Universidad del País Vasco Euskal Herriko Unibertsitatea

BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

ILLINOIS INSTITUTE OF TECHNOLOGY

MÁSTER UNIVERSITARIO EN

INFORMATION TECHNOLOGY AND MANAGEMENT

TRABAJO FIN DE MÁSTER

NG911 INDOOR LOCATION ANDROID APP

Alumna	*Sirés, Artázcoz, Carmen*
Directora	*Davids, Carol*
Departamento	Real Time Communications
Curso académico	*2018/2019*

Chicago, 22 de julio de 2019

Abstract

This paper describes the NG911 Indoor Location project and, more specifically, the Android app that has been developed for it. Since current GPS technology is unable to locate people in buildings in cases of emergency, this project offers a solution to this problem, providing a way to obtain the indoor location of a person. The NG911-Indoor-Location app will obtain the indoor location of the calling device using nearby Bluetooth beacons and the BOSSA Platform.

First, the app will scan for Bluetooth beacons in the building, obtaining a list of them. Then, it will query the BOSSA Platform with this list which will answer with the indoor location (address, building, floor, x, y). Finally, it will establish a SIP call, sending the received indoor location in it. This will be done to provide a more accurate location for the person to be easily located in the case of an emergency.

Resumen

Este documento describe el proyecto NG911 Indoor Location (ubicación en interiores) y, más específicamente, la aplicación para Android que se ha desarrollado para ello. Dado que el GPS no puede ubicar con precisión a las personas dentro de edificios en casos de emergencia, se ha desarrollado una manera de obtener la ubicación en el interior de un edificio de una persona. La aplicación NG911-Indoor-Location obtendrá la ubicación interior del dispositivo que realiza la llamada mediante el uso de las señales recibidas de los beacons de Bluetooth más cercanos y la plataforma BOSSA.

En primer lugar, realizará una búsqueda de señales Bluetooth que provienen de los beacons en el edificio, obteniendo una lista de ellas. Posteriormente, realizará una consulta a la plataforma BOSSA con esta lista, que responderá con la ubicación interior (dirección, edificio, piso, x, y). Finalmente, establecerá una llamada SIP, enviando en ella la ubicación interior recibida. El objetivo de esto es proporcionar una ubicación más precisa para que la persona pueda ser encontrada más fácilmente en caso de emergencia.

Laburpena

Dokumentu honek NG911 Indoor Location (barne-kokapena) proiektua deskribatzen du eta, zehatzago esanda, horretarako garatu den Android aplikazioa deskribatzen du. Larrieldietan GPSak eraikinen barruko pertsonak aurkitu ezin dituenez gero, eraikinen barrualdean dauden persona horien kokapena lortzeko bidea garatu egin da. NG911 Indoor Location aplikazioak deiak egiten dituen gailuaren kokapena lortuko du, gertuen dauden Bluetooth beaconen bidez jasotako seinaleak eta BOSSA plataforma erabiliz.

Lehendabizi, eraikinean jarritako beaconetatik datozen Bluetooth seinaleen araketa egingo da, haien lista bat sortuz. Ondoren, BOSSA plataformari kontsulta bat egingo dio lista horrekin, eta BOSSA plataformak barne-kokapenarekin erantzungo du (helbidea, solairua, x, y). Azkenik, SIP deia burutuko da, dei horretan lortutako barne-kokapena bidaliz. Honen helburua kokapen zehatzago bat lortzea da, larrialdi bat egonez gero, pertsona azkar eta errazago aurki dadin.

Table of Contents

Table of figures

Acronyms

- **ACK**: Acknowledgement
- **API**: Application Programming Interface
- **AWS**: Amazon Web Services
- **BCF**: Border Control Function
- **BLE**: Bluetooth Low Energy
- **BOSSA**: Bluetooth and Sensors Array
- **DNS**: Domain Name Server
- **ECRF**: Emergency Call Routing Function
- **ESInet**: Emergency Services IP backbone Network
- **ESRP**: Emergency Service Routing Proxy
- **ESXI**: Elastic Sky X Integrated
- **FCC**: Federal Communications Commission
- **GPS**: Global Positioning System
- **HTTP**: HyperText Transfer Protocol
- **ID**: Identifier
- **IIT**: Illinois Institute of Technology
- **IP**: Internet Protocol
- **JSON**: JavaScript Object Notation
- **LoST**: Location to Service Translation
- **MIME**: Multipurpose Internet Mail Extensions
- **NAT**: Network Address Translation
- **NG911**: Next-Generation 911
- **PAN**: Personal Area Network
- **PIDF-LO**: Presence Information Data Format – Location Object
- **PSAP**: Public Safety Answering Point
- **RSSI**: Received Signal Strength Indicator
- **RTC**: Real Time Communications
- **RTP**: Real-time Transport Protocol
- **SBC**: Session Border Control

- **SDP**: Session Description Protocol
- **SIP**: Session Initiation Protocol
- **UI**: User Interface
- **URI**: Uniform Resource Identifier
- **URL**: Uniform Resource Locator
- **URN**: Uniform Resource Name
- **UUID**: Universally Unique Identifier
- **VM**: Virtual Machhine
- **XML**: eXtensible Markup Language

Introduction

Nowadays, everybody is very reliable on the current emergency system. While the existing 911 system has been very successful for years, it has been stretched to its limit with regards to technology advances. Besides, people place false trust on the current system, relying on assumption that it is easy for the police to locate them if they call for an emergency.

When most people call the police, they assume that their location will be reliably acquired by the police, using GPS. This is the first problem with GPS, because, according to a FCC (Federal Communications Commission) study [1], most 911 dispatch centers do not receive GPS information from a cell phone caller. For this reason, usually, it is the caller that needs to describe the indoor location in a building, which can be time consuming or even difficult to explain in some situations.

On the other hand, there is the problem of GPS itself (which is the technology that could be used by the police to locate the caller). It is true that most cellphones are equipped with GPS hardware, which can locate callers via satellites. This can be a good solution when the caller is in an open field, but when we are talking about inside homes, apartments, stores, hotels, or even in downtowns where tall buildings block signals, the GPS technology performs poorly, especially when the call comes from inside a tall building. Moreover, approximately 58 percent of wireless calls to 911 come from indoors, so they are not likely to deliver a location at all.

If a person was calling from a 23rd floor of a 50 story building, it is very likely that they would not be located in time. In some types of emergencies (like a shooting or a fire), this could lead to horrible consequences for the caller.

For these reasons, the FCC requires mobile carriers to improve the indoor location service when calling 911, and this is why the NG911 Indoor Location System has been created. Using Bluetooth beacons and the BOSSA Platform, the indoor location of a person in a building will be calculated, and will be sent to the police in order for them to locate the calling device more easily.

If this system is implemented, the time saved to locate the 911 callers could be saving 10,000 lives every year.

Goals

The main goal of this project is to develop an Android App capable of fulfilling the following requirements:

- **Get a list of nearby beacons**: scan for Bluetooth beacons signals and obtain a list of the closest beacons inside of a building.
- **Get the indoor location**:
    - Make a HTTP request to the location server of the BOSSA platform, including the list of received beacons in the URL.
    - Receive the HTTP response from the BOSSA platform, containing the indoor location in XML format.
- **Establish the call**: establish a SIP call with the ESInet, including the received indoor location in the INVITE message.

The app will be easy to use for the user, with a very simple UI and limited functionality.

Technologies involved

Bluetooth

Bluetooth [10] is a wireless technology standard for exchanging data between fixed and mobile devices over short distances, using short-wavelength UHF radio waves in the frequency of 2.4 GHz, and building personal area networks (PANs).

A beacon is a small Bluetooth radio transmitter, powered by batteries. It transmits Bluetooth Low Energy (BLE) signals. The Bluetooth enabled smartphones are capable of scanning and displaying these signals. It is called Low Energy because the power that it requires is very low.

Within this project, Bluetooth is the technology used between the beacons and the Android App. The beacons are constantly broadcasting Bluetooth Low Energy signals, and the phone needs to scan them.

SIP (Session Initiation Protocol)

SIP [3] is an application-layer control (signaling) protocol for creating, modifying, and terminating sessions with one or more participants. SIP uses different type of messages. The invitations are used to create sessions. They carry session descriptions that allow participants to agree on some parameters.

SIP makes use of elements called proxy servers to help route requests to the user's location and authentication and authorization of users, among others.

SIP also provides a registration function that allows users to upload their current locations for use by proxy servers.

Once the user is registered, the messages that are exchanged between the two parts (and the proxies in between them) are shown in Figure 1:

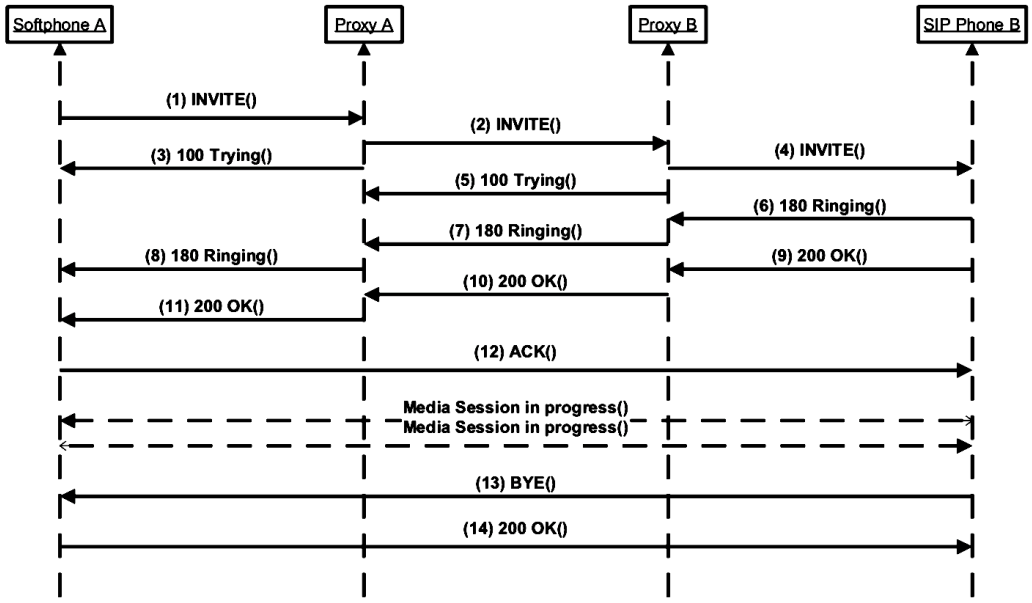

Figure 1: SIP Messages flow of a call [4]

The format of the SIP URI (which is the resource that identifies the user, and goes in the messages) is the following: sip:alice@atlanta.com.

Within this project, SIP is the protocol that is used to establish the calls between the calling device and the PSAP which is answering the call. The SIP call will be routed through the ESInet to the nearest PSAP, depending on the location, that is inserted in the INVITE message.

HTTP (HyperText Transfer Protocol)

The Hypertext Transfer Protocol (HTTP) [5] is an application-level generic, stateless protocol which can be used for many tasks through the extension of its request methods, error codes and headers. HTTP requests are messages sent by the client to initiate an action on the server. HTTP Response is the message that the server sends back to the client, answering to the HTTP Request.

HTTP defines methods to indicate the desired action to be performed on the identified resource. The main methods of HTTP are the following:

- **GET**: requests a representation of the specified resource. Requests using GET should only retrieve data, but some parameters can be specified in the URL.
- **POST**: requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI. The actual function performed by

the POST method is determined by the server and is usually dependent on the Request-URI.

- **PUT**: requests that the enclosed entity be stored under the supplied URI. If the URI refers to an already existing resource, it is modified; if the URI does not point to an existing resource, then the server can create the resource with that URI.
- **DELETE**: requests that the specified resource is deleted.

Within this project, HTTP will be used to perform a GET Request. The mobile device will make a HTTP GET Request to the BOSSA Platform API, including a list of beacons in JSON format, in order to obtain the indoor location. The location server of the BOSSA Platform will answer with an HTTP Response, which contains the indoor location in XML format.

JSON (JavScript Objetct Notation)

JavaScript Object Notation (JSON) [16] is a text format for the serialization of structured data. It is derived from the object literals of JavaScript. It defines a small set of formatting rules for the portable representation of structured data. An example of JSON is the following:

```
{
   "name":"John",
   "age":30,
   "car":null
}
```

XML (eXtensible Markup Language)

Extensible Markup Language (XML) [17] is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is a textual data format with strong support via Unicode for different human languages. An example of XML is the following:

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Next Generation 911

In this paper, the Next Generation 911 system will be described, explaining the different modules that are used to provide an accurate indoor location. Besides, the NG911-Indoor-Location Android App will be described in more detail.

Next Generation 911 Introduction

Nowadays, technology brings us the opportunity to improve the emergency systems and make them more efficient. A part of this improvement comes from providing an accurate location of the caller when calling 911. This has a double purpose.

- First of all, the location is used to route the call to the closest PSAP (Public Safety Answering Point) to the caller.
- The second one is to provide a very accurate indoor location for the dispatcher and first responder in order to find the person.

For this purpose to be achieved, a way to provide the accurate location is needed. This system is divided in 4 main quadrants, shown in Figure 2. The top-left quadrant represents the array of Bluetooth LE beacons. The top-right quadrant represents the Location server and database, used to provide the indoor location. These 2 quadrants form the BOSSA Platform. The bottom-left quadrant represents the Bluetooth Indoor Location Android App, which is the one that the caller will use to call 911. Finally, the bottom-right quadrant represents the ESInet (Emergency Services IP backbone Network).

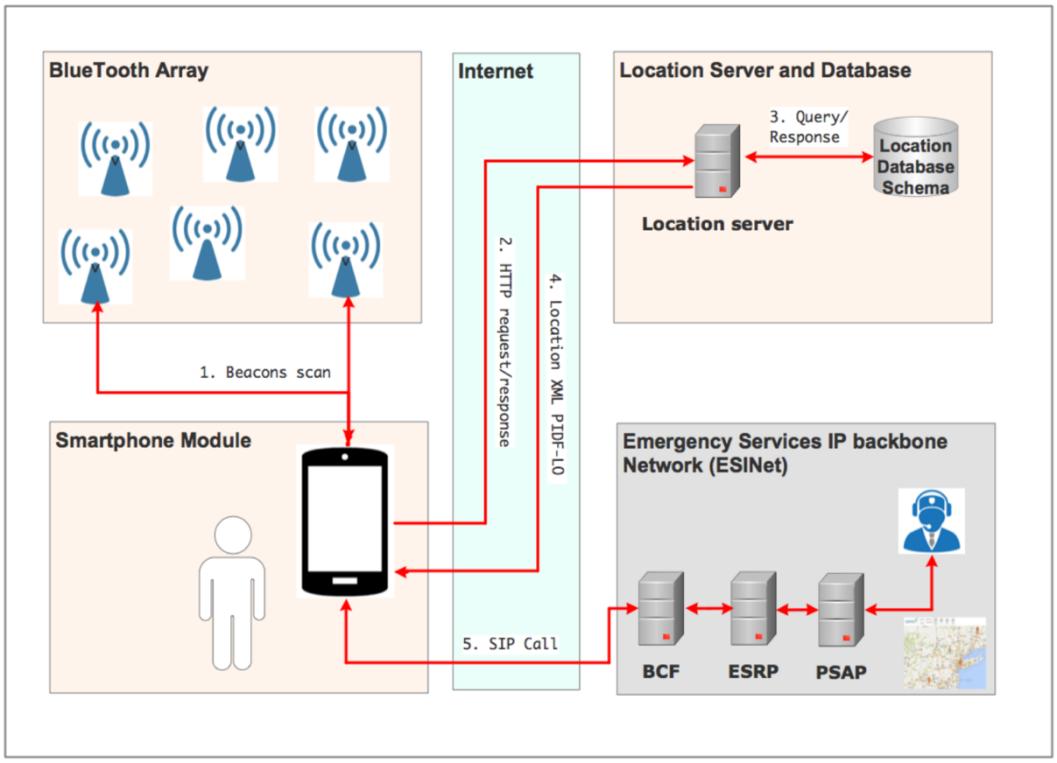

Figure 2: Indoor Location System

- First of all, the user presses “Call 911” in an Android App running on a mobile device. In this moment, the mobile device scans for Bluetooth signals from the Beacons that are close to it. The mobile app receives a set of beacons IDs.
- Then, the mobile app makes an HTTP GET Request to the BOSSA Platform. It sends a list of beacons IDs and their RSSI (Received Signal Strength Indicator) in JSON format. The location server of the BOSSA Platform answers with the indoor location that corresponds to that list of coordinates. This is given in XML Format and contains the address and the indoor location (Building, Floor, Room and x, y coordinates).
- The mobile app inserts this XML indoor location in the MIME Body of the SIP INVITE, and sends it to the ESInet (Emergency Services IP Network).
- This call is routed across the ESInet to the nearest PSAP, basing on the address contained in the SIP INVITE.

Next Generation 911 Components

The four principal components that are part of this project are the following:

An Array of BLE Beacons and Gateways

In each building, there is a group of beacons and gateways.

- The BLE beacons (AXA/RUUVI) are Bluetooth Low Energy devices. They are clustered around gateways. They have two functions:
    - They are used to determine the indoor location. They emit Bluetooth signals with their ID, so that the calling phone will receive a list of ID and their RSSI. This will be used to determine the indoor location of the caller by querying the BOSSA platform.
    - They are also used to send information about the environment conditions to the gateways.
- The gateways (RedBear) have Bluetooth and Wi-Fi. They are used for configuration and monitoring of the beacons. They interact with the beacons via Bluetooth and with the database via Wi-Fi. Each gateway is responsible for the beacons that have been assigned to them. They do the following:
    - Collect the temperature and humidity of the beacons.
    - Send the temperature and humidity reported by the beacons to the database periodically, and their battery level (of the gateways). This information is not sent directly to the database, but through the Particle Cloud.

These devices are located in Stuart Building and Alumni Memorial Hall. A map of the beacons and gateways can be seen by logging into http://rtcmaps.herokuapp.com/login. The Figure 3 shows the map of the first floor of Stuart Building. Clicking on the blue circles (beacons) or red circles (gateways), some information about them can be seen.

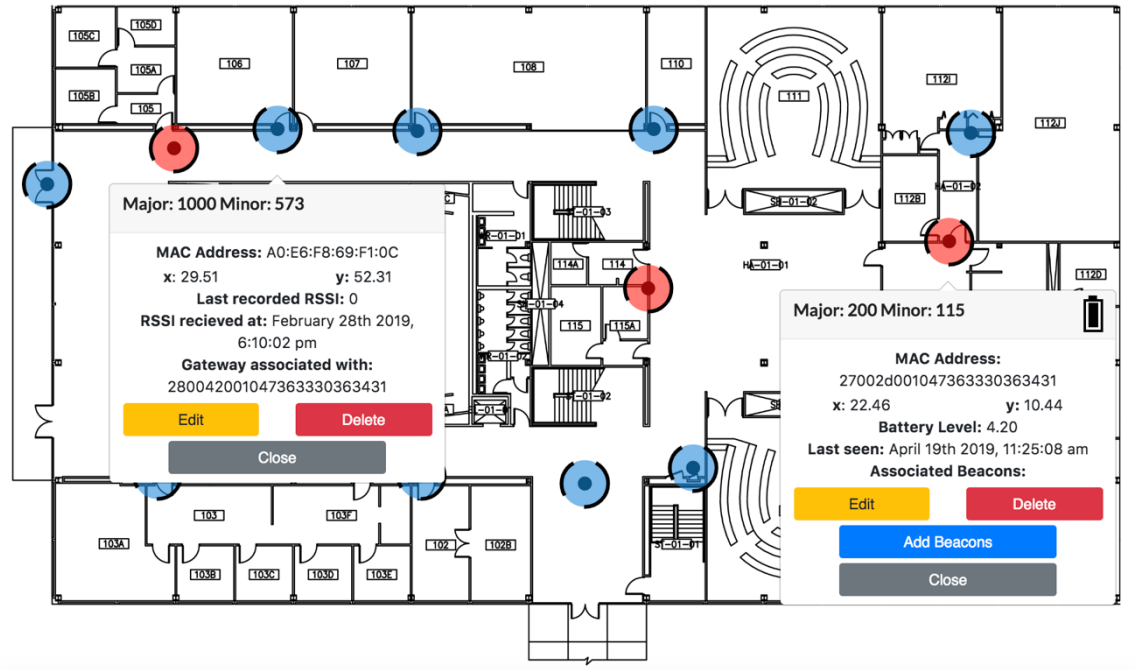

Figure 3: BLE Beacons and gateways map in Stuart Building

A Database and Backend Server

These components are located remotely on an Amazon Web Services (AWS) site.

- The database will be used in 2 different cases.
    - On one hand, it will be used to determine the indoor location, as it has the physical location of the beacons.
    - On the other hand, it will be used to store other information about the beacons, collected by the gateways, like temperature and humidity.


```
{
    "beacon_id": "A0:E6:F8:69:F9:DD",
    "x": 7.69024806020303,
    "y": 22.942882404540125,
    "major": 1000,
    "loc_id": 484,
    "minor": 569,
    "temperature": null,
    "humidity": null,
    "updatetimestamp": "2018-11-15T18:52:57.000Z",
    "building_id": 31,
    "floor": 2
}
```


Figure 4: Stored data about the beacons in JSON Format

- The backend server hosts a number of API's that enable developers to easily extract information from the database and to process that information to produce useful results.
    - One of the important roles of this server is to provide the indoor location. It queries for the physical location of each of the beacons in the received list and then performs an algorithm to calculate the indoor location of the caller.

Mobile device Applications

There are 3 main applications used in this project:

- **End user application**: this is the android application that the caller will use in case of emergency. When the user presses "call 911", the app scans for Bluetooth signals and receives some from the beacons nearby. Each signal contains the ID of the beacon. Then, the app sends a list of {ID,RSSI} to the API, which queries the database to obtain the physical locations of the beacons and, after performing a location algorithm, answers with the indoor location of the caller (in XML format). Finally, the app includes this location in the MIME Body of the SIP INVITE, which will be sent to the ESInet, in order to be routed to the nearest PSAP. This app will be explained with further detail in this document.
    - The current state of the application can be seen in GitHub [9].

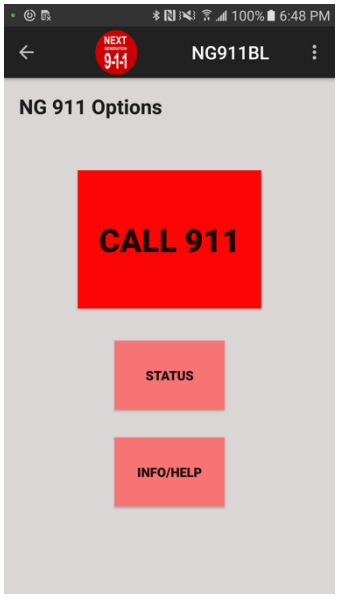

Figure 5: User Application home screen

- **Tester application**: this app allows a tester to generate data sets in different locations of a building. The data is collected and stored, in order to be used for analysis purposes, failures' detection and location algorithms' improvements.
    - The current state of the application can be seen in GitHub [14].

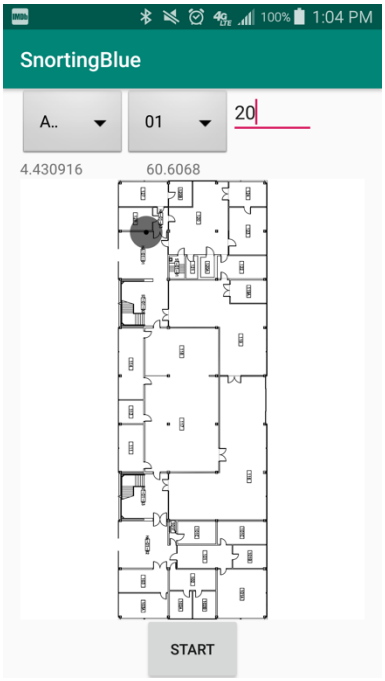

Figure 6: Tester Application Home screen

- **Operations applications**: these were built to help Operations and Support teams learn the battery level of the gateways and the temperature and humidity sensed by the beacons.

Emergency Services IP Network

The ESInet is the network that will be routing the calls from the user to the corresponding PSAP. This network has got the components that are shown in Figure 7:

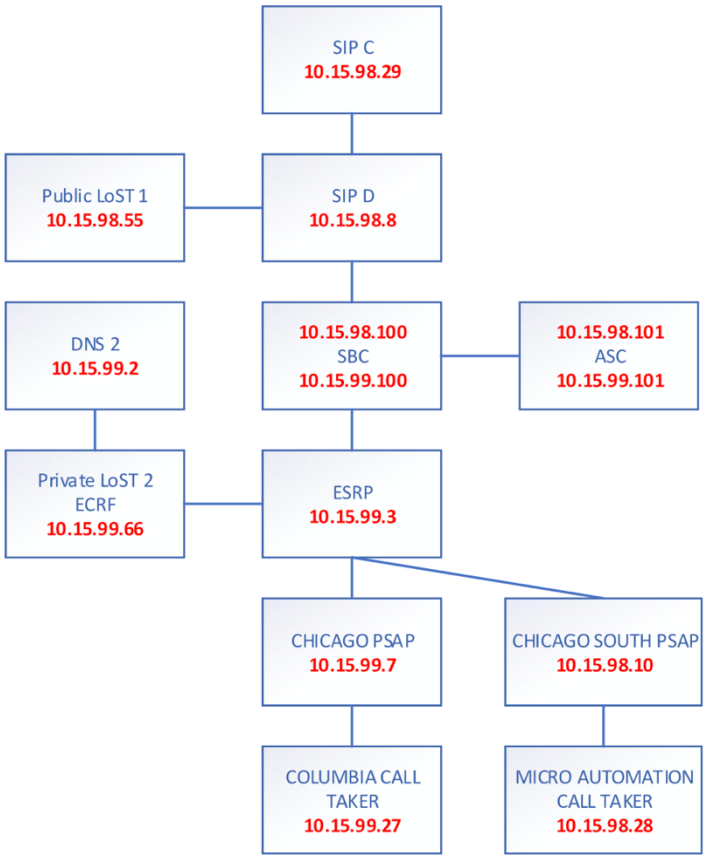

Figure 7: ESInet complete architecture

This is the complete architecture, which is followed when the call is made from the SIP C (as in the RTC Lab), but when we make the call from the android app in a mobile device, as described in this document, the call goes directly from the mobile app to the SBC, in which it is registered.

When the call is made from the app and it goes into the ESInet, it performs the following steps:

- To enter the ESInet, the call goes through the BCF/SBC, which sits between the external networks and the ESInet.
- The SBC forwards the call to the ESRP.
- To be able to forward the call to the corresponding PSAP, the ESRP has to query the ECRF (LOST2), which will respond with the PSAP to which the call has to be forwarded.
- Then the ESRP forwards the call to the PSAP that the ECRF has answered, which is the one that corresponds with the caller's location.

When the call is made from the SIP-C, it performs the following steps:

- The SIP-C routes the call to the SIP-D (sends the SIP INVITE).
- The SIP-D queries the Lost1 server to obtain the address of the SBC.
- The SBC forwards the SIP INVITE to the ESRP.
- From here onwards, the steps are the same as in the previous case.

The main functions will be now described:

SBC (Session Border Control)

The SBC (Session Border Control), is an element of the BCF (Border Control Function). It controls the borders to resolve problems such as NAT (Network Address Translation) or firewall traversal.

Among the main functions of the SBC, the most important ones are the following:

- Identification of emergency calls and priority handling for their IP flows.
- Management of priority marking based on policy for emergency calls.
- Forwarding of an emergency call to an ESRP.
- To make an emergency call from the external network, the session agent has to be registered in the SBC to be able to enter the ESInet.
    - To register a Session Agent (Appendix B: SBC: Manage Session Agents), the IP Address needs to be provided, so any SIP User Agent using this IP Address is registered in the SBC. For example, if the Session Agent 64.131.109.30 has been registered, the SIP User Agent android2@64.131.109.30 will be able to establish a SIP call.

ESRP (Emergency Service Routing Proxy)

The function of the ESRP is to route a call to the next hop. It receives the call from the SBC and has to forward it to the PSAP. To be able to know to which PSAP it has to forward the call, it needs to make a query to the ECRF. It sends the location of the caller and the ECRF responds with the URI of the PSAP to which it has to forward the call.

ECRF (Emergency Call Routing Function)

The ECRF is the functional element which is responsible for providing routing information to the various querying entities. It is queried using the LoST Protocol (Location to Service Translation Protocol).

The ECRF gets as an input from the ESRP:

- the location information (either civic address or geocoordinates)
- a Service URN

It performs a mapping function which gives as a result the URI which is used to forward the call to the appropriate PSAP. Depending on the request, the response may identify the PSAP or the ESRP. In addition, the ECRF provides the capability to retrieve other location related URIs, such as Additional Data URIs.

The ECRF will decide which PSAP to forward the call to, basing on the information received in the PIDF-LO, in particular, checking the a1 and a2 fields in the PIDF-LO.

```
<ca:A1>IL</ca:A1>
<ca:A2>Chicago</ca:A2>
```

Then, it will check which PSAP is associated to that location in the Lost2 table that corresponds to the US and send this information to the ESRP. Some examples of the current PSAPs that appear in this database can be seen in Figure 8.

source character varying (255)	source_id character varying (255)	service character varying	sn chara	display_name character varying (2	uri character varying (255)	last_ time	country character	a1 chara	a2 character varyi
lost.cs.columbia.edu	4c58c0a0-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	Wyoming Poison ...	sip:poison_wy@irt.cs.columbia.edu	2...	us	wy	*
lost.cs.columbia.edu	4c58ca64-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	Wyoming Animal...	sip:animal_wy@irt.cs.columbia.edu	2...	us	wy	*
lost.cs.columbia.edu	4c58d428-1389-11de-bfdc-8da325ae7c2f	urn:service:sos	911	Wyoming PSAP	sip:psap_wy@irt.cs.columbia.edu	2...	us	wy	*
lost.cs.columbia.edu	4c58ddec-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	Puerto Rico Polic...	sip:police_pr@irt.cs.columbia.edu	2...	us	pr	*
lost.cs.columbia.edu	4c58e7e2-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	Puerto Rico Fire ...	sip:fire_pr@irt.cs.columbia.edu	2...	us	pr	*
lost.cs.columbia.edu	4c58f1a6-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	Puerto Rico EMS	sip:ems_pr@irt.cs.columbia.edu	2...	us	pr	*
lost.cs.columbia.edu	4c58fb6a-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	Puerto Rico Pois...	sip:poison_pr@irt.cs.columbia.edu	2...	us	pr	*
lost.cs.columbia.edu	4c59052e-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	Puerto Rico Anim...	sip:animal_pr@irt.cs.columbia.edu	2...	us	pr	*
lost.cs.columbia.edu	4c590f7e-1389-11de-bfdc-8da325ae7c2f	urn:service:sos	911	Puerto Rico PSAP	sip:psap_pr@irt.cs.columbia.edu	2...	us	pr	*
lost.cs.columbia.edu	4c591b5e-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	New York EMS	sip:ems_ny@irt.cs.columbia.edu	2...	us	ny	*
lost.cs.columbia.edu	4c592522-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	New York Poison ...	sip:poison_ny@irt.cs.columbia.edu	2...	us	ny	*
lost.cs.columbia.edu	4c592ef0-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	New York Animal...	sip:animal_ny@irt.cs.columbia.edu	2...	us	ny	*
lost.cs.columbia.edu	4c5938b4-1389-11de-bfdc-8da325ae7c2f	urn:service:so...	911	New York Fire De...	sip:fire_ny@irt.cs.columbia.edu	2...	us	ny	*
lost2.ng911main.iit.edu	4c59426e-1389-11de-bfdc-8da325ae7c2f	urn:service:sos	911	Indiana PSAP	sip:psap@psap2.microautomation.ng911main.iit.edu	2...	us	in	elkhart
lost2.ng911main.iit.edu	4c594c32-1389-11de-bfdc-8da325ae7c2f	urn:service:sos	911	New York psap	sip:psap@psapd.ng911main.iit.edu	2...	us	ny	onondaga
lost2.ng911main.iit.edu	Illinois Chicago PSAP	urn:service:sos	911	chicago psap	sip:psap@psapd.ng911main.iit.edu	2...	us	il	chicago
lost2.ng911main.iit.edu	Illinois chicagosouth PSAP	urn:service:sos	911	chicagosouth psap	sip:psap@psap2.microautomation.ng911main.iit.edu	2...	us	il	chicagosouth
lost2.ng911test.iit.edu	4c4e2fa0-1389-11de-bfdc-8da325ae7c2f	urn:service:sos	911	Illinois PSAP	sip:8880@107.170.50.230	2...	us	il	webrtc

Figure 8: PSAPs in the Lost2 table in the ECRF

NG911-Indoor-Location Android App

Requirements

This project consists of the development of an Android App for the NG911 system. This app will be used to call 911, providing the indoor location of the person that is calling. This app will perform the following tasks:

- R1: The app should be able to scan for Bluetooth signals nearby.
- R2: The app should be able to detect the signals of the beacons that are in the range of the device.
- R3: The app should be able to convert this list of beacons to JSON format.
- R4: The app should be able to query the BOSSA Platform API, sending the detected beacons.
- R5: The app should be able to obtain the response from the BOSSA Platform with the indoor location in XML format.
- R6: The app should be able to create the SIP message, inserting the indoor location.
- R7: The app should be able to send the SIP INVITE message with the indoor location to the ESInet.

Physical architecture

The Figure 9 shows the physical architecture of the NG911 system:

Figure 9: Physical architecture of the NG911 System

Calling device

The device from which the user performs the NG911 Call is an Android device. It needs to have at least Android 5.1 (which corresponds to SDK version 22).

Bluetooth LE Beacons Array

The Bluetooth beacons that are used in this project are currently AXA Beacons. These devices are Low Energy. They only broadcast their ID, which will be scanned by the calling device.

Figure 10: AXA Beacons

The communication between the mobile device and the beacons is performed by the use of Bluetooth.

BOSSA Platform Location Server and Database

These are stored in AWS (Amazon Web Services). The hosting is done by the use of a middleware called Sails.js, which uses Node.js for the server. This is where the API is.

The communication between the calling device and the BOSSA Platform location server API is performed by the use of HTTP between the device and the API. The device sends an HTTP GET Request to the API. The GET request includes the JSON object that describes the data gathered by the phone application. The API takes the information in the object and uses it as input to database queries and location algorithms. The resulting location information is sent back to the phone application by the API in XML format.

ESInet

The ESInet consists of a set of servers that are currently located on the main campus of IIT (Illinois Institute of Technology), in the RTC (Real Time Communications) Lab. Some of them are servers and some others are virtual machines in an ESXi host. These ones are the Micro Automation PSAP, the Lost1, the ECRF (Lost2), the DNS server and a windows 7 Virtual Machine which runs Wireshark.

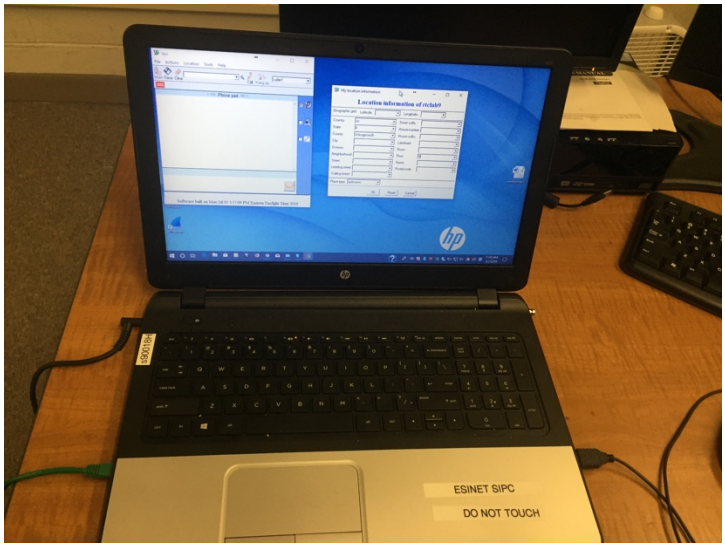

Figure 11: SIP-C. Laptop in the IIT RTC Lab

Figure 12: SIP-D. Server in the IIT RTC Lab

Figure 13: SBC. Server in the IIT RTC Lab

Figure 14: ESRP. Server in the IIT RTC Lab

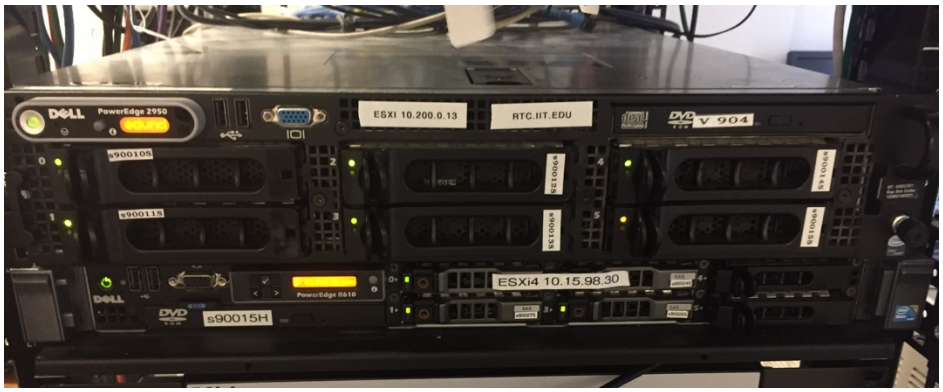

Figure 15: ESXI host. Server in the IIT RTC Lab.

There are also 2 PSAPs in the IIT RTC Lab:

- Chicago PSAP: Columbia PSAP (developed in the university of Columbia)
- Chicago South PSAP: MicroAutomation PSAP

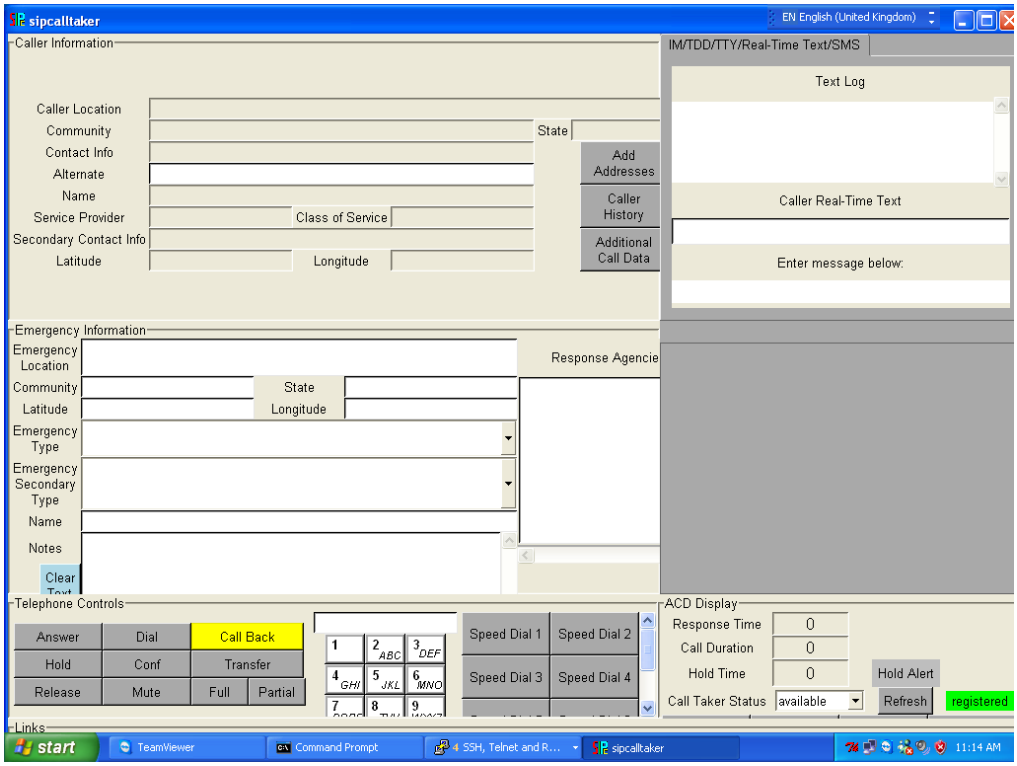

Figure 16: Columbia PSAP (server in the IIT RTC Lab)

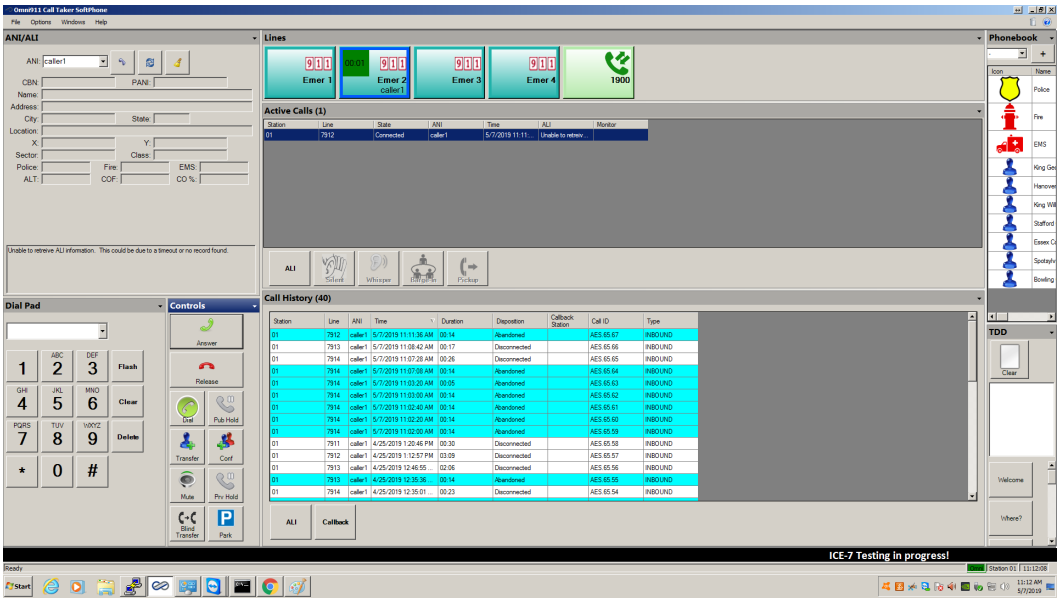

Figure 17: Micro Automation PSAP (VM in the ESXi host)

The communication between the calling device and the ESInet is performed by the use of SIP. A call is established between the calling device and the PSAP. This call goes through the ESInet.

Logical Architecture

An Android App has been developed, in order to be able to establish a call in which the indoor location is sent. Figure 18 shows the main four Functional Units of the logical architecture of the NG911 Indoor Location system. The top-left quadrant represents the array of Bluetooth LE beacons. The top-right quadrant represents the Location server and database, used to provide the indoor location. These 2 quadrants form the BOSSA Platform. The bottom-left quadrant represents the Bluetooth Indoor Location Android App, which is the one that the caller will use to call 911. Finally, the bottom-right quadrant represents the ESInet (Emergency Services IP backbone Network).

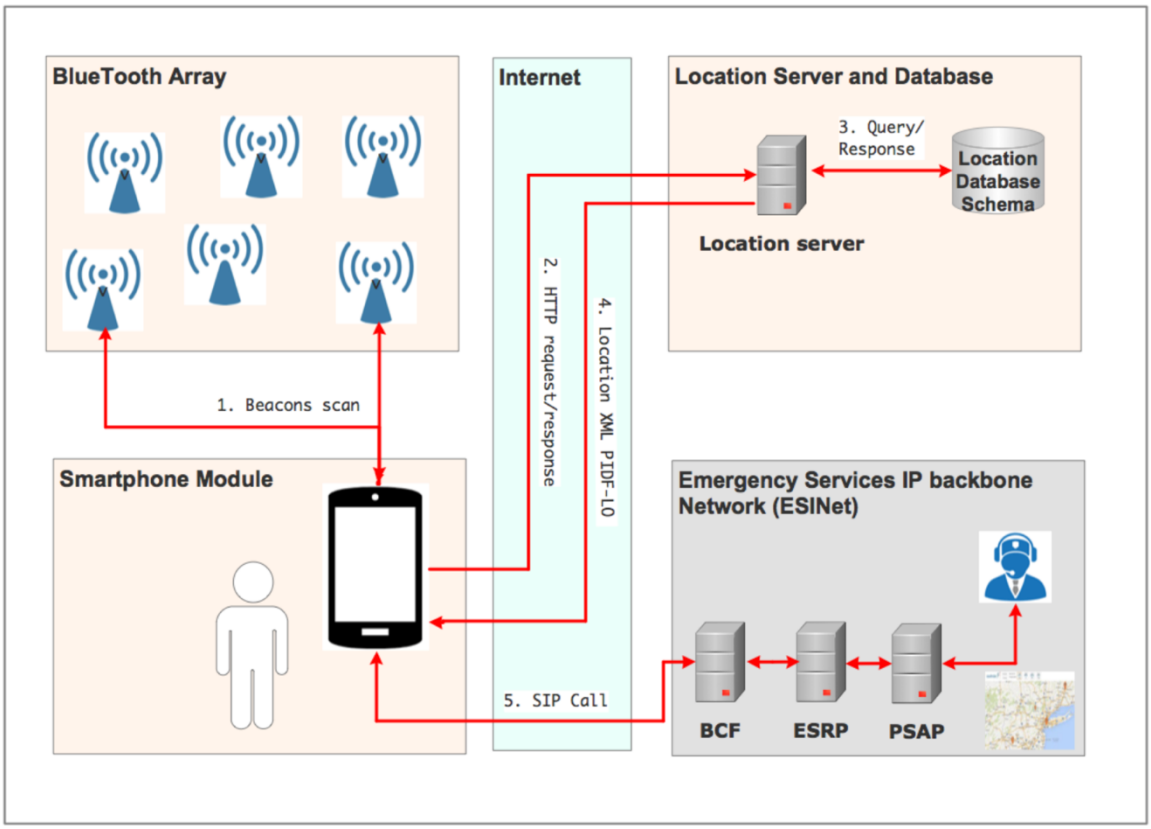

Figure 18: Indoor Location System logical architecture: Bluetooth array (top left), BOSSA Platform (top right), Android App (bottom left) and ESInet (bottom right)

Android App Steps

The android app is divided in 3 main steps:

- When the Call 911 button is pressed, the first step will be to scan for Bluetooth signals. It will receive some from the beacons nearby. Each signal contains the ID of the beacon [quadrant top left].
- Then, the app will convert the list of beacons to the required JSON format that has to be inserted in the URL. It will query the BOSSA Platform with this API, sending a list of {ID, RSSI}. The BOSSA Platform will answer with the indoor location of the caller (in XML format) [quadrant top right].
- Finally, the app will insert the indoor location in the MIME Body of the SIP INVITE, which will be sent to the ESInet, in order to be routed to the nearest PSAP [quadrant bottom right].

The steps that the android app performs to be able to establish a call and send the indoor location will be now explained in detail.

First of all, the user will press the 911 CALL button to start this process.

Beacons scanning

The first step when the user has started the process is to scan for beacons in the area. This will be done by the use of the Android Beacon Library [11]. Two classes have been created for the use of the beacons: the classes IBeacon and IBeaconScanner.

The IBeacon class defines the Beacon object. It contains the parameters that will be necessary to obtain the indoor location. The main attributes that define a beacon are the following:

- UUID (Universally Unique Identifier): The purpose of the ID is to distinguish beacons in one network, from all the beacons in other networks. All the beacons in the network will have the same UUID.
- Major, Minor: these values are used to identify one single beacon. They are values in between 0 and 65535.
- RSSI (Received Signal Strength Indicator): This is not a parameter that the beacon broadcasts, but the strength of the signal received from the beacon, measured in the Android device.

The IBeaconScanner is the class that will contain the logic to perform the scanning of the beacons. First, an object of this class will be created, specifying the period in between measures. 0.5 seconds will be the period used in this app. There is an attribute in the IBeaconScanner class in which this period is stored when the object of this class is created. The attribute is called period. This parameter has been hardcoded.

```
testCase = new IBeaconScanner(this, .5);
```

Where "this" represents the Activity from where the call is made (CallActivity). Then, the start method will be called, indicating the duration of the scanning (in seconds), which has been hardcoded too.

```
testCase.start(5);
```

During the scanning, all the detected beacons will be saved in the class IBeaconScanner, in a list of objects of the class IBeacon. When the scanning has ended, the stop() function will be called.

This function will perform the http request that follows the beacon scanning and that will be now explained.

An example of the detected list of beacons is the following:

```
[IBeacon{rssi=-91, major=1000, minor=575, uuid='fda50693-a4e2-4fb1-afcf-c6eb07647825'},
IBeacon{rssi=-84, major=1000, minor=539, uuid='fda50693-a4e2-4fb1-afcf-c6eb07647825'},
IBeacon{rssi=-91, major=1000, minor=515, uuid='fda50693-a4e2-4fb1-afcf-c6eb07647825'},
IBeacon{rssi=-94, major=1000, minor=575, uuid='fda50693-a4e2-4fb1-afcf-c6eb07647825'},
IBeacon{rssi=-91, major=1000, minor=515, uuid='fda50693-a4e2-4fb1-afcf-c6eb07647825'},
IBeacon{rssi=-92, major=1000, minor=515, uuid='fda50693-a4e2-4fb1-afcf-c6eb07647825'},
IBeacon{rssi=-85,  major=1000,  minor=552,  uuid='fda50693-a4e2-4fb1-afcf-c6eb07647825'}]
```

For this part of the app to be developed, a simple app has been created. This one only performs the beacon scanning and the HTTP Request to the API. It can be found in GitHub [15].

For this part of the project to work, the Bluetooth permission has to be activated. This is because it has to be specified in the app that it will need to make use of the Bluetooth antenna of the phone.

BOSSA Platform API HTTP Request

Once the mobile app has got a list of beacons available, it will be able to perform the HTTP GET Request to the API. This list is in the format of a list of objects of the class IBeacon. For the API to calculate the indoor location, a list of beacons in JSON format is needed.

The app contains a class called Json, that will be used to convert the list of beacons to the correct format required by the API. The URL used to make the HTTP Request to the API has the following format:

```
baseUrl?json[]={jsonObject1}&json[]={jsonObject2}&...&json[]={jsonObjectN}&algorithm=1
```

Where the base URL is https://api.iitrtclab.com/indoorlocation/xml?. Each JSON beacon will be inserted in the following format:

```
json[]={jsonObject1}
```

The JSON objects are separated by “&” between them and separated by “?” from the base URL. Each JSON object has the following format:

```
{"major":<major>,"minor":<minor>,"rssi":<rssi>}
```

An example of a complete URL is the following:

```
https://api.iitrtclab.com/indoorlocation/xml?json[]={"major":1000,"minor":575,"rssi":-
91}&json[]={"major":1000,"minor":539,"rssi":-
84}&json[]={"major":1000,"minor":515,"rssi":-
91}&json[]={"major":1000,"minor":575,"rssi":-
94}&json[]={"major":1000,"minor":515,"rssi":-
91}&json[]={"major":1000,"minor":515,"rssi":-92}&json[]={"major":1000,"minor":552
,"rssi":-85}&algorithim=1
```

The HTTP GET Request will be performed by the use of the class HttpGetRequestTask. This class performs a simple HTTP GET Request and returns the response. The response comes in XML Format. The format of the response can be seen in the following example:

```
<presence>
    <tuple>
        <status>
            <gp:geopriv>
                <gp:location-info>
                    <ca:civicAddress>
                        <ca:country>US</ca:countr y>
                        <ca:A1>IL</ca:A1>
                        <ca:A2>Chicago</ca:A2>
                        <ca:A6>31st</ca:A6>
                        <ca:PRD>W</ca:PRD>
                        <ca:STS>St</ca :STS>
                        <ca:HNO>10</ca:HNO>
                        <ca:FLR>1</ca:FLR>
                        <ca:x>19.247809719012466</ca:x>
                        <ca:y>20.38418379 3285832</ca:y>
                    </ca:civicAddress>
                </gp:location-info>
            </gp:geopriv>
            <timestamp>2019-04-24T22:35:11.4 01Z</timestamp>
        </status>
    </tuple>
</presence>
```

As it can be observed, the answer provides the country, state, city, address, floor, x and y.

For this part of the project to work, the internet permission has to be activated in the app. This allows the app to connect to the internet and send or receive information.

Establish SIP Call through the ESInet

Finally, once, the app has got the indoor location available, it will have to insert it in the SIP INVITE message and establish the call. For the call to be established, the calling device has to send a SIP INVITE into the ESInet. The application has to be registered with the ESInet. In our case, we register the application with the Session Border Controller (SBC) at the edge of the ESInet.

Before sending the SIP INVITE, the app will insert the indoor location into the MIME Body of the message. The app will make use of two classes that have been created to insert indoor location in the MIME message of the SIP INVITE: MIMEpart and MIMEmessage.

- MIMEpart will define a part of the message. It will include the content and some parameters that describe the content.
- MIMEmessage will include all the parts of the class MIMEpart, separated by boundaries. This will be inserted in the SIP INVITE.

The class NG911MessageFactory has been created in order to create the SIP INVITE. This class contains a method called createInviteNG911. The parts that will be inserted in the SIP INVITE are the SDP (Session Description Protocol) message and the indoor location. This will be done in the method createInviteNG911 of this class.

The Sipdroid [12] class is the main activity of the Sipdroid App. Once executed, it will start the 911 SIP call. This class is part of the Sipdroid Library, and will interact with other Sipdroid classes in order to establish the SIP Call.

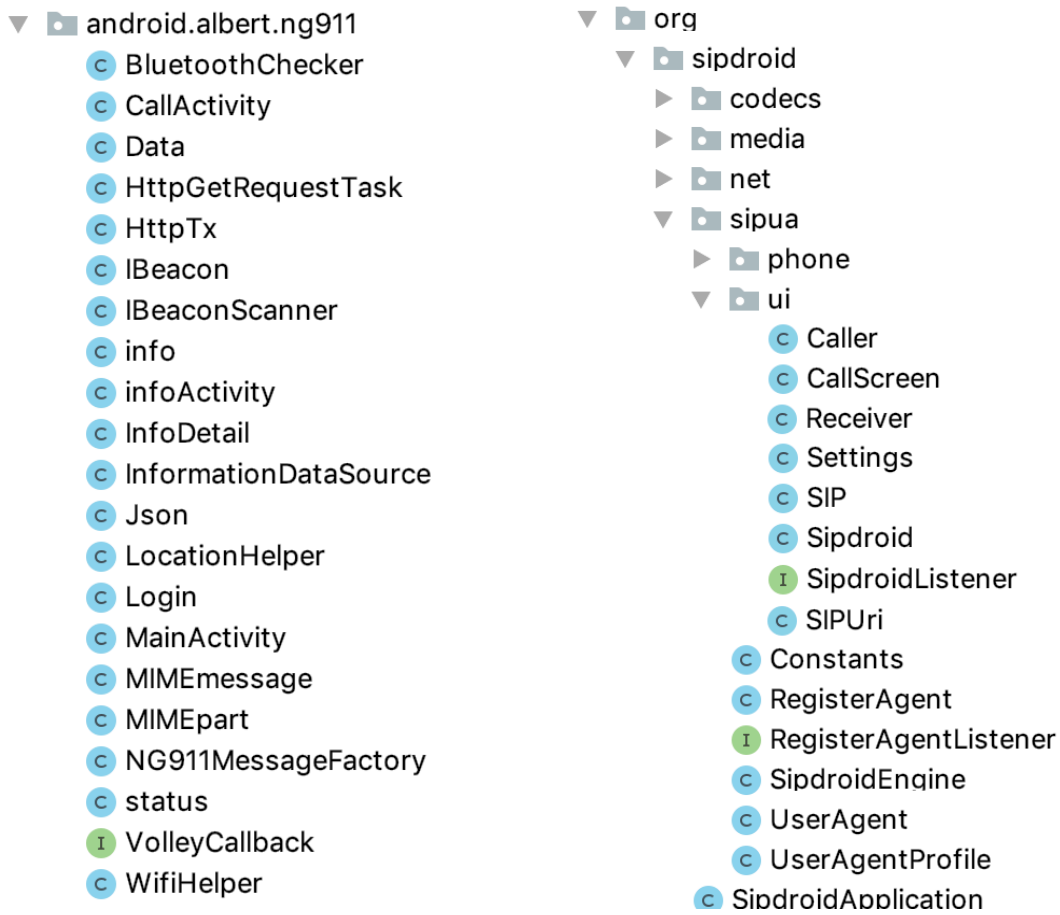

Figure 19: NG911 Created classes (left) and main Sipdroid classes (right)

Ladder diagram of a call

The ladder diagram of a call shows the points through which the call travels. This diagram has been taken out of a Wireshark trace corresponding to a call established in the lab, so instead of having a mobile device calling, the call is established from the SIP-C.

The diagram in Figure 20 has been obtained from a Wireshark trace.

Figure 20: Ladder diagram of a NG911 call from the SIP-C.

The steps are the following:

- The SIP-D send a SIP INVITE to the SIP-D.
- The SIP-D queries the LOST1 server to be able to know the address of the SBC.
- The LOST1 answers with the address of the SBC.
- The SIP-D answers with a 100 Trying message and forwards the SIP INVITE to the SBC.
- The SBC answers with a 100 Trying message and forwards the SIP INVITE to the ESRP.
- The ESRP queries the LOST2, in order to discover the address of the nearest PSAP.
- The LOST2 takes a look at the a1 and a2 parameters of the location, and answers with the address of the PSAP that corresponds to them.
- The ESRP answers with a 100 Trying message and forwards the SIP INVITE to the indicated PSAP.
- The PSAP answers with a 100 Trying message and forwards the SIP INVITE to a call taker.
- The call taker answers with a 100 Trying message.
- When the call taker answers the phone, it sends 180 Trying message to the PSAP, which will forward it in the same order as the SIP INVITE was forwarded but inverted.
- The same happens with the 200 OK message from the call taker to the SIP-C.
- Finally, the SIP-C will send an ACK, which will mean that the call is established.
- RTP Communication

If the call was established from a mobile device, the flow would be the one shown in Figure 21.

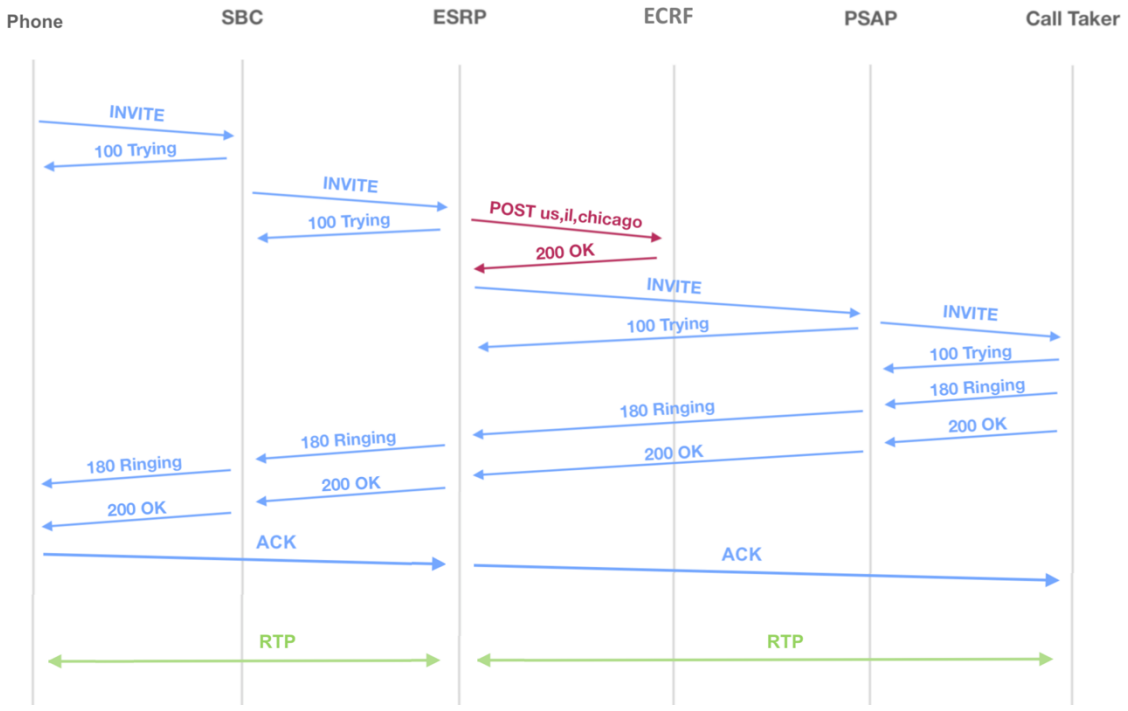

Figure 21: Ladder diagram of a NG911 call from a phone.

For this to work, the caller has to be previously registered in the SBC.

Results

Description

After the app being developed, it has been tested in Stuart Building. The flow of the app is the following:

When the user opens the app, the main screen is the one shown in Figure 22:

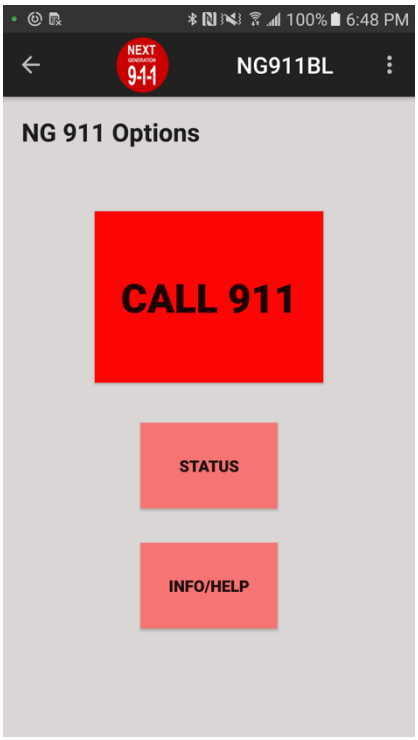

Figure 22: NG911-Indoor-Location app Main Screen.

This screen has several options:

- **Settings**: if the user clicks on the three dots on the top right corner, a “Settings” button appears there and the user can click on settings. This will redirect the user to the settings screen (Figure 23).

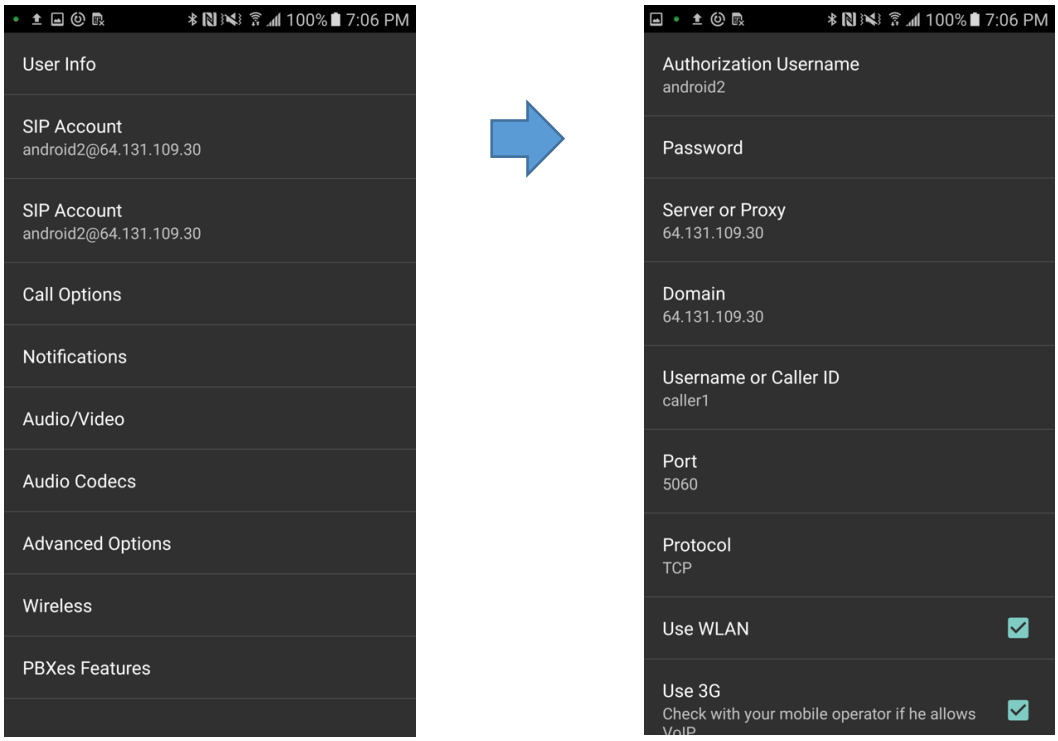

Figure 23: NG911-Indoor-Location app general Settings Screen (left) and SIP Account Settings Screen (right).

- **Status**: if the user clicks on "status", the app will show the status of the captured Bluetooth signals, in a new screen (Figure 24).

Figure 24: NG911-Indoor-Location app Status Screen.

- **Info/help**: if the user clicks on "info/help", the app will show a screen with different information that can be of help to the user in an emergency situation (Figure 25).

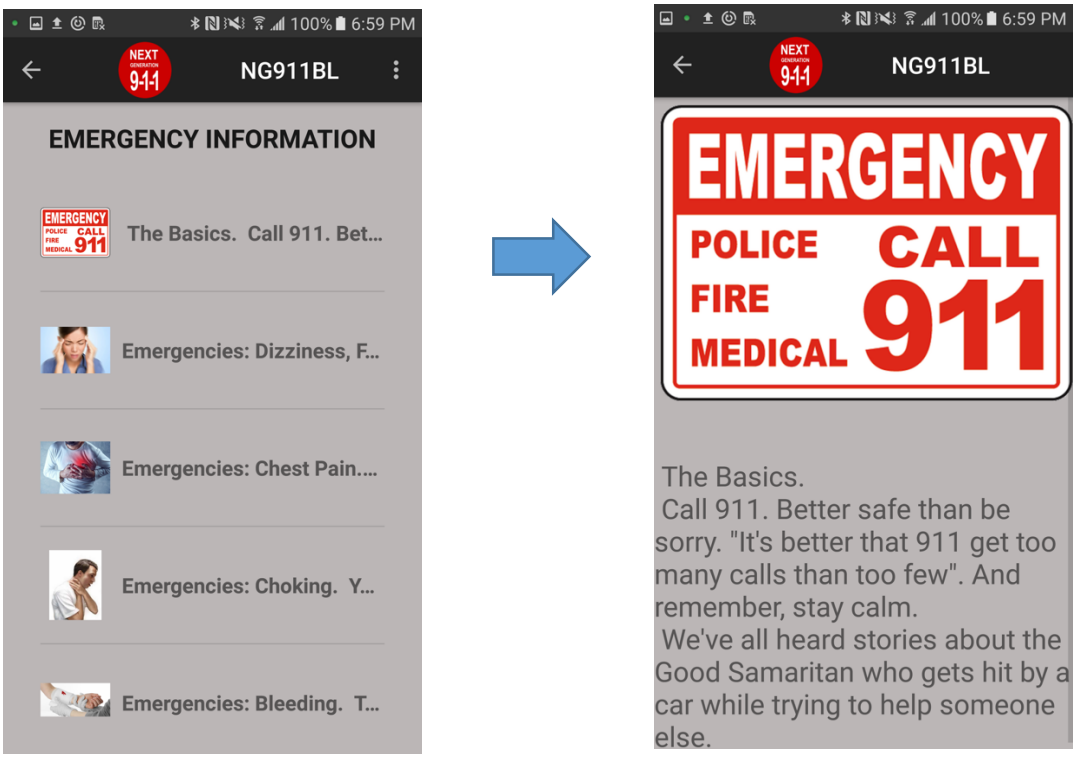

Figure 25: NG911-Indoor-Location app Info/Help Screens.

- **Call 911**: this is the main button of the NG911 app. When the user clicks on "CALL 911", the app performs the process of scanning for beacons, getting the indoor location and establishing the SIP call. While the app is scanning for beacons, it will show a splash screen (Figure 26). After obtaining the indoor location, it will establish the SIP call and show the Sipdroid Call Screen (Figure 27).

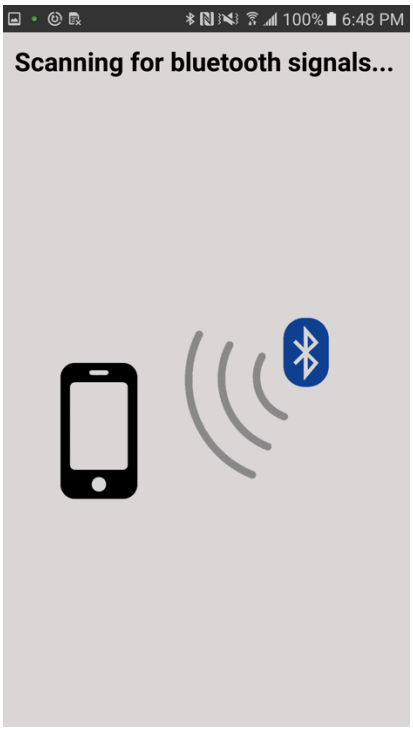

Figure 26: NG911-Indoor-Location app Call Splash Screen.

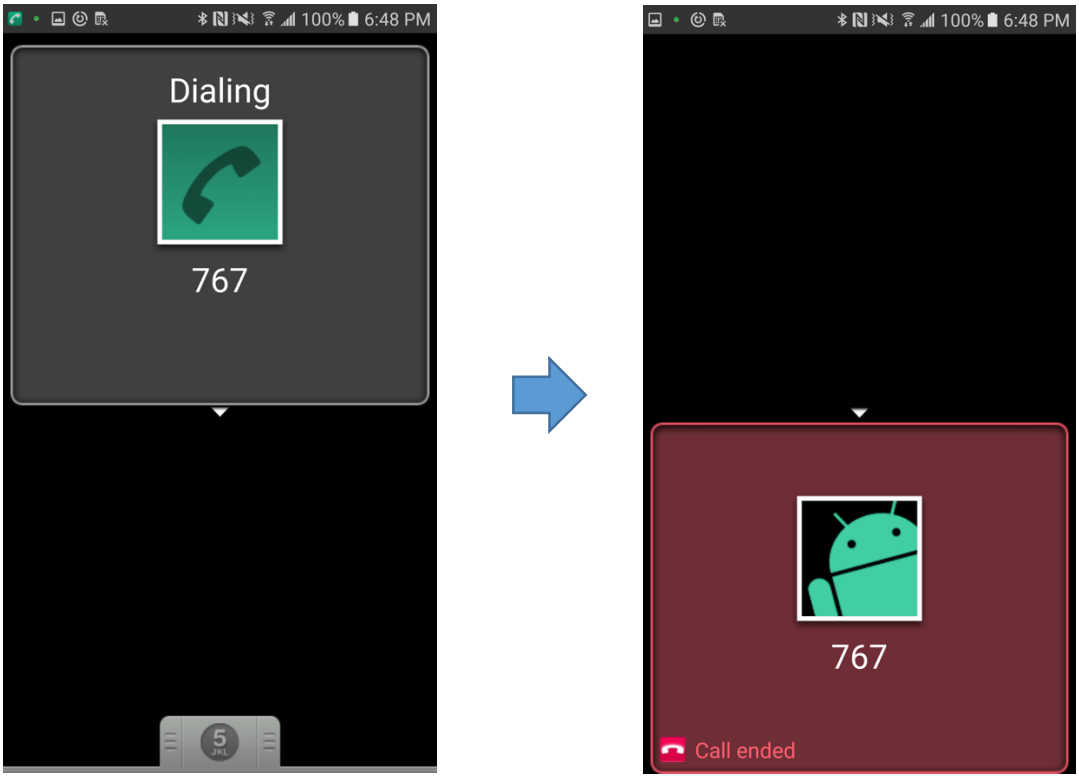

Figure 27: NG911-Indoor-Location app Sipdroid Call Screen, dialing (left) and call ended (right).

Interpretation

The app performs the following steps when “CALL NG911” button is clicked:

- In the case that the app doesn’t have the needed permissions, it requests for the permissions to the user.
- The app scans for Bluetooth beacons signals and detects the nearby beacons correctly.
- The app converts the list of detected beacons with their RSSI to JSON objects in order to be inserted in the URL that will be used to make the HTTP request.
- The app creates a URL using the JSON objects that represent the beacons.
- The app performs the HTTP request with this URL and receives the indoor location that corresponds to that spot. Tests have been performed in Stuart Building in order to test this. The floor has been correctly guessed and the x, y coordinates are accurate within approximately 5 meters.

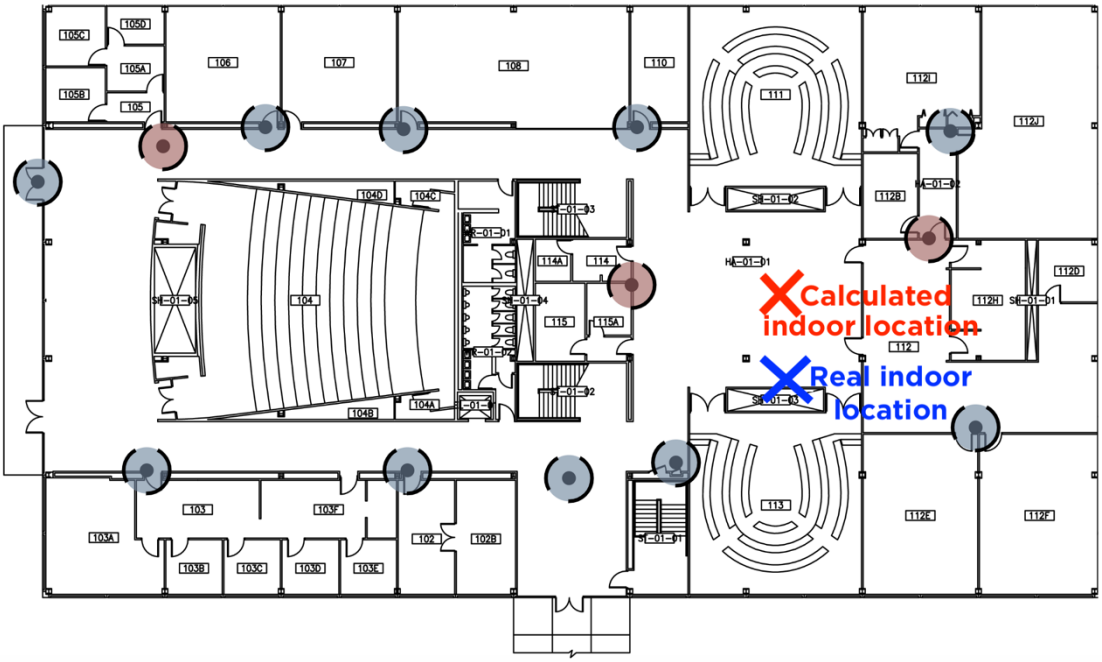

Figure 28: Calculated indoor location in Stuart Building in comparison with real indoor location.

- The app starts the Sipdroid activity after having received the correct indoor location.
- The app generates the correct INVITE message, inserting the received indoor location in the correct format. In order to know this, a comparison has been made between the created INVITE message in the NG911 App and a message captured with Wireshark for a

successful communication using the Sipdroid App. These messages can be observed in Appendix C: SIP INVITE

Apart from the correct expected behavior of the NG911-Indoor-Location app, there has been some issues when establishing the call.

The SIP Account doesn't get correctly registered in the SBC, so the call does not get correctly established. We have observed that the call does not even leave the phone, as there are no SIP messages coming from the phone that we can see in Wireshark. This needs to be examined for future development of the app.

Troubleshooting

Scenario to capture with Wireshark

In order to be able to capture the messages from the phone with Wireshark, the scenario shown in Figure 29 has been setup.

- The Android phone with which the tests have been done and that has been used in order to develop the app doesn't have any SIM card, so it needs to be connected to the Wi-Fi.
- Wireshark is running in a laptop (MacOS). In order to capture the traffic coming from the Android phone, we need to make it go through the laptop.
- For this reason, the laptop needs to provide Wi-Fi to the Android phone, by creating a Wi-Fi network to which the one the Android phone will connect.
- The laptop only has one Wi-Fi interface, so in order to use it to provide Wi-Fi to the Android phone, it won't be able to get its internet connection by connecting to a Wi-Fi network (like the Wi-Fi network of a house, university…).
- The internet has to be shared with the laptop using a cable.
- An iPhone with LTE data has been used to provide Internet connection to the laptop, connected by a wire.
- The traffic goes from the Android phone, through the laptop (where Wireshark is running), through the iPhone, which is connected to the Internet by LTE celular data.

Figure 29: Setup to capture the traffic of the phone in Wireshark (running in a laptop).

Comparison between NG911 app and Sipdroid app

We can perform a SIP call directly using the Sipdroid app. The difference between using this and using the NG911 app is that if use the Sipdroid app no indoor location will be sent in the SIP INVITE, it will only be a SIP Call.

Using the phone, several calls have been made with the Sipdroid app, which have been successfully established. The settings in order for this to succeed are the following:

- Authorization username: android2
- Password: teamc255
- Server or proxy: 64.131.109.30
- Domain: 64.131.109.30
- Username or Caller ID: caller1
- Port: 5060

When the call is successfully established we have 2 different indicators. First of all, we can see that the dialing screen changes to “Call in progress” (Figure 30). Apart from this, we can see the call coming into the Micro Automation PSAP if we connect to it (Figure 31).

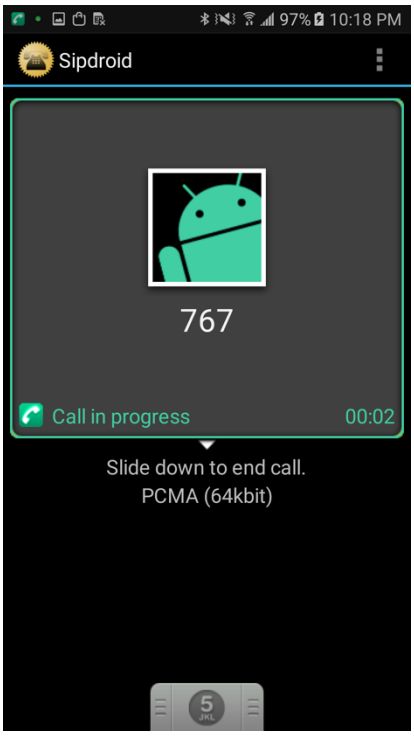

Figure 30: Screenshot of Sipdroid app when the call is in progress.

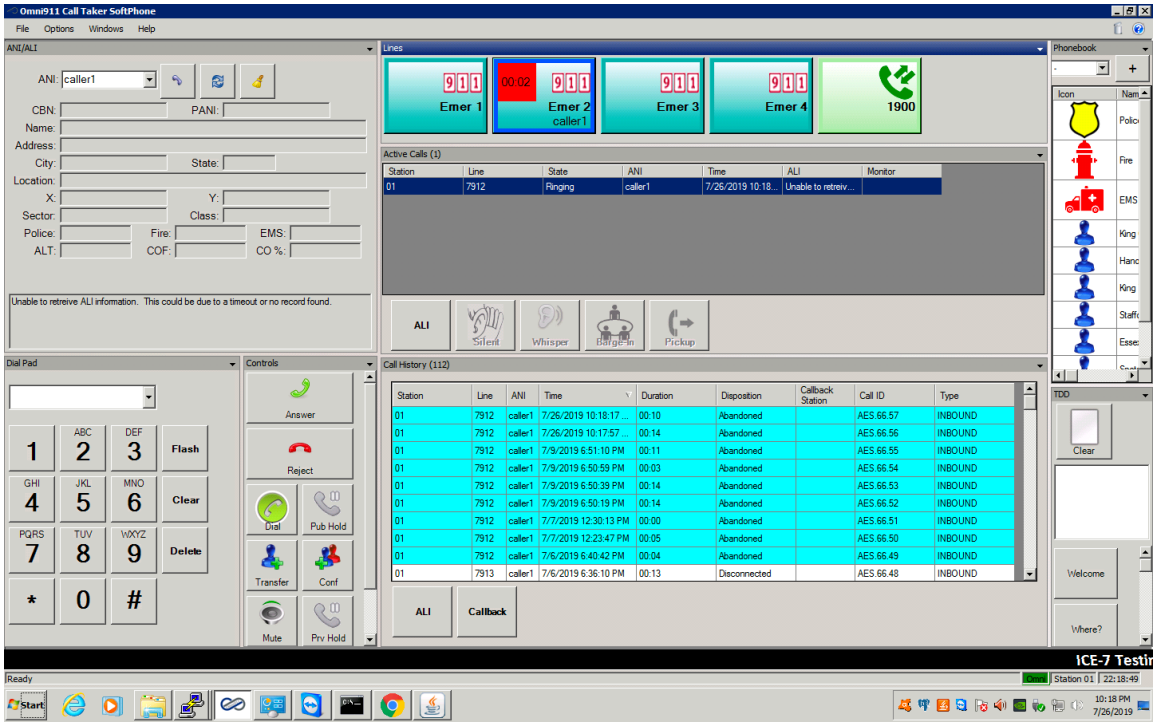

Figure 31: Screenshot of the Micro Automation where we can see the call coming in.

For Sipdroid, there are 2 possible cases:

- Call gets correctly established: the call can be seen coming into the PSAP and the SIP Messages captured in Wireshark.
- Call doesn't get established: no SIP messages can be seen leaving the phone in Wireshark. This case is the same as when the user tries to call 911 using the NG911 app.

Conclusions and Future Development

The Android app that has been developed in this project, by the collaboration with the other modules (the array of beacons, the BOSSA Platform and the ESInet), brings the emergency system to a new level, where by pressing a button, the police can obtain a very accurate location of a person in a building. By the use of very simple devices and technologies, a huge improvement can take place.

The developed app is able to combine different technologies and interact with all modules of the NG911 system, performing the needed conversions between all of them. It can obtain a list of beacons that are near the phone by the use of Bluetooth, it can perform an HTTP Request, interacting with the BOSSA Platform to obtain the indoor location and it can establish a SIP call through the ESInet, including the indoor location in it. This lets the system not only locate the caller in a more accurate and easy way, but also send the call to the nearest PSAP, so that the emergency is solved in the most efficient way.

This system is moving forward, and can still be improved in several ways. One direction in which the project can be improved is by providing the updated location to the police, after the call has been made.

The location algorithm can also be improved in accuracy, and also by providing information such as the name of the rooms instead of the x, y coordinates.

This system can be improved in many ways, but it is the base for the future emergency system. According to the FCC, more than 10.000 lives a year could be saved if the location of a person is provided when calling 911.

References

[1] GuardLlama Article on NG911, 2019 [Online] Available: https://guardllama.com/blogs/news/10-000-lives. [Accessed in 2019]

[2] FCC, Wireless E911 Location Accuracy Requirements, 2014 [Online] Available: https://www.documentcloud.org/documents/2195636-fcc-third-nprm-february-2014.html#document/. [Accessed in 2019]

[3] RFC 3261: Session Initiation Protocol (SIP), 2002 [Online] Available: https://tools.ietf.org/html/rfc3261. [Accessed in 2019]

[4] RFC 3665: Session Initiation Protocol (SIP) Basic Call Flow Examples, 2003 [Online] Available: https://tools.ietf.org/html/rfc3665. [Accessed in 2019]

[5] RFC 2616: HyperText Transfer Protocol (HTTP/1.1), 1999 [Online] Available: https://tools.ietf.org/html/rfc2616. [Accessed in 2019]

[6] C. Davids, V. K. Gurbani, S. Loreto y R. Subramanyan, «Next Generation 911: Where Are We? What Have We Learned? What lies ahead?» IEEE Communications Magazine, January 2017.

[7] C. Davids, J. Moreno Valdecantos, B. Dworak, C. Tovar, B. Ramaswamy Nandakumar and M. Patil, "Dispatchable Indoor Location for Mobile Phones Calling for Emergency Services," pp. 21-27, 2015.

[8] C. Davids, C. Davids, B. Ramaswamy Nandakumar, N. Okhandiar, F. Rois, C. Ljazouli and A. Calle Murillo, "Dispatchable Indoor Location System for Mobile Phones based on a Bluetooth Low Energy Array," IEEE, pp. 1-8, 2017.

[9] NG911-Indoor-Location Android App GitHub Repository, Carmen Sirés, 2018 [Online] Available: https://github.com/carmensires/NG-911-Bluetooth-Indoor-Location.

[10] IEEE 802.15.1-2002 - IEEE Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN - Specific Requirements [Online]. Available: https://standards.ieee.org/standard/802_15_1-2002.html [Accessed in 2019].

[11] Android Beacon Library [Online]. Available: https://github.com/AltBeacon/android-beacon-library. [Accessed in 2019]

[12] Sipdroid [Online] Available: https://github.com/i-p-tel/sipdroid. [Accessed in 2019]

[13] Wireshark trace for a NG911 call from the SIP-C in IIT RTC Lab, September 2018 [Online] Available: https://drive.google.com/file/d/1D6Mt1iVbcmTjAiPcR4f2F4klOvYQNRDZ/view?usp=sharing [Accessed in 2019]

[14] Test App GitHub Repository, Luke Logan, 2018 [Online] Available: https://github.com/lukemartinlogan/TestApp.

[15] Bluetooh and HTTP Test App GitHub Repository, Carmen Sirés, 2019 [Online] https://github.com/carmensires/Bluetooth_http_TestApp.

[16] RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format, 2017 [Online]. Available: https://tools.ietf.org/html/rfc8259. [Accessed in 2019]

[17] W3C: Extensible Markup Language, 2008 [Online]. Available: https://www.w3.org/TR/xml/. [Accessed in 2019]

Appendices

Appendix A: NG911-Indoor-Location Android App Code

The code can be found on the GitHub repository for the app [9]. The main files are the following:

AndroidManifest.xml

The android manifest is the file that specifies the configuration for the app. It describes the activities used by the app and declares the needed permissions for the app.

Permissions

```
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-permission android:name="android.permission.CALL_PHONE" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.WRITE_INTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_INTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"></uses-
permission>
    <uses-permission android:name="android.permission.RECORD_AUDIO"></uses-permission>
    <uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS"></uses-
permission>
    <uses-permission android:name="android.permission.WRITE_SETTINGS"></uses-permission>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"></uses-
permission>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"></uses-
permission>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"></uses-
permission>
    <uses-permission android:name="android.permission.READ_CONTACTS"></uses-permission>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"></uses-permission>
    <uses-permission android:name="android.permission.WAKE_LOCK"></uses-permission>
    <uses-permission android:name="android.permission.DISABLE_KEYGUARD"></uses-
permission>
    <uses-permission android:name="android.permission.CAMERA"></uses-permission>
    <uses-permission android:name="android.permission.VIBRATE" ></uses-permission>
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE" ></uses-
permission>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" ></uses-
permission>
    <uses-permission android:name="android.permission.GET_ACCOUNTS" ></uses-permission>
    <uses-permission android:name="android.permission.BROADCAST_STICKY" ></uses-
permission>
    <uses-permission android:name="android.permission.READ_CALL_LOG"></uses-permission>
    <uses-permission android:name="android.permission.WRITE_CALL_LOG"></uses-permission>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"></uses-
permission>
    <uses-permission android:name="android.permission.INTERACT_ACROSS_USERS_FULL"></uses-
permission>
```

Activities

```
        <activity
            android:name="com.android.albert.ng911.MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar"
            android:launchMode="singleInstance">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
                <category android:name="android.intent.category.DEFAULT"/>
            </intent-filter>
        </activity>
          <activity
              android:name="com.android.albert.ng911.CallActivity"
              android:label="@string/main_name"
              android:theme="@style/AppTheme.NoActionBar">
              <meta-data
                  android:name="android.support.PARENT_ACTIVITY"
                  android:value="com.android.albert.ng911.MainActivity" />
          </activity>
          <activity
              android:name="com.android.albert.ng911.status"
              android:label="@string/main_name"
              android:theme="@style/AppTheme.NoActionBar">
              <meta-data
                  android:name="android.support.PARENT_ACTIVITY"
                  android:value="com.android.albert.ng911.MainActivity" />
          </activity>
          <activity
              android:name=".infoActivity"
              android:label="@string/main_name"
              android:theme="@style/AppTheme.NoActionBar">
              <meta-data
                  android:name="android.support.PARENT_ACTIVITY"
                  android:value="com.android.albert.ng911.MainActivity" />
          </activity>
        <activity
            android:name="com.android.albert.ng911.InfoDetail"
            android:label="@string/main_name"
            android:theme="@style/AppTheme.NoActionBar">
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value="com.android.albert.ng911.MainActivity" />
        </activity>
```

MainActivity.java

This is the first activity that will be shown when the app is opened. The function that is called when the screen is shown is the onCreate() method.

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        getSupportActionBar().setLogo(R.drawable.ng911icon2);
        getSupportActionBar().setDisplayUseLogoEnabled(true);
        //Bluetooth turn on
        bluetooth=new BluetoothChecker(getApplicationContext());
        bluetooth.enableBluetooth();
        //turn wifi on
        WifiHelper wfhelper = new WifiHelper(this);
        wfhelper.enableWifi();
        //create views
        bindViews();
        bindButtons();

        fa=this;

        //Show first name of the user
        SharedPreferences SP =
PreferenceManager.getDefaultSharedPreferences(getBaseContext());
        String strFName = SP.getString("fname", "");
        nameView = (TextView) findViewById(R.id.firstname);
        //if registered get from facebook account
        try {
            if (Profile.getCurrentProfile().getFirstName()!="")
                strFName=Profile.getCurrentProfile().getFirstName();
        }catch(Exception e){System.out.println(e);}
        nameView.setText(strFName);
        permissionsCheck();
    }
```

If the user has not granted the needed permissions before opening, he/she will be asked to grant them when opening the app.

```
    private void permissionsCheck() {
        Log.i("AAAA " + MAIN_ACTIVITY,"checking permissions");
        if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)) {
            finish();
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.BLUETOOTH) !=
PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.BLUETOOTH},
                    1);
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.BLUETOOTH_ADMIN)
!= PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
```

```
                    new String[]{Manifest.permission.BLUETOOTH_ADMIN},
                    2);
        }
        if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.ACCESS_COARSE_LOCATION},
                    3);
        }
        if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
                    4);
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.INTERNET) !=
PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.INTERNET},
                    5);
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_CALL_LOG) !=
PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.READ_CALL_LOG},
                    6);
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.WRITE_CALL_LOG)
!= PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.WRITE_CALL_LOG},
                    7);
        }
        if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_NETWORK_STATE) != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.ACCESS_NETWORK_STATE},
                    7);
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_CONTACTS) !=
PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.READ_CONTACTS},
                    7);
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.WRITE_CONTACTS)
!= PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.WRITE_CONTACTS},
                    7);
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.WRITE_SETTINGS)
!= PackageManager.PERMISSION_GRANTED) {
            boolean settingsCanWrite = Settings.System.canWrite(MainActivity.this);
            if (!settingsCanWrite) {
                Intent intent = new Intent(Settings.ACTION_MANAGE_WRITE_SETTINGS);
                startActivity(intent);
            }
        } }
```

The MainActivity has got several buttons that can be pressed. Their listeners are the following:

```
public class StartOnClickListener implements View.OnClickListener {

        @Override
        public void onClick(View v) {

                switch (v.getId()) {
                    case R.id.callButton:
                        Intent intent = new Intent(v.getContext(), CallActivity.class);
                        startActivity(intent);
                        Log.i("AAAA " + MAIN_ACTIVITY,"Call button pressed");
                        break;

                    case R.id.statusButton:
                        Intent intent2 = new Intent(v.getContext(), status.class);
                        startActivity(intent2);
                        Log.i("AAAA " + MAIN_ACTIVITY, "Status button pressed");
                        break;

                    case R.id.infoButton:
                        Intent intent3 = new Intent(v.getContext(), infoActivity.class);
                        startActivity(intent3);
                        Log.i("AAAA " + MAIN_ACTIVITY, "Info button pressed");
                        break;
                }

        }
    }
```

On the top of the screen, a menu bar is shown, with the option to go to the app settings;

```
@Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        switch (item.getItemId()) {

            //noinspection SimplifiableIfStatement
            case R.id.action_settings:
                Intent myIntent = new Intent(this, org.sipdroid.sipua.ui.Settings.class);
                startActivity(myIntent);
                return true;
            // For going back to home. Handle back button toolbar
            case android.R.id.home:
                finish();
                return true;
        }

        return super.onOptionsItemSelected(item);
    }
```

CallActivity.java

When the user clicks on the CALL 911 button, the CallActivity is started. This is the activity that calls the class to scan for the Bluetooth beacons, and when it ends, performs the http request to the location server of the BOSSA platform. After this has finished, it opens the Sipdroid activity, to establish the call. While performing all of this logic, it shows a splash activity.

When the CallActivity is created, it starts an instance of the IBeaconScanner activity, which performs the scanning and when it finishes, it performs the HTTP GET request.

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_splash);
        sended=false;

        //When the Splash time finishes, it will open the Sipdroid Activity
        new Handler().postDelayed(new Runnable() {
            public void run() {
                Intent intent = new Intent();
                intent.setClass(CallActivity.this, Sipdroid.class);
                CallActivity.this.startActivity(intent);
                CallActivity.this.finish();
            }
        }, SPLASH_DISPLAY_TIME);

        try {
            //Create bluetooth scanner
            bluetoothScanner = new IBeaconScanner(this, period);
            bluetoothScanner.start(scan_period);
        }
        catch(Exception e) {
            e.printStackTrace();
            finish();
        }
    }
```

IBeaconScanner.java

This class is initialized with the scanning duration (5 seconds) and the period between scanning (0.5 seconds). After creating the instance, the start method is called.

```
    //Starts the scanning during the established scan_period
    public void start(int scan_period) {
        try {
            this.scan_period = scan_period;
            this.current_time = 0;
            this.beaconManager.bind(this);
            this.timer.schedule(this, 0, Math.round(period * 1000));
            this.scan_enabled = true;
        }
        catch(Exception e) {
            e.printStackTrace();
            stop();
        }
    }
```

The detected beacons are stored in a list in this same class.

```
    private final RangeNotifier beaconCB = new RangeNotifier() {
        @Override
        public void didRangeBeaconsInRegion(Collection<Beacon> beacons, Region region) {
            if(!scan_enabled)
                return;
            for(Beacon x:beacons){
                IBeacon beacon = new IBeacon(x.getRssi(),
x.getId2().toInt(),x.getId3().toInt(),x.getId1().toString());
                Log.i(IBEACON_SCANNER,"beacon detected: "+beacon.toString());
                beaconList.add(beacon);

            }
            Log.i(IBEACON_SCANNER,"beacon list: "+beaconList.toString());
        }
    };
```

When the scanning stops, the stop method is called.

```
    //This is called when the scanning stops
    public void stop() {
        timer.cancel();
        timer.purge();
        beaconManager.unbind(this);
        scan_enabled = false;
        this.finished= true;
        makeHttpRequest();
    }
```

The makeHttpRequest method performs the http get (using the HttpTx class) request and saves the data in a static data class, to be used by Sipdroid. In this method, the JSON class is used to

convert the list of beacons (objects of the class IBeacon) to a JSON string that needs to be used in the URL to perform the HHTTP GET request.

```
//Makes an HTTP get request to obtain the indoor location
    public void makeHttpRequest(){
        for(IBeacon b: beaconList){
            try {

json.updateMyJsonIndoor(String.valueOf(b.getMajor()),String.valueOf(b.getMinor()),String.
valueOf(b.getRssi()));
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }

        try {
            String result = httptx.HttpGetRequest(json.readMyJson());
            Data d = Data.getInstance();
            d.setReceived(result);
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
```

IBeacon.java

This class defines the beacons. It uses the altbeacon library.

```
public class IBeacon {
    private int major, minor;
    private String uuid;
    private LinkedList<Integer> rssiSet;
    private double x, y;

    public IBeacon(int rssi, int major, int minor, String uuid) {
        rssiSet = new LinkedList<>();
        rssiSet.add(rssi);
        this.major = major;
        this.minor = minor;
        this.uuid = uuid;
    }

    public int getRssi() {
        int sum = 0;
        for (int x:rssiSet)
            sum+=x;
        return sum/rssiSet.size();
    }

    public void add(int rssi) {
        rssiSet.add(rssi);
    }

    public int getMajor() {
        return major;
    }
```

```
    public void setMajor(int major) {
        this.major = major;
    }

    public int getMinor() {
        return minor;
    }

    public void setMinor(int minor) {
        this.minor = minor;
    }

    public void setPosition(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public String getUuid() {
        return uuid;
    }

    public void setUuid(String uuid) {
        this.uuid = uuid;
    }
    public long getKey(){
        return getKey(major,minor);
    }
    public static long getKey(int major,int minor){
        return Long.parseLong(""+major+minor);
    }

    @Override
    public String toString() {
        return "IBeacon{" +
                "rssi=" + getRssi() +
                ", major=" + major +
                ", minor=" + minor +
                ", uuid='" + uuid + '\'' +
                '}';
    }
}
```

HttpTx.java

This class is used to perform the Http request. It receives the JSON string to insert in the URL.

```
public static String baseUrl = "https://api.iitrtclab.com/indoorlocation/xml?";
public String HttpGetRequest(String json) {
        String url = baseUrl+json;
        try {
            result = new HttpGetRequestTask().execute(url).get();
        } catch (ExecutionException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return result;
    }
```

Json.java

To set the Json coordinates:

```
public void updateMyJsonIndoor( String Major, String Minor, String Rssi) throws
JSONException {
        JSONObject json = new JSONObject();
        json.put("major", Integer.parseInt(Major));
        json.put("minor", Integer.parseInt(Minor));
        json.put("rssi", Integer.parseInt(Rssi));
        beaconsJson.put(json);
    }
```

To read the coordinates in order to be inserted in the URL:

```
public String readMyJson()throws JSONException {
        String jsonStr = "";
        JSONObject json_i;
        for(int i=0;i<beaconsJson.length();i++){
            json_i = beaconsJson.getJSONObject(i);
            jsonStr+="json[]="+json_i.toString()+"&";
        }
        jsonStr+="algorithim=1";
        return jsonStr;
    }
```

Data.java

When the indoor location (in xml) is received from the server, it is stored in the static class Data.java. It will be retrieved from here when it needs to be inserted in the MIME body of the SIP INVITE.

```
public class Data {

    public String captured,json;
    public final static Data data= new Data( );
```

```
    final String DATA="Data";

    public Data(){ }

    /* Static 'instance' method */
    public static Data getInstance( ) {
        return data;
    }

    public String getReceived()
    {
        return this.captured;
    }

    public void setReceived(String captured)
    {
        this.captured = captured;
    }

    public String getJson()
    {
        return this.json;
    }

    public void setJson(String b)
    {
        this.json = b;
    }

    public void deleteReceived()
    {
        this.captured=null;
    }

    public void clear() {
        this.captured="";
    }
}
```

MIMEpart.java

The indoor location in XML format has to be inserted in the MIME body of the SIP INVITE. For this, first the MIME part is represented by this class.

```
public class MIMEpart {
    String version;
    String contentID;
    String contentType;
    String contentTransferEncoding;
    String content;

    /**
     * Constructor with all the fields
     *
     * @param version
     * @param contentID
     * @param contentType
     * @param contentTransferEncoding
```

```
     * @param content
     */
    public MIMEpart(String version, String contentID, String contentType, String
contentTransferEncoding, String content) {
        this.version = version;
        this.contentID = contentID;
        this.contentType = contentType;
        this.contentTransferEncoding = contentTransferEncoding;
        this.content = content;
    }

    /**
     * Constructor without specifying MIME version
     *
     * @param contentID
     * @param contentType
     * @param contentTransferEncoding
     * @param content
     */
    public MIMEpart(String contentID, String contentType, String contentTransferEncoding,
String content) {
        this.version = "1.0";
        this.contentID = contentID;
        this.contentType = contentType;
        this.contentTransferEncoding = contentTransferEncoding;
        this.content = content;
    }

    /**
     * Most basic constructor.
     *
     * @param contentID
     * @param contentType
     * @param content
     */
    public MIMEpart(String contentID, String contentType, String content) {
        this.version = "1.0";
        this.contentID = contentID;
        this.contentType = contentType;
        this.contentTransferEncoding = "8bit";
        this.content = content;
    }

    @Override
    public String toString() {
        String part = "";
        part += "\r\nMIME-Version: " + version;
        part += "\r\nContent-ID: <" + contentID + ">";
        part += "\r\nContent-Type: " + contentType;
        part += "\r\nContent-Transfer-Encoding: " + contentTransferEncoding + "\r\n";
        part += "\r\n" + content;
        part += "\r\n";
        return part;
    }
}
```

MIMEmessage.java

When the MIME part is created, the MIME message can be formed by the different MIME parts.

```
public class MIMEmessage {

    String version;
    String contentType;
    String boundary;
    List<MIMEpart> parts;

    public MIMEmessage(String version, String contentType, String boundary,
List<MIMEpart> parts) {
        this.version = version;
        this.contentType = contentType;
        this.boundary = boundary;
        this.parts = parts;
    }

    /**
     *
     * @param contentType
     * @param boundary
     * @param parts
     */
    public MIMEmessage(String contentType, String boundary, List<MIMEpart> parts) {
        this.version = "1.0";
        this.contentType = contentType;
        this.boundary = boundary;
        this.parts = parts;
    }

    @Override
    public String toString() {
        String mime = "";   //no need to introduce CRLF, it's introduced in the method
'setBody' of 'BaseMessage.java'

        for(MIMEpart part: parts) {
            mime += "--" + boundary;
            mime += part.toString();
        }

        mime += "\r\n--" + boundary + "--\r\n";//last boundary indicating end of MIME
message
        return mime;
    }
}
```

NG911MessageFactory.java

The invite message is created in this class.

```
public static Message createInviteNG911(String call_id, SipProvider sip_provider, SipURL
request_uri, NameAddress to, NameAddress from, NameAddress contact, String body, String
icsi, String location) {
        long cseq = SipProvider.pickInitialCSeq();
        String local_tag = SipProvider.pickTag();
        // String branch=SipStack.pickBranch();
        if (contact == null)
            contact = from;

        String method = SipMethods.INVITE;
        String remote_tag = null;
```

```
        String branch = null;

        String via_addr = sip_provider.getViaAddress();
        int host_port = sip_provider.getPort();
        boolean rport = sip_provider.isRportSet();
        String proto;
        if (request_uri.hasTransport())
            proto = request_uri.getTransport();
        else
            proto = sip_provider.getDefaultTransport();

        String qvalue = null;

        //creation of new message
        Message req = new Message();
        // mandatory headers first (To, From, Via, Max-Forwards, Call-ID, CSeq):

        request_uri.setURL("urn:service:sos");  //NG911
        req.setRequestLine(new RequestLine(method, request_uri));

        ViaHeader via = new ViaHeader(proto, via_addr, host_port);
        if (rport)
            via.setRport();
        if (branch == null)
            branch = SipProvider.pickBranch();
        via.setBranch(branch);
        req.addViaHeader(via);
        req.setMaxForwardsHeader(new MaxForwardsHeader(70));

        //[NG911] Set 'To: ' header to urn:service:sos
        req.setHeader(new Header(SipHeaders.To, "urn:service:sos"));
        //NG911

        req.setFromHeader(new FromHeader(from, local_tag));
        req.setCallIdHeader(new CallIdHeader(call_id));
        req.setCSeqHeader(new CSeqHeader(cseq, method));
        // optional headers:
        // start modification by mandrajg
        if (contact != null) {
            if (((method == "REGISTER")||(method == "INVITE")) && (icsi != null) ){
                MultipleHeader contacts = new MultipleHeader(SipHeaders.Contact);
                contacts.addBottom(new ContactHeader(contact, qvalue, icsi));
                req.setContacts(contacts);
            }
            else{
                MultipleHeader contacts = new MultipleHeader(SipHeaders.Contact);
                contacts.addBottom(new ContactHeader(contact));
                req.setContacts(contacts);
            }
            // System.out.println("DEBUG: Contact: "+contact.toString());
        }
        if ((method == "INVITE") && (icsi != null) ){
            req.setAcceptContactHeader(new AcceptContactHeader(icsi));
        }
        // end modifications by mandrajg
        req.setExpiresHeader(new ExpiresHeader(String
                .valueOf(SipStack.default_expires)));
        // add User-Agent header field
        if (SipStack.ua_info != null)
            req.setUserAgentHeader(new UserAgentHeader(SipStack.ua_info));
```

```
        /**
         * Set custom headers for NG911
         */

        //Expires:
        if(!(req.hasHeader(SipHeaders.Expires))) req.setExpiresHeader(new
ExpiresHeader(3600));
        //Accept-Language:
        Header acceptLanguage = new Header("Accept-Language", "en");
        req.setHeader(acceptLanguage);

        //Priority:
        Header priority = new Header("Priority", "emergency");
        req.setHeader(priority);

        //Geolocation:
        String from_uri = from.getAddress().toString().substring(4);    //remove 'sip:'
part from the URI
        Header geolocation = new Header("Geolocation", "<cid:"+from_uri+">; inserted-
by=\"" + from.getAddress().getHost() + "\"; used-for-routing");
        req.setHeader(geolocation);

        //Date:
        DateFormat dateFormat = new SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss ZZZZ");
        Date date = new Date();
        req.setDateHeader(new DateHeader(dateFormat.format(date)));

        /**
         * Add location to body
         *
         * Location will be added as part of a MIME message
         */

        String randomString = Random.nextString(50);
        String boundary = "NG911" + randomString;   //this is the boundary of the MIME
body
        String mimeType = "multipart/mixed";

        //SDP part of the MIME body
        MIMEpart sdpMIME = new MIMEpart(call_id, "application/sdp", body);

        //Location part of the MIME body
            MIMEpart locationMIME = new MIMEpart(from_uri, "application/pidf+xml",
location);

            //MIME body
            List<MIMEpart> content = new ArrayList<MIMEpart>();
            content.add(sdpMIME);
            content.add(locationMIME);
            MIMEmessage mimeBody = new MIMEmessage(mimeType, boundary, content);

            req.setBody(mimeType + "; boundary=" + boundary + "", mimeBody.toString());
        return req;
    }
```

Appendix B: Management of the elements of the ESInet

ECRF: Manage PSAPs

The first step is to login in the ESXi vmware (10.15.98.30)

- Username: root
- Password: Joejoe11!

Then choose Lost2 (which has the ECRF). The terminal opens and the processes need to be restarted:

```
Username: root
Password: teamc255
service postgresql restart
service tomcat7 restart
```

Then, to add any PSAPs or modify/delete them, the following steps are necessary:

In pgAdmin (127.0.0.1:57660/browser/), we go to lost → lost2 (the password is 911test).

The hierarchy to follow is Databases → lost → schemas → public → tables. Right-clicking on civic_us and choosing View/Edit Data → all rows, a table displays, with all the PSAPs. In the bottom, there is the possibility to add, edit or delete any PSAP.

For example, for the Chicago PSAP, it would be sip:psap@psapd.ng911main.iit.edu. The a1 is "il" and the a2 is "chicago".

After saving it, the processes would need to be restarted as explained previously.

To check that the PSAP has been correctly added, a call can be made and captured with Wireshark.

SBC: Manage Session Agents

The first step is to login using Putty, into 10.15.99.100:

- Username: user
- Password: 0r@cle!!

To add a new session agent, the following steps need to be performed:

```
Main_Campus_SBC> enable
Main_Campus_SBC# configure terminal
Main_Campus_SBC(configure)#session-router
Main_Campus_SBC(session-router)#session-agent
```

```
Main_Campus_SBC(session-agent)# sel
```

By using sel, all the actual session agents are shown. In each step, all the options can be displayed by using "?". After doing "sel", and choosing a session agent, by writing "show" command, all the details of the selected session agent are shown. To add a session agent:

```
Main_Campus_SBC(session-agent)# hostname <ip address>
Main_Campus_SBC(session-agent)# ip_address <ip address>
Main_Campus_SBC(session-agent)# port 5060
Main_Campus_SBC(session-agent)# realm-id public
```

After the question of saving the changes, write "yes", and the session agent would be added. After exiting configuration mode:

```
Main_Campus_SBC# save-config
Main_Campus_SBC# activate-config
```

To delete a session agent:

```
Main_Campus_SBC(session-agent)# no <ip address>
```

All this configuration can be saved with a name:

```
Main_Campus_SBC(configure)# backup-config <name>
```

And restored:

```
Main_Campus_SBC(configure)# restore-backup-config <name>
```

SIP-D: Start process

Connect via Putty to 10.15.98.8:

- Username: root
- Password: teamc255

```
cd /ng911/sipd
./sipd.sh
```

ESRP: Start process

Connect via Putty to 10.15.99.3:

- Username: root
- Password: teamc255

```
cd /ng911/sipd
./sipd.sh
```

PSAPD: Start process in the Columbia Call-taker

Connect via Putty to 10.15.99.7:

- Username: root
- Password: teamc255

```
cd /ng911/psapd
./psapd.sh
```

Appendix C: SIP INVITE

SIP Invite created by NG911

```
INVITE urn:service:sos SIP/2.0
Via: SIP/2.0/TCP 98.193.88.66:33988;rport;branch=z9hG4bK38745 Max-Forwards: 70
To: urn:service:sos
From: <sip:android2@64.131.109.30>;tag=z9hG4bK06143326
Call-ID: 457029456297@98.193.88.66
CSeq: 1 INVITE
Contact: <sip:android2@98.193.88.66:33988;transport=tcp>
Expires: 3600
User-Agent: Sipdroid/1.0/SM-G920T
Accept-Language: en
Priority: emergency
Geolocation: <cid:android2@64.131.109.30>; inserted-by="64.131.109.30";
used-for-routing
Date: Wed, 3 Jul 2019 21:25:59 GMT-05:00 Content-Length: 1146
Content-Type: multipart/mixed;
boundary=NG911f052KQD9zCUS0bkPZgoQFWGJSbmENL88CAspmv1aAoJWEiz2H2
--NG911f052KQD9zCUS0bkPZgoQFWGJSbmENL88CAspmv1aAoJWEiz2H2 MIME-Version: 1.0
Content-ID: <457029456297@98.193.88.66>
Content-Type: application/sdp
Content-Transfer-Encoding: 8bit
v=0
o=android2@64.131.109.30 0 0 IN IP4 98.193.88.66 s=Session SIP/SDP
c=IN IP4 98.193.88.66
t=0 0
m=audio 21000 RTP/AVP 8 0 101
a=rtpmap:8 PCMA/8000
a=rtpmap:0 PCMU/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
m=video 21070 RTP/AVP 103
a=rtpmap:103 h263-1998/90000
--NG911f052KQD9zCUS0bkPZgoQFWGJSbmENL88CAspmv1aAoJWEiz2H2 MIME-Version: 1.0
Content-ID: <android2@64.131.109.30>
Content-Type: application/pidf+xml Content-Transfer-Encoding: 8bit
<presence><tuple><status><gp:geopriv><gp:location-
info><ca:civicAddress><ca:country>US</ca:country><ca:A1>IL</ca:A1><ca:A2>Chica
go</ca:A2><ca:A6>31st</ca:A6><ca:PRD>W</ca:PRD><ca:STS>St</ca:STS><ca:HNO>10</
ca:HNO><ca:FLR>1</ca:FLR><ca:x>19.247809719012466</ca:x><ca:y>20.3841837932858
32</ca:y></ca:civicAddress></gp:location-info></gp:geopriv><timestamp>2019-07-
04T02:25:56.895Z</timestamp></status></tuple></presence>
--NG911f052KQD9zCUS0bkPZgoQFWGJSbmENL88CAspmv1aAoJWEiz2H2—
```

"--NG911f052KQD9zCUS0bkPZgoQFWGJSbmENL88CAspmv1aAoJWEiz2H2" would be the separator between MIME Parts of the MIME Body of the message. The last MIME Part is the PIDF-LO (the XML representing the physical location of the calling device).

SIP Invite captured by Wireshark for a successful call from Sipdroid

```
▼ Request-Line: INVITE sip:767@64.131.109.30 SIP/2.0
    Method: INVITE
  ▶ Request-URI: sip:767@64.131.109.30
    [Resent Packet: False]
▼ Message Header
  ▶ Via: SIP/2.0/UDP 172.56.13.95:49191;rport;branch=z9hG4bK51558
    Max-Forwards: 70
  ▶ To: <sip:767@64.131.109.30>
  ▶ From: <sip:caller1@64.131.109.30>;tag=z9hG4bK60118925
    Call-ID: 932147062522@172.56.13.95
  ▶ CSeq: 1 INVITE
  ▶ Contact: <sip:caller1@172.56.13.95:49191;transport=udp>
    Expires: 3600
    User-Agent: Sipdroid/3.9 beta/SM-G920T
    Content-Length: 389
    Content-Type: application/sdp
▶ Message Body
```