TKN Telecommunication Networks Group

Technical University Berlin

Telecommunication Networks Group

Data Management Services for Evaluation of RF-based Indoor Localization

Filip Lemic and Vlado Handziski

lemic@tkn.tu-berlin.de

Berlin, July 2014

TKN Technical Report TKN-14-002

TKN Technical Reports Series

Editor: Prof. Dr.-Ing. Adam Wolisz

Abstract: This report presents R2DM (Raw Ranging Data Management) and PDM (Processed Data Management) services. R2DM service is a web service developed for storing, managing and accessing the raw data from indoor localization benchmarking experiments. By raw ranging data we mean the measured physical values used as input for the localization algorithms, such as Angle of Arrival (AoA), Time of Arrival (ToA) and Received Signal Strength (RSS). Storing the raw data of indoor localization benchmarking experiments, together with the ground truth location and time of where and when the raw data was obtained and the device used for obtaining the data, gives the possibility of reusing the same raw data for benchmarking other indoor localization algorithms, thus achieving entirely objective comparison of performance of different algorithms. PDM service is the web service developed for storing the processed data, i.e. results of indoor localization benchmarking experiments, namely sets of ground truths and location estimates, together with latencies of location estimation, power consumptions, etc. Having the database of benchmarked solutions gives users the possibility of comparing their solutions with the performance of other benchmarked solutions and gives them the possibility of finding the best solution for their requirements.

The report begins with the presentation of usual procedures (multilateration / multiangulation, fingerprinting, proximity) of estimating indoor location using usual types of raw data, namely RSS (Received Signal Strength), ToA (Time of Arrival) and AoA (Angle of Arrival). Moreover, it gives the design overview of developed services and it puts these services in the wider context of experimental benchmarking of indoor localization. Furthermore, it gives the detailed overview of the R2DM and PDM architectures, the hierarchy of data and metadata storage, and the description of the used message types. Finally, it provides descriptions of the functions that APIs of both services provide with examples of their usage.

Keywords: raw ranging data, benchmarking experiment indoor localization, Received Signal Strength Indicator, Angle of Arrival, Time of Arrival, RESTful, Protocol Buffers, MongoDB, web service, Amazon, HTTP requests

Note: This document gives the overview of the current state of the R2DM and PDM services. It should be expected that both services will change and update in time. According to the changes and additions in the services, the document will be updated. Be sure to always use the version of the document related to the same version of the R2DM and PDM services.

Acknowledgments: This work has been partially funded by the European Commission (FP7-ICT-FIRE) within the project EVARILOS (grant No. 317989).

Contents

Chapter 1

Introduction

People spend most of their time inside buildings and indoor environments. Due to the poor performance of GPS (Global Positioning System) in the indoor and urban environments, a lot of research effort has been pushed towards alternative techniques for indoor localization. There has been a number of different indoor localization solutions proposed in the research community. Different indoor localization solutions are usually presented together with the evaluation of their performance. Unfortunately, their performance results are usually achieved in hardly repeatable manner, which makes them incomparable with other benchmarks. To make the comparison objective, we present the R2DM (Raw Ranging Data Management) service for managing raw ranging data (RSSI, ToA, AoA). Our intention is to collect and offer to the research community different types of usually used raw ranging data, together with the ground truth location and time where and where the data was obtained and the device used for collecting the data, and R2DM service will serve as an enabler for this purpose. Researchers will then be able to use the same raw ranging data, offered in a simple way through the R2DM service, for benchmarking their algorithms and solutions. Furthermore, we present PDM (Processed Data Management) service for storing the results of benchmarking experiments, i.e. location estimates with ground truths, latencies of indoor localization, power consumptions, etc. The data stored in the PDM service will be publicly available and users will be able to compare the performance of different indoor localization algorithms and solutions in one place.

R2DM and PDM services are a Python based RESTful APIs (Application Programming Interface) running on the EC2 (Elastic Compute Cloud) instance in the AWS (Amazon Web Services). The raw ranging data and results of benchmarking experiments are stored in the cloud using a NoSQL based database MongoDB. The message types are defined using the Protocol Buffer message definition and serialization format and transferred as binary streams provided by Protocol Buffer serializer. This type of R2DM and PDM architecture has numerous advantages such as extensibility, reliability, changeability, simplicity of usage, etc. The R2DM and PDM services give a possibility of storing, retrieving, changing or deleting the data using only standard HTTP requests.

This work is structured as follows. Chapter II gives a description of how different types of raw ranging data are used for estimating location. We find it necessary to give a short overview of most common approaches in using different types of ranging data for localization procedure before going deeper into the R2DM and PDM services. Chapter III gives the overview of the whole indoor localization benchmarking system we envision and it puts R2DM and PDM services in the higher context of benchmarking of indoor localization algorithms and solutions. Chapter IV gives the architectural overview of the services, together with the implementation ideas that were leading us during the development. Chapter V and VI respectively present R2DM and PDM services in more details. These chapters give the overview of the used message types and provided API functions. Finally, Chapter VII concludes the report and gives directions for the future work.

Chapter 2

Usage of Raw Ranging Data for Indoor Localization

Indoor localization solutions can, based on the signal types used for gathering localization data, roughly be classified as infra-red, ultrasound, ultra-wideband (UWB), and radio-frequency (RF) based [1]. Technology that can be used for RF-based indoor localization in the 2.4 GHz ISM (Industrial, Scientific and Medical) band is divided into WiFi (IEEE 802.11), ZigBee (IEEE 802.15.4), and Bluetooth (IEEE 802.15.1). Different types of metrics derived from the signal can be used for indoor localization procedure and we consider them as the raw ranging data of indoor localization. Based on the used signal metrics indoor localization solutions can be classified as AoA (Angle of Arrival), ToA (Time of Arrival), TDoA (Time Difference of Arrival) and RSSI (Received Signal Strength Indicator) based. Metric processing procedures used for indoor localization can also vary. Based on the metric processing method indoor localization solutions can be classified as proximity, multilateration / multiangulation and fingerprinting based [2]. Proximity solutions use different signal metrics for reporting the proximity of user's device and an AP (Anchor Point) [3], but those solutions do not have the practical usage in precise indoor localization. Multilateration / multiangulation based indoor localization solutions are using geometrical dependencies between user's device and APs for estimating location. Finally, fingerprinting, also known as scene analysis, computes the similarities between stored fingerprints and the user's fingerprints for estimating location.

Usual indoor localization procedure can be described as shown in Figure 2.1. In the first step of the procedure a measurement of some physical values of the environment is taken (e.g. signal power, arrival time). Physical values are then processed or filtered in order to obtain the raw data that will be used as the input to the processing algorithm. Processing algorithm here can be considered in an relatively broad scope, including e.g. pattern matching, geometrical calculation and possibly post-processing such as kNN method. The output of the processing algorithm is the location estimate.

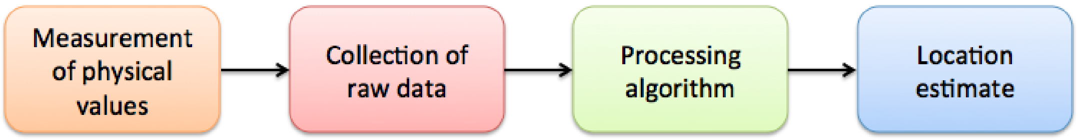

Figure 2.1: Steps of indoor localization procedure at one measurement location

Bellow in text we try to detect the usual processing procedures (multilateration, multiangulation, fingerprinting) that use different types of raw data for RF-based indoor localization. As mentioned before, as raw ranging data for indoor localization we consider RSSI, ToA and AoA information.

The message types for storing the raw ranging data and experimental results in both services will be defined according to this classification, although the defined messages can easily be extended or changed.

2.1 Multilateration and Multiangulation

Multilateration and multiangulation are range-based procedures that use the geometrical dependencies between client and APs (Anchor Points) for indoor localization. In the multilateration information used for localization are the distances between the client's device (device that needs to be localized) and all APs used for localization. In order to obtain the distances between the client and APs different signal metrics can be used, such as received signal strength levels (RSSI), times of arrival (ToA), and time differences of arrival (TDoA). On the other side, multiangulation is using the angles of arrival (AoA) between the client and APs for localization procedure.

2.1.1 RSSI-based Multilateration

RSSI-based multilateration indoor localization algorithms are mapping RSSI values received from APs to distances, i.e. RSSI is considered as the raw data type, while multilateration is the processing procedure that gives the location estimate. The detailed description of calculating the unknown users position is given in [4]. Assume that there are N anchors with known positions (x_i,y_i) and the goal is to determine the user's unknown position (x_u,y_u). Using the Pythagoras theorem, it is possible to write the problem as a set of N equations as follows:

$$(x_i - x_u)^2 + (y_i - y_u)^2 = r_i^2, i = 1, ..., N. \tag{2.1}$$

The way to solve this set of equations is to to remove quadratic terms x_u^2 and y_u^2 which can be done by subtracting the last equation from the previous ones. After the subtractions and rearrangements, the terms can be rewritten in a matrix form as follows:

$$2 \cdot \begin{bmatrix} x_n - x_1 & y_n - y_1 \\ \vdots & \vdots \\ x_n - x_{n-1} & y_n - y_{n-1} \end{bmatrix} \cdot \begin{bmatrix} x_u \\ y_u \end{bmatrix} = \begin{bmatrix} (r_1^2 - r_n^2) - (x_1^2 - x_n^2) - (y_1^2 - y_n^2) \\ \vdots \\ (r_{n-1}^2 - r_n^2) - (x_{n-1}^2 - x_n^2) - (y_{n-1}^2 - y_n^2) \end{bmatrix} \tag{2.2}$$

For this system of linear equations the solution that minimizes the mean square error (MSE) is the pair (x_u, y_u) that minimizes $||\mathbf{Ax\text{-}b}||_2$. This mathematical problem is possible to solve using various mathematical methods, and one example is QR-factorization (substituting $\boldsymbol{A} = \boldsymbol{QR}$, where $\boldsymbol{Q}$ is an orthonormal and $\boldsymbol{R}$ an upper triangular matrix).

2.1.2 ToA-based Multilateration

Time of arrival (ToA) technology is well known and frequently used raw data for obtaining range information by using signal propagation time and it can be combined with multilateration as teh procesing procedure for estimating location. The most spread and well known localization system that uses ToA techniques is GPS (Global Positioning System). The assumption of the ToA technique is that the signal propagates in the space with the constant speed (v). Most of the research indicate that the propagation speed is 2/3 of the speed of light [4]. By measuring the propagation time ($t_{propagation}$)

between the transmitter (anchor point) and the receiver (client) it is possible to determine their distance.

$$d = v \cdot t_{propagation} = \frac{2}{3} \cdot c \cdot t_{propagation} \tag{2.3}$$

After estimating the distance between the client and anchor points the procedure of determining the position of the client is multilateration, as presented in previous section. The main advantage of using Time of Arrival information for indoor localization purposes is their insensitivity to the environmental noise which makes it more accurate than RSS indicators based localization. The main drawback of the ToA-based indoor localization is a need for accurate synchronisation between the receiver and transmitter in order to obtain accurate propagation time. That kind of additional hardware capable of precise synchronisation is time and energy consuming, as well as expensive.

Time difference of arrival technique is quite similar to ToA. The only difference in the TDoA approach is that the localization system assumes the presence of two simultaneously transmitted signals, with different propagation speeds (v_1 and v_2). The time difference (t_{dif}) between those two signals is then used for determining the distance between transmitter and receiver (d). The advantage gained from this approach is that there is no need for the synchronization hardware. Still, additional hardware is needed for the generation of two simultaneously emitted signals of different speed.

$$t_{dif} = t_2 - t_1 = \frac{d}{v_2} - \frac{d}{v_1} \tag{2.4}$$

$$d = \frac{v_1 v_2}{v_1 - v_2} t_{difference} \tag{2.5}$$

2.1.3 AoA-based Multiangulation

AoA-based indoor localization uses multiangluation procedure for processing the angles between the client and APs in order to derive client's location. AoA is then considered as the raw data used in the indoor localization. AoA-based localization also assumes specialized hardware in order to determine the angles of arrival of the signal. Using the angles of signals' arrival to determine the position of the client is called multiangulation. Let the unknown position of the client in 2D environment be determined with the coordinates (x_u, y_u). Angle of arrival from access point with coordinates (x_i, y_i) is α_i. From here, unknown clients coordinates can be expressed as:

$$x_u = x_i + d_{i,u} \cdot \cos \alpha_i, i = 1, ..., N \tag{2.6}$$

$$y_u = y_i + d_{i,u} \cdot \sin \alpha_i, i = 1, ..., N. \tag{2.7}$$

where $d_{i,u}$ is the range between the access point i and the client. The problem above can be written in the matrix form:

$$\underbrace{\left[\begin{array}{cc} 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \\ \hline 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \end{array}\right]}_{\boldsymbol{H}_A} \underbrace{\begin{bmatrix} x_u \\ y_u \end{bmatrix}}_{\boldsymbol{x}_u} = \underbrace{\left[\begin{array}{c} x_1 \\ \vdots \\ x_N \\ \hline y_1 \\ \vdots \\ y_N \end{array}\right]}_{\boldsymbol{v}_{xy}} + \underbrace{\left[\begin{array}{ccc} \cos\alpha_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \cos\alpha_N \\ \hline \sin\alpha_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sin\alpha_N \end{array}\right]}_{\boldsymbol{G}_a} \underbrace{\begin{bmatrix} d_{1,u} \\ \vdots \\ d_{N,u} \end{bmatrix}}_{\boldsymbol{d}} \tag{2.8}$$

Matrix equation can be simplified as:

$$\hat{\boldsymbol{x}}_u = \boldsymbol{H}_A^{\dagger}(\boldsymbol{v}_{xy} + \hat{\boldsymbol{G}}_\alpha d) \tag{2.9}$$

Usually, ranges $d_{i,u}$ are unknown and must be estimated from the angles and anchor coordinates. If ranges are known, they may be just used without estimation. From here, it is possible to write $2N$ equations based on pairwise coordinate differences. That kind of the over-determined system of equations can be solved using the matrix-vector form. The matrix-vector form is given as follows:

$$\hat{\boldsymbol{d}} = \hat{\boldsymbol{H}}_\alpha^{\dagger} \boldsymbol{u}_{xy} \tag{2.10}$$

$$\underbrace{\left[\begin{array}{ccccc} -\cos\alpha_1 & \cos\alpha_2 & 0 & \dots & 0 \\ 0 & -\cos\alpha_2 & \cos\alpha_3 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \ddots & \cos\alpha_N \\ \cos\alpha_1 & 0 & 0 & \dots & -\cos\alpha_N \\ \hline -\sin\alpha_1 & \sin\alpha_2 & 0 & \dots & 0 \\ 0 & -\sin\alpha_2 & \sin\alpha_3 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \ddots & \sin\alpha_N \\ \sin\alpha_1 & 0 & 0 & \dots & -\sin\alpha_N \end{array}\right]}_{\boldsymbol{H}_\alpha} \underbrace{\begin{bmatrix} d_{1,u} \\ \vdots \\ d_{N,u} \end{bmatrix}}_{\boldsymbol{d}} = \underbrace{\left[\begin{array}{c} x_1 - x_2 \\ \vdots \\ x_{N-1} - x_N \\ \hline y_1 - y_2 \\ \vdots \\ y_{N-1} - y_N \end{array}\right]}_{\boldsymbol{u}_{xy}} \tag{2.11}$$

From the above equations it is possible to derive the clients position as:

$$\hat{\boldsymbol{x}}_u = \boldsymbol{H}_A^{\dagger}(\boldsymbol{v}_{xy} + \hat{\boldsymbol{G}}_\alpha \hat{\boldsymbol{H}}_\alpha^{\dagger} u_{xy}) \tag{2.12}$$

2.2 RSSI fingerprinting

Scene analysis or fingerprinting is a different processing procedure from previously described one. Fingerprinting is usually divided in two phases. First phase is learning, *offline* or training phase. During that phase some characteristics of the environment are collected. Usually they are referred to as fingerprints. Fingerprints of different locations can be collections of RSSI indicators, signal-to-noise ratios (SNRs), estimated signal probability distributions, etc. The most commonly used and the most promising approach is fingerprinting using signals' RSSI values, i.e. RSSI is considered as the raw data for the processing procedure. Second phase is running, *online* or runtime phase. In this phase client collects runtime fingerprints in the same manner as in the training phase and compares them with the training dataset. Based on the defined criterion some location (usually called cell) from the training dataset is reported as the client's location.

Chapter 3

Design Overview of Desired System

The envisioned system for indoor localization benchmarking is presented in Figure 3.1. Given the information about ground truth location, Executor should request raw data or location estimate from the system under test (SUT). Executor can then store the raw data into the R2DM service, together with the information about ground truth location, i.e. the location where a measurement was taken. Further, executor can store the location estimate, together with the ground truth, into the PDM service. Moreover, executor can calculate metrics from the obtained location estimates. Metrics that can be calculated are geometrical and room level accuracy, latency of indoor localization and energy consumption. Central engine communicates with the SUT over HTTP protocol. The minimum requirement for the SUT is to provide the location estimation when requested. Finally, in order to be able to store raw ranging data, SUT should also provide it when requested.

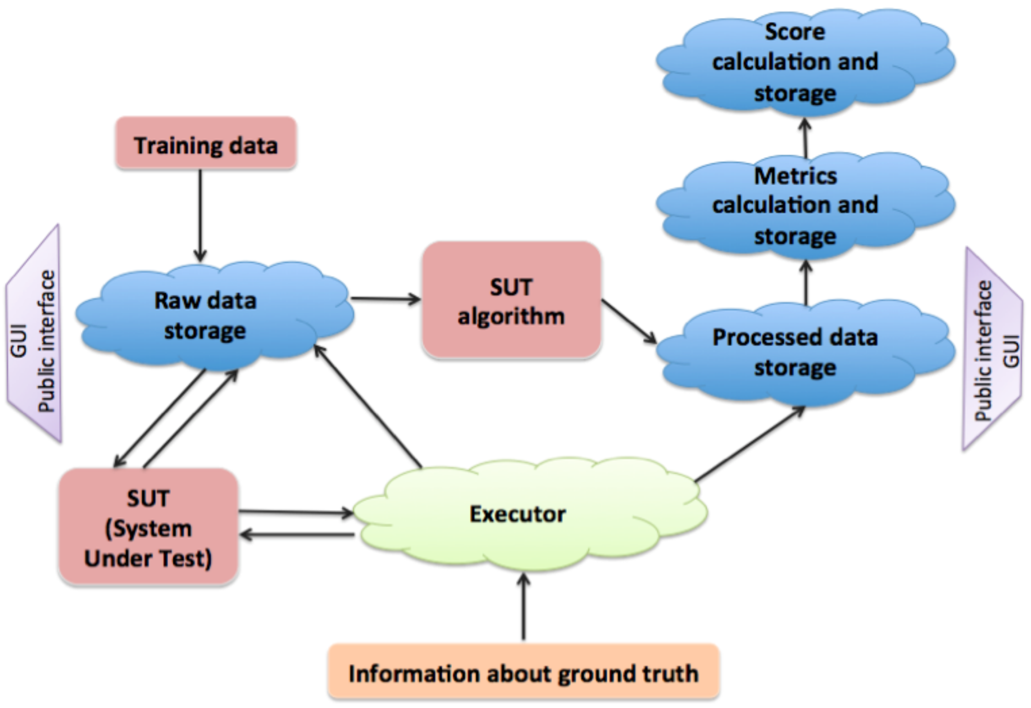

Figure 3.1: Overview of the system design

Raw data stored in the R2DM service can reused by different processing procedures (SUT algorithms). For example, in fingerprinting based indoor localization the whole training and evaluation dataset can be stored into the R2DM service, and then different processing methods can be applied for obtaining location estimates. These estimates can then be stored in the Processed Data Management (PDM) service. Different metrics can be calculated, so different algorithms can easily be compared on the same set of raw data, making the comparison objective.

This section presented a general overview of the envisioned indoor localization benchmarking system. This system is envisioned to work together with the interference generation and monitoring system, and with the autonomous mobility system provided by the robotic platform. Further sections are related to the R2DM and PDM services. R2DM will be used by different parts of the envisioned system. R2DM will be used by central engine for storing raw data and by different SUT algorithms for processing the stored raw data for reporting the location estimates. Similarly, PDM will be used by central engine for storing the results of benchmarking experiments. Both services will be publicly available for all users.

Chapter 4

Implementation of R2DM and PDM services

R2DM and PDM services are web services developed in Python 2.7 using the Flask module [6]. Flask is a micro framework for Python based on Werkzeug and Jinja 2 library. Flask module provides a simple way of creating a RESTful web services [7]. Our development was the addition of specific functions into the already provided framework in order to support raw ranging data management. Raw ranging data is stored into the MongoDB database [8]. MongoDB is an open-source document database and the leading NoSQL database written in C++. It gives a possibility of storing JSON (JavaScript Object Notation) [9] based messages using BSON (Binary JSON) format of data. Our data messages are defined as Protocol Buffers structures [10]. Google provided Protocol Buffers are a way of encoding structured data using an efficient and extensible format. Developed services, together with the MongoDB databases, are running on the EC2 instance in AWS [11]. Amazon Elastic Compute Cloud (Amazon EC2) [12] is a web service that provides resizeable computing capacity in the cloud. The user is able to communicate with the services through properly defined HTTP requests. The control messages are JSON based, while the data messages are Protocol Buffer encoded. The overview of the architecture of services is given in Figure 4.1.

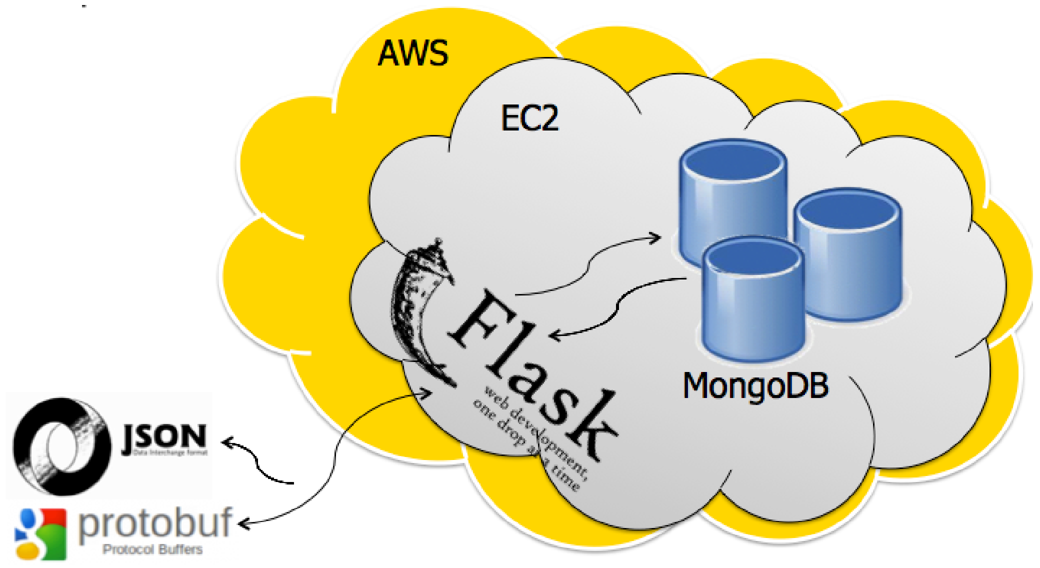

Figure 4.1: R2DM and PDM architecture

4.1 Implementation Ideas

This section shortly presents the design ideas that we used as a guidance for developing the R2DM service. Main requests for the service are extensibility, reliability, fast data flow, simple and remote usage of the service, and language and platform independence.

4.1.1 Extensibility

Our main concern while developing R2DM was extensibility. In other words, we wanted to make a service in such a manner that user is able to add or remove parameters from the data messages according to her specific request. Furthermore, even those types of user's defined messages should be possible to store into the database using the R2DM service. We achieved those goals using Protocol Buffer for defining message types and MongoDB for storing those messages.

In Protocol Buffers, each data structure that needs to be encoded is encapsulated in the form of a message. The specification of the internal structure of the messages is done in special protocol files that have the *.proto* extension. The specification is performed using a simple, but powerful, domain specific data specification language, that allows easy description of the different message fields, their types, optionality, etc. Using the Protocol Buffer compiler *protoc* the *.proto* specification files can be compiled to generate data access classes in number of languages like C++, Java, Python, etc. These classes provide simple accessors for each field (like query() and set_query()) as well as methods to serialize/parse the whole structure to/from raw bytes. For example, the following command:

```
protoc -I=\$SRC_DIR --python_out=DST_DIR SRC_DIR/example.proto
```

creates a python file *example_pb2.py* in the specified output directory that contains data access classes for the Example message. The automatically generated code enables simple creation, manipulation, serialization and deserialization of the messages.

By using a NoSQL or schemaless type of database, i.e. MongoDB, R2DM service enables the storage of any type of defined message, without the need of changing the code and/or the database itself. The enabler for this is a fact that NoSQL databases employ less constrained consistency models than traditional relational databases.

4.1.2 Fast, Reliable and Simple Remote Access

RESTful web services enable fast and remote access to the data just by using HTTP requests. This makes them simple to use for the end users. Furthermore, using HTTP requests enable the possibility of remote access to the data. R2DM is running on the Amazon EC2 instance, i.e. in the cloud which makes it sufficiently reliable for usage. Also, cloud services tend to provide almost infinite processing and storing capabilities. This enables efficient and fast communication between users and cloud and there is practically no possibility of overflowing the memory. Protocol Buffer serialization serializes a messages into binary streams which also enables the sufficiently fast communication between users and cloud. To sum up, RESTful web service as a part of R2DM enables the possibility of simple and remote access to the service. Reliability of usage is supported by running the R2DM in the Amazon cloud. Finally, the fast data flow from user to database is achieved by using Protocol Buffer serialization and MongoDB schemaless database for storing the binary data.

4.1.3 Language and Platform Independence

As mentioned before, using the Protocol Buffer compiler message specification files (*.proto*) can be compiled to generate data access classes in number of languages like C++, Java and Python. In other words, it is possible and simple to use Protocol Buffers with a variety of different programming languages. Furthermore, due to the fact that communication with the cloud is done using HTTP requests, it is possible to manage data from different users' platform, and also using different programming languages (most of the programming languages today support HTTP requests). To conclude, by using Protocol Buffers and RESTful web service it is possible to manage raw ranging data from different platforms and using different programming languages.

4.2 Data Storage Hierarchy

NoSQL databases usually provide the following hierarchy. On top level there is a set of databases. Each database consists of a number of collections (known as tables in SQL-based databases). Finally, each collection consists of a set of messages which contain data. We follow the same principles in our services. The data storage hierarchy in R2DM and PDM services is given in Figure 4.2. The database for storing the raw data from the indoor localization experiments consists of a set of collections (called "Experiments" in our case), where collection is a measurement survey for gathering raw data. Each collection consists of messages. Messages are related to locations where the measurements are made. Besides the databases for storing raw ranging data, there is also a database for storing the metadata of the raw ranging data. This database contains descriptions of different experiments or measurements surveys done to collect the raw ranging data. Similar approach is followed with PDM service. The database has set of collections storing experimental results obtained during the indoor localization benchmarking experiments.

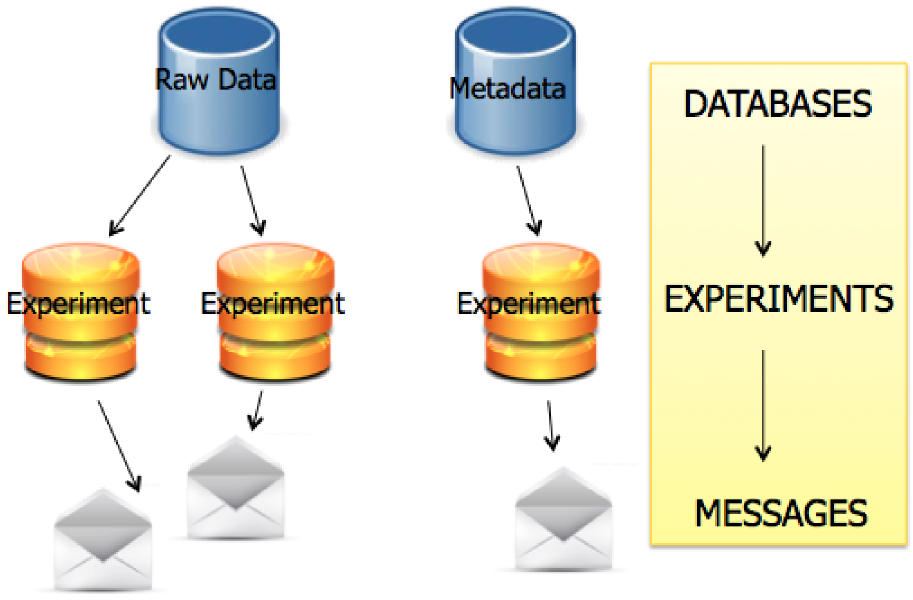

Figure 4.2: R2DM data storage hierarchy

Chapter 5

R2DM - Raw Ranging Data Management Service

This chapter presents the R2DM (Raw Ranging Data Management) service. R2DM is a RESTful web service developed in Python 2.7 [5]. The API provides different functions for managing the stored raw ranging data of indoor localization experiments. For handing raw data different types of messages are used. In the text bellow we firstly give the architectural overview of R2DM and the hierarchy of data storage. Furthermore, to justify the usage of specific architectural components, we present the design ideas that led us through the development of the R2DM service. Finally, we describe the types of messages used and give a short description of the function provided by R2DM's API.

5.1 Message Types

This section presents different types of data messages used for storing and retrieving the raw ranging data from the R2DM service. There are four message types defined. Raw RSSI, AoA and ToA message types are the types used for storing raw RSSI, AoA and ToA data, respectively.The last message type is Metadata, which is used to describe the information about the raw data messages. All of the message types are in fact a Protocol Buffer file with the extension *.proto*. All messages have required, repeated and optional fields. Required fields are the field that have to be filed before the message serialization, otherwise the serialization error occurs. On the other side, depending on the users' preferences, optional fields can be filed, but not necessary. Repeated fields can be repeated any number of times in a well defined message.

5.1.1 Raw Data Message

This section presents the Protocol Buffer structure of the raw data message type. Using the *protoc* compiler the *.proto* messages can be translated into the Python, Java and C++ code. The *raw_Data.proto* consists of the number of raw measurements taken at one location. Each measurement is stored in a *RawDataReading* collection of parameters of the message and can consist of raw RSSI, raw ToA and raw AoA information related to particular time and location in the environment. One raw reading contains a number of optional parameters (timestamp, run number, etc.) and a raw RSSI measurement a a required parameter. As mentioned before, data messages are related to the location where raw measurements are taken. Location can be defined with *(x,y,z)* coordinates, with room label and/or with an additional label, as presented in the parameter *Location*. Except for the mentioned parameters, one message also contains the ID of metadata that connects a reading with the description, internal ID given by the database, etc. The whole structure of one raw data message is presented in the code bellow.

```
package example;

message RawDataReading {
    optional string sender_ssid = 1;          // SSID of the sender
    optional string sender_bssid = 2;         // BSSID of the sender
    optional string channel = 3;              // Channel
    required int32 rssi = 5;                  // RSSI (Received Signal Strength)
    optional float lqi = 6;                   // LQI (Link Quality Indication)
    required int64 timestamp_utc = 7;         // Milliseconds from 1.1.1970.
    required int32 run_nr = 8;                // Run number
}

message RawDataReadingCollection {
    required int64 timestamp = 1;             // TImestamp
    required string metadata_id = 2;          // ID of the metadata
    required string receiver_id = 3;          // ID of the receiver
    repeated RawRSSIReading rawRSSI = 4;      // Collections of raw RSSI data
    repeated RawToAReading rawToA = 5;        // Collections of raw ToA data
    repeated RawToAReading rawAoA = 6;        // Collections of raw AoA data
    required Location location = 7;           // Location of the measurements
    required string data_id = 8;              // ID of the measurement
    optional int32 seq_nr = 9 [default = 1];  // Sequence number
    optional bytes _id = 10;                  // Internal ID given by MongoDB

    message Location {
      optional double coordinate_x = 1;       // x-coordinate
      optional double coordinate_y = 2;       // y-coordinate
      optional double coordinate_z = 3;       // z-coordinate
      optional string room_label = 4;         // Room label
      optional string node_label = 5;         // Additional label
    }
}
```

5.1.2 Metadata Message

Metadata message defines the format for storing the metadata describing the stored raw ranging data. The structure of the Protocol Buffer message is given below.

```
package example;

message Metadata {
    required string data_id = 1;                    // ID of the data
    required string metadata_id = 2;                // ID of the metadata
    required int64 timestamp_utc = 3;               // Milliseconds from 1.1.1970.
    required Scenario_description scenario = 4;     // Description of the scenario
    optional bytes _id = 5;                         // Internal ID given by MongoDB

    message Scenario_description {
      required string receiver_description = 1;     // Receiver description
      required string sender_description = 2;       // Sender description
      required string environment_description = 3;  // Environment description
      required string experiment_description = 4;   // Experiment description
      required string type_of_raw_data = 5;         // Type of raw data
      optional string interference_description = 6; // Interference description
    }
}
```

5.2 API Functions

This section gives an overview of the functions that API provide and that can be used for managing the raw ranging data. The types of HTTP requests can generally be divided into the GET, POST, PUT, PATCH and DELETE requests, so the same division is followed here. The frontpage of the R2DM service is given with the following URL:

```
http://ebp.evarilos.eu/page/r2dm.html
```

5.2.1 POST Requests

POST requests are usually used for creating and storing different data over HTTP protocol. R2DM service API provides three types of POST requests, i.e. creating the database, creating the collection in a given database and adding the message in a collection.

Create database: The command given bellow is used to create a new database named <*database_name*> in the R2DM service.

```
curl --request POST http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database --data '<database_name>'
```

Create collection: The command given bellow is used to create a collection named <*collection_name*> in the database <*database_name*>.

```
curl --request POST http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>/collection --data '<collection_name>'
```

Add message into a collection: Command bellow can be used to add a message into collection <*collection_name*>. Message has to be a properly defined following a Protocol Buffer definition. Furthermore, message also has to be serialized using the Protocol Buffer serialization. Only properly defined serialized messages will be stored in the collection. Serialized message (*message.pb*) has to be sent as a data parameter in the HTTP POST request. Message identificator in the R2DM service is a message field *data_id*.

```
curl  --request POST http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>/collection/<collection_name> \
--data message.pb
```

5.2.2 GET Requests

GET requests are usually used for retrieving data from the service over HTTP protocol. R2DM service API provides four different types of GET requests. It is possible to get a list of databases, list of collections in a given database, list of all messages in a collection and just one message from the collection.

Get a list of databases: In order to get a list of all databases currently present in a R2DM service, following command can be used. The command will return a list of names of all databases together with URLs for them in a JSON format.

```
curl --request GET http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database
```

Get a list of all collections: The command bellow can be used in order to get a list of all collections in a given database <*database_name*>. THe request will return a JSON formatted message containing names and URLs of all collections in a given database.

```
curl --request GET http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>/collection
```

Get a list of all messages: A following request will return a list of all messages in a collection <*collection_name*>. The response will be a JSON formatted message containing all messages, where one message is presents with its <*data_id*>, *metadata_id*, *_id* and an URL to get to the message.

```
curl --request GET http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>/collection/<collection_name>/message
```

Get one message: One message from the collection <collection_name> can be requested using following command. As mentioned, identificator for the message is <*data_id*> and it should me used as <*message_name*> parameter. Depending on the data parameter of the request, message can be requested as a Protocol Buffer serialized string or JSON formatted message. Namely, if data parameter of the request is defined as *'protobuf'*, message will be returned as a Protocol Buffer string. In any other case message will be returned in a JSON format.

```
curl --request GET http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>/collection/<collection_name>/\
message/<message_name> --data 'protobuf/json'
```

5.2.3 PUT Requests

PUT requests in a HTTP protocol are usually use for replacing data. In R2DM there is just one PUT request, used for replacing a message in a given collection.

Replace the message: The following message can be used in order to replace a message <*message_name*> in a collection <*collection_name*> with a new message. Same as in POST request for storing the message, message has be formatted as a Protocol Buffer stream, where *data_id* field will be used as a message identificator.

```
curl  --request PUT http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>/collection/<collection_name>/
message/<message_name> --data new_message.pb
```

5.2.4 PATCH Requests

PATCH request are similar to PUT requests and in the HTTP protocol are usually used from changing some parameters of the data. R2DM supports two types of PATCH requests. User is able to change the collection name or parameters of messages.

Change the name of a collection: A command given bellow gives an example of changing the name of a collection <*collection_name*>. The new name of a collection <*collection_name_new*> is given as a data parameter in a PATCH request.

```
curl --request PATCH http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>/collection/<collection_name>
--data '<collection_name_new>'
```

Change parameters of a message: In order to change the parameters of the message <*message_name*> following command can be used. New parameters have to be given as a JSON formatted message.

```
curl --request PATCH http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>/collection/<collection_name>/
message/<message_name> --data '{"<parameter_name>": "<parameter_value>"}'
```

5.2.5 DELETE Requests

DELETE requests in HTTP protocol are usually used for deleting the data. In the R2DM service DELETE requests are used in order to delete databases, collections and messages.

Delete message: In order to delete the message <*message_name*> from the collection <*collection_name*> the command given bellow can be used.

```
curl --request DELETE http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>/collection/<collection_name>/\
message/<message_name>
```

Delete Collection: For deleting the collection <*collection_name*> from the database <*database_name*> example command is as follows.

```
curl --request DELETE http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>/collection/<collection_name>
```

Delete database: Finally, for deleting the database <*database_name*> a following command can be used.

```
curl --request DELETE http://ebp.evarilos.eu:5000/\
evarilos/raw_data/v1.0/database/<database_name>
```

Chapter 6

PDM - Processed Data Management Service

This chapter presents the message types and API functions of the PDM (Processed Data Management) service. This service is used for storing, managing and using the results of indoor localization benchmarking experiments. Similarly to the R2DM, PDM service is developed as a RESTful interface in Python 2.7 [5].

6.1 Message Types

This service stores the data using only one message format, named *experiment_results.proto*. This message consists of a set of locations, related to different measurement points in the environment. The location is defined with the ID, coordinates (x, y, z) and room in which the measurement point is. Further, the message defined the estimated coordinates provided by the system under test (SUT), together with latency of indoor localization. Moreover, message defines different types of statistical information regarding the localization error, latency and power consumption of SUT devices, together with average data per experiment (called Primary metrics). Finally, message defines a scenario in which the experiment has been executed, serving as a metadata describing the stored experimental results.

```
package example;

message Experiment {
  required int64 timestamp_utc = 1;                // Timestamp - milliseconds from 1.1.1970
  required string experiment_label = 2;            // Name of the experiment
  repeated Measurement_point locations = 3;        // Evaluation points
  required Scenario_description scenario = 4;      // Scenario Description
  required Primary_metrics primary_metrics = 5;    // Primary metrics

  message Measurement_point {
    required int32 point_id = 1;                   // ID of the each point in the experiment
    optional int32 localized_node_id = 2;          // ID of the localized node
    optional string point_label = 3;               // Zero_truth: point label
    required double true_coordinate_x = 4;         // Zero-truth: x-coordinate
    required double true_coordinate_y = 5;         // Zero-truth: y-coordinate
    optional double true_coordinate_z = 6;         // Zero-truth: z-coordinate
    optional string true_room_label = 7;           // Zero-truth: room
    optional string est_point_label = 8;           // Estimated location: point label
    required double est_coordinate_x = 9;          // Estimated location: x-coordinate
    required double est_coordinate_y = 10;         // Estimated location: y-coordinate
    optional double est_coordinate_z = 11;         // Estimated location: z-coordinate
    optional string est_room_label = 12;           // Estimated location: room
    optional double latency = 13;                  // Latency of the location estimation
    optional double localization_error_2D = 14;    // 2D localization error
    optional double localization_error_3D = 15;    // 3D localization error
    optional double localization_correct_room = 16; // Room level localization error
    optional double power_consumption = 17;         // Power consumption estimate
  }

  message Primary_metrics {
    optional double error_2D_average = 1;           // Average 2D localization error
    optional double error_2D_median = 2;            // Median 2D localization error
    optional double error_2D_std = 3;               // 2D localization error st. deviation
    optional double error_2D_min = 4;               // Min. 2D localization error
    optional double error_2D_max = 5;               // Max. 2D localization error
    optional double error_3D_average = 6;           // Average 3D localization error
    optional double error_3D_median = 7;            // Median 3D localization error
    optional double error_3D_std = 8;               // 3D localization error st. deviation
    optional double error_3D_min = 9;               // Min. 3D localization error
    optional double error_3D_max = 10;              // Max. 3D localization error
    optional double room_error_average = 11;        // Average room accuracy error
    optional double latency_average = 12;           // Average latency
    optional double latency_median = 13;            // Latency median
    optional double latency_std = 14;               // Latency standard deviation
    optional double latency_min = 15;               // Min. latency
    optional double latency_max = 16;               // Max. Latency
    optional double power_consumption_average = 17; // Average power consumption
    optional double power_consumption_std = 18;     // St. deviation of power consumption
    optional double power_consumption_min = 19;     // Min. power consumption
    optional double power_consumption_max = 20;     // Max. power consumption
    optional double power_consumption_median = 21;  // Median power consumption
  }

  message Scenario_description {
    required string testbed_label = 1;              // Testbed label
    required string testbed_description = 2;        // Testbed description
    required string experiment_description = 3;     // Experiment description
    required string sut_description = 4;            // System under test description
    required string receiver_description = 5;       // Receiver description
    required string sender_description = 6;         // Sender description
    required string interference_description = 7;   // Interference description
  }
}
```

6.2 API Functions

Similar to the R2DM service, the API of PDM service defines functions for storing, getting, changing and deleting the data, implemented as standard HTTP requests. The frontpage of the PDM service is given with the following URL:

```
http://ebp.evarilos.eu/page/pdm.html
```

6.2.1 POST Requests

POST request is generally used for storing the data over HTTP protocol. PDM service API uses POST requests for creating the database, creating the experiment in a given database and adding the data in an experiment.

Create database: For creating a new database named <*database_name*> in the PDM service the following command is used.

```
curl --request POST http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database --data '<database_name>'
```

Create experiment: The command given bellow is used to create an experiment named <*experiment_name*> in the database <*database_name*>.

```
curl --request POST http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database/<database_name>/experiment --data '<experiment_name>'
```

Add message in the experiment: command bellow can be used to add a message in the experiment <*experiment_name*>. Message has to be a properly defined following a Protocol Buffer definition. Furthermore, message also has to be serialized using the Protocol Buffer serialization. Only properly defined serialized message can be stored in the experiment. Serialized message (*experiment_results.pb*) has to be sent as a data parameter in the HTTP POST request. Message identificator in the PDM service is a message field *experiment_id*.

```
curl  --request POST http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database/<database_name>/experiment/<experiment_name> \
--data experiment_results.pb
```

6.2.2 GET Requests

GET requests are usually used for retrieving data from the service over HTTP protocol. PDM service API provides three types of GET requests. It is possible to get a list of databases, list of experiments in a given database or the data from an experiment.

Get a list of databases: In order to get a list of all databases currently present in a R2DM service, following command can be used. The command will return a list of names of all databases together with URLs for them in a JSON format.

```
curl --request GET http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database
```

Get a list of all experiments: The command bellow can be used in order to get a list of all experiments in a given database <*database_name*>. THe request will return a JSON formatted message containing names and URLs of all experiments in a given database.

```
curl --request GET http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database/<database_name>/experiment
```

Get data from the experiment: One message from the experiment <experiment_name> can be requested using following command. As mentioned, identificator for the message is <*experiment_id*> and it should me used as <*message_name*> parameter. Depending on the data parameter of the request, message can be requested as a Protocol Buffer serialized string or JSON formatted message. Namely, if data parameter of the request is defined as *'protobuf'*, message will be returned as a Protocol Buffer string. In any other case message will be returned in a JSON format.

```
curl --request GET http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database/<database_name>/experiment/<experiment_name>/\
--data 'protobuf/json'
```

6.2.3 PUT Requests

PUT requests in a HTTP protocol are usually use for replacing data. In PDM service PUT request is used for replacing the message in a given experiment.

Replace the message: The following message can be used in order to replace a message <*message_name*> in the experiment <*experiment_name*> with a new message. Same as in POST request for storing the message, message has be formatted as a Protocol Buffer stream, where *experiment_id* field will be used as a message identificator.

```
curl  --request PUT http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database/<database_name>/experiment/<experiment_name>/
--data new_message.pb
```

6.2.4 PATCH Requests

PATCH request are similar to PUT requests and in the HTTP protocol are usually used from changing some parameters of the data. PDM supports two types of PATCH requests. User is able to change the experiment name or data in the experiment.

Change the name of an experiment: A command given bellow gives an example of changing the name of the experiment <*experiment_name*>. The new name of a experiment <*experiment_name_new*> is given as a data parameter in a PATCH request.

```
curl --request PATCH http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database/<database_name>/experiment/<experiment_name>
--data '<experiment_name_new>'
```

Change the data in an experiment: In order to change the parameters of the message <*message_name*> following command can be used. New parameters have to be given as a JSON formatted message.

```
curl --request PATCH http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database/<database_name>/experiment/<experiment_name>/
message/<message_name> --data '{"<parameter_name>": "<parameter_value>"}'
```

6.2.5 DELETE Requests

DELETE requests in HTTP protocol are usually used for deleting the data. In the PDM service DELETE requests are used in order to delete databases and experiments.

Delete experiment: For deleting the experiment <*experiment_name*> from the database <*database_name*> example command is as follows.

```
curl --request DELETE http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database/<database_name>/experiment/<experiment_name>
```

Delete database: Finally, for deleting the database <*database_name*> a following command can be used.

```
curl --request DELETE http://ebp.evarilos.eu:5001/\
evarilos/metrics/v1.0/database/<database_name>
```

Chapter 7

Conclusion and Future Work

The report presented the R2DM and PDM services for managing the raw ranging data and experimental results of indoor localization benchmarking experiments. Our intention was to offer a simple, yet reliable and extensible service for managing the data. Furthermore, we wanted to make services reasonably fast, language and platform independent, as well as remotely usable. The building blocks of the services were carefully chosen in order to satisfy the given requests. We used reliable cloud computing and storage services provided by Amazon, NoSQL database provided by MongoDB, Protocol Buffer and JSON message types and RESTful web service provided by Python Flask module.Finally, both services offer a possibility of managing the data using only HTTP standard requests through the well defined APIs. Further work includes adding new possibilities such as filtering and listing into the service. Finally, in the scope of EVARILOS project [13] we will provide a number of experimental surveys for different kinds of raw ranging data and experimental results of different indoor localization solutions solutions and offer them to the public using provided services.

Bibliography

[1] H. Lim, L. Kung, J. Hou and H. Luo: "Zero-Configuration, Robust Indoor Localization: Theory and Experimentation", Proceedings of IEEE INFOCOM, 2006.

[2] J. S. Lee, S. Yu-Wei, and S. Chung-Chou: "A comparative study of wireless protocols: Bluetooth, UWB, ZigBee, and Wi-Fi." Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE. IEEE, 2007.

[3] R. Stoleru, H. Tian, and J. A. Stankovic: "Range-free localization." Secure Localization and Time Synchronization for Wireless Sensor and Ad Hoc Networks. Springer US, 2007. 3-31.

[4] H. Karl, A. Willig: "Protocols and Architectures for Wireless Sensor Networks", John-Wiley, New York, 2005.

[5] "Python Programming Language": http://www.python.org/

[6] "Flask: Python Microframework": http://www.pocoo.org/projects/flask/#flask/

[7] L. Richardson, and R. Sam: "RESTful web services", O'Reilly, 2008.

[8] K. Chodorow: "MongoDB: The definitive guide", O'Reilly, 2013.

[9] "Introducing JSON": http://www.json.org/

[10] Buffers, Protocol: "Google's Data Interchange Format", 2011.

[11] "Amazon Web Services (AWS)": http://aws.amazon.com/

[12] "Amazon Elastic Compute Cloud (Amazon EC2)": http://aws.amazon.com/ec2/

[13] "EVARILOS project": http://www.evarilos.eu/