Universitat Autònoma de Barcelona

MASTER'S THESIS

MASTER IN TELECOMMUNICATION ENGINEERING

Design and Implementation of a Positioning Service in the Context of Smart Cities

Alejandro Pérez Conesa

THESIS ADVISOR: Jose A. López Salcedo

DEPARTMENT OF TELECOMMUNICATIONS AND SYSTEMS ENGINEERING

ESCOLA D'ENGINYERIA (EE)

UNIVERSITAT AUTÒNOMA DE BARCELONA

Bellaterra, February 2020

El sotasignant, José A. López Salcedo, Professor de l'Escola d'Enginyeria (EE) de la Universitat Autònoma de Barcelona (UAB),

CERTIFICA:

Que el projecte presentat en aquesta memòria de Treball Final de Master ha estat realitzat sota la seva direcció per l'alumne *Alejandro Pérez Conesa*.

I, perquè consti a tots els efectes, signa el present certificat.

Bellaterra, 2, gener 2022

Signatura: *José A. López Salcedo*

Resum: *en les últimes dècades, les ciutats s'han convertit en els nuclis mundials de comerç, cultura, ciència i societat, sent també les majors consumidores d'energia i les més grans emissores de carboni. Amb l'objectiu de solucionar aquesta problemàtica, les ciutats sostenibles o "SmartCities" són un dels objectius a complir en l'Agenda 2030. Amb aquest objectiu en ment i en el context del projecte "Navigation and GNSS inSmart Cities – Testbed Concept Definition"(HANSEL), l'estudiant pretén dissenyar i desenvolupar un servei a càrrec del posicionament de sensors basats en tecnologies GNSS i cel·lular per al posterior tractament de la informació generada per a diverses finalitats, com la detecció i localització de fonts d'interferència o la hibridació GNSS i cel·lular, donant lloc a posicions híbrides, més precises que les de cada sistema per separat. Aquest servei pretén ser accessible mitjançant Internet al públic general (com un Software com a servei o SaaS), i aprofita els avantatges que la computació en el núvol és capaç d'oferir tant a nivell de prestacions com a nivell d'estalvi d'energia respecte als dispositius de navegació actuals.*

Resumen: *en las últimas décadas, las ciudades se han convertido en los núcleos mundiales de comercio, cultura, ciencia y sociedad, siendo también las mayores consumidoras de energía y las más grandes emisoras de carbono. Con el objetivo de solucionar esta problemática, las ciudades sostenibles o "Smart Cities" son uno de los objetivos a cumplir en la Agenda 2030. Con este objetivo en mente y en el contexto del proyecto "Navigation and GNSS in Smart Cities – Testbed Concept Definition" (HANSEL), el estudiante pretende diseñar y desarrollar un servicio a cargo del posicionamiento de sensores basados en tecnologías GNSS y celular para el posterior tratamiento de la información generada para diversos fines, como la detección y localización de fuentes de interferencia o la hibridación GNSS y celular, dando lugar a posiciones híbridas, más precisas que las de cada sistema por separado. Dicho servicio pretende ser accesible mediante Internet al público general (como un Software como servicio o SaaS), y aprovecha las ventajas que la computación en la nube es capaz de ofrecer tanto a nivel de prestaciones como a nivel de ahorro de energía con respecto a los dispositivos de navegación actuales.*

Summary: *in recent decades, cities have become the global hubs of commerce, culture, science and society, being also the largest consumers of energy and the largest carbon emitters. With the objective of solving this problem, sustainable cities or "Smart Cities" are one of the objectives to be fulfilled in the 2030 Agenda. With this objective in mind and in the context of the project "Navigation and GNSS in Smart Cities - Testbed Concept Definition" (HANSEL), the student intends to design and develop a service in charge of sensor positioning based on GNSS and Cellular technologies for the subsequent treatment of the information generated for various purposes, such as the detection and location of sources of interference or GNSS and Cellular hybridization, obtaining hybrid positions, more precise than those of each system separately. This system or service is intended to be accessible to the general public via Internet (as a Software as a Service or SaaS), and takes advantage of the all the features cloud computing has to offer, both at performance and energy consumption level.*

Acknowledgments

This Master's Thesis represents the end of my student career (for the moment). These almost seven years as university student (seven!) and three working for SPCOMNAV group have formed my character and have given me knowledge and opportunities I never thought would be within my reach, such as setting foot in the European Space Agency.

First of all, I would like to thank my boss, supervisor, professor and friend, José A. López Salcedo, for being my guide in the professional stage that I started in the group, for giving me advice and for granting me the opportunities that he has given me during these three years working in SPCOMNAV.

To my coworkers, those who were and those who are still there: Sergi Locubiche, Daniel Egea, José Antonio del Peral Rosado, Alda Xhafa, Ning Chang, Alessandro Pin. Special mention to Quim Gáñez, who has always been at the frontline when I needed him.

To my friends and master's classmates, Víctor Piñero, Saurabh Kulkarni, Borja Herranz, for the mutual help and support during the Master's years. Also to Ander Pardo and Pablo Garrido, Bachelor classmates, who have always been good friends.

To my lifelong friends, Albert Soldevila, Alexandru Oprea, for bearing my complaints about almost everything and helping me out when I needed it.

And finally, as the most important thing is left for last, to my family and specially my partner, Andrea Morante, for the overall support and the happiness you bring to my life. Thank you.

Alejandro Pérez Conesa

Bellaterra, February 2020

Table of contents

List of figures

List of tables

1 Introduction

In the past decades, cities worldwide have experienced a dramatic growth moving to what they have become nowadays: powerful hubs for commerce, culture, science and social. Presently, cities occupy just 3% of the Earth's land, while they account for 60-80% of the energy consumption worldwide and 75% of carbon emissions, according to the United Nations (UN) [1].

To redress such an unbalanced situation, the UN included sustainable cities as one of the goals to be achieved within the 2030 Agenda for Sustainable Development, in particular through goal #11 on "Sustainable cities and communities" [2]. A crystallization of such goal is the emergence of the so-called "*Smart Cities*": urban developments meant to improve the quality of life, increasing the efficiency of services and better meeting the residents' needs through the use of technology.

In this context, the European GNSS Agency (GSA) and the UN Office for Outer Space Affairs (UNOOSA) have both acknowledged that GNSS is expected to be a key enabler for the design of new city services, thus helping to achieve the UN Sustainable Development Goal on cities [3]. GNSS-driven features such as autonomous machinery, provision of ambient intelligence combining GNSS with IoT sensors, and positioning intelligence are naturally embraced by Smart Cities, which become fertile ground for technological breakthroughs in the years to come.

All the reasons stated above lead to the European Space Agency Invitation To Tender (ITT) AO/1-9494/18/NL/CRS [4], entitled Navigation and GNSS in Smart Cities – Testbed Concept Definition (codenamed HANSEL), which considers as baseline the availability of a network of distributed smart sensors and aims to design and develop a testbed to manage and efficiently exploit the data generated by this network.

This project, framed in a collaboration between the consortium formed by the SPCOMNAV group (Spain), Rokubun (Spain), Trafficnow (Spain), LINKS Foundation (Italy) and Politecnico di Torino (Italy) and the European Space Agency, started in February 2018, aims to cover all the aspects and requirements of this ITT, with the focus of the development of a navigation-based cloud Testbed in the context of Smart Cities.

1.1 Motivation

The main motivation of this work is to develop a service that efficiently, both in computing and energetic terms, generates and manages navigation information, taking a step forward to a future where every device will be connected thanks to what we know as Internet of Things or IoT and all generated data will be analyzed searching for patterns thanks to Big Data analytics. This work then is nothing but the prelude of all the data management technologies that are to come in the following years, when IoT, Big Data, Smart Cities and all the related terms that now are novelty become mundane and a staple in technology development.

It is important to note that the work presented in this thesis is just a fragment of the whole developed project. While the contributions from other partners will be mentioned and shown, it will be clarified whether the development belongs to the author or to a project associate.

1.2 Objective

The main objective of the whole HANSEL project is to develop a Testbed to demonstrate a series of services around navigation and localization that could be implemented in the context of Smart Cities. Such services are based on GNSS technologies as well as wireless communication signals of opportunity (3G/4G/5G, WiFi).

Specifically to the present work, the main objective of this Master's Thesis is to:

1. Design, develop and validate a positioning platform that conforms one of the Testbed services, covering the needs of GNSS positioning using snapshot-based cloud processing (2.2.1),
2. jammer detection and localization using GNSS snapshots (2.2.2),
3. positioning using cellular infrastructure and hybridized measurements (2.2.3), and
4. the need of data storage and intercommunication with the rest of the platform, both internal and external (2.2.4).

Due to the needs of the service, this thesis groups several engineering branches whose knowledge was acquired in different subjects during the Master's Degree. This knowledge, in addition to the acquired knowledge in SPCOMNAV group, conforms a solid knowledge basis that is able to be added as a valuable contribution to the main project. The main engineering and technology topics and its related subject this work covers are, mainly:

- System modeling, design, operation, administration and maintenance. Related to Communication Systems Design (CSD) and Telecommunication Projects (TP) subjects.
- Software architecture design.
- Network and Internet architectures. Related to Advanced Networks and Security (ANS) subject.
- Security mechanisms. Related to Advanced Networks and Security (ANS) subject.
- GNSS positioning and GNSS technologies. Related to SPCOMNAV group knowledge.
- Cellular positioning and Cellular technologies. Related to SPCOMNAV group knowledge.

1.3 Methodology

The methodology used to obtain results in the present work, mainly obtained by fulfilling the project objectives, has been the following:

1. Bibliographic review. The first step of the methodology consisted in an extensive review of all the provided information that came with the project, such as proposal, statement of work and specific objectives of the project.
2. Conceptual development. This phase is the one where the framework for the project was decided along with the conceptual design of the platform i.e. identification and grouping of the set of actions that the service has to perform.
3. Design development. In this phase, all the conceptual action sets took form as different modules and submodules, each one with a unique and atomized task in order to fulfill a specific requirement.
4. Software development. Each one of the designed modules and submodules are coded following the previously selected framework, carefully following the stated design.
5. Test phase. Each one of the software modules are exhaustively tested in order to correct the possible software errors, thus assuring the reliability of the platform.

1.4 Thesis outline

As for this thesis outline, Section 2 provides a general overview of the Testbed that is being to be developed, specifically the main functionalities (2.2), based on project objectives and how these functionalities are going to be implemented in terms of hardware (2.1) and software (2.3) architecture.

Section 3 explains and develops the service inside the Testbed that is being developed by the student, showing and detailing the components of the service (3.1), its internal design (3.2, 3.3, 3.4, 3.5) and the service functionalities (3.6). Also, a high-level explanation of the developed software code is presented (3.7).

Section 4 intends to depict an end to end execution example of the software, in order to demonstrate and validate the correct functioning and behavior of the platform.

Sections 5 and 6 contain the Conclusions and the Bibliography of the present work, respectively.

Section 7 presents de appendices of the project. They are, mainly, the documentation of the developed platform, including all the developed requests, parameters and examples.

2 HANSEL Positioning Testbed

2.1 Testbed architecture

The Testbed platform architecture at system level description is shown in *Figure* 1. Blue elements in the figure represent the elements that are being fully designed and developed by the project partners. The different components of the architecture are:

- Operators/Administrators. Users of the Testbed, that directly interact with the Testbed front-end.
- Nodes/Users. Either sensors or active equipment providing data to the server.
- Server. The server or Central Processing Facility (CPF), where most of the computation is performed.
- Infrastructure. Consisting of GNSS transmitters, 4G/5G and WiFi access points, will provide the Testbed with the needed signals or information, used so the Testbed users can properly develop its activities. It is important to highlight that, whilst some of this infrastructure is under the control and monitoring of the Testbed administration (WiFi APs, sensors, smartphones), other is totally detached from it (4G/5G towers, GNSS satellites), under the control of other agencies.
- Network. The network of the Testbed encompasses the contents of the previously mentioned components, and as a baseline will be deployed in UAB's facilities. UAB campus stands as a Smart Campus Living Lab, the first of its kind to receive the title of a certified member of the European Network of Living Labs (ENoLL), in 2014 [5].

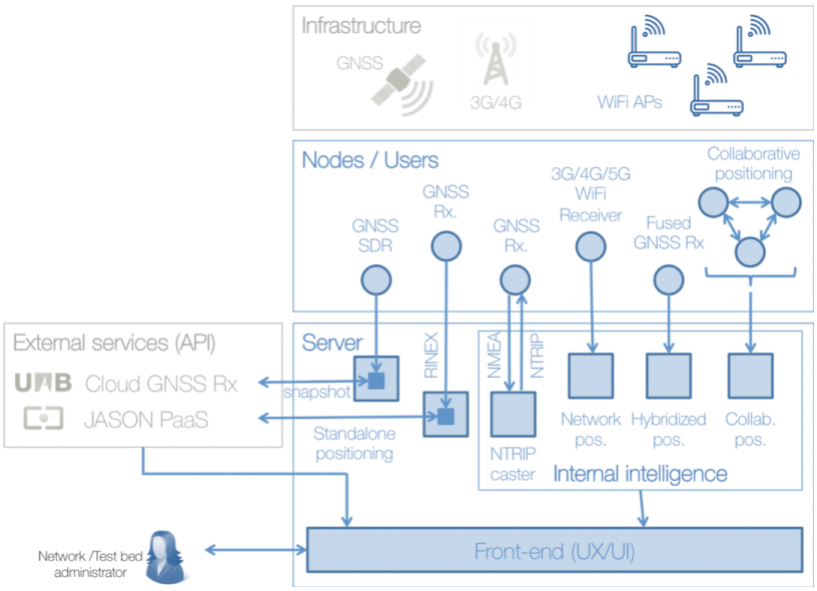

Figure 1. Testbed system level overview.

The cornerstone of the Testbed is the server (CPF), which on one side allows the data exchange between all the Testbed components, and on the other side has enough computational power to either compute navigation algorithms to obtain positions or to facilitate the data provision to the Testbed users to they can perform its position computation. The implementation of the CPF is based on various services encapsulated in containers, which are each one of the blue squares in *Figure* 1.

All the previously mentioned functionalities materialize in a list of services that HANSEL contains in order to satisfy the Testbed requirements. The architecture is distributed among three different kinds of software: services in server, pieces of software running physically inside the HANSEL CPF, applications, software applications running in nodes outside the server (smartphones, sensors), which directly communicate to its corresponding services, and external services, performing subtasks associated to its related services, already developed at the start of the project by the corresponding partner. It is possible to see in Figure 2 where each service is located and its corresponding developed applications (if any), that give support to the services. This information is presented along with a color code, clarifying what is the contribution of each partner to the project.

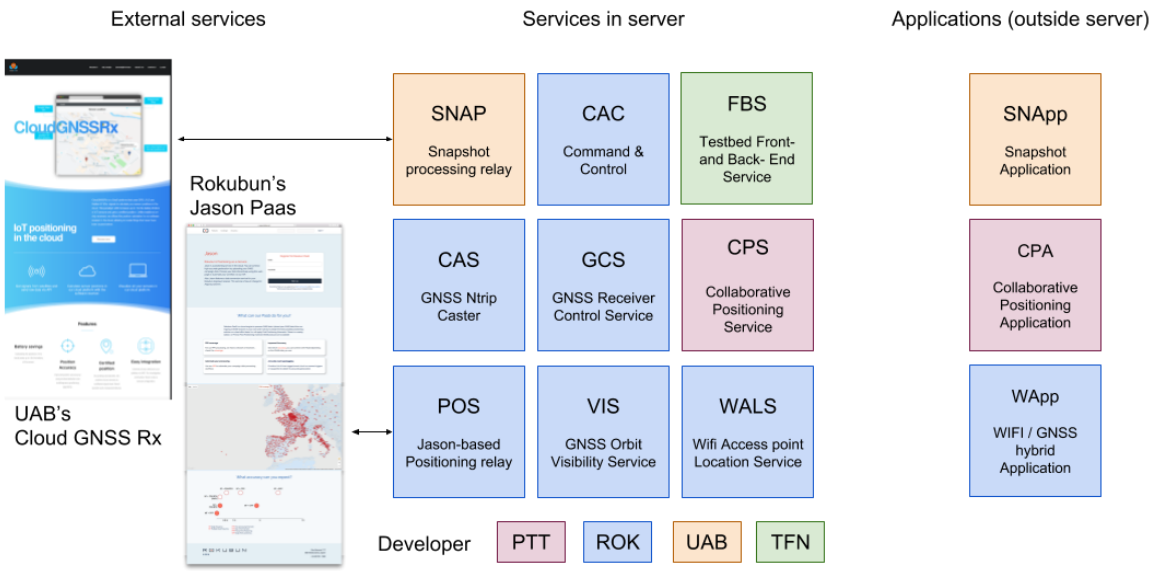

Figure 2. HANSEL service structure.

A brief summary of each service and application follows:

- CAC (Command And Control). This service is in charge of monitoring the Testbed user position and (optionally) a route identifier. With this information the service sends notifications to the user (e.g. via Android app). This module is the foundation that will, in the future, a full-fledged command and control of vehicle platoons, formations, etc.
- POS (Positioning service). This module is a relay module that interfaces with Rokubun's Jason Positioning-as-a-Service in the cloud. The user is able to upload RINEX files and compute its position (dynamic or kinematic) by means of PPK or SPP (depending on available nearby stations). This service is a thin interfacing layer to forward the requests and store the metadata that will be used for bookkeeping purposes.
- GCS (GNSS Receiver Control Service). This service is in charge of monitoring the data provided by the GNSS receivers of the Testbed.
- CAS (GNSS Ntrip Caster). This service broadcasts the data from the Testbed GNSS receivers so that users can perform DGPS or RTK if they are in the area of influence of the Testbed.
- VIS (GNSS satellite visibility service). Provides the necessary tools to visualize the status of the GNSS constellation above the Testbed (number of satellites, azimuth, elevation, etc.)
- WALS (WiFi Access Location Service). The service that ingests data points from an external application (described below) and will routinely compute the WiFi Access point location within the Testbed.
- SNAP (Snapshot service). Service charge of providing the position and time of the GNSS and cellular sensors, as well as the intermediate raw observables that were obtained from the received signals. The service internally provides the option to hybridize both GNSS and cellular observables in order to obtain a hybrid position solution. The processing of real signals, either GNSS or cellular, is performed with the UAB's CloudRx Positioning-as-a-Service in the cloud, which hence will be an *external service* to the Testbed. For cellular signals, SNAP additionally incorporates a simulator to allow the generation of synthetic raw observables, in order to circumvent the positioning problems posed by most of the existing cellular deployments.
- CPS (Collaborative Positioning Interface Service). The service in charge of coordinating the exchange of raw measurements and navigation solutions among the agents registered to the service (i.e. smartphones equipped with GNSS receivers). It provides the temporary hosting of the data and it will store the positioning solution

obtained by means of the CP to provide monitoring of the localization performance of the cooperative algorithm implemented at the user application level.

- FBS (Front- and Back-end Service). Responsible of the hosting, managing, configuration, monitoring and results data visualization of all the other services. The user authentication and management also takes place in this service. It also provides tools for the management and monitoring of all the hardware components of the testbed.
- SNAP Application (SNApp). Software (installed in a lite operative system) required by the GNSS and cellular sensors interacting with the SNAP service in order to control, manage, and configure them through the testbed.
- CP Application (CPA). Android-based application that will provide the interface between the GNSS receiver integrated in the smartphones and the CPS. Guarantees the reception and transmission of time-stamped raw measurements and navigation solutions towards agents registered to the CPS. The CPA will implement a CP algorithm running locally, thus distributing the logic of the cooperation at the node/user level.
- WALS Application. Android-based application that will interact with the Testbed WALS service to upload data measurements from WiFi hotspots, among other features, described below.

Besides the internal services running in the server, as already stated, the Cloud Receiver (CloudRx) service and Jason Positioning-as-a-Service will run externally to the Testbed and will be accessible via network connectivity.

2.2 Description of the main testbed functionalities

2.2.1 Positioning using GNSS snapshot-based cloud processing

Internet of Things (IoT) sensors, key component of any “Smart City” development, have experienced a dramatic growth in the recent years due to the advent of low-power wireless communication technologies such as LoRa, SigFox or narrowband IoT (NB-IoT). However, most of the sensors implementing positioning functionalities are still relying on GNSS chipsets that process the received GNSS samples, compute the user’s position, and then report this position together with additional data to the remote server. This approach implies high power consumption that compromises the battery lifetime of these sensors. A means to reduce this consumption is by moving from conventional “always-on” GNSS chipsets to ultra-low-power

GNSS chipsets implementing “on-off” strategies by means of snapshot-based processing. The user’s position is computed on-demand by processing a snapshot of received GNSS samples, and the result is then reported to the remote server. While snapshot processing significantly reduces the power consumption of IoT GNSS chipset, there are still many concerns when it turns to implement advanced features such as authentication, multi-constellation (e.g. GPS, Galileo, GLONASS, Beidou) and multi-frequency (e.g. L1/E1 and L5/E5a,b), where an additional constraint appears due to the limited computational capabilities of GNSS-enabled IoT devices.

Furthermore, more computationally demanding techniques must be applied when the GNSS-enabled IoT device operates under harsh environments such as urban canyons or soft-indoors, typically encountered in Cities. In that case high-sensitivity techniques (i.e., using much longer coherent and non-coherent integration times) are required at acquisition level. Similarly, advanced signal processing techniques are needed to detect and mitigate interferences e.g. jammers, a threat that could break the security and reliability of the GNSS data. This could jeopardize the efficiency of the Smart City management and endanger GNSS-based safety-critical or autonomous driving services.

All the mentioned problematics are solved through a paradigm shift in the way GNSS positioning is carried out; where the snapshots of received GNSS samples are processed remotely instead of locally at the user’s terminal. This concept is depicted in *Figure* 3.

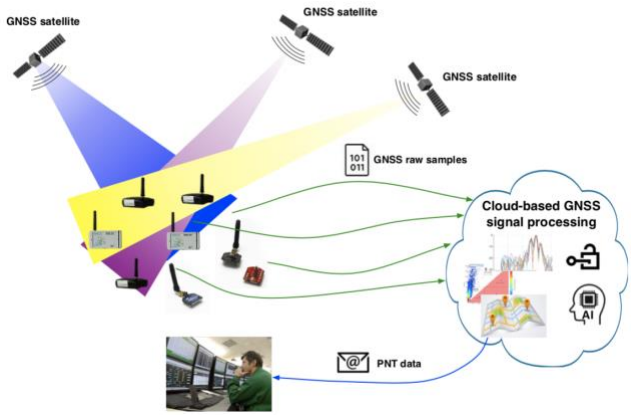

Figure 3. UAB cloud GNSS receiver operating on a network of heterogeneous GNSS-enabled IoT sensors.

2.2.2 Localization of jammers using GNSS snapshots

The presence of either intentional or unintentional interferences is known to be a major threat to the operation of GNSS receivers and therefore, to the services they provide in the context of Smart Cities. The underlying reason is the extremely weak power of GNSS signals received on Earth, which is one-billionth of a billionth the power consumed by a single 100W light bulb. As a result, an interference source emitting with just 1W EIRP at a few kilometers distance from a GNSS reference station, is received with an interference power to noise spectral C_i/N_0 of about 100 dBHz. This is more than 50 dB larger than the nominal carrier to noise spectral density C/N_0 of GNSS signals outdoors (on the order of 40–50 dBHz), and exceeds by large the 24-27 dB of inherent interference protection that is provided by GNSS spreading codes. In these circumstances, the performance of GNSS receivers dramatically degrades, and for the case of Smart Cities, it poses a serious threat to safety-critical applications.

A pragmatic approach is followed by selecting the most appropriate technique considering the nature of the Smart City users, whose GNSS-enabled devices are mostly based on mass-market implementations specifically tailored to the IoT domain. Thus, limited computational capabilities and limited resources will be assumed to be available. In order to compensate this limitation, localization of jammers using GNSS snapshots is carried out at the UAB's cloud GNSS receiver platform, by fusing and jointly processing the GNSS snapshots delivered by the Smart City users, as illustrated in Figure 4. This strategy has been previously considered in the literature as the best approach to exploit information from neighboring sensors with the aim of detecting and localizing the presence of potential threats [6].

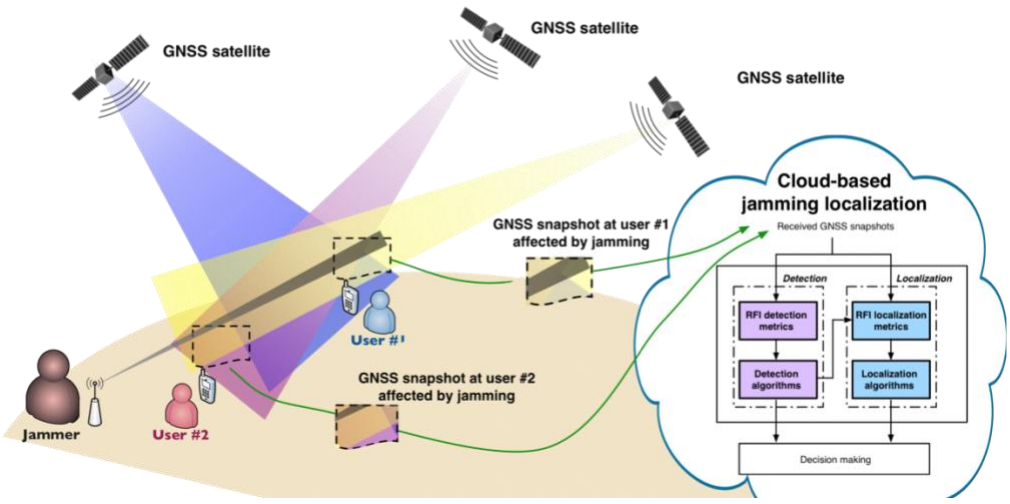

Figure 4. Jamming detection and localization by cloud-processing GNSS snapshots.

2.2.3 Positioning using terrestrial infrastructure and measurement hybridization

As it is known, GNSS performs ideally in open-sky conditions where the satellite visibility is not impaired. However, the performance of GNSS systems starts to degrade as more obstacles and signal blockages occur. Such environment is found especially in cities (urban canyons) and buildings (indoors). In these scenarios, other terrestrial-based signals-of-opportunity are being explored to complement GNSS: cellular and WiFi.

Cellular networks not only play a key role on the provision of communications capabilities to mobile devices, but they are also of importance for mobile positioning. The large cellular communication infrastructure deployed by the different network operators can be re-used for positioning purposes, providing an additional terrestrial positioning system in Smart Cities. The re-use of cellular deployments for positioning has been studied with simulations and field experimentations for the different network generations [7]. Results show a clear improvement when comparing the stand-alone GNSS performance and the hybridized performance using measurements from cellular networks such as 4G/5G.

In addition, governmental bodies have reinforced the need for complementary cellular-based location methods, such as in emergency services [8]. This interest has led to the specification of dedicated positioning signals, methods and protocols in 4G Long Term Evolution (LTE) systems. Although these location methods are only adopted in certain 4G LTE commercial networks (due to additional deployment costs), this trend is expected to change in the future with the inherent technological improvements of future 5G networks [9].

Positioning engines based on signals-of-opportunity from cellular networks are severely limited by the facts that network operators do not disclose the location of the base stations on one hand, and the lack of time synchronization between base stations on the other hand, both required to compute the user's location. For this reason, besides the Testbed Cellular signal processing, which provides the Testbed with the raw Cellular measurements, the Testbed also incorporates a software engine capable of performing a simulation of the scenario with network infrastructure and provide 4G/5G users' corrected observables and/or position solutions.

Besides cellular-based positioning, WiFi-based positioning has been also explored as a means to provide with indoor localization. Up to today, most approaches where based on the Received

Signal Strength Indicator (RSSI) as a proxy of the distance between the WiFi router and the terminal and then infer the positioning probabilistically by means of fingerprinting [10]. However, with the announcement of the Android P OS for smartphones, Google is giving access to the Fine Time Measurements (FTM) of the Round Travel Time (RTT), i.e. Time-Of-Arrival, which triggers the possibility to achieving meter-level accuracy indoors for devices compliant with the 802.11mc protocol. Smartphone technologies will be then used to both locate the position of WiFi routers as well as obtain positioning based on WiFi-ranges.

This functionality, comprising WiFi and Cellular positioning plus the hybridization of these terrestrial systems with GNSS technologies, are in charge of Rokubun regarding WiFi positioning plus WiFi and GNSS hybridization and of UAB when it comes to Cellular positioning, Cellular signal processing and the hybridization between this technology and GNSS.

2.2.4 Centralized control, storage and visualization

Each of the previous explained functionalities separate themselves into a different set of services, each one of them having different needs in terms of visualization and user-interaction. These different needs lead to the necessity of a way of homogenize the user interface, so an operator interacting with the Testbed is able to see the platform as it is, a whole of connected services, not as separated services put together in the same environment. Also, a storage system and a centralized control of the platform is needed in order to have a general overview of the whole system (visible GNSS satellites, Cellular base station and WiFi Access Point positioning, users connected to the platform, network connections, etc.) so the administrator of the system can easily access all the Testbed data and diagnose/solve problems if they appear.

The centralized control is refers to all the activities in charge of the data gathering from the different services and the treatment of these data, being also responsible for the visualization of the state of the GNSS satellites, the terrestrial infrastructure such as WiFi APs and Cellular base stations, and the visualization of all the present nodes in the Testbed (sensors, smartphones, etc.) out of the given data from the services.

2.3 Software architecture

One of the main requirements of the Testbed is the scalability and ease to add new services in the future (i.e. expandability). These requirements shape the development approach that needs

to be adopted for the implementation. Fundamentally, these requirements translate to the following requirements in terms of architecture:

- There needs to be a strategy for service encapsulation.
- Standardized interfaces to/from the service must be enforced.

In view of these requirements, the Testbed is based on the deployment of Docker containers [11] that will encapsulate all the components required by the service (executables, databases, software libraries, etc.). Within the container, a component that will provide an API will be put in place so that the communication to/from the service is done via a HTTP RESTful protocol [12]. All the nodes/users that need to communicate with the Testbed will also use this protocol. These technologies are explained in the following subsections.

2.3.1 Docker; packaging software into standardized units for development

Docker is a tool designed to create, deploy and run applications by using containers. A container is nothing but a unit of software that packages code and its dependencies so the application can run reliably from one computer environment to another without any hardware dependencies. A Docker container is lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. The usage of Docker containers has also beneficial side-effects to the development of the HANSEL server:

- Ensures the expandability of the platform, making it as easy as developing a new container and adding it to the Testbed.
- The implementation language or environment within the container is not determined by the host machine which gives a higher degree of independence among the different parties and developers involved in the implementation.
- The definition of the Docker container freezes the environment (e.g. software versions, required development libraries, etc.) which guarantees a higher degree of traceability.
- Ensures the usage of the developed software out of the HANSEL framework, thus adding value to the developed software out of the project.

2.3.2 Application Programming Interfaces: the RESTful API design

An Application Programming Interface (API) is an interface or communication protocol between different parts of a computer program intended to simplify the implementation and maintenance of software [13]. An API is, then, a layer added between the user and a software core (service functionality), intended to homogenize, filter and manage all the requests, while also protecting the service by isolating its direct interaction with the user. It could be considered the entrance door to the service laying behind, which will only receive the requests that are in line with the API documentation. A simplified diagram of this operation is depicted in Figure 5.

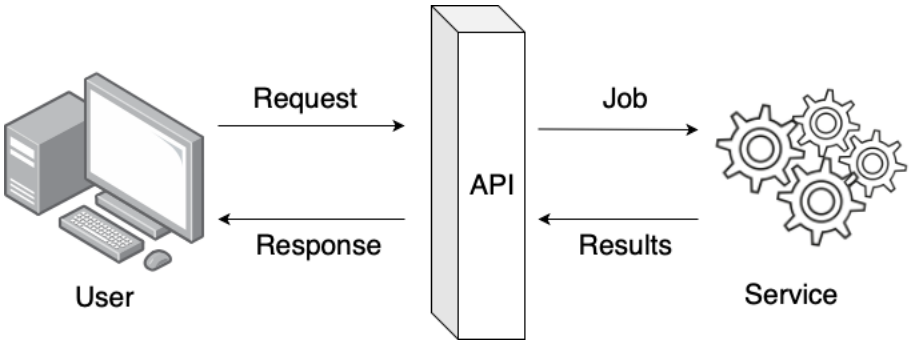

Figure 5. Simplified API functionality.

As stated before, the communication to and from the Testbed services is done in accordance to the RESTful design (Representational State Transfer), taking advantage of HyperText Transfer Protocol (HTTP). RESTful API design provides flexibility when it comes to data exchanges, since the REST has the ability to handle multiple types of calls and return different data formats. Main features of REST designs are:

- Stateless protocol. Every HTTP query includes all the necessary information to be executed, eliminating the necessity of keeping track of a session/state.
- Uniform interface. Every query is unequivocally identified by specific action (GET, POST, PUT, DETELE, etc.) and a Uniform Resource Identifier (URI). The structure and format of the queries is also determined, and defined by a few parameters:
    - Action or method. Defines what the query does. These actions are already designed. All available actions can be seen in [14].
    - Headers. Help the receiving entity to identify properties of the request, such as content type, authorization (if any), message length, etc.
    - Base URL. Defines the service that is being accessed e.g. www.google.com

- URL parameters. Define the specific parameters sent to the service, so the service can identify the specific functionality of the service that the user is searching for e.g. www.google.com?search=how_to_pass. In the previous example, "search" would be the parameter and "how_to_pass" the parameter value. In this case, the accessed service (Google) is going to know that the user wants to make a search (due to the parameter name), and what the user wants to search is "how_to_pass" (due to the parameter value).
- Body. Contains parameters not able to be located into the URL, such as passwords of another sensitive data.

The most widely used input/output format (for the request body and the response message) is the JSON format [15], although not being officially defined.

For confidentiality purposes, HTTP is used under Transport Layer Security (TLS) protocol, which encrypts requests from source to destination, also known as HTTPS. Moreover, for Authentication and Authorization purposes all the platform services use OAuth (access tokens) instead of basic HTTP authentication [16].

3 Development of a software-based snapshot positioning service (SNAP)

SNAP is the service developed in the framework of the present thesis. Following the project requirements, it should be in charge of providing to the Testbed an interface for processing snapshots of GNSS/Cellular raw samples data gathered by the network of deployed GNSS/Cellular sensors, obtaining information such as the position, time and measurements of the sensors (and related information: visible satellites, visible LTE base stations, C/N_0 values, etc.). It should also in charge of the detection and localization of jammer interferences, the simulation of Cellular measurements (observables), and also carries out the hybridization between GNSS measurements and Cellular ones in order to improve the position accuracy. This set of requirements posed the problem on how to combine and organize all the different subfunctionalities, which actually comprise heterogeneous tasks such as data management, storage, processing, communication and synchronization. In order to create a service with an arranged structure, thus homogenizing what is common and isolating what it is not, a few points were raised:

- All the signal processing should be done in the external service for the sake of computational savings to the CPF.
- Each of the different subfunctionalities (GNSS processing and Cellular processing) although related in concept, should be separated in modules and work standalone.
- Hybridization functionality of the Testbed should make use of GNSS and Cellular processing modules.
- There is a need of a centralized storage facility i.e. a database.
- All the communication, data management and synchronization should be done by a superior entity with respect to the processing modules, namely the service API.

It was decided to use Python v3 [17] as main tool for the service development, due to the language being a very powerful to API development thanks to software packages such as Flask [18] (and sublibraries), and Requests [19]. Python also possesses a lot of packages to ease the database creation an interaction via the ORM technique [20], such as psycopg2 [21], SQLAlchemy [22] and Marshmallow [23]. All these features, in addition to the widely-known package for scientific computing Numpy [24] made Python the perfect candidate to carry on all the development presented in this thesis.

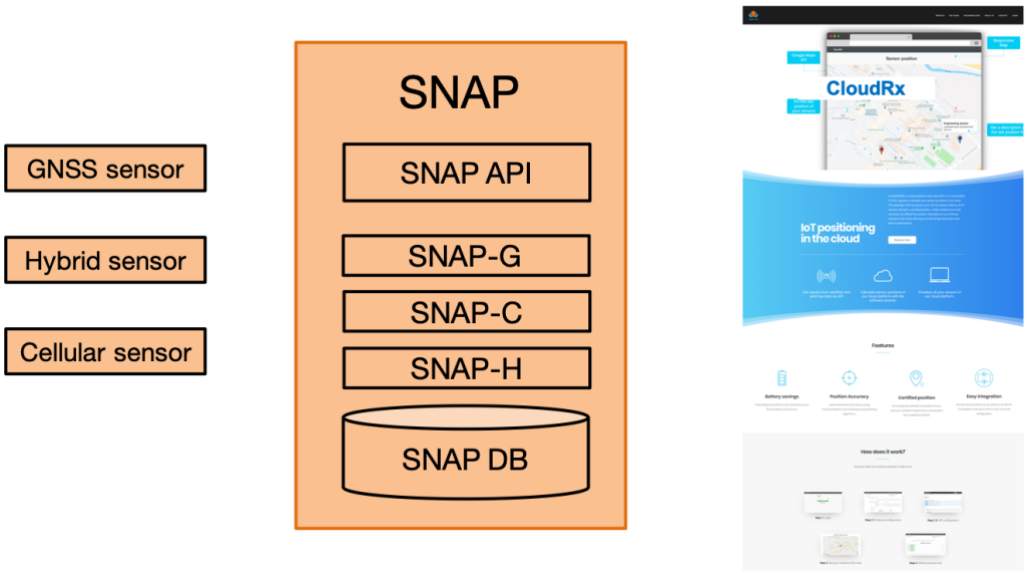

Figure 6. High-level components of SNAP service.

Previous points are related to some project requirements the software has to fulfill, namely:

1. The GNSS infrastructure should include constellations, signals and messages for GPS and Galileo as defined in their respective interface specifications.
2. The GNSS signals in centre frequencies L1/E1 (1575.42 MHz) and L5/E5a (1176.45 MHz) should be used.
3. The ground cellular infrastructure should consist as a minimum of the mobile broadband standards commonly known as 3G, 4G/LTE and 5G.
4. It should be possible to define the infrastructure attributes by means of internal and external configuration data.
5. The infrastructure should be split between external and internal infrastructure.
6. The server should be designed and implemented in a modular way for both software and hardware.
7. The (HANSEL) server should be able to execute all positioning and navigation algorithms and processing tasks related to use cases and scenarios except when external services are required.
8. It should be possible to define different areas within the network geography, as urban/open sky areas, indoor/outdoor areas and controlled access areas.
9. The server software should be portable.
10. Services should be flexible with respect to the number of nodes.
11. The used protocols should enforce by design data integrity and security measures for network traffic.

3.1 SNAP architecture

Based on the points presented in the previous section, the service has been built over five different components:

- `SNAP-G`, a subservice whose main function is to relay GNSS raw measurements to the external service `CloudRx` for its processing, for both position obtainment the jammer detection and localization.
- `SNAP-C`, a subservice whose main functions are a) to relay LTE real samples to the external service in order to its processing, and b) to simulate cellular-based position fixes.
- `SNAP-H`, a subservice whose main function is to hybridize GNSS observables and cellular observables in order to obtain a hybrid position fix.
- `SNAP DB`, a SQL database based in the PostgreSQL technology [25], which stores all data flow of the service, including the position fixes from SNAP sensor network, all its related parameters (such as sensor configurations and results), samples and related files.
- `SNAP API`, the solely entrypoint of the service, which interfaces the external network to the `SNAP-G`, `SNAP-C` and `SNAP-H` subservices, as well as the subservices with `SNAP DB`. The purpose of this component is to validate all the input and output data flow (parameters), to manage all the data storing and retrieval and to perform the service synchronization with the sensor network.

In line with the different subservices, the network of sensors if composed by:

- G Sensors (GNSS). Sensors capable of gathering GNSS signals by means of a radiofrequency front-end and sending them via Internet to the service for its processing.
- C Sensors (Cellular). There are two types of Cellular sensors. As stated in 2.2.3, the limitations on the corrected Cellular measurements computation raises the need of two types of cellular sensors, developing different functions:
    - CP Sensor (Cellular Physical). These sensors gather real Cellular signal by means of a radiofrequency front-end, with the main objective of sending them to the external service and computing the raw Cellular observables (not suitable for position computation).
    - CL Sensor (Cellular Logical). These sensors are called Logical because they are not a physical entity, just a logical piece of software. They are in charge of triggering a simulation in order to obtain corrected Cellular observables and

compute a simulated Cellular position with them. They were created to simulate a real-world scenario, where it would be possible to obtain the position out of real Cellular signal if there were no constraints.

- H sensors (Hybrid). These sensors are in able to both capture real GNSS signals and trigger the simulation of corrected Cellular observables, with the main objective of measurement hybridization. These sensors are needed since the hybridized measurements necessarily need to have the same reference position, so instead of having two sensors (one G and another CL) in the same position, it was decided to create the hybrid sensors for simplicity reasons. That means if there were no limitations computing cellular positions out of real signal, this type of sensor would be a physical sensor with both GNSS and Cellular front-ends.

A high-level perspective of the service is shown in Figure 6. A more detailed view, including the communication workflows between each of the service components and the external world, both with the sensor network and the visualization service is depicted in Figure 7.

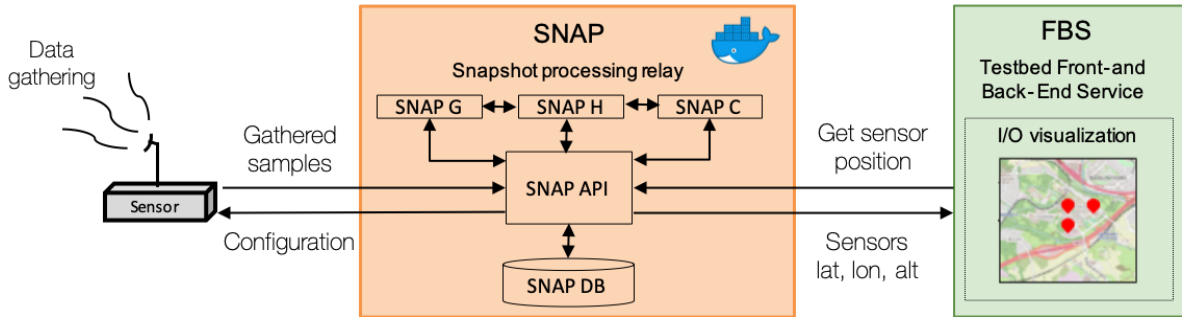

Figure 7. Detailed view of the service, including communication flows between components and external network.

The communication flow of the SNAP service is structured as follows:

1. SNAP sensors are listening for orders or jobs.
2. Upon receiving a request, SNAP API detects the action and parameters and validates them.
3. If the previous validation is successful, SNAP API communicates with the corresponding service functionality (SNAP-G, SNAP-C, SNAP-H).
4. The service functionality eventually interacts with its corresponding sensors by making them capture samples.
5. When the sensor sends the samples back to the service, the corresponding module communicates with a specific service tool (navigation algorithm) and triggers a specific computation, either in the external service or inside the service.

6. The results obtained from the computation are returned to SNAP API, which stores them in SNAP DB.

3.1.1 SNAP's external service; CloudRx

CloudRx is the external service that complements the SNAP service of the HANSEL Testbed, powered by Amazon Web Services (AWS) [26] and developed by the SPCOMNAV research group at UAB. The CloudRx service is responsible for all the Testbed snapshot processing. Users communicating with the Testbed server will have access to the cloud GNSS receiver [27] where snapshot processing and cooperative localization of jammers will take place. The basic principle of the platform is the externalization of the signal processing and position computation to a cloud platform (cloud SaaS), contrary to the (classical) procedure of the actual receivers, performing the position computation in situ. This is performed by means of sending to the cloud the gathered raw measurements for them to be processed. This approach is identified to be key for future GNSS positioning applications in the context of ubiquitous Positioning, Navigation and Timing (PNT) for connected IoT devices and Smart Cities [28]. Some obstacles that this approach solves are:

- Battery life. Mathematical complex operations like position computation constitute a high energy consumption due to all the related processes to satellite acquisition and its processing. Externalization of such procedure constitute high energy savings that translate into a longer battery life.
- Computational power. Limited device size avoids the implementation of powerful computational cores, so the processing is bounded both temporarily and computationally (there are functionalities that cannot be included due to computational limitations). Cloud computing solves these issues, being capable of an optimal resources assignment to optimize computation.
- Security. Due to the lack of computational power, devices are not able to perform certification at signal level in order to verify its authenticity. Unlimited (virtually) cloud computational resources enable the certification at signal level in order to avoid spoofing or jamming.
- Cost. Actual devices need a full GNSS module, formed by radiofrequency front-end and acquisition, tracking and navigation modules to compute a position. A CloudRx-designed device is only in need of a radiofrequency front-end, so the fabrication cost of the device is reduced.

- Precision. Lack of computation power compromises the maximum precision that a standard device can achieve. Cloud computational power also solves this issue.
- Flexibility. Actually, device updates are done through firmware, which limits the flexibility of those updates (normally requires to physically access all the devices). Cloud receiver software is easily updatable and allows high flexibility.

A high-level approach of the user the platform can be seen in Figure 8. Note that all the subblocks inside the AWS main block, corresponding to AWS services, are totally transparent to the user. The actual development within the framework of the project modifies the interactions in the figure by adding an extra layer between the User and the AWS block; SNAP and all its components, represented in the figure as "Web/API".

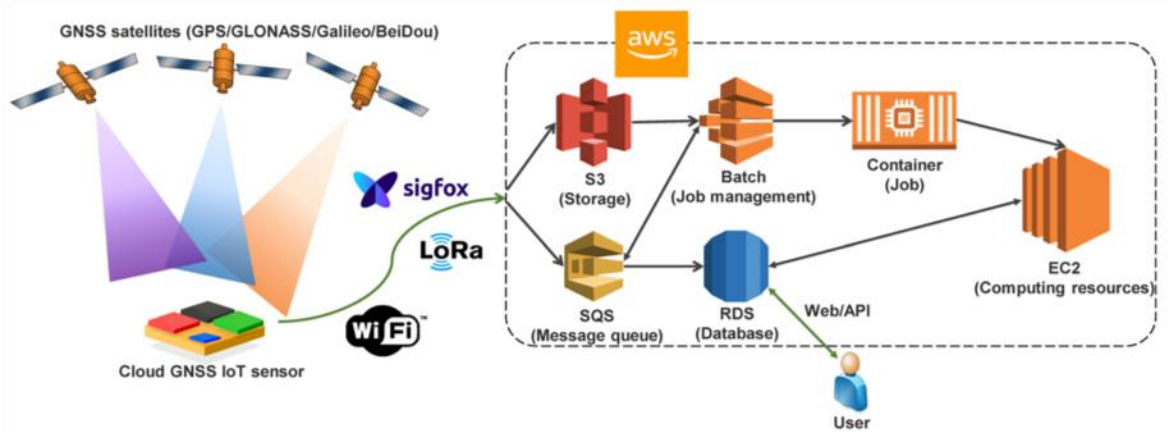

Figure 8. Interactions of a device with CloudRx platform.

3.2 SNAP-G subservice design

SNAP-G subservice, as its only function is to relay raw GNSS measurements to the CloudRx external service, his main and only component is a relay, which is in charge of this information forwarding. Following the points in 3, the structure was properly defined by atomizing the functionalities of each subservice block. A block diagram of the subservice and interactions with other components of the service can be seen in Figure 9.

As can be seen in the figure, the main SNAP-G interactions come from the sensor network, which is periodically polling for position computation requests (more information in 3.6.2). When one of these requests is detected, the demanded sensor captures GNSS signal and sends it to SNAP-G via SNAP API. The service then relays the raw measurements file to the CloudRx, which does the corresponding processing and returns to SNAP-G the obtained results, namely

the PVT of the sensor (if possible to compute), status of GNSS signals, generated observables, and the location of interference sources if any was detected.

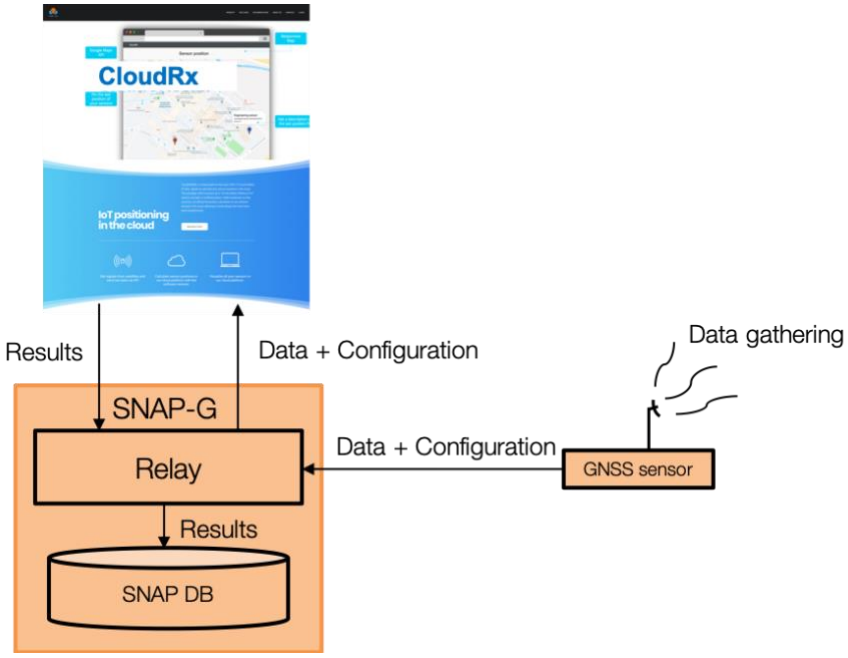

Figure 9. SNAP-G structure and interactions.

This module also has post-processing capabilities, allowing the operator to directly inject a GNSS measurements file to the service, without the need of a sensor to gather the measurements. The raw GNSS measurements file then follows the same workflow as if it was gathered by the sensor, being sent to the CloudRx for its processing and its results being stored in the database.

3.3 SNAP-C subservice design

SNAP-C subservice, due to its clear division between signal processing and simulation capabilities was divided in two main functionalities in order to have clear functionalities per software and easily work with each piece of software, Cellular simulated positioning and real Cellular data processing. These functionalities divide the subservice in two main modules according to the points in section 3, mainly the one stating that all the snapshot processing should be done in CloudRx external service. These two main modules are:

- Cellular positioning simulator. It is a software that simulates a 4G/5G cellular infrastructure and provides observables and/or position solutions. The simulation is configurable by the user, by specifying several parameters e.g., reference position, base station location, positioning method, etc. With this configuration, a deployment of base stations is performed and used to compute the sensor position. As mentioned in 2.1, the

sensors that make use of this simulation module are the so-called Cellular Logical (CL) sensors, since they are actually virtual sensors part of the simulation software in charge of generating cellular observables and position. These components can be seen in the lower part of Figure 10. While sensors are part of the software, have been depicted outside it for the sake of clarity.

- Cellular signal relay. Relay service in charge of forwarding real Cellular signals to the CloudRx, where they are processed. The results (including raw observables, spectrum image, detected base stations, etc.) returned to SNAP-C in order for them to be stored in SNAP DB.. These components can be seen in the upper part of Figure 10.

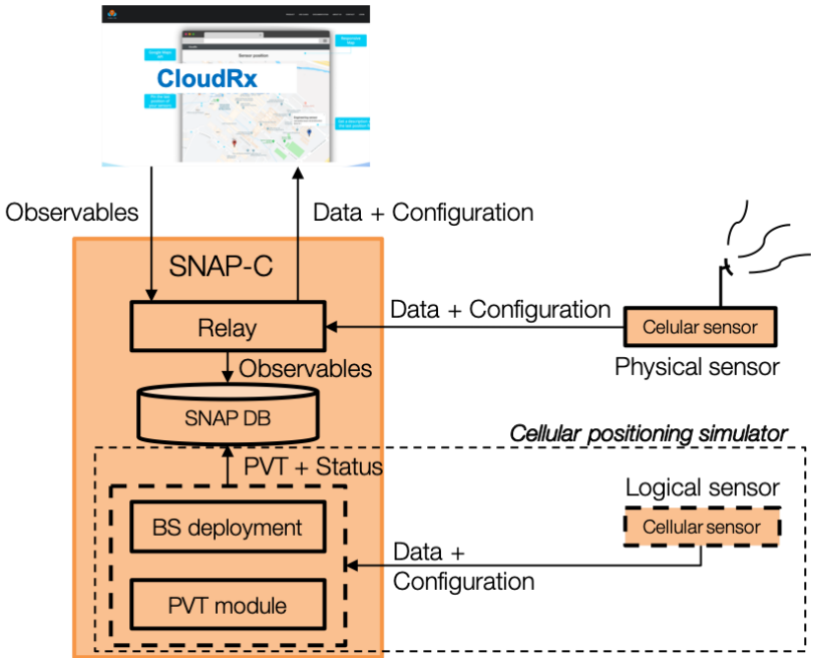

Figure 10. SNAP-C architecture and interactions.

The workflow for SNAP-C computations depends on the module that is going to be used. Regarding the Cellular signal relay, the workflow is very much the same as for SNAP-G, since the carried task is the same: to forward samples to the external service (CloudRx) for its processing. This module also has post-processing capabilities when it comes to the Cellular signal relay, performing the same functionality as SNAP-G out of a previously recorded raw Cellular measurements file.

When it comes to the Cellular positioning simulator, things are a little bit different; since the sensors interacting with this piece of software are logical (a piece of software), there is no need to interact and wait for data gathering as in previous cases. Once all the request parameters are validated by the SNAP API, the simulator is triggered inside the own service (so no need to communicate with the external service), and its results are directly stored in SNAP DB.

3.4 SNAP-H subservice design

SNAP-H subservice mainly contains a hybrid PVT algorithm that uses GNSS and Cellular observables as input. These observables are computed on demand when a hybrid position is requested using SNAP-G and SNAP-C modules. The workflow of this service is, then, once a Hybrid (H) sensor detects the position request, on one side gathers GNSS raw measurements for its processing in the CloudRx, and on the other side triggers the Cellular positioning simulator. Both computations are launched in parallel, and when both executions are finished all the generated outputs are directly injected to the hybridization algorithm as input parameters. The module performs the corresponding GNSS + Cellular measurement hybridization and outputs the hybrid sensor position plus related parameters, such as the HDOP (Horizontal Dilution Of Precision). A detailed block diagram of the subservice is shown in Figure 11. This module also has post-processing capabilities in the same way as SNAP-G and SNAP-C (Cellular signal relay).

The use of simulated Cellular observables instead of the ones obtained from real Cellular signal computation (, upper part) is limited by implementation constraints out of the scope of the development. Those are, on one side, the lack of knowledge about the exact position of the base stations, and on the other side the lack of time synchronization between base stations. Even assuming the base station positions are accurate enough, the synchronization constraint cause the Cellular signal computation module only capable of obtaining the raw Cellular observables i.e. observables without time synchronization corrections.

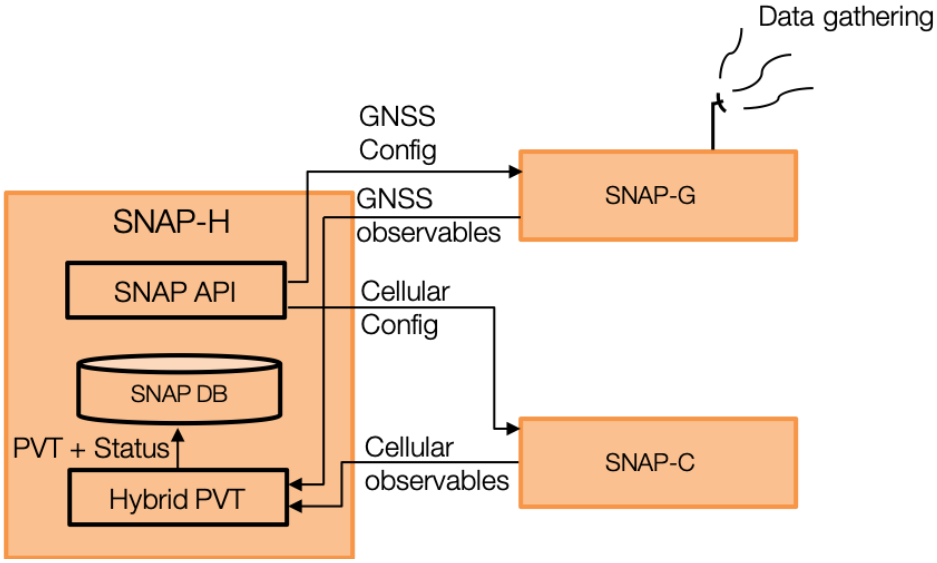

Figure 11. SNAP-H architecture and interactions.

3.5 SNAP DB design

SNAP DB, the central storage facility of the SNAP service, is a database created with PostgreSQL technology, an open source object-relational database system that uses and extends the SQL language. As stated previously, the existence of Python v3 packages such as psycopg2 and SQLAlchemy, ORM (Object Relational Mappers) for this kind of SQL databases and specifically PostgreSQL one, eases both the database creation, interaction and modification as time progresses and the requirements are polished. This technology also possesses parameter validation capabilities, thus adding another security layer to the service avoiding database corruption and easing the API parameter validation process.

Database structure was designed following the Testbed hierarchy itself, where operators are on top and each of them possesses nodes such as sensors or smartphones, and these nodes are positioned by means of different technologies. SNAP comprises two different technologies (GNSS, Cellular) and 4 different applications (GNSS positioning, Cellular processing, Cellular simulated positioning, GNSS plus Cellular hybridization), giving place to the database structure depicted in the UML diagram in Figure 12.

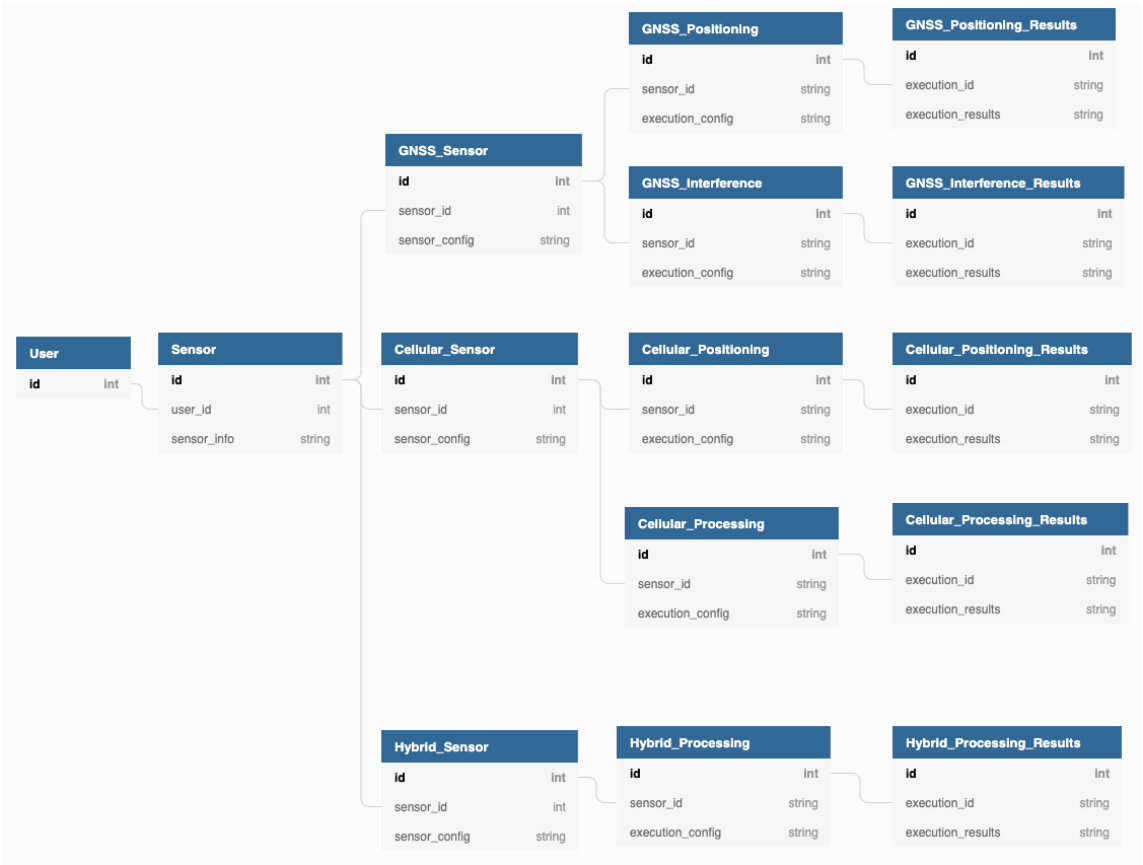

Figure 12. Simplified UML diagram of the stored information in SNAP DB..

3.6 SNAP API design

The SNAP API is nothing but the entrypoint of the SNAP service, so it is the component responsible of all the input/output (I/O) data of the service. It acts as a black box with respect to other services, so it is important for SNAP to be standalone and reliable when it comes to its internal procedures.

This need of reliability made the API to be built upon the Representational State Transfer design (RESTful), which creates an environment with tools capable of handling high number of request while also being modular, easing the scalability and modification of the service inner functionalities. Since the design also detaches client and server side (while also providing a homogeneous interface), any functionality modification is totally transparent to the user.

Security features are also important when it comes to a service that is going to be live on the internet, and it is known that plain HyperText Transfer Protocol (HTTP) does not provide enough safety. This reason led to place the RESTful API over Transport Layer Security protocol (TLS), also known as HTTPS. This protocol stack provides data encryption and certification at transport layer, thus assuring confidentiality and integrity of the data exchange. Security features have been also added to the service at application level, such as username/password log in and, most importantly, token-based identification for each performed interaction (OAuth).

The following subsections describe the different actions that are covered by the API. For the sake of clarity, next subsections just provide a superficial description of the actions and its main API queries. For detailed information, please refer to Appendix 1. Table 1 shows a summary of API actions and direct access to its descriptions.

Subservice/ Component	**Functionality**	**Brief description**	**Detailed description**
General	User management	3.6.1	7.1.1 User creation 7.1.2 User deletion 7.1.3 User information obtainment 7.1.4 Authorization token obtainment 7.1.5 User validation 7.1.6 User invalidation 7.1.7 User account upgrade 7.1.8 User account downgrade

			7.1.9 Give a user account admin privileges 7.1.10 Revoke privileges to an admin account
All subservices	Sensor management	3.6.2	7.1.11 Sensor 7.1.12 Sensor information modification 7.1.13 Sensor deletion 7.1.14 Sensor information obtainment 7.1.15 Sensor configuration obtainment 7.1.16 Sensor configuration modification 7.1.17 Sensor configuration update
SNAP-G	GNSS signal processing	3.6.3.1 3.6.3.2	7.1.18 Sensor location 7.1.23 Post-processing recorded GNSS data 7.1.19 Interference location 7.1.20 Located interference historic obtainment
SNAP-C	Cellular signal processing	0	7.1.21 Observables computing from live TLE signals 7.1.24 Post-processing recorded real Cellular data
	Cellular position simulation	3.6.4	7.1.18 Sensor location
SNAP-H	GNSS data & simulated Cellular observables hybridization	3.6.5	7.1.18 Sensor location 7.1.25 Recorded GNSS signal with simulated LTE signal hybridization
SNAP DB	Results and data management	3.6.6	7.1.22 Signal data uploading 7.1.26 Adding results to SNAP DB 7.1.27 SNAP database result obtainment 7.1.28 SNAP DB result deletion

Table 1. Summary of API actions and access to its descriptions.

3.6.1 User management

As a standalone service, SNAP has its own user system. That means the service has to provide the necessary means for user management. Those are mainly the following points:

- User privileges. Some users are designated as service administrators, who have a more general overview of the environment and are able efficiently manage the users registered into the platform.

- User registration. A Testbed operator must be able to register himself into the SNAP environment, and also to edit his information if needed. That also includes the capability of deleting a user.
- User access. The service provides the means for a secure access to the platform, which is implemented by means of OAuth.
- User account capabilities. The service has two type of platform accounts, "basic" and "pro". The first provides basic operations, while the latter offers additional features (such as better computational capabilities), added on top of the basic ones.
- User validation. Since the service is intended to be hermetic, the administrators are able to grant or deny access to standard users.

Main API queries for user management are the use registration query (7.1.1), by which a user registers in the SNAP service:

```
POST https://hansel.com/snap/api/user
```

and the user validation query (7.1.5), by which an administrator allows a user to access all the SNAP capabilities:

```
POST https://hansel.com/snap/api/validate?user=john_doe@gmail.com
```

3.6.2 Sensor management

When it comes to controlling the network of SNAP sensors, it is necessary to give to the Testbed operator the possibility of registering a sensor in the service, as well of modifying the sensor inner configuration, which defines how the devices captures GNSS and Cellular signal. Also, since the sensors are designed to interact with the Testbed on an autonomous way, they have to be able to reconfigure themselves without operator physical interaction.

The mentioned objectives are accomplished by programming the sensor to periodically poll the service for new configuration and jobs (signal gathering tasks). That is, asking the service "*Do I have to reconfigure myself? If yes, with what configuration?*" and "*Do I have any job to do? If yes, what job?*", respectively. The followed approach for the achievement of this behaviors is:

- For the new configuration retrieval, the service keeps a copy of the sensor configuration in the database, with a flag indicating if the configuration has been modified since the

last poll. Every time a sensor connects (asynchronously) to the service, the service checks if the sensor configuration has been changed by the user since the last sensor connection. If the flag says that configuration has been changed since the last time the sensor accessed, that means that the current configuration of the sensor does not match with the one present in the service database. Then, the new configuration is delivered to the sensor so it can reconfigure himself. With this approach the dependence of continuous service-sensor communication is eliminated, since the configuration synchronization is done in an asynchronous way.

- For the job retrieval, the behavior is achieved by building a FIFO (First In First Out) queue inside of the service database. Every time the sensor accesses the services, it checks if there is any job in its queue. If there is, extracts the least recent job, so the jobs are extracted giving priority to the ones that were firstly introduced. That approach avoids the sensor to be overloaded with job requests.

Main queries for this set of service capabilities are the sensor registration (7.1.11), used by a user to register a sensor in the environment:

```
POST https://hansel.com/snap/api/sensor
```

the sensor configuration query (7.1.16), used to change sensor capture parameters such as sampling frequency, frequency shift, format, encoding, etc.:

```
PUT https://hansel.com/snap/api/configure?sensor=DoeSensor
```

the update request (7.1.17), responsible for the sensor to remotely update its configuration parameters and check its corresponding job queue:

```
GET https://hansel.com/snap/api/update_request?sensor=DoeSensor
```

and the data uploading request (7.1.22), the way a sensor is able to upload the gathered data to the service for its forwarding to CloudRx.

```
POST https://hansel.com/snap/api/signal?execution=DoeExecution
```

3.6.3 GNSS & Cellular snapshot signal processing

GNSS and Cellular signal processing, have as main purposes to compute, on one side, the position of the SNAP GNSS sensors and on the other side to obtain Cellular measurements (observables) from real Cellular data. Another objective is to detect and locate possible

interference sources (jammers) surrounding the sensor network. It also provides to the hybridization module the necessary data to perform the measurement hybridization. The overall architecture supporting these services can be seen in *Figure* 7.

3.6.3.1 GNSS snapshot positioning

GNSS positioning provided by SNAP-G relies on a software receiver running in AWS, the one responsible of performing all the computationally demanding tasks in the cloud. As stated before, the cloud GNSS receiver is accessed by the SNAP-G service via a relaying connection, forwarding all requests for GNSS positioning that arrive to the SNAP service.

The receiver implements a high-sensitivity architecture where the user can make use of long coherent and non-coherent correlations in order to improve the sensitivity. As snapshot receiver, it only implements the acquisition stage, and the double-FFT algorithm is used to perform the code-delay and frequency search (Doppler) in an optimal manner, making extensive user of FFT operations to ease the implementation in devices already specialized in this type of computation e.g. wireless communication devices already implementing the FFT for OFDM signal processing. The GNSS software receiver is compatible with GPS L1/L5 and Galileo E1C/E5a, and makes use of assistance information based on the prior knowledge of a coarse position to boost the acquisition computation. The full list of specifications and functionalities can be found in [29]. The position solution is obtained using a classical WLS approximation, considering that there is no access to the navigation message thus using the estimated Doppler frequency [30] for the resolution of the navigation system.

The main request that allows access to this resource is the sensor location request (7.1.18):

```
GET https://hansel.com/snap/api/g/position?sensor=DoeSensor
```

3.6.3.2 GNSS snapshot-based interference localization

In addition to the provision of GNSS position fixes, the SNAP-G service is also capable of detecting the presence of interference signals within the GNSS band and to locate the position of the emitting source by making use of measurements collected by the sensor network. The adopted approach assumes that the sensors are not time nor frequency synchronized, so that the only observations that become meaningful for localization purposes are Received Signal Strength (RSS) measurements. Non-calibrated RSS-based localization algorithms have

previously been reported in applications dealing with interference localization in GNSS. This is the case for instance of [31], where the centroid method is used in the context of crowdsourcing localization with a population of GNSS sensors. SNAP-G adopts this method to localize potential detected interferences.

Main request that allows access to this resource is the interference location request (7.1.19), which accesses to a SNAP-G service and performs the interference location:

```
POST https://hansel.com/snap/api/g/interference
```

3.6.3.3 Cellular measurements computation

The SNAP-C module can also process real LTE signals captured from a sensor. The LTE downlink cell-specific reference signals (CRS) are used to estimate and track the time delay between the receiver and the detected base stations. The LTE CRS follows the same diagonal time-frequency pilot allocation pattern that significantly improves the robustness against inter-cell interference or interference between neighbor BSs with respect to the LTE downlink synchronization signals, which are overlapped in the same time-frequency resources.

For the module to perform optimally, the SDR sensor must be surrounded by at least one 4G LTE cellular base station within a radius of one km from the sensor. This prerequisite is derived from the low sampling frequency and low sensitivity of the low-cost sensors considered for testing. The BS proximity of 1 km is a good trade-off in urban areas to acquire several BSs (and at least one BS) due to the sufficient received power that compensates for the losses of the front-end equipment. If LTE samples are captured with a large sampling frequency and in good propagations conditions, the cellular base stations can be located at larger distances. LTE signal processing is based on three main blocks:

- LTE samples loading. The LTE baseband signals are captured by the sensor and loaded for data processing.
- LTE signal processing. The UAB LTE software receiver processes the snapshots of the captured LTE downlink signals. This data processing is based on the cell detection, signal acquisition and signal tracking for each detected BS.
- LTE output generation. The receiver provides as outputs the timing or pseudorange observables of the tracked BSs, their estimated SNR level and a time-frequency spectrum of the tracked signals.

The main request that allows access to this resource is the sensor observables computation request (7.1.21):

```
GET https://hansel.com/snap/api/c/observables?sensor=DoeSensor
```

3.6.4 Cellular position simulation

Due to the Cellular positioning simulator, SNAP-C module can also provide Cellular positioning. This type of positioning is based on a simulator for the deployment of 4G/5G cellular networks. The module, given the user position and a network deployment (either simulated or provided as input), generates downlink TDoA observables as the ones mainly adopted for positioning in the 4G/5G standard. Having these observables, the localization engine only requires to know the network synchronization offset between the considered base stations along with its accurate coordinates to compute the mobile position. These parameters should be known, and provided as input to the module.

The main request that allows access to this resource is the sensor location request (7.1.18):

```
GET https://hansel.com/snap/api/c/position?sensor=DoeSensor
```

3.6.5 GNSS data & simulated Cellular observables hybridization

The SNAP-H modules computes a 3D PVT position with a tightly-coupled hybridization of the available GNSS and cellular TDoA 4G/5G observables. These are provided by the SNAP-G and SNAP-C subservices respectively, making use of the built-in simulator of 4G/5G deployment of BSs in the case of SNAP-C. Given the use of 4G/5G TDoA observables, the reference transmission time of the BSs can be independent of the GNSS reference time. Again, the classical WLS solution is considered to solve this positioning problem.

The main request that allows access to this resource is the sensor location request (7.1.18):

```
GET https://hansel.com/snap/api/h/position?sensor=DoeSensor
```

3.6.6 Data management

Main sources of data generation in the SNAP service are the samples files, both GNSS and Cellular, and ancillary information for the results obtainment, such as RINEX files or base

station position dictionary (.kml files). Since a huge amount of data is generated on normal usage, data management is an essential task for SNAP service. For this reason, a tools or ways for accessing and managing that data is mandatory.

Main SNAP services that allow data managing are the results storage query (7.1.26), performed either by CloudRx or the inner SNAP computation modules:

```
POST https://hansel.com/snap/api/results?execution=DoeExecution
```

The results retrieval query (7.1.27), able to output total or partial information about the results stored in database:

```
GET https://hansel.com/snap/api/results?sensor=DoeSensor
```

and the data deletion query (7.1.28), which allows the system administrators to manage the service storage and bookkeeping-related activities:

```
DELETE https://hansel.com/snap/api/results?sensor=DoeSensor
```

3.7 SNAP high-level source code structure

SNAP API service, created from scratch, has now over five thousand (5000) code lines considering just the architectural source code, not the functional one. The source code is written in the image of the architecture, divided first in the two main components of the API (database and subservices) and then the subservices are divided into the general functionalities, common to all three different blocks, and the specific functionalities of each subservice, giving place file structure shown in *Figure* 13. The code is written following Python v3 style constraints [32], and takes as basis the common software file structure, tailored to suit the specific service needs, such as the presence of functional code (MATLAB compiled code). It is important to note that this code is also responsible for the interactions with the functional code. That is, to make the corresponding calls either to the compiled MATLAB modules or to CloudRx. Then, the API retrieves the module's results and stores it into de database.

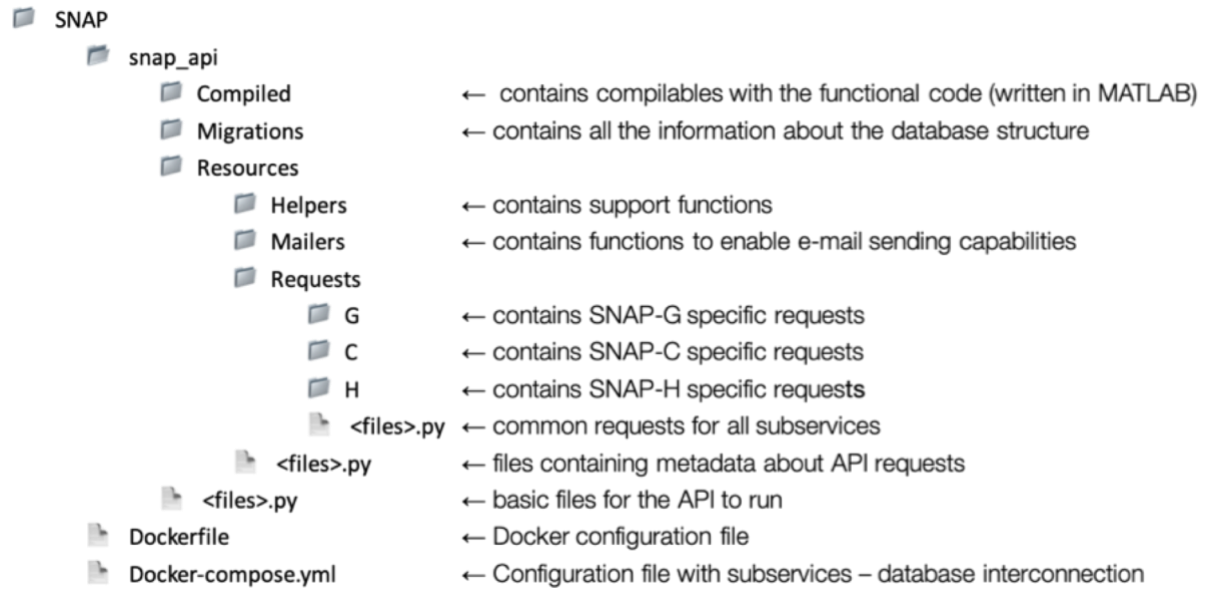

Figure 13. SNAP service source code structure.

As an additional feature to the developed software, the service API has been fully documented regarding all the available queries, necessary parameters, accepted values and examples.

4 Service validation

This section intends to demonstrate and validate the correct behavior of the developed service by means of a practical example. The practical example is going to be made from an operator point of view, and it is going to consist of launching some requests to the SNAP environment and see how the service reacts. This example departs from a registered and validated user with already registered and configurated sensors. To see more information about user registration/validation and sensor registration/configuration, see Sections 7.1.1, 7.1.5, 7.1.11 and 7.1.16, respectively. Important to note that the web environment that is shown in the following figures was developed by one of the consortium partners, not by the student. Web environment is nothing but a visual layer on top on the API responsible for the API request triggers via a user-friendly interface.

Let's assume that the considered user, John Doe, with access token "I_Pa5S" and ID 1, has one registered Hybrid (H) sensor, with Name "DoeSensor" and ID 1, located in the roof of his car. Let's also assume that the sensor is in standby mode i.e. listening its corresponding job queue, waiting for a job to come (Figure 15, left).

John Doe has parked in a very big, outdoor parking and does not remember where his car is. He takes out his mobile phone, enters the HANSEL environment, SNAP service, selects the option to locate his sensor (Figure 14) and sets up the corresponding parameters. As soon as John Doe sends his request via the environment, which triggers the corresponding API request (see 7.1.18 for more information) he is informed about the ID for the execution he launched job is created in the queue for his hybrid sensor, which as the same time triggers a SNAP-G and SNAP-C job, as can be seen in Figure 16. The sensor detects the jobs and starts gathering GNSS signal (Figure 15, right) while, in parallel, a Cellular simulation is running inside the SNAP service (Figure 17). Once the sensor finishes gathering the GNSS raw measurements, sends them to SNAP API using the corresponding query, and SNAP-G relays the samples to CloudRx for its processing (Figure 18). Eventually, both executions finish and SNAP-H hybridization algorithm is triggered (Figure 19). As SNAP-H module finishes its job (Figure 20), it is possible for John to see the results of the hybridization i.e. where his car is (Figure 21).

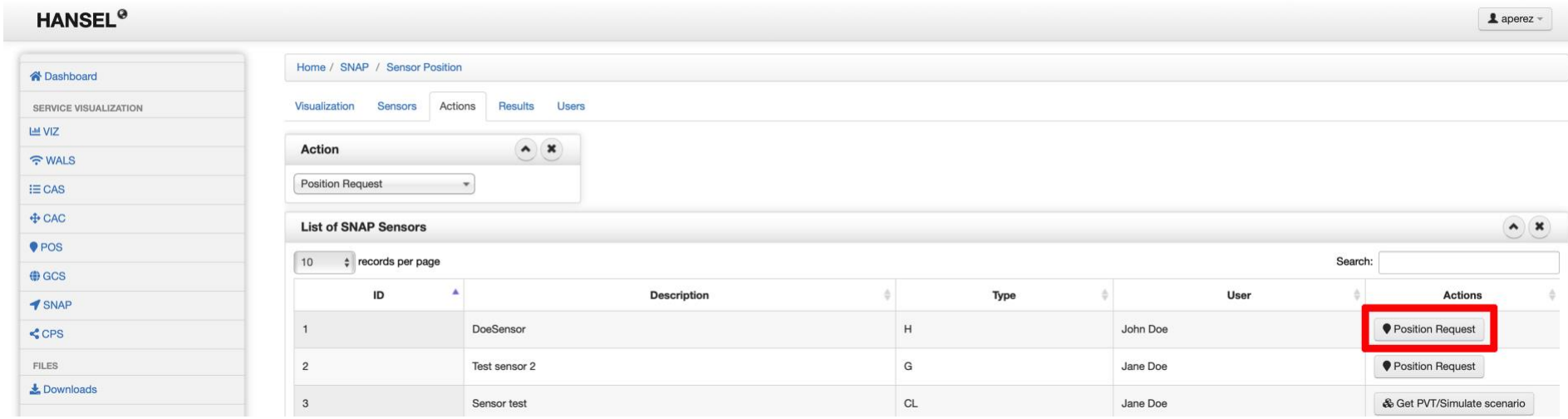

Figure 14. Sensor position request.

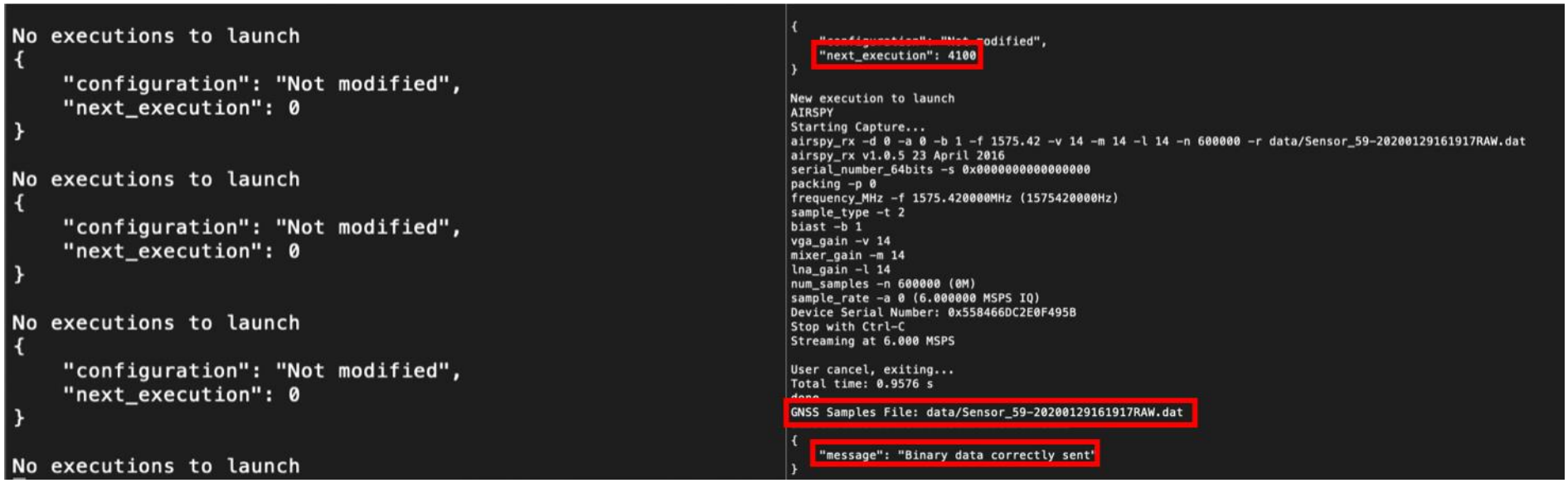

Figure 15. Left: sensor polling for new executions. Right: execution detected, signal gathering and deliver to SNAP service for its forwarding.

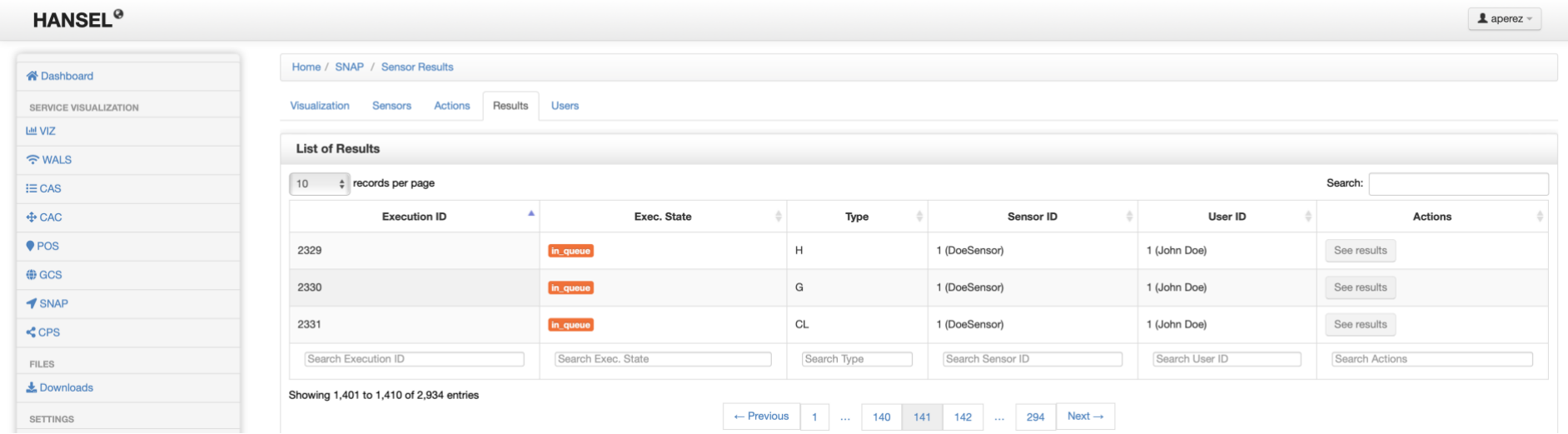

Figure 16. Jobs triggered by a hybrid position request.

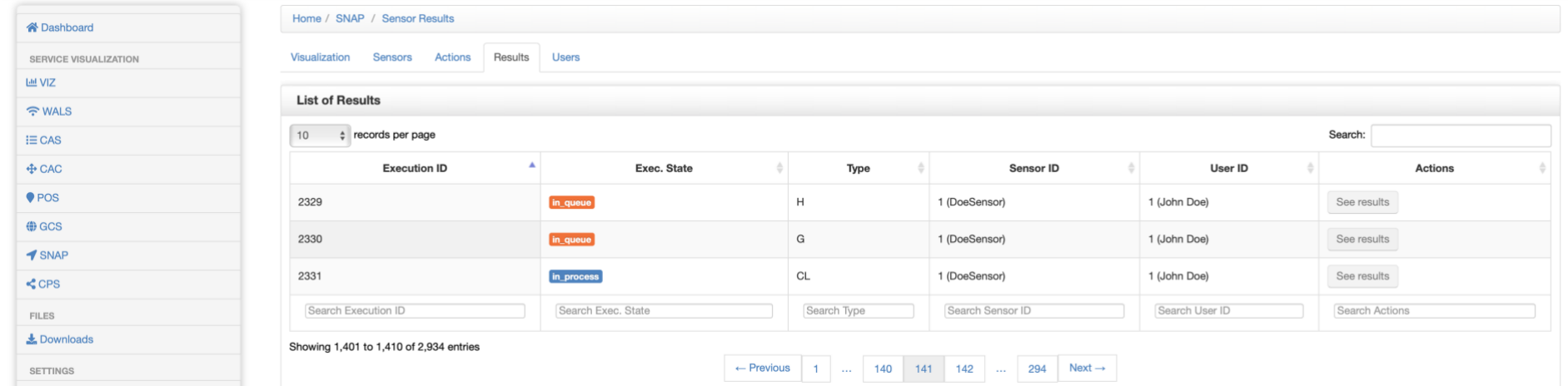

Figure 17. Cellular positioning simulator running while hybrid sensor captures GNSS signal.

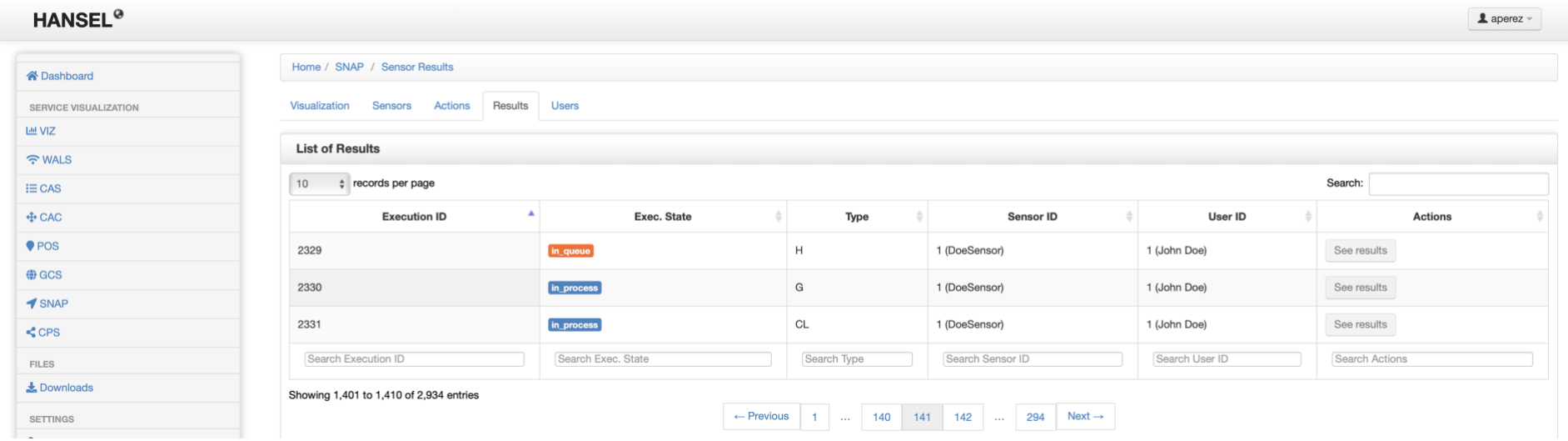

Figure 18. GNSS position computation started by external service (CloudRx).

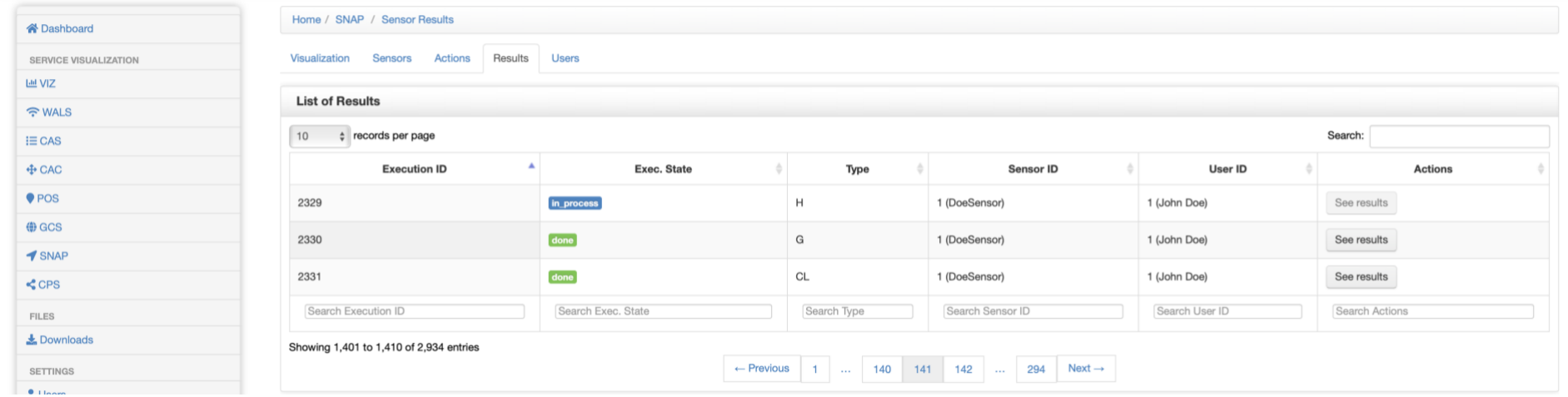

Figure 19. GNSS computation and Cellular computation finished. Hybridization of the measurements started as consequence.

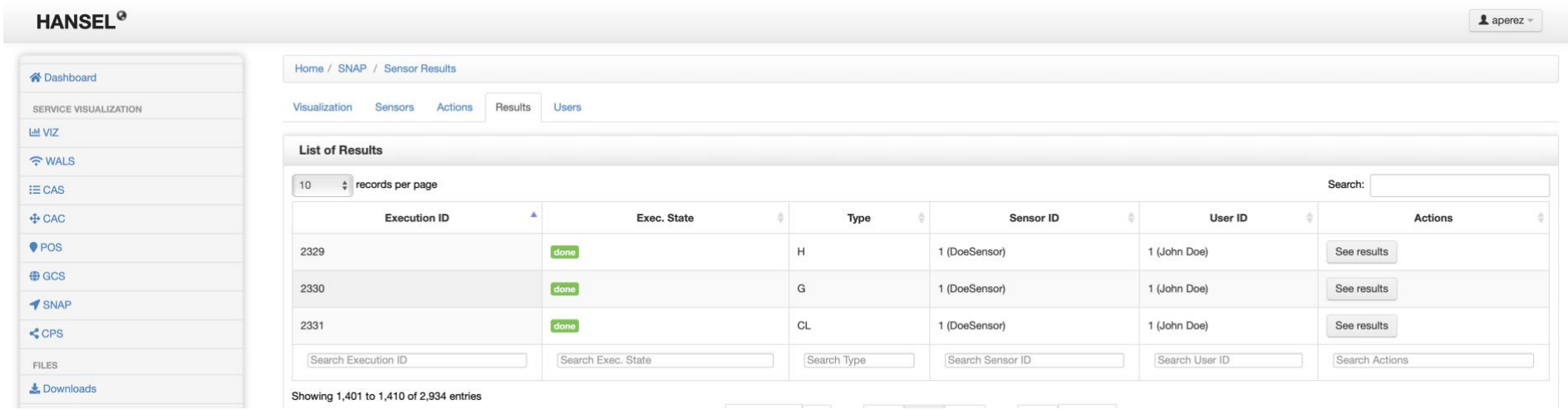

Figure 20. Hybridization finished. Results now available for the user to watch.

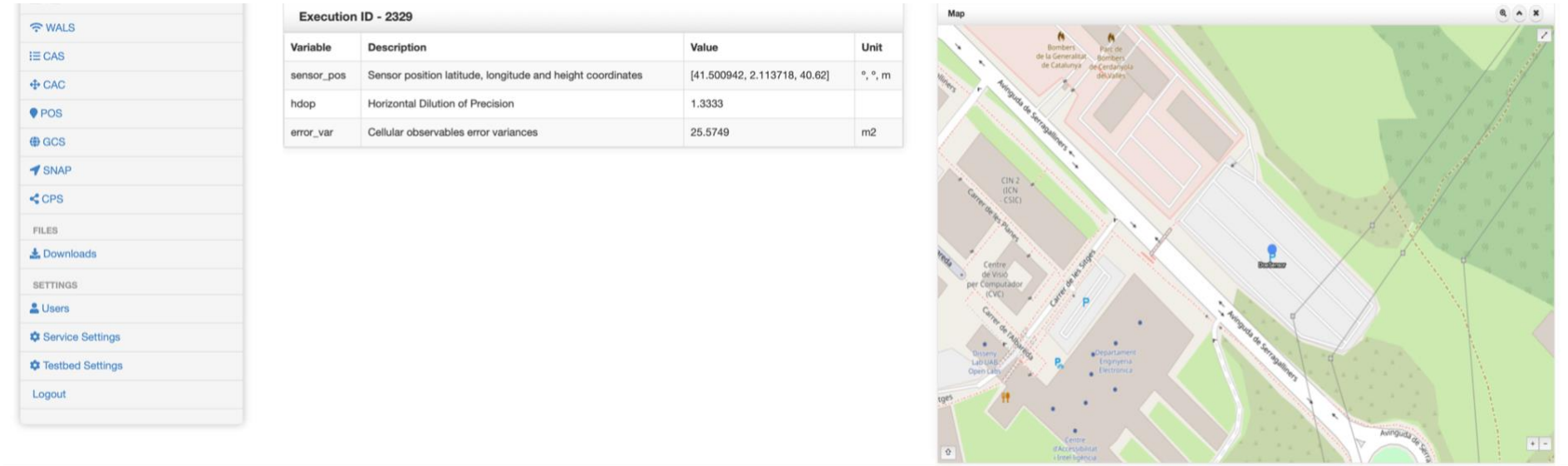

Figure 21. Hybrid position computation results.

5 Conclusions and future lines

The present work had the main objective of creating a service capable of managing, processing and storing the data generated by a distributed sensor network for the specific needs of positioning and interference localization in the context of Smart Cities. The developed software, actually functional and running in a real-life Testbed server, meets all the requirements defined by the end user (defined in Section 3), in this case the European Space Agency through the work statement of the ITT with reference AO/1-9494/18/NL/CRS. From all the work carried out since the beginning of the project (February 2019), the following conclusions can be extracted:

1. A fully functional service has been added to the HANSEL Testbed, thus providing to the server all the initially expected functionalities.
2. A modular an atomized service has been achieved, so any update or modification to any particular module of the service can be easily done.
3. A homogeneous interface, achieved by means of the RESTful design methods, led to an easy-to-access API, facilitating the consortium partners work in terms of integration while also providing a simple access for any external entity.
4. Related with the previous point, the created documentation is also a breaking point in terms of accessibility for any user or entity unrelated to the project, who now has available all the guidelines and necessary examples to successfully use the service.
5. The externalization of all the compute-intensive processes into a cloud platform is a step forward in the right direction for sustainable, energy-efficient devices as compared with conventional technologies.
6. The previously mentioned computation externalization in addition to the encapsulation of all the service logic thanks to Docker has led to a lightweight, portable software, thus highly facilitating the deployment in any kind of platform and environment.

Moreover, while the development of the platform is already finished, several future lines are still on the table to consider and further develop:

1. As software development is known to be an iterative process, adjustments in the service are going to be necessary as new issues appear during the testing phase.
2. Once the project finishes, further API interactions/functionalities are planned to be developed, such as device calibration, which is now done manually. That would bring

to the platform even more versatility when it comes to the user, providing a one-in-all service when it comes to device management.

3. As the development of the API has grouped all the logic in one place (the API itself), now a dedicated, proprietary web interface could be developed without further problem, since all the necessary data management and checks are done in the API core.
4. All the present submodules present inside the service (GNSS positioning service, Cellular positioning service, Hybrid algorithm and interference detection and localization) are planned to be improved, thus ending up providing a better service to the user while having to perform little or no modification to the service architecture.

6 Bibliography

[1] "Sustainable cities and communities: Facts and figures," United Nations, 2015. [Online]. Available: https://www.un.org/sustainabledevelopment/cities/.

[2] "Goal 11: Sustainable cities and communities," United Nations Development Programme, 2015. [Online]. Available: https://www.undp.org/content/undp/en/home/sustainable-development-goals/goal-11-sustainable-cities-and-communities.html.

[3] "Sustainable Development Goal 11: on Earth Observation and GNSS data support," United Nations: Office for Outer Space Affairs, 2018. [Online]. Available: https://www.unoosa.org/oosa/en/ourwork/space4sdgs/sdg11.html.

[4] "ITT - AO9494: Navigation and GNSS in Smart Cities: Testbed Concept Definition," European Space Agency, 16 July 2018. [Online]. Available: http://www2.rosa.ro/index.php/en/esa/oferte-furnizori/3003-navigation-and-gnss-in-smart-cities-testbed-concept-definition-expro.

[5] "UAB Smart and Sustainable Campus Living lab," European Network of Living Labs (ENoLL), 2014. [Online]. Available: https://enoll.org/network/living-labs/?livinglab=uab-smart-and-sustainable-campus-living-lab#description.

[6] J. A. Garcia-Molina and J. M. Parro-Jimenez, "Cloud-based GNSS Processing of Distributed Receivers of Opportunity: Techniques, Applications and Data-collection Strategies," October 2017.

[7] J. A. d. Peral-Rosado, R. Raulefs, J. A. López-Salcedo and G. Seco-Granados, "Survey of Cellular Mobile Radio Localization Methods: From 1G to 5G," 2018.

[8] FCC Record; Volume 20, No. 3, February 17 - March 20, 2015.

[9] R. D. Taranto, S. Muppirisetty, R. Raulefs, D. T. M. Slock, T. Svensson and H. Wymeersh, "Location-Aware Communications for 5G Networks: How location information can improve scalability, latency, and robustness of 5G," November 2014.

[10] V. Honkavirta, T. Perälä, S. Ali-Löytty and R. Piché, "A comparative survey of WLAN location fingerprinting methods," April 2009.

[11] "Docker main page," Docker, [Online]. Available: https://www.docker.com.

[12] "REST API guidelines," [Online]. Available: https://restfulapi.net.

[13] "Application Programming Interface article in Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Application_programming_interface.

[14] "RESTful API Methods," [Online]. Available: https://restful-api-design.readthedocs.io/en/latest/methods.html.

[15] "JSON format main page," [Online]. Available: https://www.json.org/json-en.html.

[16] "OAuth main page," [Online]. Available: https://oauth.net.

[17] "Python main page," [Online]. Available: https://www.python.org/.

[18] "Flask main page," [Online]. Available: https://www.palletsprojects.com/p/flask/.

[19] "Requests: HTTP for Humans, main page," [Online]. Available: https://requests.readthedocs.io/en/master/.

[20] "Object-Relational Mapping article on Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Object-relational_mapping.

[21] "Psycopg main page," [Online]. Available: https://pypi.org/project/psycopg2/.

[22] "SQLAlchemy main page," [Online]. Available: https://www.sqlalchemy.org.

[23] "Marshmallow main page," [Online]. Available: https://marshmallow.readthedocs.io/en/stable/.

[24] "Numpy main page," [Online]. Available: https://numpy.org.

[25] "PostgreSQL main page," PostgreSQL, [Online]. Available: https://www.postgresql.org.

[26] "Amazon Web Services main page," Amazon, [Online]. Available: https://aws.amazon.com.

[27] "CloudGNSSrx main page," SPCOMNAV group - Universitat Autònoma de Barcelona, [Online]. Available: http://www.cloudgnssrx.com.

[28] "GNSS User Technology Report, Issue 2," European GNSS Agency (GSA), 2018. [Online]. Available: https://www.gsa.europa.eu/system/files/reports/gnss_user_tech_report_2018.pdf.

[29] "CloudGNSSrx documentation," SPCOMNAV group - Universitat Autònoma de Barcelona, [Online]. Available: http://www.gloudgnssrx.com/documentation/.

[30] B. Peterson, R. Hartnett and G. Ottman, "GPS Receiver Structures for the Urban Canyon".

[31] D. Borio, C. Gioia, A. Stern, F. Dimc and G. Baldini, "Jammer Localization: from Crowdsourcing to Synthetic Detection," 2016.

[32] "Python Style Constraints," [Online]. Available: https://www.python.org/dev/peps/pep-0008/.

7 Appendices

7.1 Appendix 1: API action description

7.1.1 User creation

Relative URL	**/user**	
Action	Adds a new user to SNAP service	
Method	POST	
Access	External (can be used by anyone)	
Authorization	Not needed	
Querystring	-	-
Request body	JSON (application/json)	
Returns	201 CREATED	
	400 BAD REQUEST	
	422 UNPROCESSABLE ENTITY	
Use case	Entity wants to create an account in the SNAP service	
Example	`https://hansel.rokubun.cat/snap/api/user`	

The request body includes the information of the user to be registered, following the structure:

```
{
  "email": "<email>",
  "name": "<name>",
  "password": "<password>",
  "organization": "<organization to which the user belongs>",
  "purpose": "<purpose of the user>"
}
```

An example of request body:

```
{
  "email": "user@test.com",
  "name": "User",
  "password": "sample_password123",
  "organization": "ESA",
  "purpose": "Testing",
}
```

The query will return, upon successful status code, a token that will be used to access the platform upon validation:

```
{
  "token": "<token>"
}
```

An example of JSON response:

```
{
  "token": "DSTX1AvboFSy0yxVRgjV2XHbFh0dSfueNEfMVJDR4OpFs-
QY5O062vob_o5yvG7Gsp_HUqZp3XXINt8cM0zMLMU1LFySFftOX4TgpVGLALwBuSZB
5tAqchFFgdwwoig"
}
```

7.1.2 User deletion

Relative URL	**/user**	
Action	Deletes a user from SNAP service	
Method	DELETE	
Access	External (can be used by anyone)	
Authorization	Valid access token and admin privileges needed	
Querystring	user = <user_email>	The user to be deleted
Request body	-	
Returns	200 OK	
	400 BAD REQUEST	
	404 NOT FOUND	
Use case	A HANSEL administrator wants to delete a user from SNAP service; a SNAP user wants to delete his account	
Example	`https://hansel.rokubun.cat/snap/api/user?user=email@test.com`	

7.1.3 User information obtainment

Relative URL	**/user**	
Action	Gets the users from SNAP database. – Admin: obtains all the users from the database. – Normal user: obtains the information of the requesting user.	
Method	GET	
Access	External (can be used by anyone)	
Authorization	Valid access token needed.	
Querystring	(optional) (admin) user=<user_email>	Obtains the information of the requested user.
Request body	-	
Returns	200 OK & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	Admin wants to obtain user information; user wants to know his information	
Example	`https://hansel.rokubun.cat/snap/api/user`	

The query will return, upon successful status code, a response based on the permissions of the requesting user, following the structures:

- User is admin, no querystring

```
{
  "users": [
              {
                "email": "<email 1>",
                "name": "<name 1>",
                "organization": "<organization to which the user 1
belongs>",
                "purpose": "<purpose of the user 1>",
                "account_type": "<account_type>",
                "validated": <true or false>,
                "admin": <true or false>,
                "id": <user id>
              },
              …
```

```
              {
                "email": "<email N>",
                "name": "<name N >",
                "organization": "<organization to which the user N
belongs>",
                "purpose": "<purpose of the user N>",
                "account_type": "<account_type>",
                "validated": <true or false>,
                "admin": <true or false>,
                "id": <user id>
              }
           ]
}
```

- User is not admin, no querystring, or user is admin with querystring

```
{
  "user":
          {
            "email": "<email>",
            "name": "<name>",
            "organization": "<organization to which the user
belongs>",
            "purpose": "<purpose of the user>",
            "account_type": "<account_type>",
            "validated": <true or false>,
            "admin": <true or false>,
            "id": <user id>
          }
}
```

Examples of request JSON response:

- User is admin, no querystring

```
{
  "users": [
              {
                "email": "user_1@email.com",
                "name": "User N1",
                "organization": "UAB",
                "purpose": "Testing",
                "account_type": "pro",
                "validated": true,
                "admin": true,
                "id": 1
              },
              {
                "email": "user_2@email.com",
                "name": "Validation user",
                "organization": "ESA",
                "purpose": "Validation",
```

```
                "account_type": "basic",
                "validated": false,
                "admin": false,
                "id": 2
            }
        ]
}
```


- User is not admin, no querystring, or user is admin with querystring


```
{
  "user":
        {
            "email": "user_2@email.com",
            "name": "Validation user",
            "organization": "ESA",
            "purpose": "Validation",
            "account_type": "basic",
            "validated": false,
            "admin": false,
            "id": 2
        }
}
```


7.1.4 Authorization token obtainment

Relative URL	**/token**	
Action	Delivers an authorization token to the requesting user.	
Method	GET	
Access	External (can be used by anyone)	
Authorization	User password needed.	
Querystring	-	-
Request body	JSON (application/json)	
Returns	200 OK & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
Use case	SNAP user wants to obtain a new authentication token to interact with the API	
Example	`https://hansel.rokubun.cat/snap/api/token`	

The request information includes the information of the user, following the structure:

```
{
  "email": "<email>",
  "password": "<password>"
}
```

The query will return, upon successful status code, a JSON-response containing the token

```
{
  "token": "<token>"
}
```

An example of request body:

```
{
  "email": "user@test.com",
  "password": "sample_password123"
}
```

An example of JSON response:

```
{
  "token": "hOLDq7FHFvPeoiRRZktXe_-
8w_GULBHBVkh25b54zJBEZiSPdn0_Nz9hIRtEDtQPZfdzO-
HU4jnSVR6v3pLJf_fuR6MRN2klB8dHt-QpW6Oz8U7qQCF3fdjcpm4kgO0"
}
```

7.1.5 User validation

Relative URL	**/validate?user=<user_email>**	
Action	Validates a user	
Method	POST	
Access	External (can be used by anyone)	
Authorization	Valid access token and admin privileges needed	
Querystring	user=<user_email>	The user to validate
Request body	-	
Returns	200 OK	
	400 BAD REQUEST	

	401 UNAUTHORIZED
	404 NOT FOUND
Use case	SNAP administrators want to validate a user so he can interact with the platform
Example	`https://hansel.rokubun.cat/snap/api/validate?user=email@test.com`

Upon successful status code, the introduced user will be validated to interact with the service by user the defined queries.

7.1.6 User invalidation

Relative URL	**`/validate?user=<user_email>`**	
Action	Invalidates an already validated user	
Method	DELETE	
Access	External (can be used by anyone)	
Authorization	Valid access token and admin privileges needed	
Querystring	user=<user_email>	The user to invalidate
Request body	-	
Returns	200 OK	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	Platform administrators want to invalidate a user to prevent him from accessing the platform.	
Example	`https://hansel.rokubun.cat/snap/api/validate?user=email@test.com`	

Upon successful status code, the introduced user will be invalidated, thus will not be allowed of using the SNAP API queries.

7.1.7 User account upgrade

Relative URL	**/upgrade?user=<user_email>**	
Action	Upgrades a user account from *'basic'* to *'pro'*	
Method	POST	
Access	External (can be used by anyone)	
Authorization	Valid access token and admin privileges needed	
Querystring	user=<user_email>	The user to upgrade
Request body	-	
Returns	200 OK	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	Administrator wants to upgrade a user. Users with '*pro'* accounts have access to better machines and faster processing of the data.	
Example	`https://hansel.rokubun.cat/snap/api/upgrade?user=email@test.com`	

Upon successful status code, the introduced user will be upgraded, so could make use of the characteristics of a pro account.

7.1.8 User account downgrade

Relative URL	**/upgrade?user=<user_email>**	
Action	Downgrades a user account from '*pro'* to *'basic'*	
Method	DELETE	
Access	External (can be used by anyone)	
Authorization	Valid access token and admin privileges needed	
Querystring	user=<user_email>	The user to downgrade
Request body	-	

Returns	200 OK
	400 BAD REQUEST
	401 UNAUTHORIZED
	404 NOT FOUND
Use case	Administrator wants to revoke a *'pro'* account.
Example	`https://hansel.rokubun.cat/snap/api/upgrade?user=email@test.com`

Upon successful status code, the introduced user will be downgraded, so will be no longer capable of using of the characteristics of a pro account.

7.1.9 Give a user account admin privileges

Relative URL	**`/admin?user=<user_email>`**	
Action	Promotes a user account to admin.	
Method	POST	
Access	External (can be used by anyone)	
Authorization	Valid access token and admin privileges needed	
Querystring	user=<user_email>	The user to promote to admin
Request body	-	
Returns	200 OK	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	Administrator wants to promote a user account to admin account.	
Example	`https://hansel.rokubun.cat/snap/api/admin?user=email@test.com`	

Upon successful status code, the specified user has administrator permissions and capabilities.

7.1.10 Revoke privileges to an admin account

Relative URL	**/admin?user=<user_email>**	
Action	Revokes admin privileges to an administrator	
Method	DELETE	
Access	External (can be used by anyone)	
Authorization	Valid access token and admin privileges needed	
Querystring	user=<user_email>	The admin user to downgrade
Request body	-	
Returns	200 OK	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	Administrator wants to revoke the administrator privileges to an admin account.	
Example	`https://hansel.rokubun.cat/snap/api/admin?user=email@test.com`	

Upon successful status code, the introduced user will no longer have administrator permissions and capabilities.

7.1.11 Sensor register

Relative URL	**/sensor**	
Action	Adds a new sensor to the database that belongs to the user that sent the request and creates a default configuration for the registered sensor	
Method	POST	
Access	External (can be used by anyone)	
Authorization	Valid access token needed	
Querystring	-	-

Request body	JSON (application/json)
Returns	201 CREATED & JSON (application/json)
	400 BAD REQUEST
	401 UNAUTHORIZED
	404 NOT FOUND
Use case	User wants to register one of their sensors to the platform
Example	`https://hansel.rokubun.cat/snap/api/sensor`

The request information includes the information of the sensor that wants to be registered, following the structure:

```
{
  "description": "<sensor description>",
  "type": <sensor type>
}
```

The query will return, upon successful status code, the basic information of the registered sensor, following the structure:

```
{
   "sensor:" {
               "id": <sensor id>,
               "user_id": <user id>,
               "description": <sensor description>,
               "type": <sensor type>,
               "last_position": <last position known for the
sensor>,
               "creation_date": "<creation date of the sensor>"
            }
}
```

An example of request body:

```
{
  "description": "Roof sensor",
  "type": "G"
}
```

An example of JSON response:

```
{
   "sensor": {
               "id": 2,
               "user_id": 7,
               "description": "Roof sensor",
               "type": "G",
               "last_position": [41.99635, 2.112547,  140],
```

```
                "creation_date": "2019-10-01T10:19:58.00+00:00"
            }
}
```

7.1.12 Sensor information modification

Relative URL	**/sensor?sensor=<sensor_id>**	
Action	Modifies the description of a given sensor	
Method	PUT	
Access	External (can be used by anyone)	
Authorization	Valid access token needed, user must be the owner of the sensor to update	
Querystring	sensor=<sensor id>	The sensor to be updated
Request body	JSON (application/json)	
Returns	200 OK & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	User wants edit the information of one sensor	
Example	`https://hansel.rokubun.cat/snap/api/sensor`	

The request information includes the information of the sensor that wants to updated, following the structure:

```
{
  "description": "<new sensor description>"
}
```

The query will return, upon successful status code, the updated information of the registered sensor, following the structure:

```
{
  "id": <sensor id>,
  "user_id": <user id>
  "description": "<new sensor description>",
  "type": "<sensor type>",
```

```
  "last_position": <last position known for the sensor>,
  "creation_date": <creation date of the sensor>
}
```

An example of request body:

```
{
  "description": "Roof sensor moved to lab"
}
```

An example of JSON response:

```
{
  "id": 2,
  "user_id": 7,
  "description": "Roof sensor moved to lab",
  "type": "G",
  "last_position": [41.99635, 2.112547,  140],
  "creation_date": "2019-10-01T10:19:58.00+00:00"
}
```

7.1.13 Sensor deletion

Relative URL	**/sensor?sensor=<sensor_id>**	
Action	Deletes a given sensor	
Method	DELETE	
Access	External (can be used by anyone)	
Authorization	Valid access token needed, user must be the owner of the sensor to delete or have admin privileges	
Querystring	sensor=<sensor id>	The sensor to be deleted
Request body	-	
Returns	200 OK	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	User wants edit the information of one sensor	
Example	`https://hansel.rokubun.cat/snap/api/sensor`	

7.1.14 Sensor information obtainment

Relative URL	**/sensor**	
Action	Gets present sensors in the database and its basic information. – Admin: obtains all the sensors in the database. – Normal user: obtains only his sensors.	
Method	GET	
Access	External (can be used by anyone)	
Authorization	Valid access token needed	
Querystring	(optional) (admin) user=<user_email>	Returns the sensors of the specified user.
	(optional) sensor=<sensor_id>	· Admin: returns the specified sensor. · Normal user: returns the specified sensor if belongs to the user who made the request.
Request body	-	
Returns	200 OK & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	Administrator wants to see registered sensors, user wants to see his registered sensors	

Example	`https://hansel.rokubun.cat/snap/api/sensor`
	`https://hansel.rokubun.cat/snap/api/sensor?user=email@test.com`
	`https://hansel.rokubun.cat/snap/api/sensor?sensor=13`	

The query will return, upon successful status code, the requested information based on the performed query, following the structure:

- Response for no querystring or querystring user=<user_email>

```
{
  "sensors": [
```

```
                {
                   "id": <sensor 1 id>,
                   "user_id": <user id>,
                   "description": "<sensor 1 description>",
                   "type": "<sensor 1 type>",
                   "last_position": <last position known>,
                   "creation_date": "<creation date of sensor 1>"

                },
                …
                {
                   "id": <sensor N id>,
                   "user_id": <user id>,
                   "description": "<sensor N description>",
                   "type": "<sensor N type>",
                   "last_position": <last position known>,
                   "creation_date": "<creation date of sensor N>"
                }
             ]
}
```

- Response for querystring sensor=<sensor_id>

```
{
   "sensor": {
                   "id": <sensor id>,
                   "user_id": <user id>,
                   "description": "<sensor description>",
                   "type": "<sensor type>",
                   "last_position": <last position known>,
                   "creation_date": "<creation date of sensor>"
             }
}
```

Examples of JSON response:

- Example response for no querystring or querystring user=<user_email>:

```
{
  "sensors": [
                {
                   "id": 2,
                   "user_id": 7,
                   "description": "Roof sensor moved to lab",
                   "type": "G",
                   "last_position": [41.99635, 2.112547, 140],
                   "creation_date": "2019-10-01T10:19:58.00+00:00"
                },
                {
                   "id": 3,
                   "user_id": 7,
                   "description": "Roof sensor",
                   "type": "CL"
                   "last_position": [41.99436, 2.11225, 160],
```

```
                    "creation_date": "2019-10-02T20:19:58.00+00:00"
                },
                {
                    "id": 4,
                    "user_id": 7,
                    "description": "Parking sensor",
                    "type": "H",
                    "last_position": [41.98456, 2.11225, 60]
                    "creation_date": "2019-11-29T20:09:47.00+00:00"
                }

            ]
}
```


- Example response for querystring sensor=<sensor_id>:


```
{
   "sensor": {
                    "id": 2,
                    "user_id": 7,
                    "description": "Roof sensor moved to lab",
                    "type": "G",
                    "last_position": [41.99635, 2.112547, 140],
                    "creation_date": "2019-10-01T10:19:58.00+00:00"
            },
}
```


7.1.15 Sensor configuration obtainment

Relative URL	**/g/configuration** **/c/configuration** **/h/configuration**	
Action	Gets present sensor configurations in the database. – Admin: obtains all the sensor configurations in the database. – Normal user: returns all the configurations of the requesting user.	
Method	GET	
Access	External (can be used by anyone)	
Authorization	Valid access token needed	
Querystring	sensor=<sensor_id>	Returns the specified sensor configuration. An admin can request for the configuration of any sensor, while a normal user can only ask about its sensors
Request body	-	

Returns	200 OK & JSON (application/json)
	400 BAD REQUEST
	401 UNAUTHORIZED
	404 NOT FOUND
Use case	Administrator wants to know the configuration of the registered sensors, an user wants to know the configuration of his sensors

Example	`https://hansel.rokubun.cat/snap/api/g/configure`
	`https://hansel.rokubun.cat/snap/api/c/configure`
	`https://hansel.rokubun.cat/snap/api/h/configure`
	`https://hansel.rokubun.cat/snap/api/g/configure?sensor=13`
	`https://hansel.rokubun.cat/snap/api/c/configure?sensor=14`
	`https://hansel.rokubun.cat/snap/api/h/configure?sensor=15`

The query will return, upon successful status code, the requested configuration, following the structure:

- Response JSON for no querystring:

```
{
  "configurations": [
                      {
                       "sensor_id": <sensor 1 id>,
                       "<conf parameter 1>": "<parameter 1
value>",
                       …
                       "<conf parameter N>": "<parameter N
value>"
                      },
                       …
                      {
                        "sensor_id": <sensor N id>,
                       "<conf parameter 1>": <parameter 1
value>,
                       …
                       "<conf parameter N>": <parameter N
value>
                      }
                    ]
}
```

- Response JSON for sensor=<sensor_id>

```
{
   "configuration": {
                      "sensor_id": <sensor id>,
                       "<conf parameter 1>": <parameter 1
value>,
                       …
                       "<conf parameter N>": <parameter N
value>
                     }
}
```

Examples of JSON responses:

- Example of JSON response for no querystring: (/g, /h)

```
{
  "configurations": [
                      {
                       "sensor_id": 2,
                       "ion": false,
                       "sampling_freq": 2.8,
                       "bandwidth": 6.0,
                       "intermediate_freq": 0.6,
                       "quantization": 1,
                       "format": "IQ"
                       "encoding": "SIGN",
                       "delay": 10,
                       "signal_length": 100,
                       "update_period": 1,
                      },
                      {
                       "sensor_id": 7,
                       "ion": false,
                       "sampling_freq": 2.6,
                       "bandwidth": 2.8,
                       "intermediate_freq": 1.06,
                       "quantization": 16,
                       "format": "IQ",
                       "encoding": "INT",
                       "delay": 0,
                       "signal_length": 20,
                       "update_period": 5.0
                      }
                    ]
}
```

- Example of JSON response for sensor=<sensor_id> (/g, /h)

```
{
   "configuration": {
```

```
                        "sensor_id": 2,
                        "ion": false,
                        "sampling_freq": 2.8,
                        "bandwidth": 6.0,
                        "intermediate_freq": 0.6,
                        "quantization": 1,
                        "format": "IQ"
                        "encoding": "SIGN",
                        "delay": 10,
                        "signal_length": 100,
                        "update_period": 1.0,
                       }
}
```


- Example of JSON response for sensor=<sensor_id> (/c)


```
{
   "configuration": {
                        "sensor_id": 13,
                        "sampling_freq": 2.8,
                        "quantization": 1,
                        "format": "IQ",
                        "encoding": "SIGN",
                        "update_period": 1.0,
                       }
}
```


Configuration parameters of the sensors can be seen in Appendix 2, Table 2 for GNSS sensors, Table 5 for Cellular sensors and Table 11 for Hybrid sensors.

7.1.16 Sensor configuration modification

Relative URL	**/g/configuration?sensor=<sensor_id>** **/c/configuration?sensor=<sensor_id>** **/h/configuration?sensor=<sensor_id>**	
Action	Modifies a sensor configuration if the sensor belongs to the requesting user.	
Method	PUT	
Access	External (can be used by anyone)	
Authorization	Valid access token needed, the specified sensor must belong to the requesting user.	
Querystring	sensor=<sensor_id>	ID of the sensor whose configuration is going to be changed.

Request body	JSON (application/json)
Returns	200 OK
	400 BAD REQUEST
	401 UNAUTHORIZED
	404 NOT FOUND
Use case	User wants to change the configuration of one of his sensors

Example	`https://hansel.rokubun.cat/snap/api/g/configure?sensor=13`
	`https://hansel.rokubun.cat/snap/api/c/configure?sensor=14`
	`https://hansel.rokubun.cat/snap/api/h/configure?sensor=15`

The request information includes the values of the specific parameters that want to be changed for a given sensor, following the structure:

```
{
   "<parameter 1 name>": "<new parameter 1 value>",
   …
   "<parameter N name>": "<new parameter N value>"
}
```

Configuration parameters of the sensors that it is possible to modify can be seen in Appendix 2, Table 2 for GNSS sensors, Table 5 for Cellular sensors and Table 11 for Hybrid sensors.

An example of request body:

```
{
   "bandwidth": 3.5,
   "intermediate_freq": 3.056,
   "delay": 15
}
```

7.1.17 Sensor configuration update

Relative URL	**/g/update_request?sensor=<sensor_id>** **/c/update_request?sensor=<sensor_id>** **/h/update_request?sensor=<sensor_id>**	
Action	Delivers to the specified sensor: 1. New configuration if available 2. Next execution to process Handles the periodical requests coming from the GNSS sensors. This action is necessary because the communication between the SNAP service and the GNSS sensor must be initiated by the latter.	
Method	GET	
Access	Internal (only accessible to sensors)	
Authorization	Valid token needed, the sensor must belong to the requesting user	
Querystring	sensor=<sensor_id>	The sensor checking if there is new configuration and/or new execution to process.
Request body	-	
Returns	200 OK & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	Sensor wants to know if their user has changed its configuration and if there is any new execution to be launched	

Example	`https://hansel.rokubun.cat/snap/api/g/update_request?sensor=13`
	`https://hansel.rokubun.cat/snap/api/c/update_request?sensor=14`
	`https://hansel.rokubun.cat/snap/api/h/update_request?sensor=15`	

The query will return, upon successful status code, a JSON response containing the new configuration if there is any and the ID of the execution that needs to be launched by the sensor, following the structure:

```
{
   "configuration":   {
                        <New sensor configuration>
                      }
   "next_execution":    <execution id of the next execution>
```

```
}
```

In case there is no new configuration available, the value of the field "`configuration`" will be exactly "`Not modified`". If there is no execution to launch, the value of the field "`next_execution`" will be exactly `0`.

An example of JSON response:

```
{
   "configuration": {
                      "sensor_id": 2,
                      "ion": false,
                      "sampling_freq": 2.8,
                      "bandwidth": 3.5,
                      "intermediate_freq": 3.056,
                      "quantization": 1,
                      "format": "IQ"
                      "encoding": "SIGN",
                      "delay": 15,
                      "signal_length": 100,
                      "update_period": 1.0,
                     },
   "next_execution": 155
}
```

7.1.18 Sensor location

Relative URL	**/g/position?sensor=<sensor_id>** **/c/position?sensor=<sensor_id>** **/h/position?sensor=<sensor_id>**	
Action	Gets the position of a given sensor using the corresponding technique.	
Method	GET	
Access	External (can be used by anyone)	
Authorization	Valid access token needed and the specified sensor must belong to the requesting user.	
Querystring	sensor=<sensor_id>	ID of the sensor whose position is requested.
Request body	JSON (application/json)	
Returns	201 ACCEPTED & JSON (application/json)	

	400 BAD REQUEST
	401 UNAUTHORIZED
	404 NOT FOUND
Use case	User wants to know the position of one of his sensors

Example	`https://hansel.rokubun.cat/snap/api/g/position?sensor=13`
	`https://hansel.rokubun.cat/snap/api/c/position?sensor=14`
	`https://hansel.rokubun.cat/snap/api/h/position?sensor=15`

The type of sensor should be consistent with the type of positioning method being requested. In particular:

- for GNSS positioning the /g field must be present in the URL query, right after the /snap field, and the sensor being requested must be a physical sensor of type "G" capable of gathering GNSS samples. This information is introduced at the time of registering the sensor into the testbed (see "Register sensor" action described above).
- for cellular positioning the /c field must be present in the URL query, right after the /snap field and the sensor being requested must be a logical sensor of type "CL". This information is introduced at the time of registering the sensor into the testbed (see "Register sensor" action described above).
- for hybrid positioning the /h field must be present in the URL query right after the /snap field and the sensor being requested must be a sensor of type "H". This information is introduced at the time of registering the sensor into the testbed (see "Register sensor" action described above).
- and be of the corresponding type (type G for /g, type CL for /c, type H for /h).

The request information includes all the configurable execution parameters, following the structure:

```
{
   "<execution parameter 1>": "<parameter 1 value>",
   …
   "<execution parameter N>": "<parameter N value>",
}
```

The query will return, upon successful status code, a JSON object containing the ID of the execution that just launched to the platform, following the structure:

```
{
   "execution_id": <execution id>
}
```

Execution parameters can be found in Appendix 2, Table 3 for GNSS sensor positions, Table 9 for simulated Cellular positions and Table 12 for Hybrid positions.

An example of request body for G executions (/g/position):

```
{
   "nearfar": 0,
   "band": "1",
   "system": "GPS",
   "num_snap": 1,
   "delta_snap": 0,
   "coh_time": 20,
   "num_noncoh": 1,
   "gpdit": 0,
   "sam": 0,
   "dec_rate": 1,
   "interference": 0,
   "generate_agnss": 1,
   "manual_sat_search": 0,
   "assisted_dopp": 1,
   "ref_pos": [41.5002, 2.11292, 140.0]
}
```

An example of request body for C simulation execution (/c/position):

```
{
   "prs_bandwidth": 100,
   "carrier_freq": 4,
   "max_bs": 7,
   "deployment": 0,
   "num_measurement": 1,
   "sync_error": 0,
   "network_type": "4G",
   "pos_method": "OTDOA",
   "ref_pos": [41.5002, 2.11292, 140.0]
}
```

An example of request body for hybrid processing (/h/position):

```
{
   "nearfar": 0,
```

```
    "band": "1",
    "system": "GPS",
    "num_snap": 1,
    "delta_snap": 0,
    "coh_time": 20,
    "num_noncoh": 1,
    "gpdit": 0,
    "sam": 0,
    "dec_rate": 1,
    "interference": 0,
    "generate_agnss": 1,
    "manual_sat_search": 0,
    "assisted_dopp": 1,
    "prs_bandwidth": 100,
    "carrier_freq": 4,
    "max_bs": 7,
    "deployment": 0,
    "num_measurement": 1,
    "sync_error": 0,
    "network_type": "4G",
    "pos_method": "OTDOA",
    "ref_pos": [41.5002, 2.11292, 140.0]
}
```


7.1.19 Interference location

Relative URL	**/g/interference**	
Action	Gets the location of interference sources (if any) surrounding the deployed SNAP GNSS sensors.	
Method	POST	
Access	External (can be used by anyone)	
Authorization	Valid access token needed	
Querystring		
Request body		
Returns	200 OK & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	

Use case	User wants to know if there is any interference source surrounding his GNSS sensors
Example	`https://hansel.rokubun.cat/snap/api/g/interference`

The query will return, upon successful status code, either a JSON object containing the jammer position if detected or a message informing that no jammer was localized, following the structure:

```
{
   "jammer_pos": [<jammer latitude>,
                  <jammer_longitude>,
                  <jammer_height>]
}
```

or

```
{
   "message": "No Jammer was located."
}
```

7.1.20 Located interference historic obtainment

Relative URL	**`/g/interference`**	
Action	Retrieves the historic of located interferences.	
Method	GET	
Access	External (can be used by anyone)	
Authorization	Valid access token needed.	
Querystring	-	-
Request body	-	
Returns	200 OK & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	User wants to know what interference sources were located in the past	

Example	`https://hansel.rokubun.cat/snap/api/g/interference`

The query will return, upon successful status code, either a JSON object containing the jammer position if detected or a message informing that no jammer was localized, following the structure:

```
{
   "jammers": [
                  {
                      "jammer_pos": [
                                          <jammer1 latitude>,
                                          <jammer1 longitude>,
                                          <jammer1 height>
                                      ],
                       "id": <jammer1 id>,
                       "creation_date": <date when jammer 1
was                     detected>
                  },
                  …
                  {
                      "jammer_pos": [
                                          <jammerN latitude>,
                                          <jammerN longitude>,
                                          <jammerN height>
                                      ],
                       "id": <jammerN id>,
                       "creation_date": <date when jammer N
was                     detected>
                  },

}
```

An example of JSON response:

```
{
   "jammers": [
                  {
                      "jammer_pos": [
                                          42.4992,
                                          2.115,
                                          140
                                      ],
                       "id": 1,
                       "creation_date": "2019-11-
22T15:37:19+00:00"
                  },
                  {
```

```
                    "jammer_pos": [
                                    45.3688,
                                    3.663,
                                    70
                                ],
                     "id": 2,
                     "creation_date": "2019-11-
23T19:34:53+00:00"
                },

}
```

7.1.21 Observables computing from live TLE signals

Relative URL	**/c/observables?sensor=<sensor_id>**	
Action	Computes the raw observables from real cellular signal.	
Method	GET	
Access	External (can be used by anyone)	
Authorization	Valid access token needed and the specified sensor must belong to the requesting user.	
Querystring	sensor=<sensor_id>	ID of the sensor whose observables are requested.
Request body	JSON (application/json)	
Returns	201 ACCEPTED & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	User wants to know the observables that a Cellular sensor is gathering	
Example	`https://hansel.rokubun.cat/snap/api/c/observables?sensor=16`	

The type of sensor that is allowed to send that request are only CP sensors, since are the only ones that capture real Cellular signal. The request information includes all the configurable execution parameters, following the structure:

```
{
   "<execution parameter 1>": "<parameter 1 value>",
```

```
   …
   "<execution parameter N>": "<parameter N value>",
}
```

The query will return, upon successful status code, a JSON object containing the ID of the execution that just launched to the platform, following the structure:

```
{
   "execution_id": <execution id>
}
```

Execution parameters can be found in Appendix 2, Table 6.

An example of request body:

```
{
   "bandwidth": 100
}
```

An example of JSON response:

```
{
   "execution_id": 369
}
```

7.1.22 Signal data uploading

Relative URL	**/signal?execution=<exec_id>&ref_time=<ref_time>**	
Action	Adds the samples file for the specified execution.	
Method	POST	
Access	Internal (only accessed by sensors).	
Authorization	Valid access token needed, the execution must belong to the requesting user.	
Querystring	execution=<exec_id>	ID of the execution whose samples are being delivered
	ref_time=<ref_time>	Time in which the samples were gathered. Format should be YYYYMMDDHRMNSC.
Request body	Binary data (application/octet-stream)	

Returns	202 ACCEPTED
	400 BAD REQUEST
	401 UNAUTHORIZED
	404 NOT FOUND
Use case	The FBS has requested a given sensor to provide its location and after the request fetching by the sensor, the latter responds to this action by gathering and sending the signal samples to the testbed.
Example	`https://hansel.rokubun.cat/snap/api/signal?execution=50`

This action is used by sensors to deliver signal samples to the testbed. That is, a given sensor captures GNSS/cellular live signals (as specified in the corresponding execution ID configuration file) and sends the gathered samples to the SNAP service, which will relay these samples to the external cloud platform in charge of the samples processing.

7.1.23 Post-processing recorded GNSS data

Relative URL	**`/g/process`**	
Action	Computes the position for a given raw GNSS samples file along with its configuration	
Method	POST	
Access	External (can be used by anyone)	
Authorization	Valid access token needed.	
Querystring	-	-
Request body	JSON (application/json)	
Returns	200 OK & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	User wants to post-process a previously recorded GNSS raw samples file	
Example	`https://hansel.rokubun.cat/snap/api/g/process`	

The request body includes all the configurable execution parameters under the key `'params'`, following the structure:

```
{
   "<execution parameter 1>": "<parameter 1 value>",
   …
   "<execution parameter N>": "<parameter N value>",
}
```

Also the request body should include the samples file, under the key `'samples'`. Optionally the request body could include a RINEX file for the processing, under the key `'rinex'`. If the RINEX file is not included, the receiver will look for it automatically.

The query will return, upon successful status code, a JSON object containing the ID of the execution that just launched to the platform, following the structure:

```
{
   "execution_id": <execution id>
}
```

Execution parameters can be found in Appendix 2, Table 3.

An example of request body as well as for JSON response are the same as for G position (/g/position) plus the samples file and (optionally) a RINEX file.

7.1.24 Post-processing recorded real Cellular data

Relative URL	**/c/process**	
Action	Computes the observables for LTE real signal samples, given the parameters of the samples.	
Method	POST	
Access	External (can be used by anyone)	
Authorization	Valid access token needed.	
Querystring	-	-
Request body	JSON (application/json)	
Returns	200 OK & JSON (application/json)	

	400 BAD REQUEST
	401 UNAUTHORIZED
	404 NOT FOUND
Use case	User wants to post-process a previously recorded Cellular raw samples file
Example	`https://hansel.rokubun.cat/snap/api/c/process`

The request body includes all the configurable execution parameters under the key `'params'`, following the structure:

```
{
   "<execution parameter 1>": "<parameter 1 value>",
   …
   "<execution parameter N>": "<parameter N value>",
}
```

Also, the request body should include the samples file, under the key `'samples'`.

The query will return, upon successful status code, a JSON object containing the ID of the execution that just launched to the platform, following the structure:

```
{
   "execution_id": <execution id>
}
```

Execution parameters can be found in Appendix 2, Table 6.

An example of request body as well as for JSON response are the same as for GET /c/observables plus the cellular samples file.

7.1.25 Recorded GNSS signal with simulated LTE signal hybridization

Relative URL	**`/h/process`**
Action	Computes the hybridized position given a raw GNSS samples file, a Cellular data simulation and its configuration parameters.
Method	POST
Access	External (can be used by anyone)

Authorization	Valid access token needed.	
Querystring	-	-
Request body	JSON (application/json)	
Returns	200 OK & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	User wants to post-process a previously recorded GNSS raw samples file and hybridize it with a Cellular data simulation	
Example	`https://hansel.rokubun.cat/snap/api/h/process`	

The request body includes all the configurable execution parameters under the key `'params'`, following the structure:

```
{
   "<execution parameter 1>": "<parameter 1 value>",
   …
   "<execution parameter N>": "<parameter N value>",
}
```

Also the request body should include the GNSS raw samples file, under the key `'samples'`. Optionally the request body could include a RINEX file for the processing, under the key `'rinex'`. If the RINEX file is not included, the receiver will look for it automatically.

The query will return, upon successful status code, a JSON object containing the ID of the execution that just launched to the platform, following the structure:

```
{
   "execution_id": <execution id>
}
```

Execution parameters can be found in Appendix 2, Table 12.

An example of request body as well as for JSON response are the same as for H position (/h/position))plus the samples file and (optionally) a RINEX file.

7.1.26 Adding results to SNAP DB

Relative URL	**/results?execution=<exec_id>**	
Action	Adds to database the results for the specified execution.	
Method	POST	
Access	Internal (only accessed by the external service, CloudRx).	
Authorization	Valid access token needed, the execution must belong to the requesting user.	
Querystring	execution=<exec_id>	ID of the execution whose results are being commited
Request body	JSON (application/json)	
Returns	200 OK	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	CloudRx stores all the execution results into SNAP DB	
Example	`https://hansel.rokubun.cat/snap/api/results?execution=50`	

The request body includes the information about the results of the execution, that will be identified by its ID number. The result parameters can be seen in Appendix 2, Table 4 for GNSS executions, Table 7 for real Cellular signal executions, Table 10 for simulated Cellular signal executions and Table 14 for Hybrid executions.

7.1.27 SNAP database result obtainment

Relative URL	**/results**
Action	Gets execution solutions present in the database. – Admin: obtains all the results in the database. – Normal user: obtains the results of his executions only.
Method	GET
Access	External (can be used by anyone)

Authorization	Valid access token needed, execution must belong to the user that requested the results.	
Querystring	(optional) (admin) user=<user_email>	Obtains the execution results of the executions launched by a given user.
	(optional) sensor=<sensor_id>	· Admin: obtains all the results of the executions launched by given sensor. · Normal user: obtains the results of the executions launched by the given sensor if it belongs to the requesting user.
	(optional) execution=<execution_id>	· Admin: obtains the results of the given execution. · Normal user: obtains the results of the given execution if it belongs to the requesting user.
Request body	JSON (application/json)	
Returns	200 OK & JSON (application/json)	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	FBS wants to represent execution results, administrator wants to see an overview of the platform execution results, user wants to see the results of his executions	

Example	`https://hansel.rokubun.cat/snap/api/results`
	`https://hansel.rokubun.cat/snap/api/results?user=email@test.com`
	`https://hansel.rokubun.cat/snap/api/results?sensor=13`
	`https://hansel.rokubun.cat/snap/api/results?execution=50`	

The query will return, upon successful status code, the information about the executions results, following the structure:

- Response JSON for no querystring or querystring user=<user_email>:

```
{
   "results": {
```

```
                "G": [
                        {
                            <GNSS execution 1 results>
                        },
                        …
                        {
                            <GNSS execution N results>
                        }
                    ],
                "CP": [
                        {
                            <Cell execution 1 results>
                        },
                        …
                        {
                            <Cell execution N results>
                        }
                    ],
                "CL": [
                        {
                            <Cell execution 1 results>
                        },
                        …
                        {
                            <Cell execution N results>
                        }
                    ],
                 "H": [
                        {
                            <Hybrid execution 1 results>
                        },
                        …
                        {
                            <Hybrid execution N results>
                        }
                    ]
            }
}
```

- Response JSON for querystring sensor=<sensor_id>: (G sensor)

```
{
   "results": [
                            {
                                   <Execution 1 results>
                            },
                            …
                            {
                                   <Execution N results>
                            }
            ]
}
```

7.1.28 SNAP DB result deletion

Relative URL	**/results?execution=<execution_id>**	
Action	Allows the platform administrators to delete execution results.	
Method	DELETE	
Access	External (can be used by anyone)	
Authorization	Valid access token needed, user must have admin privileges	
Querystring	(optional) (admin) execution=<execution_id>	Deletes the execution results for the execution with ID <execution_id>
	(optional) (admin) sensor=<sensor_id>	Deletes the execution results for all the executions that belong to the sensor with id <sensor_ID>
	(optional) (admin) user=<user_id>	Deletes the execution results for all the executions that belong to the user with id <user_ID >
Request body	-	
Returns	200 OK	
	400 BAD REQUEST	
	401 UNAUTHORIZED	
	404 NOT FOUND	
Use case	System administrator wants to free machine storage	

Example	`https://hansel.rokubun.cat/snap/api/results?user=4`
	`https://hansel.rokubun.cat/snap/api/results?sensor=13`
	`https://hansel.rokubun.cat/snap/api/results?execution=50`	

- Response JSON for querystring execution=<execution_id>: (G execution)

```
{
   "result": {
                    <Execution results>
             }
}
```

If results of one specific execution are not yet available, the value for <execution results> field will follow the structure:

```
{
   "execution_id": <execution id>,
   "execution_state": <state of the execution>
}
```

Examples of JSON responses:

- Example response for no querystring or querystring user=<user_email>:

```
{
   "results": {
                "G": [
                        {
                            "execution_id": 1,
                            "sensor_pos": [41.49963, 2.112638, 176,433],
                            "observables": [23875764, 24678082, 22787820,
21896799, 22897650],
                            "sat_pos":
[[4273454.6,21090102.1,15656972.0],
                                                [15940910,-
14818167.7,15751421.1],

[21454094.27430468.65,14446305.1],

[26334172.9,250734.11,4225974.83],

[11226598.9,12057242.7,20782552]],
                            "cn0": [30.86, 33.65, 30.83, 36.39, 40.25],
                            "prn": [13, 16, 23, 2, 7],
                            "dopp": [2815.69, 12.40, -2209.43, 100.67,
2795.88],
                            "tow": 402547,
                            "detected": null,
                            "jammer_pos": null,
                            "user_id": 1,
                            "sensor_id": 1,
                            "creation_date": "2019-10-
01T22:22:22.0+00:00"
                        },
                        {
                            "execution_id": 2,
                            "execution_state": "in_process",
                        },
                        {
                            "execution_id": 3,
                            "execution_state": "in_queue",
                        }
                    ],
                "CP": [
                        {
                            "execution_id": 19,
                            "snr": [6.98, 2.98, 1.08, 3.85, 10.43]
                            "observables": [90.59, 20.098, 87.667,
50.0874, 20.0858],
                            "bs_id": [1, 5, 7 9, 15],
                            "spectrum_image": true,
                            "spectrum_image_url": http://www.url.com,
                            "creation_date": "2019-12-
13T00:22:13.0+00:00"
```

```
                    },
                    {
                        "execution_id": 12,
                        "execution_state": "failed"
                    }
                ],
            "CL": [
                    {
                        "execution_id": 16
                        "bs_pos":
                        "pos_method": "OTDOA",
                        "scenario": "UMa",
                        "observables": [90.987, 57.675, 30.451,
10.857, 100.8598],
                        "error_var": [1.875, 2.08, 4.09, 2.09, 1.09],
                        "flag_los": true,
                        "snr": [10.50, 3.988, 6.09, 8.987, 3.097],
                        "ref_pos": [41.49989, 2.112643, 170.0],
                        "sensor_pos": [41.49963, 2.112638, 176,433],
                        "sensor_pos_error": 4.076,
                        "creation_date": "2019-10-
01T07:32:43.0+00:00"
                    }
                ],
             "H": [
                    {
                        "execution_id": 17,
                        "sensor_pos": [41.49463, 2.112638, 176,433],
                        "hdop": 1.5,
                        "error_var": 3.65,
                        "creation_date": "2019-10-
01T02:12:03.0+00:00"
                    },
                    {
                        "execution_id": 21,
                        "execution_state": "in_process"
                    }
                ]
            }
}
```


- Example response for querystring sensor=<sensor_id>:


```
{
   "results": {
                [
                    {
                        "execution_id": 1,
                        "sensor_pos": [41.49963, 2.112638, 176,433],
                        "observables": [23875764, 24678082, 22787820,
21896799, 22897650],
                        "sat_pos":
[[4273454.6,21090102.1,15656972.0],
                                        [15940910,-
14818167.7,15751421.1],

[21454094.27430468.65,14446305.1],

[26334172.9,250734.11,4225974.83],
```

```
[11226598.9,12057242.7,20782552]],
                            "cn0": [30.86, 33.65, 30.83, 36.39, 40.25],
                            "prn": [13, 16, 23, 2, 7],
                            "dopp": [2815.69, 12.40, -2209.43, 100.67,
2795.88],
                            "tow": 402547,
                            "detected": null,
                            "jammer_pos": null,
                            "user_id": 1,
                            "sensor_id": 1,
                            "creation_date": "2019-10-
01T22:22:22.0+00:00"
                        },
                        {
                            "execution_id": 2,
                            "execution_state: "in_process"
                        }
                    ]

              }
}
```

- Example response for querystring execution=<execution_id>:

```
{
   "result": {
                    "execution_id": 1,
                    "sensor_pos": [41.49963, 2.112638, 176,433],
                    "observables": [23875764, 24678082, 22787820,
21896799, 22897650],
                    "sat_pos": [[4273454.6,21090102.1,15656972.0],
                                [15940910,-14818167.7,15751421.1],
                                [21454094.27430468.65,14446305.1],
                                [26334172.9,250734.11,4225974.83],
                                  [11226598.9,12057242.7,20782552]],
                      "cn0": [30.86, 33.65, 30.83, 36.39, 40.25],
                      "prn": [13, 16, 23, 2, 7],
                      "dopp": [2815.69, 12.40, -2209.43, 100.67,
2795.88],
                      "tow": 402547,
                      "detected": null,
                      "jammer_pos": null,
                      "user_id": 1,
                      "sensor_id": 1,
                      "creation_date": "2019-10-01T22:22:22.0+00:00
             }
}
```

7.2 Appendix 2: SNAP API I/O

Table 2. Configuration parameters of SNAP-G sensors.

Variable	Description	Type	Value	Size	Unit	Requirement
ion	Enable ION GNSS Metadata Standard decoding file	Bool	{0, 1}	-	-	-
sampling_freq	Sampling frequency	Float	Strictly positive	-	MHz	ion = 0
bandwidth	Bandwidth	Float	Strictly positive	-	MHz	ion = 0
intermediate_freq	Intermediate frequency	Float	-	-	kHz	ion = 0
format	Sample format	String	{IF, IQ, QI, SC16}	-	-	ion = 0
encoding	Encoding	String	{INT, FLOAT, SIGN}	-	-	ion = 0
quantization	Quantization bits	Unsigned int	Strictly positive	-	Bits	ion = 0
delay	File pointer offset	Unsigned int	Positive	-	ms	-
signal_length	Length of the signal to be captured	Unsigned int	Strictly positive	-	ms	-
update_period	Time between /update_request updates	Float	Positive	-	s	-

Table 3. Input configuration parameters of the SNAP-G service.

Variable	Description	Type	Value	Size	Unit	Requirement
nearfar	Enable near-far detector	Bool	{0, 1}	-	-	-
band	GNSS band	String	{1, 5}	-	-	-
system	GNSS system	String	{GPS, Galileo}	-	-	-
num_snap	Number of snapshots	Unsigned int	Strictly positive	-	-	-
delta_snap	Time gap (delta) between snapshots	Unsigned int	Positive	-	ms	-
coh_time	Coherent integration time	Unsigned Int	Strictly positive	-	ms	-
num_noncoh	Number of non-coherent integrations	Unsigned int	Strictly positive	-	-	-
gpdit	Enable Generalized Post-Detection Integration Truncated Technique	Bool	{0, 1}	-	-	-
sam	Enable multipath analysis (SAM metric)	Bool	{0, 1}	-	-	-
dec_rate	Decimation rate	Unsigned int	Strictly positive	-	-	-

interference	Enable interference detection and localization	Bool	{0, 1}	-	-	-
manual_sat_search	Enable manual satellite search	Bool	{0, 1}	-	-	-
sat_list	Satellites to be searched	Integer array	-	1 x N_sat	-	manual_sat_search = 1
assisted_dopp	Enable (manually) assisted Doppler frequency	Bool	{0, 1}	-	-	if manual_sat_search = 0, assisted_dopp must be 1
generate_agnss	Enable automatic generation of assistance data (visSat and dopp#)	Bool	{0, 1}	-	-	assisted_dopp = 1
max_doppler_search	Maximum Doppler frequency search	Unsigned int	-	-	kHz	assisted_dopp = 0
visible_sat	Visible satellites	Unsigned int array	-	1 x N_sat	-	generate_agnss = 0 (so, assisted_dopp = 1), sorted with dopp
dopp	Doppler frequency of each visible satellite	Float array	-	1 x N_sat	kHz	generate_agnss = 0

						(so, assisted_dopp = 1), sorted with visible_sat
ref_pos	Reference position latitude, longitude and height coordinates. Has to be ±75km accurate to solve the navigation equations. In testing operations, should be the true position so position error becomes the actual error between the estimated position and the true one.	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x 1	º, º, m	-

Table 4. Output parameters of the SNAP-G service.

Variable	Description	Type	Value	Size	Unit	Requirement
sensor_pos	Sensor position latitude, longitude and height coordinates	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x 1	º, º, m	-
tow	Sensor time of week	Float	0 to 604799	-	s	-
cn0	Satellites C/N0	Float array	Positive	1 x N_sat	dBHz	sorted with observables

observables	GNSS corrected pseudoranges	Float array	Positive	1 x N_sat	m	sorted with observables
sat_pos	Satellite positions in ECEF coordinates (x, y, z)	Float matrix	x, y and z strictly positive	3 x N_sat	[m, m, m]	sorted with observables
detected	Detection of interference/jammer	Bool	{0, 1}	-	-	(optional)
sensor_pow	Power values of the samples	Float matrix	Positive	1 x N_pow	W	(optional) detected = 1
prn	PRN identification for each satellite in acquisition stage	Integer matrix	Valid PRN	1 x N_sat	-	sorted with observables
prn_pvt	PRN identification for each satellite in PVT stage	Integer matrix	Valid PRN	1 x N_sat	-	
dopp	Doppler shift for each satellite	Float matrix	Float	1 x N_sat	Hz	sorted with observables
creation_date	Timestamp when the execution results where stored	Timestamp	-	-	-	-

Table 5. Configuration parameters of SNAP-C (Physical) sensors.

Variable	Description	Type	Value	Size	Unit	Requirement
sampling_freq	Sampling frequency	Float	Strictly positive	-	MHz	-

format	Sample format	String	{IF, IQ, QI, SC16}	-	-	-
encoding	Encoding	String	{INT, FLOAT, SIGN}	-	-	-
quantization	Quantization bits	Unsigned int	Strictly positive	-	Bits	-
update_period	Time between /update_request updates	Float	Positive	-	s	-

Table 6. Input configuration parameters of the "Process real cellular signal" of SNAP-C service.

Variable	Description	Type	Value	Size	Unit	Requirement
bandwidth	System bandwidth	Float	Strictly positive	-	MHz	-

Table 7. Output parameters of the "Process real cellular signal" of SNAP-C service.

Variable	Description	Type	Value	Size	Unit	Requirement
snr	Signal-to-Noise Ratio	Float array	-	1 x N_bs	dB	-
observables	Cellular observables	Float array	Positive	1 x N_bs	-	-
bs_id	Physical cell identifier	Integer array	Valid identifiers	1 x N_bs	-	-

spectrum_image	Indicates if a time-frequency spectrum is generated	Bool	{0,1}	-	-	(optional)
spectrum_image_url	Time-frequency spectrum image url	String	-	-	-	(optional) spectrum_image = 1

Table 8. Configuration parameters of SNAP-C (Logical) sensors.

Variable	Description	Type	Value	Size	Unit	Requirement
SNAP-C Logical sensors have no inner configurations since they are just conceptual sensors, not real ones						

Table 9. Input configuration parameters of the "Get cellular PVT" of SNAP-C service.

Variable	Description	Type	Value	Size	Unit	Requirement
deployment	Deployment mode	Bool	{0, 1}	-	-	-
ref_pos	Reference position in latitude, longitude, height coordinates True position of the sensor so the generated Cellular data is consistent.	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x 1	º, º, m	-

set_ref_coord	Use reference coordinate for network deployment	Bool	{0, 1}	-	-	deployment = 1
ref_coord	Reference coordinate position for network deployment	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x 1	º, º, m	deployment = 1, set_ref_coord = 1
ratio_rural	Probability or ratio to be RMa (rural) deployment	Float	0 to 1	-	%	deployment = 1, set_ref_coord = 0, ratio_rural + ratio_suburban + ratio_urban = 1
ratio_suburban	Probability or ratio to be UMa (sub-urban) deployment	Float	0 to 1	-	%	deployment = 1, set_ref_coord = 0, ratio_rural + ratio_suburban + ratio_urban = 1
ratio_urban	Probability or ratio to be UMi (urban/dense urban) deployment	Float	0 to 1	-	%	deployment = 1, set_ref_coord = 0, ratio_rural + ratio_suburban + ratio_urban = 1
isd_rural	Inter-site distance in RMa (rural)	Float	Positive	-	m	deployment = 1

isd_suburban	Inter-site distance in UMa (sub-urban)	Float	Positive	-	m	deployment = 1
isd_urban	Inter-site distance in UMi (urban/dense urban)	Float	Positive	-	m	deployment = 1
scenario	Type of scenario for ref_pos	String array	{UMi, UMa, RMa}	-	-	(deployment = 0) or (deployment = 1 and set_ref_coord = 1)
prs_bandwidth	System Positioning Reference Signal (PRS) bandwidth	Float	{1.4, 5, 10, 20, 50, 100}	-	MHz	-
sync_error	Network synchronization	Float	{0, 50}	-	ns	-
num_measurement	Number of measurement integrations per position fix	Unsigned int	{1, 2, 5, 10}	-	-	-
max_bs	Maximum number of BSs for positioning	Unsigned int	1 to 10	-	-	-
pos_method	Positioning method	String	{OTDOA, CellId}	-	-	-
network_type	Network type (4G/5G)	String	{4G, 5G}	-	-	-
carrier_freq	Carrier frequency	Float	{0.9, 2, 4, 6}	-	GHz	-

In addition to these parameters, there is the possibility to upload a KML file containing real cell. The real cell locations should be carefully provided with the corresponding physical cell ID (PCI), which matches the PCI of the acquired LTE signals in SNAP-C.

Table 10. Output parameters of the "Get cellular PVT" of SNAP-C service.

Variable	Description	Type	Value	Size	Unit	Requirement
bs_pos	BS location in latitude, longitude and height coordinates	Float matrix	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x N_bs	[º, º, m]	-
pos_method	Positioning method	String	{OTDOA, CellId}	-	-	-
scenario	Type of scenario for ref_pos	String array	{UMi, UMa, RMa}	-	-	-
observables	Cellular observables	Float array	Positive	1 x N_bs	[m]	-
error_var	Cellular observables error variances	Float array	Positive	1 x N_bs	m^2	-
flag_los	Flag of cellular observables in LoS conditions	Bool	{0, 1}	-	-	-
snr	Signal-to-noise ratio	Float array	Signed	1 x N_bs	dB	-
ref_pos	Reference position in latitude, longitude, height coordinates	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180,	3 x 1	º, º, m	-

			height positive			
sensor_pos	Sensor position in latitude, longitude, height coordinates	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x 1	º, º, m	-
sensor_pos_error	Sensor position error (in the horizontal coordinates)	Float	Positive	-	m	-

Table 11. Configuration parameters of SNAP-H sensors.

Variable	Description	Type	Value	Size	Unit	Requirement
SNAP-H sensor configuration parameters are the same as SNAP-G sensor parameters + SNAP-C (logical) sensor parameters, since they have to launch a GNSS execution to obtain GNSS observables and launch a (simulated) Celullar execution to obtain Cellular observables. Since there is no SNAP-C (logical) sensor parameters, this configuration remains the same as SNAP-G sensor parameters.						
ion	Enable ION GNSS Metadata Standard decoding file	Bool	{0, 1}	-	-	-
sampling_freq	Sampling frequency	Float	Strictly positive	-	MHz	ion = 0
bandwidth	Bandwidth	Float	Strictly positive	-	MHz	ion = 0

intermediate_freq	Intermediate frequency	Float	-	-	kHz	ion = 0
format	Sample format	String	{IF, IQ, QI, SC16}	-	-	ion = 0
encoding	Encoding	String	{INT, FLOAT, SIGN}	-	-	ion = 0
quantization	Quantization bits	Unsigned int	Strictly positive	-	Bits	ion = 0
delay	File pointer offset	Unsigned int	Positive	-	ms	-
update_period	Time between /update_request updates	Float	Positive	-	s	-

Table 12. External parameters of the SNAP-H service (input from FBS).

Variable	Description	Type	Value	Size	Unit	Requirement
Configuration parameters to launch a hybrid execution (h/position) should be the necessary to launch a GNSS execution (g/position) + a Cellular execution (c/position) in order to obtain the observables to hybridize them. (remains necessary to define if some parameter should have a default value in order to avoid the user entry)						
nearfar	Enable near-far detector	Bool	{0, 1}	-	-	-
band	GNSS band	String	{1, 5}	-	-	-
system	GNSS system	String	{GPS, Galileo}	-	-	-

num_snap	Number of snapshots	Unsigned int	Strictly positive	-	-	-
delta_snap	Time gap (delta) between snapshots	Unsigned int	Positive	-	ms	-
coh_time	Coherent integration time	Unsigned Int	Strictly positive	-	ms	-
num_noncoh	Number of non-coherent integrations	Unsigned int	Strictly positive	-	-	-
gpdit	Enable Generalized Post-Detection Integration Truncated Technique	Bool	{0, 1}	-	-	-
sam	Enable multipath analysis (SAM metric)	Bool	{0, 1}	-	-	-
dec_rate	Decimation rate	Unsigned int	Strictly positive	-	-	-
interference	Enable interference detection and localization	Bool	{0, 1}	-	-	-
manual_sat_search	Enable manual satellite search	Bool	{0, 1}	-	-	-
sat_list	Satellites to be searched	Integer array	-	1 x N_sat	-	manual_sat_search = 1
assisted_dopp	Enable (manually) assisted Doppler frequency	Bool	{0, 1}	-	-	if manual_sat_search = 0, assisted_dopp must be 1

generate_agnss	Enable automatic generation of assistance data (visSat and dopp#)	Bool	{0, 1}	-	-	assisted_dopp = 1
max_doppler_search	Maximum Doppler frequency search	Unsigned int	-	-	kHz	assisted_dopp = 0
visible_sat	Visible satellites	Unsigned int array	-	1 x N_sat	-	generate_agnss = 0 (so, assisted_dopp = 1), sorted with dopp
dopp	Doppler frequency of each visible satellite	Float array	-	1 x N_sat	kHz	generate_agnss = 0 (so, assisted_dopp = 1), sorted with visible_sat
ref_pos	Reference position in latitude, longitude, height coordinates True position of the sensor so the generated Cellular data is consistent.	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x 1	º, º, m	-
deployment	Deployment mode	Bool	{0, 1}	-	-	-
set_ref_coord	Use reference coordinate for network deployment	Bool	{0, 1}	-	-	deployment = 1

ref_coord	Reference coordinate position for network deployment	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x 1	º, º, m	deployment = 1, set_ref_coord = 1
ratio_rural	Probability or ratio to be RMa (rural) deployment	Float	0 to 1	-	%	deployment = 1, set_ref_coord = 0, ratio_rural + ratio_suburban + ratio_urban = 1
ratio_suburban	Probability or ratio to be UMa (sub-urban) deployment	Float	0 to 1	-	%	deployment = 1, set_ref_coord = 0, ratio_rural + ratio_suburban + ratio_urban = 1
ratio_urban	Probability or ratio to be UMi (urban/dense urban) deployment	Float	0 to 1	-	%	deployment = 1, set_ref_coord = 0, ratio_rural + ratio_suburban + ratio_urban = 1
isd_rural	Inter-site distance in RMa (rural)	Float	Positive	-	m	deployment = 1
isd_suburban	Inter-site distance in UMa (sub-urban)	Float	Positive	-	m	deployment = 1

isd_urban	Inter-site distance in UMi (urban/dense urban)	Float	Positive	-	m	deployment = 1
scenario	Type of scenario for each of the entries of user_pos	String array	{UMi, UMa, RMa}	1 x N_pos	-	(deployment = 0) or (deployment = 1 and set_ref_coord = 1)
prs_bandwidth	System Positioning Reference Signal (PRS) bandwidth	Float	{1.4, 5, 10, 20, 50, 100}	-	MHz	-
sync_error	Network synchronization	Float	{0, 50}	-	ns	-
num_measurement	Number of measurement integrations per position fix	Unsigned int	{1, 2, 5, 10}	-	-	-
max_bs	Maximum number of BSs for positioning	Unsigned int	1 to 10	-	-	-
pos_method	Positioning method	String	{OTDOA, CellId}	-	-	-
network_type	Network type (4G/5G)	String	{4G, 5G}	-	-	-
carrier_freq	Carrier frequency	Float	{0.9, 2, 4, 6}	-	GHz	-

Table 13. Internal input parameters of the SNAP-H service (communication between SNAP-G & SNAP-C to SNAP-H).

Variable	Description	Type	Value	Size	Unit	Requirement
Internal parameters for a hybridized position (h/position) will be the ones coming from GNSS execution and Cellular execution and will only be used internally in SNAP.						
ref_pos	Reference user position in latitude, longitude and height coordinates	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x 1	[º, º, m]	External input from h/position
bs_pos	BS locations	Float matrix	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x N_bs	[º, º, m]	Output from c/position
pos_method	Positioning method	String	{OTDOA, CellId}	-	-	Output from c/position
cellular_observables	Cellular observables	Float array	Positive	1 x N_bs	Depends on pos_method	Output from c/position
error_var	Cellular observables error variances	Float array	Positive	1 x N_bs	m^2	Output from c/position
gnss_observables	GNSS corrected pseudoranges	Float array	Positive	1 x N_sat	m	Output from g/position

cn0	CN0 of the satellites	Float array	-	1 x N_sat	dBHz	Output from g/position
sat_pos	Satellite positions in ECEF coordinates (x, y, z)	Float matrix	x, y and z strictly positive	3 x N_sat	[m, m, m]	Output from g/position

Table 14. Output parameters of the SNAP-H service.

Variable	Description	Type	Value	Size	Unit	Requirement
sensor_pos	Sensor position	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x 1	[º, º, m]	-
hdop	Horizontal Dilution of Precision	Float	Strictly positive	-	-	-
sensor_pos_error	Sensor position error (in the horizontal coordinates)	Float	Positive	-	m	-

Table 15. Input parameters of the interference location service.

Variable	Description	Type	Value	Size	Unit	Requirement
Internal parameters for the jammer location service (g/interference) will be coming from SNAP-DB and will only be used internally in SNAP-G interference module.						

sensor_positions	Latitude, longitude and height coordinates of each sensor.	Float matrix	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x N_sensors	[º, º, m]	Positions should be the ones coming from the sensors possible affected by jammers.
sensor_powers	RSS samples levels from each sensor	Float matrix	Positive number	N_RSS x N_sensors	W	Sorted with sensor_positions.

Table 16. Output parameters of the interference location service.

Variable	Description	Type	Value	Size	Unit	Requirement
Internal parameters for a hybridized position (h/position) will be the ones coming from GNSS execution and Cellular execution and will only be used internally in SNAP.						
jammer_pos	Latitude, longitude and height coordinates of the jammer	Float array	0≤ latitude ≤ 90, -180 ≤ longitude ≤ 180, height positive	3 x 1	º, º, m	-

7.3 Appendix 3: Software licenses

Software	License Type	License Website description
Docker	Apache 2.0	https://github.com/docker/docker/blob/master/LICENSE
Gitlab CE	MIT	https://gitlab.com/gitlab-org/gitlab-foss/blob/master/LICENSE
Python	BSD 3-Clause	https://docs.python.org/3/license.html
Flask	BSD 3-Clause	https://github.com/pallets/flask/blob/master/LICENSE.rst
Flask_restful	BSD 3-Clause	https://github.com/flask-restful/flask-restful/blob/master/LICENSE
Flask_script		https://github.com/smurfix/flask-script/blob/master/LICENSE
Flask_migrate	MIT	https://github.com/miguelgrinberg/Flask-Migrate/blob/master/LICENSE
Flask_sqlalchemy	BSD 3-Clause	https://github.com/pallets/flask-sqlalchemy/blob/master/LICENSE.rst
Flash_marshmallow		https://flask-marshmallow.readthedocs.io/en/latest/license.html
Flask-Bcrypt		https://github.com/maxcountryman/flask-bcrypt/blob/master/LICENSE
Bcrypt		https://github.com/grnet/python-bcrypt/blob/master/LICENSE
numpy	BSD	https://numpy.org/license.html
marshmallow		https://marshmallow.readthedocs.io/en/stable/license.html
marshmallow-sqlalchemy	MIT	https://github.com/marshmallow-code/marshmallow-sqlalchemy/blob/dev/LICENSE
requests	Apache 2.0	https://github.com/psf/requests/blob/master/LICENSE
psycopg2	GNU Lesser Public License	http://initd.org/psycopg/license/
passlib		https://passlib.readthedocs.io/en/stable/copyright.html
sqlalchemy	MIT	https://docs.sqlalchemy.org/en/13/copyright.html
boto3	Apache 2.0	https://github.com/boto/boto3/blob/develop/LICENSE