universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Exploring ActivityPub Interoperability for Bluesky“

verfasst von / submitted by

Martin Sonnberger BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2026 / Vienna, 2026

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 935
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Math. Dr. Peter Reichl, Privatdoz.

Acknowledgements

Thank you very much!!

Abstract

This LaTeX template provides example on how to format and display text, mathematical formulas, and insert tables or images. There is a lot more you can do with LaTeX, for more information check out https://en.wikibooks.org/wiki/LaTeX.

Kurzfassung

Das ist eine deutsche Kurzfassung meiner in Englisch verfassten Masterarbeit.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

1. Introduction

2. Background and Related Work

2.1. AT Protocol

2.2. ActivityPub

2.3. Related Work

2.3.1. Bridgy Fed

2.3.2. Wafrn

3. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS

Fedisky is a sidecar service that a Bluesky PDS operator can deploy alongside their PDS to enable federation with the Fediverse. It acts as a bridge between the AT Protocol and ActivityPub, allowing users on Mastodon instances to discover, follow, and interact with users on the Bluesky PDS. Fedisky is designed to be a lightweight and modular service that can be easily deployed and maintained by PDS operators, without requiring any modifications to the PDS itself. In this chapter, we will provide an overview of the design and implementation of Fedisky, including its architecture, data model, key components, and operational considerations. Fedisky's source code is available on GitHub at `https://github.com/msonnb/fedisky`.

3.1. Overview

3.1.1. System Context

Figure 3.1 shows the wider system context of Fedisky and how it interacts with other systems and users. At the core of the system is the PDS host, which runs both the ATProto PDS and the Fedisky sidecar and is managed by the PDS operator. Fedisky uses ATProto XRPC APIs to read and write records from the PDS, and subscribe to its firehose endpoint to receive a real-time stream of all new and updated records.

Fedisky exposes ActivityPub endpoints, which are used by external Fediverse instances to deliver ActivityPub content to their users and to receive incoming activities from the Fediverse. These endpoints include a Webfinger endpoint for user discovery, an actor endpoint for representing Bluesky users as ActivityPub actors, and an inbox endpoint for receiving incoming activities from the Fediverse.

To fetch Bluesky records from users on other PDS instances, Fedisky fetches records from the Bluesky AppView. In addition, Fedisky periodically polls the ATProto Constellation API[1], an external service that aggregates and indexes backlinks across the entire AT Protocol, allowing Fedisky to discover interactions such as replies from users on other PDS instances, and federating them to the Fediverse.

[1] `https://constellation.microcosm.blue/`

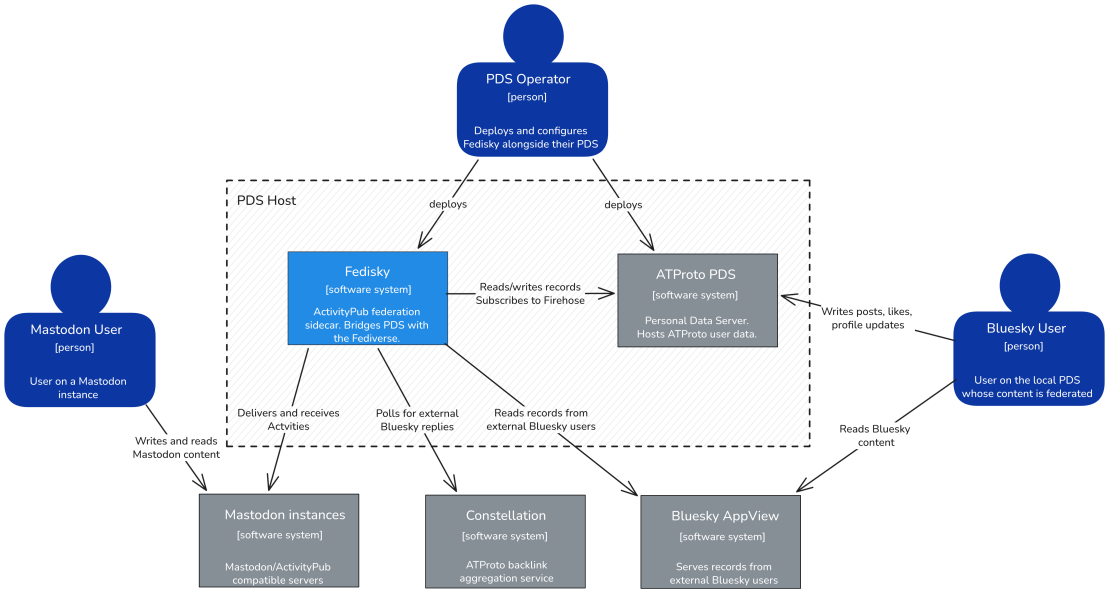

Figure 3.1.: System Context Diagram showing how the Fedisky sidecar interacts with other systems and users.

3.1.2. Technology Stack

Fedisky is implemented in TypeScript and runs on Node.js. It stores its data in a SQLite[2] database using Kysely[3] as a type-safe query builder. For federation functionality, Fedisky uses Fedify[4], a TypeScript library for building ActivityPub servers.

Fedify provides a high-level API for defining ActivityPub actors, registering dispatchers for handling incoming activities, and sending outgoing activities to other Fediverse instances. It also handles the underlying ActivityPub protocol primitives such as signing and verifying HTTP requests, providing type-safe objects for Activity Vocabulary [5] such as `Create`, `Follow`, and `Note`. In addition, Fedify includes scalability and reliability features such as retry logic for failed deliveries, a message queue for processing incoming and outgoing activities, and a KV-store for caching and storing federation-related data such as public keys and remote actor information.

[2] `https://sqlite.org/`
[3] `https://kysely.dev/`
[4] `https://fedify.dev/`
[5] `https://www.w3.org/TR/activitystreams-vocabulary/`

3.1.3. High-Level Data Flow

3.1.4. Component Overview

3.1.5. Dependency Injection

3.2. Data Model

3.2.1. Database Schema

Identity & Cryptography

- `ap_key_pair` – Stores the cryptographic key pairs used for HTTP signature signing. Each local PDS user gets two key pairs generated on first access: one RSA for compatibility with older ActivityPub implementations, and one Ed25519 for modern servers. The keys are stored as PEM-encoded strings.

- `ap_bridge_account` and `ap_bluesky_bridge_account` – Singleton tables that store the credentials for the two bridge accounts (see section 3.3).

Social Graph

- `ap_follow` – Records which ActivityPub actors follow which local PDS users. This is the core table for activity delivery, as it determines which users should receive which activities based on their follow relationships. It stores the follower's inbox URL and shared inbox URL for efficient delivery.

Content Mapping

- `ap_post_mapping` – Maps ATProto post URIs to their original ActivityPub Note IDs and author information. This table is essential for correct reply threading.

- `ap_external_reply` – Stores external Bluesky replies from other PDS instances that have been federated via the Constellation processor. Primarily for deduplication as the Constellation API is polled repeatedly.

- `ap_monitored_post` – The work queue for the Constellation processor, which tracks which posts need to be checked for new external replies.

Engagement Tracking

- `ap_like` and `ap_repost` – Stores incoming ActivityPub `Like` and `Announce` activities targeting local posts. The tables main purpose are to track engagements for display (e.g. showing like counts) and batching engagement for DM notifications.

3.2.2. Migration Strategy

Migrations are implemented using Kysely's built-in migration system, which allows us to define schema changes in a type-safe manner. Each migration is defined as a numbered TypeScript file exporting `up()` and `down()` functions. On service startup, pending migrations are automatically applied in order. This approach allows us to evolve the database schema over time while ensuring data integrity and providing a clear history of schema changes.

3.3. Bridge Account System

3.3.1. Why Two Bridge Accounts?

Fedisky uses two special "bridge accounts" to facilitate bridging between the AT Protocol and ActivityPub. The first is the Mastodon bridge account, which is hidden from users in ActivityPub and is used to post incoming Fediverse replies as ATProto posts on the PDS, so that they appear in the Bluesky user's thread and can be replied to and interacted with like normal posts. Its handle and display name can be configured by the operator, with the default being `mastodon.{hostname}` and "Mastodon Bridge" respectively.

The second one is the Bluesky bridge account, which is used to federate replies from Bluesky users on other PDS instances to the Fediverse. This allows users on Mastodon to see and interact with replies from users on other PDS instances, which would otherwise be invisible to the Fediverse. The Bluesky bridge account's default handle is `bluesky.{hostname}` and display name is "Bluesky Bridge", again configurable by the operator via environment variables.

4. Results

5. Discussion

5.1. Sidecar Architecture

As described in the previous chapter, Fedisky is designed as a *sidecar container* as defined in [1, Ch. 3] and runs alongside its *application container*, a Bluesky PDS. Sidecars are a common architectural pattern, where a secondary container provides auxiliary functionality to the primary application container, often without the application knowing. Sidecar containers share system resources and are scheduled to run in sync with the application container [1, p. 21]. This design allows Fedisky to operate independently while still closely integrating with the PDS.

An alternative design could have been a centralized bridging service that bridges *all* of Bluesky into the Fediverse, which is realized by Bridgy Fed (subsection 2.3.1). Compared to a centralized bridging service, a sidecar container running alongside a single Bluesky PDS instance, can share resources with the PDS, most importantly the server's hostname, which allows it to operate under the same domain and thus provide a single shared identity for users across Bluesky and Mastodon, e.g. `alice.fedisky.social` in Bluesky becomes `@alice@fedisky.social` in Mastodon. Additionally, the sidecar pattern allows for operator autonomy, where each PDS operator can choose to deploy Fedisky independently, or not. It also reduces the scope of the bridge, as it only needs to handle traffic for a single PDS instance, simplifying development and maintenance.

Another approach would be to fork and modify the PDS itself, thus integrating bridging functionality directly into the PDS. We considered this approach initially when designing Fedisky, but chose the sidecar pattern instead for several reasons. Forking the PDS would significantly increase the maintenance burden and development complexity, as one would need to keep up with upstream changes constantly to ensure compatibility. The sidecar avoids this by only consuming the PDS's public APIs, which are stable contracts based on AT Protocol Lexicons [2]. In addition, the sidecar pattern allows for better separation of concerns, as all bridging logic is contained within the sidecar, rather than the PDS being responsible for both AT Protocol and ActivityPub compliance and therefore tightly coupling two protocol implementations, making each harder to reason about and test in isolation. Furthermore, the sidecar gives more deployment flexibility, as operators can choose to start or stop the bridge independently of the PDS. Since communication between the sidecar and the PDS is done through HTTP APIs, the sidecar can be developed a different technology stack, and is automatically compatible with any PDS implementation that adheres to the AT Protocol specifications.

6. Conclusion

Bibliography

[1] Brendan Burns. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services, Using Kubernetes*. Second Edition. Sebastopol: O'Reilly, 2024. 200 pp. ISBN: 978-1-0981-5635-0.

[2] *HTTP Reference | Bluesky*. URL: `https://docs.bsky.app/docs/category/http-reference` (visited on 02/12/2026).

A. Appendix

here you can put further things you want to add like transcripts, questionnaires, raw data...