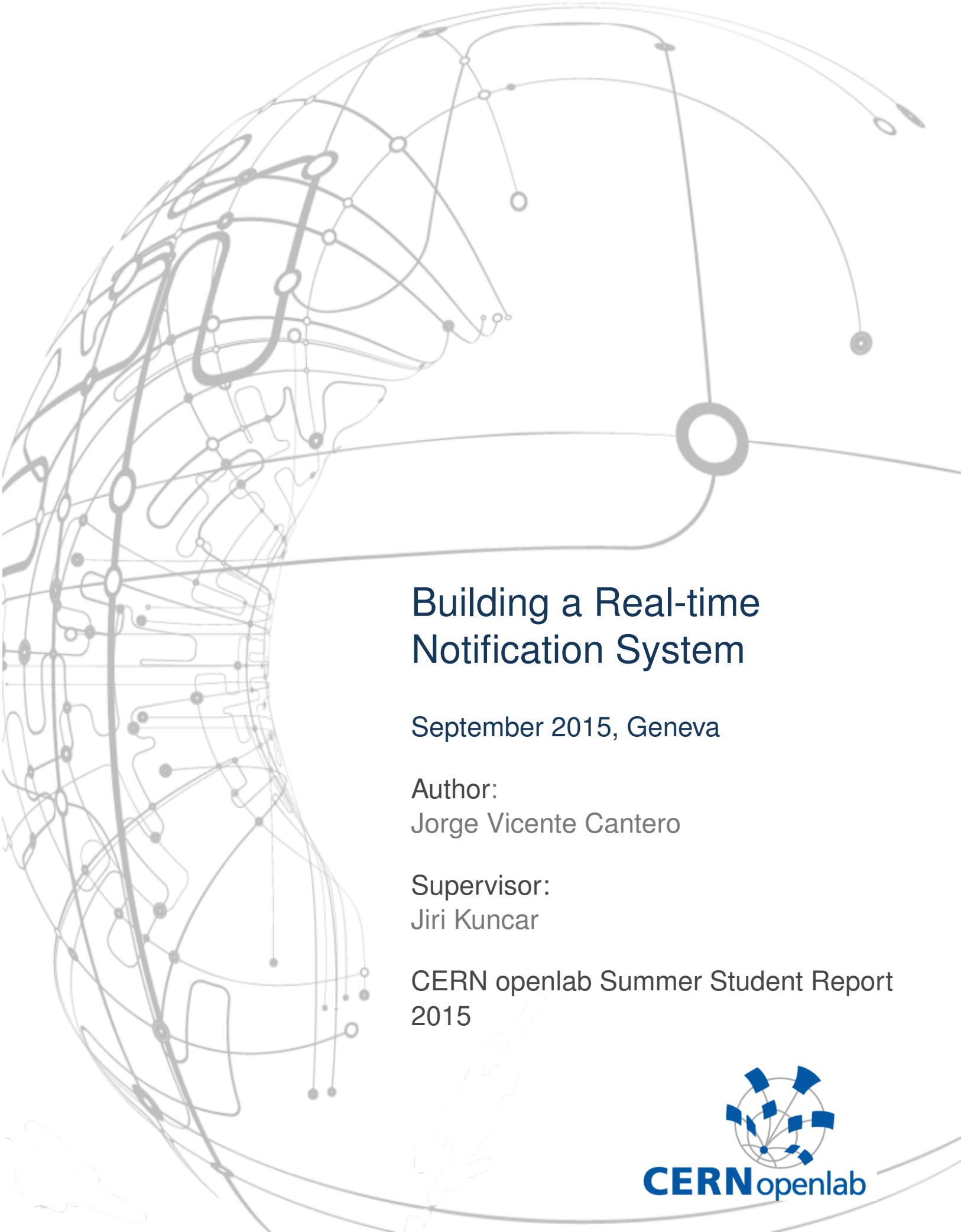

Building a Real-time Notification System

September 2015, Geneva

Author:
Jorge Vicente Cantero

Supervisor:
Jiri Kuncar

CERN openlab Summer Student Report 2015

CERN openlab

Project Specification

Configurable Notification System for Invenio Digital Library

Invenio is a free digital library framework enabling to set up document, multimedia, or data repositories on the web. It is being developed in an international collaboration comprising institutes such as *CERN*, *DESY*, *EPFL*, *FNAL* or *SLAC*.

The aim of this project is to develop configurable real-time notification system that will be used by various digital library modules in order to facilitate user interaction with the system. The student will participate in the designed implementation of plugin-based email, SMS and push notification services for various system services, such as document submission and approval, personal user query alerts, or general user messages.

The student will use `Flask`/`Jinja`/`jQuery`/`Bootstrap` technologies for the web user interface and `Python`/`SQLAlchemy`/ `Redis` technologies for the backend.

Jiri Kuncar (IT-CIS-DLT)

Abstract

A notification system is required by Invenio to fulfill the current needs and adapt to the future use cases. Such a system has to be real-time and compatible with its technology stack, notifying the Invenio users almost instantly.

In the following sections, the design of this system is described, discussing both the previous requirements and the achieved results. Also, more insight about the used key technologies is given, justifying why they were necessary for the project. The details for building such a system with Python are explained from a theoretical point of view and the necessary concepts for using it are introduced.

Lastly, the future of the project is discussed, giving a general overview of it and room for improvements.

Table of Contents

1 Introduction

Invenio Digital Library is currently supporting email delivery as the standard way of notifying users in concrete scenarios of the platform. However, a more fine-grained notification system is needed in order to allow any Invenio module to send any type of notification to any existing device. By creating such a system, more complex features like alerts matching certain queries or warning about the availability of a book could be carried out fast and easily.

The aim of this project is to develop a flexible real-time notification system that sends these notifications to the Invenio users. A real-time system is a system subject to a time constraint, performing an action within a period of time and as soon as possible. These systems are very popular nowadays because they meet the expectation of the users, who want immediate feedback when interacting with computers.

2 The system

Building a real-time notification system is not an easy task. There are a lot of details going on that should be looked at carefully. These details could break essential properties, making the system useless. It is important to know which requirements are necessary and which are not, keeping a clean design from the beginning.

One important remark is that the system to build is not a hard real-time system but a soft real-time system. This means that under a high workload, the system could process with delay some notifications, degrading the quality of service. From now on, real-time will mean soft real-time. Although it is possible to use parameters of the broker to guarantee that a notification is processed before a deadline with the *ETA* parameter in Celery, this option has not been experimented and it is out of the scope of the project.

Research show that a real-time system should have the following properties:

- **Asynchrony.** The system allows to process tasks by using independent units without any blocking operation. When a task is available, these independent units (usually workers running on several threads) get that task and run it out of the main program.
- **High availability.** The system needs to be always up and running to decrease the possibility of data loss. Otherwise, events could be unprocessed by the system, triggering the mistrust of the clients.
- **Fault tolerance.** A system is fault-tolerant when is able to recover from errors without any state or data loss. Typically, this is a property of the broker.

  > *The broker is an intermediary program that keeps all the submitted tasks and hands them out to the workers.*

  There are several available brokers. The common ones are *RabbitMQ* and *Redis*. Both have the possibility to tune the fault tolerance of the system.

- **Scalability.** The capability to get more resources when the workload is too high and to free unused resources when the worload is too low. Hence, the efficiency of these systems grows linearly with the resources. Such systems can grow in two different ways:
  - **Scale up**, by adding more cores or memory to our servers.
  - **Scale down**, by adding more distributed nodes to our system in different locations.

Although such aforementioned properties are the essential ones, the framework is not only a real-time system. Other Invenio requirements need to be met, ensuring the usability of the system for the future use cases.

- **Broker-agnostic.** Production systems using Invenio as the **CERN Document Server (CDS)** or **Zenodo** are using *RabbitMQ*. Although working well, the future is not certain and other brokers could be used. Preparing for this kind of events, the system needs to work with other brokers as well.
- **Consumer-agnostic.** The notifications are useless if they cannot be sent to any existing system or use any current technology. Invenio does not want only to send emails, but to send those events to other systems. For instance, they would like to log notifications using Logstash, persist them in a database or to make a POST request to a Webhook. The system needs to allow to plug in any consumer, from a custom user function to a more complex event processing.
- **Full device compatibility.** The system needs to be compatible with all the modern technologies without taking into account which operating system is running the user or which browser is used. For instance, Android uses a special system named Android Push Notification, while iOS uses APNs (Apple Push Notification System).
- **Flexibility and extensibility.** As the Invenio use cases could change, the framework needs to be easy and straightforward to modify. Also, the API needs to be simple but powerful.

2.1 The Invenio Environment

The notification framework is a standalone project and does not depend on other Invenio modules. However, compatibility with the Invenio technology stack is key. New technologies have been used to show the power of the system, albeit the system has been designed mostly to be easily pluggable to the existing code. Invenio infrastructure is Python-based. Python is used in both frontend and backend. In the frontend it is used mainly *Flask* and *Jinja*. That's why this system is a *Flask* extension which follows the *Flask Extension Development* guidelines.

> *Flask is a Python web microframework which allows to build fast and powerful websites in an easy way.*

2.2 Previous Research

A previous research was made in the first week of the project. In this research, a study was conducted to get to know the already aforementioned features of the system. After the theoretical analysis of the system, more practical questions and implementation doubts came up. In this section, the technologies that solve the following points will be explained:

- Construction of an asynchronous, scalable system in Python
- Persistence of notifications
- Protocols to push notifications
- Notification delivery to several clients

In this research, **Redis** was chosen as a broker and **Celery** as a job processing framework. In order to support a modern way of sending notifications to the browsers, the recent and efficient **Server-Sent Events** (SSE) technology was chosen.

2.2.1 Celery as an Asynchronous Task Queue

Asynchronous task queues allow us to execute asynchronous tasks and scale both horizontally and vertically. Celery is an asynchronous task queue that abstract over the broker (the responsible for receiving and executing the tasks), allowing the user to choose between different ones like RabbitMQ, Redis, etc.

2.2.2 Redis as a broker

Redis is an efficient data structure server. Although its popularity is due to the fact that it is a key-value store, it has broker capabilities as well. One of the reasons it has been chosen as the default broker for the system is because it has a very straightforward to use Publish/Subscribe interface in the Python library (`Py-Redis`). Also, in most of the current performance comparisons, is as efficient as RabbitMQ.

2.2.3 Server-Sent Events as default push notifications

Server-Sent Events (SSE) is a technology that allows the server to send updates to a client using an HTTP connection, removing the overhead of bidirectional communication present in alternatives like Sockets. SSE works keeping the HTTP client-server connection opened without almost any overhead, contrary to other mechanisms like HTTP polling.

There are other popular technologies with more support like WebSockets (particularly, *Socket.io*). Although this one offers full compatibility with browsers, such technology is not efficient for the targeted use cases. Hence, sockets have been dismissed and SSE is included in the framework as the default option.

In case Invenio needs compatibility with older browsers, they could fallback to sockets or HTTP polling (in fact, this is what actually Socket.io does). This behaviour is not implemented by default because it depends mostly in the application which will use the framework.

3 Results

The result of the project is a ready-to-use Python module named `Flask-Notifications`, available in the Invenio Github **here**. This module is installable via pip (`pip install Flask-Notifications`) and can be used by any *Flask* user to create its own notification system.

3.1 Basic design

`Flask-Notifications` is based on four concepts: events, hubs, filters and consumers. The client only wants to send a notification. This notifications is modeled as an event. An event have default fields that represent the bases of an event like the type, the id, the sender, the receivers, etc. Also, the event is built in a way that allows any client of the framework to add more attributes, letting the business model to be present in the event representation. This is very handy, specially when a user is building the architecture.

After declaring an event, the event need to be sent. So far, there is no way to aggregate them, neither to explicitly tell the system how it should be consumed. To solve this, one use hubs. A hub is a unit which aggregates a certain type of events that meet some conditions. It is composed of a filter and a list of consumers. Hence, this filter is applied to all the sent events. If the events pass that filter, then they are consumed by the consumers. A consumer is a function which can be registered in a hub. Several hubs can register the same consumer. Normally, they consume the event by carrying out an action in the system. A consumer is a class with two hooks: one before consuming and the other one after.

`Flask-Notifications` have already some predefined consumers. These consumers are meant to be used as an inspiration or for testing, as they carry out very simple actions, like pushing to a browser with the `PushConsumer` sending an email with both `FlaskMail` and `FlaskEmail` or logging to a file with the `LogConsumer`. Also, one can reuse them by extending the consumers and overwriting the event format, i.e. in an email or file.

Also, Invenio may want to check the efficiency or the proper behavior of the system when it is already running. By adding extra consumers, we could further extend what we can do with our system, easing the process of checking and debugging. Specifically, Invenio uses *Logstash* (built in top of ElasticSearch). By propagating the events (notifications) of our system and feeding external services, we could have a smart handling of the logs, centralizing everything in an independent system.

3.2 Thinking further

The goal is not only to develop a basic notification system but to provide a simple API so it's easy to use in the application. Therefore, it is mandatory to think over a way to control the notifications once delivered. That's why the expiration of the notifications is so important: we can allow the client to control the visibility of the notifications when using certain consumers.

This makes sense with the push notifications, which are sent at the time the user connects to the system. Nevertheless, the same does not hold in the case of the emails, since they are immediately sent and they can not be deleted. In those cases, a message will warn the user that the received notification only is valid until some date. Otherwise, the delivery could be delayed for 10 minutes.

The expiration of the notifications are not only a logical abstraction, they have an influence in the broker. The system is designed to reject to consume any event which has expired at the time it is being processed.

3.3 Complexity of delivery

Sometimes happen that events are propagated and the receivers are not logged in the system. With the current system, this is an issue as the notifications would get lost and they would never arrive to the user. In order to avoid such a problem in the future, there is a clean technique which fixes this problem.

The solution is to persist the events in a database, allowing to resend them in the future when the user is logged. By default, the system does not care about the delivery of not real-time notifications. One can implement this behavior by overriding any one of the hooks of a consumer (as specified before). Therefore, when a system is processed by the workers, it would be sent to a storage system and indexed correctly for efficiency, depending on the details of the use cases.

3.4 Asynchronous Server

The nature of `Flask` is synchronous and blocking. In the time it was designed, almost no asynchronous processing was done in the servers. Nowadays, asynchronous servers are a must in any organization or company.

One way of converting Flask into an asynchronous server is by using `uWSGI` or any similar option (like `Tornado`). This framework uses under the hood `Gevent`, which is a framework for asynchronous processing using coroutines. In case Flask is not combined with any of these frameworks, the system will not be asynchronous and the push notifications will be pushed in a blocking way. This is due to the operations interacting with the broker. Specifically, there is `Backend` class which abstracts over the *Publish/Subscribe* pattern. This class has three methods: `publish`, `subscribe` and `listen`. Only the last one is blocking, as the system

needs to be checking when a notification has been pushed and therefore redirects it to the user.

This architecture is IO-dependent and **the client using the `Flask-Notifications` module should wrap Flask with an asynchronous server if he/she wants the benefits of a real-time system**. Otherwise, it couldn't be considered so.

3.5 Architecture

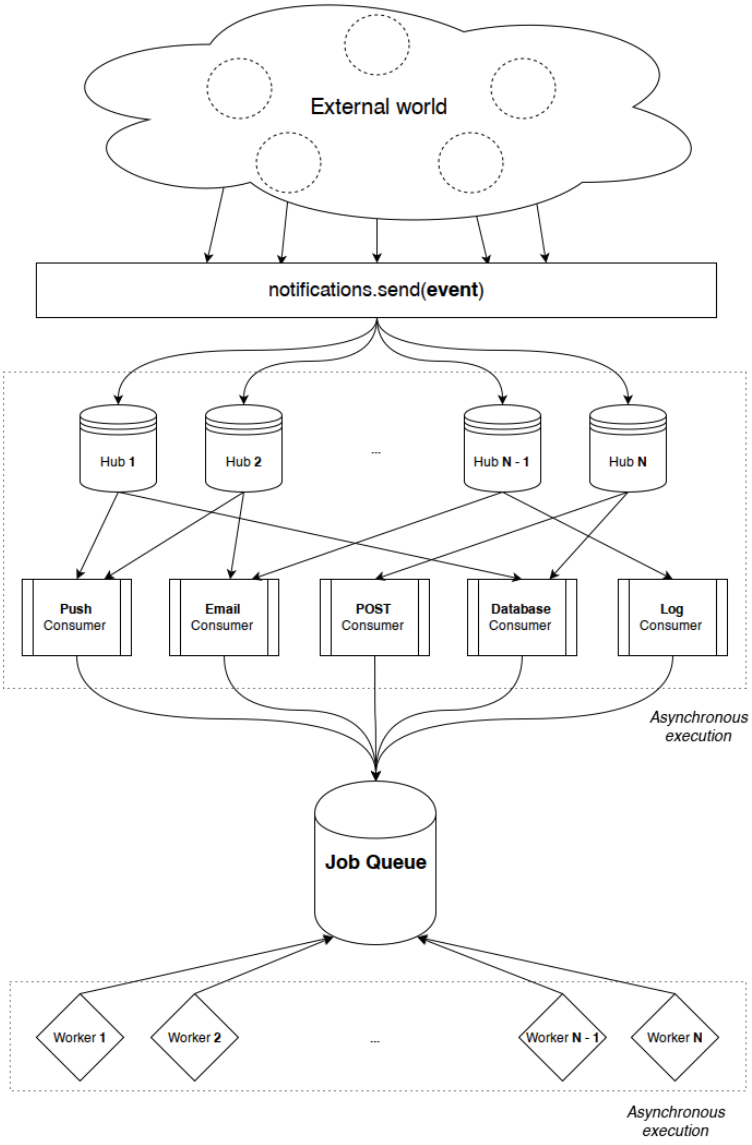

Figure 3.1: The system and its basic components

4 Future

`Flask-Notifications` is a basic module. Although most of the requirements have been met, some of them are not included by default in the system and they depend on the particular implementation of the user. Some lower-priority requirements of the system haven't been met because there was no time enough.

More work needs to be done in the following areas:

- Full compatibility with any device. Including implementation of *Apple* and *Android* Push Notifications and fallback to *Sockets*.
- To tune the broker to be more fault-tolerant and persist all the changes.

It is planned to use this module as a web service. This way, the current Invenio system only should learn how to call a simple API in `Flask-Notifications` and both systems could be totally decoupled. This scheme allows tuning system-wise for good performance in each system.

5 Conclusions

Building real-time systems is easy with Python, although sometimes it is complicated to get them right. Despite they don't have the efficiency of other systems with a different technology stack, they are very maintainable, flexible, and straightforward to build. When these properties are more important than the maximum throughput, as in the case of Invenio, Python is the perfect choice.

A successful notification system, along with its design and implementation, has been devised. It has basic set of features but it is easily extensible and open-sourced **on Github under the `inveniosoftware` organization**. `Flask-Notifications` is ready to use for any Flask user who wants to it to any Flask application.