mastodon documentation.* https://docs.joinmastodon.org/. Visited on March 15, 2022.

[6] *Decentralised user accounts – Issues – Matrix Specification.* https://github.com/matrix-org/matrix-spec/issues/246. Visited on April 15, 2022.

[7] IETF. *Certificate Transparency.* https://datatracker.ietf.org/doc/html/rfc6962. Visited on March 15, 2022.

[8] *Chrome Certificate Transparency Policy.* https://googlechrome.github.io/CertificateTransparency/ct_policy.html. Visited on April 15, 2022.

[9] Jonathan Anderson. "Privacy engineering for social networks". PhD thesis. July 2012.

[10] *fedilist.com homepage.* https://fedilist.com/. Visited on March 18, 2022.

[11] The Matrix.org Foundation. *Matrix – An open network for secure, decentralized communication.* https://matrix.org/. Visited on April 15, 2022.

[12] IETF. *The JavaScript Object Notation (JSON) Data Interchange Format.* https://datatracker.ietf.org/doc/html/rfc8259. Visited on March 24, 2022.

[13] IETF. *The Transport Layer Security (TLS) Protocol Version 1.2.* https://datatracker.ietf.org/doc/html/rfc5246. Visited on March 15, 2022.

[14] IETF. *Webfinger.* https://datatracker.ietf.org/doc/html/rfc7033. Visited on March 15, 2022.

[15] *Bluesky: Building a Social Web homepage.* https://blueskyweb.org/. Visited on March 25, 2022.

[16] *TorProject homepage.* https://www.torproject.org/. Visited on March 25, 2022.

[17] Seth Gilbert and Nancy A. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SigAct News* (June 2002).

[18] Irvine Nyandowe and Shefiu Zakariyah. *Guide to Evolutionary Prototyping.* Oct. 2014. DOI: `10.13140/2.1.1473.4080`.

[19] Free Software Foundation. *GNU General Public License v3.* `https://www.gnu.org/licenses/gpl-3.0.en.html`. Visited on March 15, 2022.

[20] Open Source Initiative. *The MIT License.* `https://opensource.org/licenses/MIT`. Visited on April 15, 2022.

[21] World Wide Web Consortium. *Activity Streams 2.0.* `https://www.w3.org/TR/activitystreams-core/`. Visited on March 27, 2022.

[22] *Friendica – A Decentralized Social Network.* `https://friendi.ca/`. Visited on March 20, 2022.

[23] Gediminas Lelešius. *Friendica bug report: Incorrect ids of ActivityPub follower/following collection pages.* `https://github.com/friendica/friendica/issues/11427`. Visited on May 12, 2022.

[24] *Peertube – free and decentralized alternative to video platforms.* `https://joinpeertube.org/`. Visited on March 25, 2022.

[25] Gediminas Lelešius. *PeerTube bug report: Incorrect ActivityPub Follower/Following collection type.* `https://github.com/Chocobozzz/PeerTube/issues/4972`. Visited on May 12, 2022.

[26] *Write Freely – An open source platform for building a writing space on the web.* `https://writefreely.org/`. Visited on March 25, 2022.

[27] Junghoo Cho and Hector Garcia-Molina. "The Evolution of the Web and Implications for an Incremental Crawler". In: *VLDB*. 2000.

[28] Ross Anderson. *Lecture notes of Software and Security Engineering.* `https://www.cl.cam.ac.uk/teaching/1920/SWSecEng/sse.pdf`. 2020.

[29] *An overview of fediverse software.* `https://fedidb.org/software`. Visited on March 25, 2022.

[30] *Pleroma – social networking software.* `https://pleroma.social/`. Visited on March 25, 2022.

[31] *PixelFed – A free and ethical photo sharing platform.* `https://pixelfed.org/`. Visited on March 25, 2022.

[32] *Lemmy – A link aggregator for the fediverse.* `https://join-lemmy.org/`. Visited on March 25, 2022.

[33] *BirdsiteLive – a Twitter to ActivityPub bridge.* `https://beta.birdsite.live/`. Visited on March 25, 2022.

[34] *Juick – microblogging service.* `https://juick.com/`. Visited on March 25, 2022.

Appendix A

Mastodon modifications

A list of files in the Mastodon code[1] that I modified or added:

```
.env.production.sample
.env.vagrant
Gemfile
Gemfile.lock
app/controllers/concerns/signature_verification.rb
app/controllers/well_known/get_from_cache_controller.rb
app/helpers/jsonld_helper.rb
app/helpers/lookup_helper.rb
app/lib/activitypub/activity/create.rb
app/lib/activitypub/linked_data_signature.rb
app/models/status.rb
app/services/activitypub/fetch_remote_account_service.rb
app/services/activitypub/fetch_remote_key_service.rb
app/services/fetch_remote_status_service.rb
app/services/fetch_resource_service.rb
app/services/resolve_account_service.rb
app/workers/thread_resolve_worker.rb
config/routes.rb
db/schema.rb
```

[1]The original can be found at `https://github.com/mastodon/mastodon`.

Appendix B

Data gathering and processing for the Evaluation

Here I describe techniques used to gather and process the data presented in the Evaluation chapter.

B.1 Lookup server

The lookup server was hosted in the cloud, on a 4 cores 24Gb Linux machine. Note that the lookup server is not multi-threaded, only cryptographic operations are executed on a separate thread.

The data for graphs was gathered by using event counter embedded in the code. Its values were saved to the database every 10 seconds. Before creating a graph, non-cumulative values were smoothed by taking a moving average over 20 entries.

Graphs were plotted using `matplotlib`. All code is included in the source repository, under `code/tools` directory.

B.2 Verifier

The verifier was hosted on my personal laptop.

Again, signing rate data was gathered using event counters in the code. It was smoothed by taking a moving average over 20 entries. RAM usage was recorded using `psutil` library's `Process.memory_info()` method, in particular Resident Set Size – the amount of physical memory allocated to the process. To ensure that there are not any memory leaks to virtual memory (swap), I used `tracemalloc` that intercepts `malloc` calls and tracks the exact amount of memory allocated by Python. It did not show any increasing trend. However, it significantly affected the performance of the program, so I did not include the graphs in the Evaluation.

Appendix C

Project Proposal

Part II Project Proposal:
Improving Resilience of ActivityPub Services

Gediminas Lelešius

October 17, 2021 (modified before releasing publicly)

C.1 Introduction

A current centralized approach to social networking, when the power and responsibility are concentrated in the hands of one (or a few) providers, results in many problems:

- censorship: a few lines of code can censor billions of people
- privacy: can you trust businesses to keep your confidential data; can you even know what data they are collecting and storing about you?
- incompatible standards: how many accounts in different social networks do you have?
- single point of failure (SPoF): undoubtedly, these companies offer the most reliable and resilient services, however, incidents are inevitable[1]

An ActivityPub[2] is a protocol that aims to tackle these issues: it describes a simple yet powerful interface for building decentralized federated social networks.

The decentralized and open-source approach means that anyone can run their own server or join any instance from the fediverse: censorship becomes limited and users only need to trust their instance owner (or create their own instance) for privacy considerations. In addition, all servers (even different implementations or services!) can communicate together: even though instances can unite people with similar interests (similar as forums used to), you can still follow and interact with content from other instances.

Even the SPoF issue is resolved up to a point: if one (or multiple) of the instances fails, users of other instances can still use the network. However, they aren't completely unaffected: posts on

[1]Recently, the most popular social network Facebook suffered an outage impacting billions of users including myself. (`https://engineering.fb.com/2021/10/04/networking-traffic/outage/`)

[2]`https://www.w3.org/TR/activitypub/`

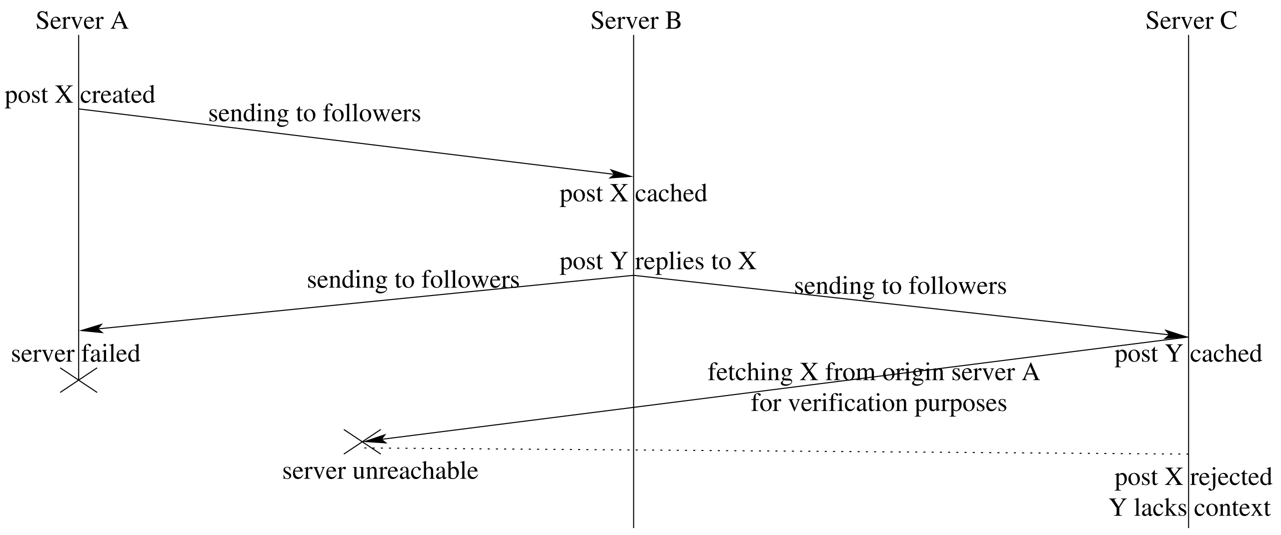


Figure C.1: *an example of sharing a post from the failed server. Time goes from top to bottom.*

other instances might contain references to posts or actors (users of the ActivityPub network) from the instance that's down. In the context of one instance, it's trivial to fix: just store each referenced post and/or actor locally.

But what if someone from server B replies to a post X from the failed server A (see Figure C.1)? That post X might be propagated further, to some instance C that hasn't ever heard about the original post. It will try to query X from the now unavailable server A and will reject our reply (or won't consider it as a reply). We (presumably) have a copy of that post, so we can incorporate it into a reply Y. But can other servers confirm that we haven't tampered with the original post? We can use the same mechanism as many ActivityPub services use for server-to-server communication: append the signature. But to get the public key of the original post author, the receiver has to query the unavailable server.

The goal of this project is to implement a solution to this problem: a public lookup server of actor public keys (a key server) that can be used to verify the integrity of cached objects in case the origin server fails.

To protect from malicious certificate servers, I will implement workers that can connect to any such server and verify its history. That way, key servers can demonstrate some level of trustworthiness.

Finally, I will create a proxy (or modify service code) that caches signatures of the incoming objects and intercepts requests to other servers. In case an outgoing certificate request fails due to the origin server being unreachable, the proxy should fall back to one of the key servers.

The ActivityPub protocol does not define how exactly authentication is done, so I will build my code for a specific service – Mastodon[3]. It's one of the most popular micro-blogging platforms in the ActivityPub federation. Multiple other ActivityPub services use a similar method for authentication.

C.2 Substance and Structure

The project can be divided into these logical parts:

[3]`https://github.com/mastodon/mastodon`

1. research into how ActivityPub and Mastodon work,
2. implementation of the key server,
3. implementation of the verifier,
4. modification of Mastodon/proxy creation,
5. evaluation,
6. dissertation writing.

Possible extensions are listed at the end of this section, they will be completed if time permits.

First, I will start by researching how ActivityPub works. Because it's an open-source protocol, proper documentation is available online, which I've already read. There is an issue, though: ActivityPub does not describe how servers should authenticate activities they receive. Therefore, I've chosen a concrete implementation of the ActivityPub server-to-server protocol – Mastodon. It looks like PeerTube[4] and FunkWhale[5] use a similar authentication mechanism. I will read the Mastodon documentation related to Security and Authentication and will probably need to inspect the code to understand the exact mechanism.

Second, I will implement the key lookup server. There are two ways to collect certificates: crawling ActivityPub fediverse or subscribing to servers (following instances) and looking through their activity feed. I will implement at least one of those. Then, I will create a web API for querying collected data.

Third, I will implement the entry verifier. It should fetch entries from the key server (or multiple servers), verify that they are correct by querying from origin servers and sign them.

Fourth, I will create a proxy and/or modify Mastodon to use my key server as a fallback. In other words, if some request to get a user certificate from the origin server fails, the proxy (or Mastodon) should try to fetch that certificate from the key server.

Then, I will do a range of synthetic tests to measure the performance of my system and ensure that it can be beneficial in real-world applications:

- key lookup server: insertion throughput (requests per second) and its dependence on database size
- key lookup server: latency (how long it takes for the key to be recorded since it becomes available in ActivityPub network) and how the number of followed instances affects it
- key lookup server web API: throughput of lookups (requests per second) and its dependence on database size
- verifier server: verification throughput (keys per second)
- proxy/Mastodon: throughput (requests per second) and request time

In addition, I will implement a simple simulator of the ActivityPub services and network. Running it with and without proxy will allow me to quantify how effective the system is: the percentage of posts that are "rescued". In other words, it would estimate the number of fallbacks to a proxy and what fraction of them the proxy was able to serve. By modifying simulation parameters, I will determine how the effectiveness depends on various properties of the network, for example, new users per second, the average number of direct connections per node, the failure probability of the node and similar.

Besides, I will host my system and capture traffic from existing ActivityPub instances. I will run the verifier to ensure that entries recorded by the key server are correct and the entire

[4]`https://github.com/Chocobozzz/PeerTube`

[5]`https://dev.funkwhale.audio/funkwhale/funkwhale/`

system works together. Next, I will simulate network partition for my ActivityPub instance and check that it works as expected: my instance can still receive posts from servers it can't reach. Keeping the system running for an extended period will act as a long smoke test – the absence of errors will mean that the system is likely correct. Whether success criteria is reached will be determined by functional testing.

Finally, I will write a dissertation about the project.

In case, something unexpected happens and I won't have enough time to implement everything described above, I might not implement the simulator as other parts of the system doesn't depend on it. If that won't be enough, I might as well skip the verifier.

Possible extensions:

- support rotation of public keys (some services might regularly issue new certificates for their actors in case old ones get stolen or cracked);
- implement both, crawler and follower, approaches for the key server;
- consider how object deletions are influenced: someone could forward signed object create activity after delete activity was issued;
- support distributed key server;
- test and implement extensions if needed so that the key server works for other ActivityPub services too, for example, PeerTube[6];
- modify Mastodon so that it stores encrypted private keys of actors and posts are signed at the user's end;
- store all crawled data, not only certificates: might be useful for tracing ActivityPub networks, monitoring for illegal, sensitive or copyrighted content, also could be served to other instances if origin server fails;
- add a personal mode to the verifier: one that only verifies that lookup servers store correct public keys for your account(s). It shouldn't be a source of trust for others. Instead, it should act as an early indicator for you that something suspicious is happening. An additional feature could be storing a backup and unified view of all owner's activities across multiple ActivityPub services.

C.3 Success Criteria

The first success criterion is correctness (both, in simulation and real ActivityPub network): the proxy works in a simulated environment and with a real Mastodon server; the lookup server stores and returns correct public keys – from simulation/as they are in origin servers; the verifier only signs logs that are consistent with the simulation/origin server responses.

Next, programs should have intended functionality: the key server should crawl or follow existing Mastodon instances; the verifier should watch key servers; the Mastodon server should be able to fall back to a key lookup server in case the origin server is unreachable.

[6] `https://joinpeertube.org/`