FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Jussi Sepponen

BLE-Data: A smartphone-based BLE-data collection tool

Master's Thesis
Degree Program in Computer Science and Engineering
<June 2019>

Sepponen J. (2019) BLE-Data: A smartphone-based BLE-data collection tool. University of Oulu, Degree Program in Computer Science and Engineering. Master's Thesis, 54p.

ABSTRACT

Smart phones together with plethora of different sensors create a massive data collection system. This system can be also used for storing, analyzing, and broadcasting data. Data can be anything that can be metered or derived, from chemical compounds to traffic congestion to advertisement data to users' activity or health statistics. Using the sensors already present in smartphones together with Bluetooth capable controller chips to add more sophisticated sensors, user creates an easily extendable monitoring system that does not require an internet connection. After embedded initialization programming, configuring and managing these chips can be done with a smartphone using Bluetooth communication stack. This study presents an Android-library for managing BLE peripherals and an app to collecting and store the recorded data. Bluetooth devices are scanned and connected through Bluetooth Low Energy interface and data is stored to persistent Room database.

Keywords: android, sensor, BLE

Sepponen J. (2019) BLE-Data: A smartphone-based BLE-data collection tool. Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 54s.

TIIVISTELMÄ

Älypuhelimet yhdessä lukuisten eri antureiden kanssa luovat massiivisen järjestelmän tietojen keräämiseksi, tallentamiseksi, lähettämiseksi ja analysoimiseksi. Nämä tiedot voivat olla mitä tahansa kemiallisista yhdisteistä liikenteen ruuhkautumiseen ja mainosdatan esittämisestä käyttäjän aktiivisuuteen ja terveystilastoihin. Käyttämällä älypuhelimen valmiiksi olemassa olevia sensoreita ja yhdistelemällä niitä Bluetooth - ominaisuudella varustettuun ohjaussiruun kehittyneempien sensorien kanssa, käyttäjä saa luotua helposti laajennettavan monitorointi verkoston, joka ei vaadi internet yhteyttä. Käyttöönotettaessa tarvitun sulautetun ohjelmakoodin jälkeen, sensorilaitteita voi ohjata älypuhelimella, käyttäen Bluetooth radiota. Tässä tutkimuksessa esitellään Android-kirjasto BLE sensorien hallintaan, yhdistettynä tietojen keräämiseksi ja tallentamiseksi rakennettuun appiin. Bluetooth-laitteet skannataan ja liitetään Bluetooth Low Energy - rajapinnan kautta ja näistä luettu data tallennetaan paikalliseen Room-tietokantaan.

Avainsanat: android, sensor, BLE

TABLE OF CONTENTS

FOREWORD

This thesis was done under guidance of Dr. Denzil Ferreira and it was partially funded by the Academy of Finland grant 318927-6Genesis Flagship.

I want to thank Matti and Kosti from RD Velho for giving me a change and thanks for RD Velho to also funding this thesis.

I want to thank Sami for encouraging me and mentoring me.

I want to thank Ville for setting an example for me in everything.

Most importantly I want to thank my wife Karoliina for understanding when I was working late again and again. I couldn't have done this without you.

Oulu, 08.07.2019

-Jussi Sepponen

ABBREVIATIONS

BLE	Bluetooth Low Energy
Android OS	Android Operating System
API	Application Programming Interface
DAO	Data Access Object
GAP	Generic Access Profile
UUID	Universally unique identifier
IPC	Inter Process Communication
APK	Android Package Kit
ATT	Attribute Protocol
GATT	Generic Attribute Profile
MIC	Message Integrity Check
L2CAP	Logical link control and adaptation protocol
CRC	Cyclic Redundancy Check
SIG	Special Interest Group
RSI	Relative Strength Index
IoT	Internet of Things
WoT	Web of Things
RFID	Radio-frequency Identification
AFK	Away from Keyboard
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
POC	Proof of Concept
UI	User Interface

1. INTRODUCTION

The world around us grows more and more connected. Accessibility to high-speed internet rises and according to International Telecommunication Union [22], at the end of 2018 51% of the world population is connected to the internet. This availability, together with increasing quality of mobile devices and embedded sensors, is one of the biggest enablers for new industry-defining concepts such as Internet of Things (IoT) [21]. Being a big topic on its' own, the concept of IoT is more closely inspected in chapter *2.4 Internet of Things*.

Sensors in smartphones, such as accelerometer, gyroscope, magnetometer, proximity sensor or ambient light sensor are useful tools for collecting data for analysing user's activities and surroundings. However, as smartphones only have sensors in one singular location at the time, they receive a small section of ambient data such as temperature, noise or amount of light. Often phones are carried inside clothes and that affects quality of data or makes it impossible. Another aspect is getting user's consent and ensuring that collection is GDPR compliant [31].

IoT- sensors allow novel data to be collected at enormous scale. Sensors can collect data from large areas and convey data to users smartphone for automated responses. Resulting smart environments can be used to create more accommodating living spaces, more efficient industrial complexes, less wasteful officebuildins and so on. These spaces are more environmental friendly by matching for example lightning conditions according to user needs, instead of user manually operating them. Another example for building wide sensor network would be humidity sensors for detecting water damage to avoid mold, or monitoring air composition in cellars to detect radon or other harmful airborne particles.

Other than sensor networks, Bluetooth [1] can be used in advertising. In mobile marketing, users who have their Bluetooth adapter enabled may receive broadcasted Bluetooth messages. Since receiving advertisement does not need pairing or an active connection, image files, virtual business cards, barcode coupons or even complete applications can be send to anyone within area of Bluetooth transmitter.

Other than advertisement, this feature can be used distributing localized information, content on demand, public health-related warnings and so on.

1.1. RD Velho Oy

RD Velho offers intelligent devices and services [16]. Products are developed from circuit boards to all the way into cloud systems by inhouse teams. An interdisciplinary team of experts from mechanics, electronics, UX, embedded- and software development ensures bleeding edge solutions to customers across Finnish industries. At the time of writing, company employs approximately 300 personnel in nine locations in Finland. Customers vary from small specialized businesses to large corporations in different industries.

1.2. My current role

I'm working at RD Velho as an Android developer. Our teams' current Android codebase is my own development. My part requires understanding in Android architecture, Java systems design, Bluetooth Low Energy stack, embedded sensors, REST APIs and tying this together with proper User Experience. Our team is small and agile and both we and our customer are happy and excited with project.

1.3. Problem statement

Creating an environment for reading and storing data from sensors in the wild is often manual labor with a lot of boilerplate code. Devices in remote areas could have limited internet capabilities due bad connection circumstances, narrow bandwidth, device has small energy source or performance issues, or device owner is preferring to stay offline. Devices also often require manual configuration and they are installed in difficult places, such as high altitudes or sealed containers. The ability to remotely configurate and debug the devices is needed.

1.4. Thesis objective

This thesis presents a proof of concept (POC) for Android app and library for configuring Bluetooth Low Energy (BLE) devices and collecting and storing the data from them. Library is meant to be a starting point that can developer can expand depending on individual needs.

1.5. Thesis structure

Related work inspects different aspects of Android development, Bluetooth low energy stack and embedded world of Internet of Things. Also, different components used in library and app are introduced. *Design and Implementation* presents the overall architecture and inspects the control flow in app. *Evaluation* explains how the app and library were tested and displays the analyzed results. *Usage* gives user a quick walk-through about how to use the app and library and provides guidance for developing with BLE. *Discussion* provides further development ideas for library and justifies decisions made on implementation. *Conclusion* gives a brief summary of this thesis.

2. RELATED WORK

Android is the dominating operating system on smartphone market [4]. Due the easy accessibility and free-of-use development environment this platform was chosen.

2.1. Android

Android OS is a mobile operating system developed and maintained by Google. User applications reside in Android runtime, which contains Delvik virtual machine and Android core libraries [6]. This runtime executes all Android programs and Android framework provides API for developers to create applications. The layer between Android runtime and device hardware is modified Linux kernel. Modified things include new Inter Process Communication system, improved memory system and removal of Linux super user.

2.1.1. Android activities and processes

Android system displays user interface screens as activities [14, 42]. Activity includes layout, application components and other additional widgets that app runs when it's active. Activities have their own lifecycle, and only one can have focus at a time. When user starts another activity, first one is put on pause. Default feature on an activity is that user can return to the old activity using back – button. Activities need to be listed to AndroidManifest.xml when creating the app.

Android keeps application processes such as activities alive for as long as possible. Eventually system runs out of memory and old processes must be removed [14, 42]. Removal depends on which of the four states (i-iv) processes are, system prioritizes keeping the first ones alive by killing last ones.

i) *Foreground* activity is the one user is currently interacting with. This is the most important process and will be killed only if it hogs more memory than system has available. Killing these processes is necessary to keep system and user interface responsive.

ii) Second important tier is *visible* activity. Visible activities are visible to the user, but not currently on foreground. For example, activity is behind foreground dialog or other multi-window mode widgets. Visible activities are killed only to keep foreground activity running.

iii) Third category is *background* activity. Background activities are not visible to user, for example when activity's `onStop()` - method is triggered, see Figure 1. States for these stopped activities can be saved. If activity is destroyed, it is restored again when user navigates back into it and `onSaveInstanceState(Bundle)` – method is triggered inside `onCreate()`. Bundle is an object which can store values across processes.

iv) On last importance tier is *empty* process. Empty processes are killed quickly, as they don't host any application components, such as activities. For example, Broadcastreceivers and Services (see Chapter *3 Design and Implementation*) are both considered empty processes if they don't belong to any activity.

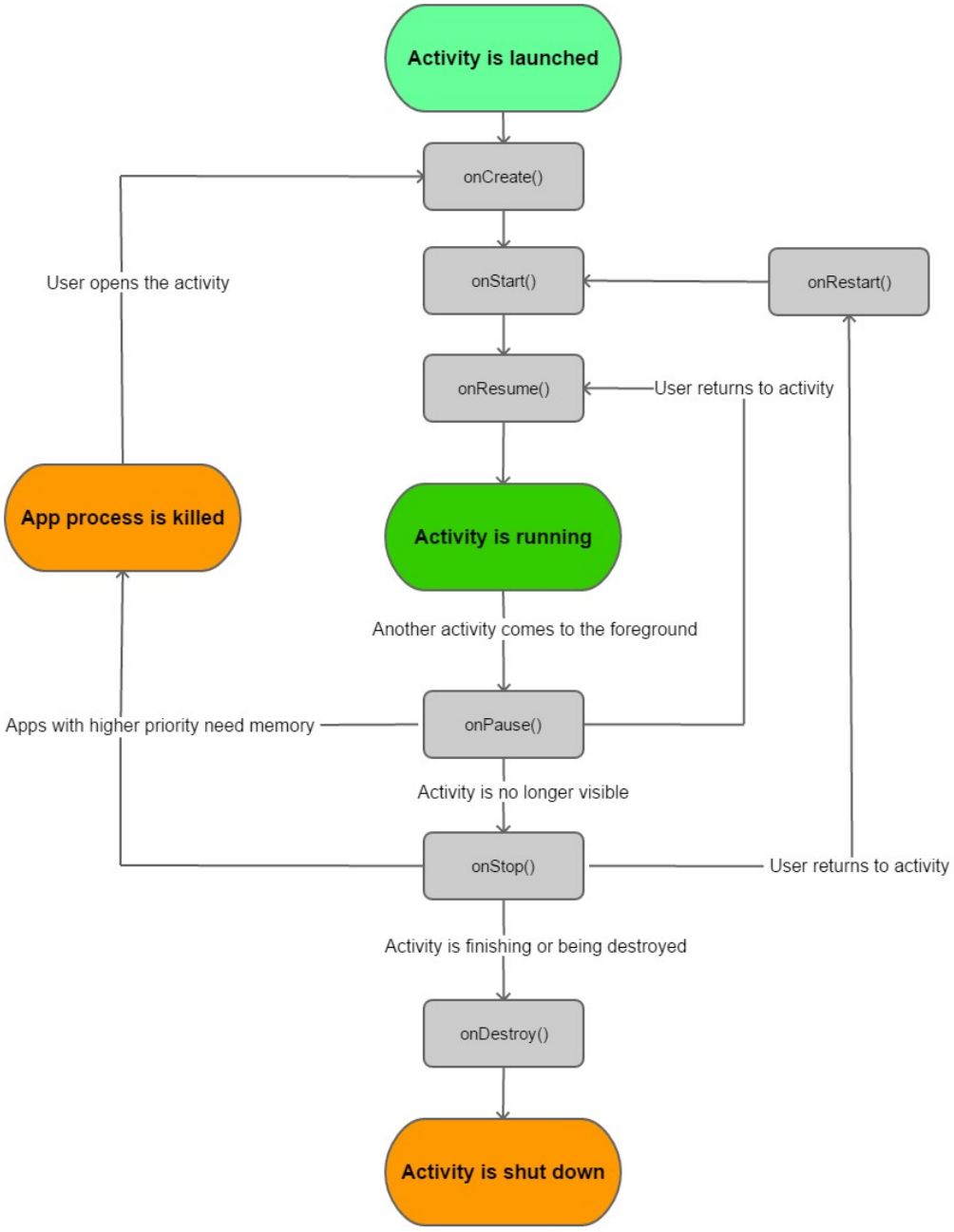

Figure 1: Android Activity Lifecycle

2.1.2. Room Persistence Library

Room is persistent data storage system from Android Architecture Components [3]. It creates a layer of abstraction on SQLite called Data Access Object (DAO). This allows a robust and simple implementation for database solution. Figure 2 presents system architecture for Room. System has three major components:

i) Database: Acts as main access point for app's relational and persistent data. Database object holds list of entities inside database.

ii) DAO: Stored entities are accessed through DAOs, that handles persistent changes to database and retrieves the data. Methods for accessing Room database are stored in DAO.

iii) Entity: Represents a table inside database.

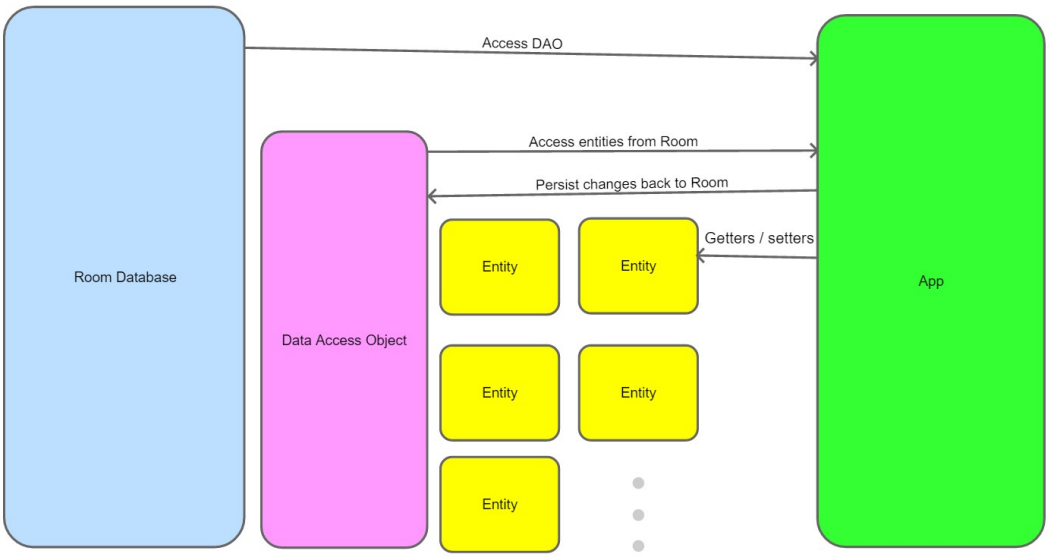

Figure 2: Room architecture [7]

2.1.3. Android Versions

Older Android version get deprecated as they grow obsolete. Current support libraries in this thesis requires Android API level 23 (Marshmallow) or newer. Android also introduced runtime permissions in Marshmallow update so developer using this library must take care of appropriate permissions. Runtime permissions may be revoked by user when application is running. Revoking permissions without proper handling from app results into undesired behavior such as app crashing, app gets stuck waiting for waiting on a resource and so forth. Developer must make sure application handles those situations gracefully.

2.1.4. Permissions

Android uses permissions to protect user from malicious operations and for user privacy protection [41, 47]. Apps that want to access sensitive data or use system features, for example reading the contacts or using the camera, must request permission. Apps by default have no permissions to impact system, user or other apps. Permissions have types, depending on how high permission is at protection level. Requesting permissions is done in app manifest. There are three levels:

i) *normal* permissions: If requested permission is a normal permission, system automatically grants permission to app. Normal permission would be for example setting system time zone. User cannot revoke normal permissions from app.

ii) *dangerous* permissions: see chapter *2.1.4.1 Dangerous permissions.*

iii) *signature* permissions: Signature permissions are granted at install time, but only if app uses same signature certificate as the app defining that permission. These are usually reserved for device firmware and are not accessible to anyone, apart the smartphone manufacturer.

2.1.4.1. Dangerous permissions

Dangerous permissions are permissions that require resources or data that involves user's private information or stored data could affect operation of other apps. If app requires a dangerous permission, such as `READ_CONTACTS`, the user must explicitly grant the permission. This is done at runtime with permission dialogue along with the initial manifest declaration.

Dangerous permissions have permission groups. Grouping permissions helps user to make informed choices, without being overwhelmed. For example, SMS group has both `READ_SMS` and `RECEIVE_SMS` permissions. User only needs to grant app permission to use SMS group. When app is granted with dangerous permission, system automatically grants other permissions in same group upon request without user interaction. However, it should be noted that newer Android SDK versions might move permissions from one group to another. Developer should make sure that also these situations will be handled gracefully.

2.1.4.2. Adapters

Currently using Bluetooth Low Energy scanner requires both Bluetooth and GPS adapters to be turned active and user must allow ether `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` permissions. `ACCESS_FINE_LOCATION`, being more restrictive also grants permission for `ACCESS_COARSE_LOCATION`.

2.2. Bluetooth Standard

Bluetooth 1.0 specification standard was published in July 1999 [18]. This royalty-free technology was developed by Bluetooth Special Interest Group SIG that included many leading manufactures in telecommunications, mobile communications and chip integrations such as IBM, Nokia and Intel [18].

Bluetooth stack is built to be modular so devices with little processing power such as headsets or keyboard/mouse can operate with small fraction of whole Bluetooth protocol stack. Bluetooth stack also allows new extensions to be built on top of existing specification [18].

2.3. Bluetooth Low Energy

Using BLE features requires BLE capable smartphone. According to Bluetooth SIG 100% of phones and tablets shipped in 2018 have Bluetooth 4.0, which includes BLE [8, 39]. This is a technology created for sending small amounts of data between nearby devices using very little energy [1]. This makes it ideal applicant for managing data sent by individual sensors or small number of sensors managed by controller chip.

BLE protocol stack inherited main parts from classic Bluetooth [19]. The controller consists of physical and link layers, and host runs an application processor with upper layers functionalities. This is displayed in Figure 3. Communication between controller and host is standardized with Host-Controller Interface, HCI. While this architecture is inherited from classic Bluetooth, these two Bluetooth protocols are incompatible due controller's differences. Logical link control and Adaptation Protocol, L2CAP in Figure 3, multiplexes the data from Attribute Protocol (ATT) and Security Manager Protocol (SMP). ATT carries the payload, while SMP is used in message exchange with Message Integrity Check (MIC) and Cyclic Redundancy Check (CRC). Generic Access Profile (GAP) is displayed in more detail in Figure 6.

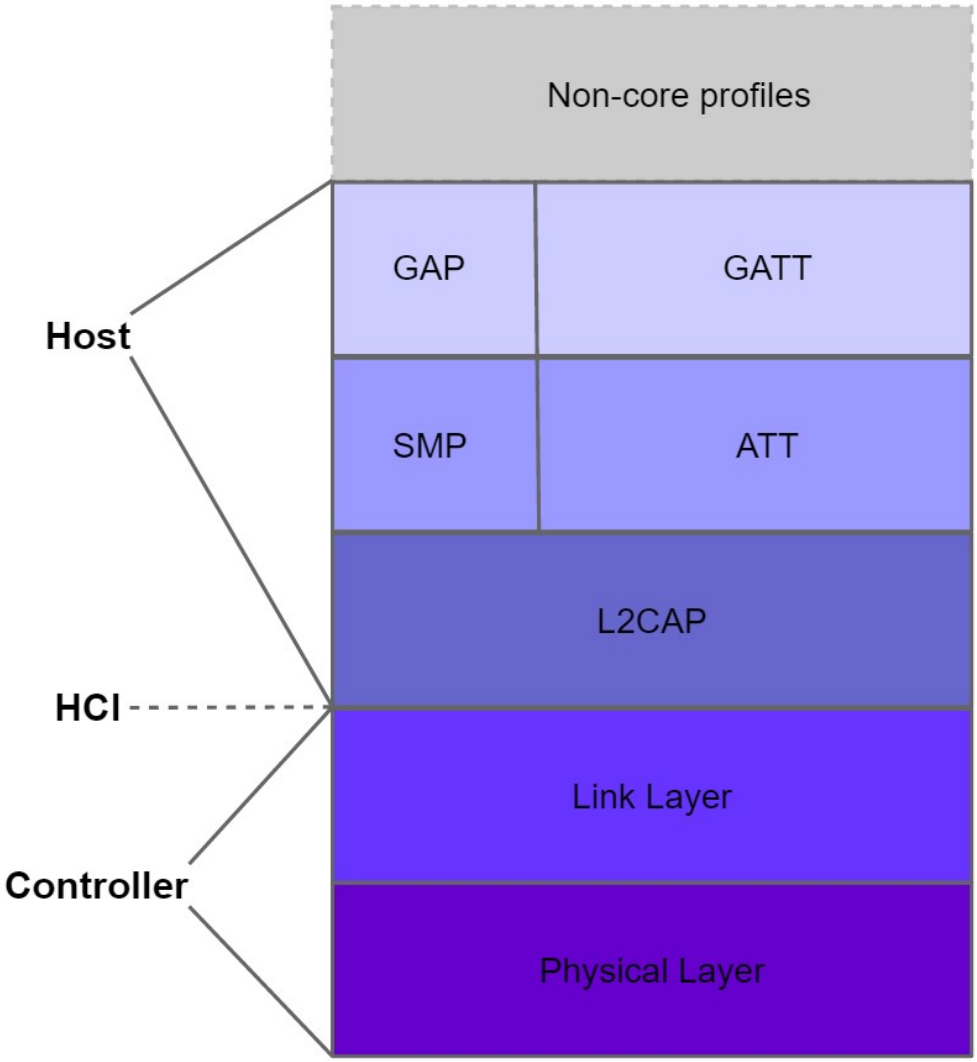

Figure 3: BLE protocol stack [19]

Android provides a support library for controlling BLE-stack [2, 15]. This library handles the low-level interaction between BLE-chip and Android OS such as establishing asynchronous communication channels and reliable write.

BLE topology is centralized around master device [19]. Master can connect to multiple peripherals, but peripheral can only have one master connection. However, peripheral can advertise itself to multiple masters within signal reach. BLE discovery is done by peripheral device advertising itself via advertising channels and master scans through these channels. BLE connection is established by both BLE radio's waking up synchronously.

For BLE packet, maximum Packet Data Unit (PDU) length is 33 bytes [39, 19]. Lower than ATT level protocols, L2CAP, MIC and CRC each reserve some bytes,

displayed in Figure 4. This leaves 23 bytes to payload, which includes the ATT attribute method name for packet (ATT OP) and 2-byte ATT handle for ATT server table lookup (Par). Thus, developer is left with 20 bytes per sent packet for application specific data. Packet configuration is displayed in Figure 4.

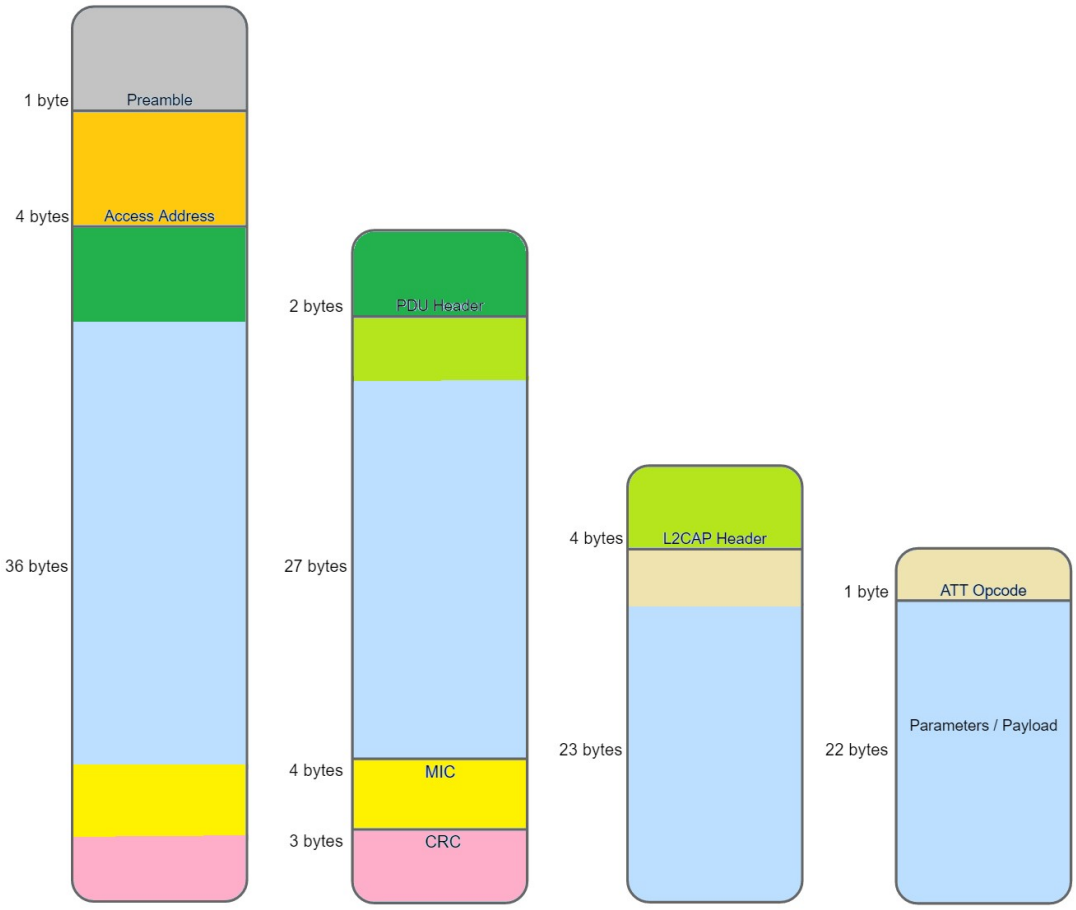

Figure 4: BLE data unit [19]

2.3.1. BLE advertising

BLE peripherals are discovered through peripherals advertising themselves to nearby BLE master devices [1, 34, 39]. Advertising is usually done through channels 37-39 displayed green in Figure 5, although advertising extensions allow advertising on other channels. Advertising packets are sent through these channels, one after another. These channels are spread across bandwidth spectrum to ensure reliable broadcasting if one channel is blocked or interfered. Advertisement channels are shown in Figure 5. Narrow bandwidth allows BLE to better manage interference from another RF devices, such as Wi-Fi, microwaves, Bluetooth classic and so forth.

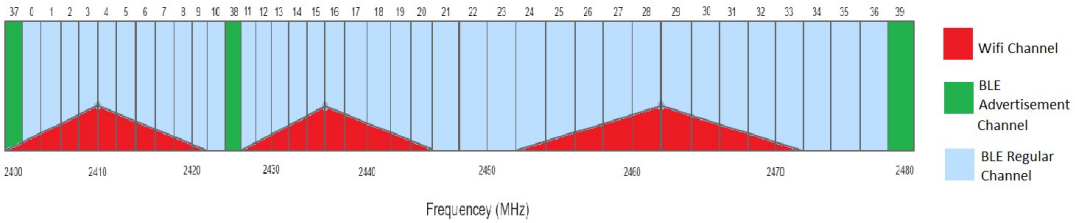

Figure 5: BLE Channels

Developer can set the BLE advertisement interval from 20ms to 10.24 seconds [34] with 0.625ms increments. Also, pseudorandom value is added automatically to every advertisement interval to reduce possibility of collisions when advertising in environment with multiple BLE devices. Using low interval advertisement helps devices to be found and connect faster but also drains device battery faster. For example, advertisement does not have to be frequent if advertisement packet data does not change much between sending.

While there are several types of advertisement packet data units, most relevant for this topic are `ADV_IND` and `ADV_NONCONN_IND` [35]. `ADV_IND` is generic advertisement used by devices that are connectable but not targeted towards any specific device. `ADV_NONCONN_IND` is typically used when peripheral does not want any connections, such as advertisement beacons or weather stations.

Typical advertisement data types are list of services peripheral offers, service class UUIDs, shortened local name and complete local name for peripheral [36]. However, developer can choose which data to advertise, staying within 31 bytes limit. Advertising services is useful when special service is needed. Developer can set a unique 128-bit UUID and master device ignores all other devices when scanning for that specific one. This makes connection faster and saves battery from peripheral and master.

2.3.1.1. UUIDs

BLE protocol uses UUIDs to uniquely identify peripheral services, characteristics and other items [36]. UUIDs are 128-bit numbers that have standardized base adopted by Bluetooth SIG. Base is standardized because constantly transmitting whole 128-bit number is wasteful as Bluetooth transmission has very limited amount of data (20 bytes or 160 bits). SIG defined base as

XXXXXXXX-0000-1000-8000-00805F9B34FB.

The rest 32-bits marked here with X are up to developer to decide. SIG has also defined a short version for some services, such as for example Heart Rate Service. 16-bit version UUID for Heart Rate Service it is just

0x180D

which in fact represents whole 128-bit number

0000180D-0000-1000-8000-00805F9B34FB

just where the bottom 16-bits are remaining zero.

When using pre-existing and numbered services or gatt profiles, developer can use shortened names. Custom services need full unique 128-bits UUID. These UUIDs can be generated for example with online UUID generators, or `java.util.UUID` class.

2.3.1.2. Scan response

When using custom UUIDs, advertisement packet can quickly run out of available data since it is only 31 bytes. Developer can include more information with special scan request – method. While scanning for peripherals, smartphone can send a scan request and peripheral responses with scan response. This has same packet format as advertisement but it's in different layer. Response can provide additional services that for example couldn't fit into advertisement package or a relative signal strength that can be used to give rough estimate to distance to peripheral.

2.3.2. Generic Attribute Profile

Generic Attribute Profile (GATT) is built atop of the Attribute Protocol (ATT) [38]. GATT is a structure for framework and operations for data that ATT transports and stores. GATT server and GATT client are two roles defined by GATT protocol. Both are essential to implement in BLE systems, as GATT is used in service discovery.

GATT server stores the data sent by ATT and accepts Attribute protocol requests such as commands or confirmations from GATT client. GATT server responds to GATT client's requests and can be configured to send asynchronous notifications to GATT client when specified events happen on GATT server. Server can also send two types of unsolicited messages to client:

i) Notifications, which don't need to be confirmed and

ii) Indications, that require confirmation form client.

In this thesis, embedded chip works as GATT server and Android smartphone is used as GATT client.

Data model contained on GATT server is specified by GATT. Attributes transported by ATT are formatted as services and characteristics. Service may contain one or more characteristics. Characteristics contain a single value and any number of descriptors for that value.

Figure 6 displays GATT profile for data exchange. Top of the hierarchy is profile. Profile contains services that fulfill use cases. Services are composed of characteristics and may have references to other services. Characteristic has a value

and may contain other information about the value. Services, characteristics and their components contain profile data and are stored as GATT server attributes.

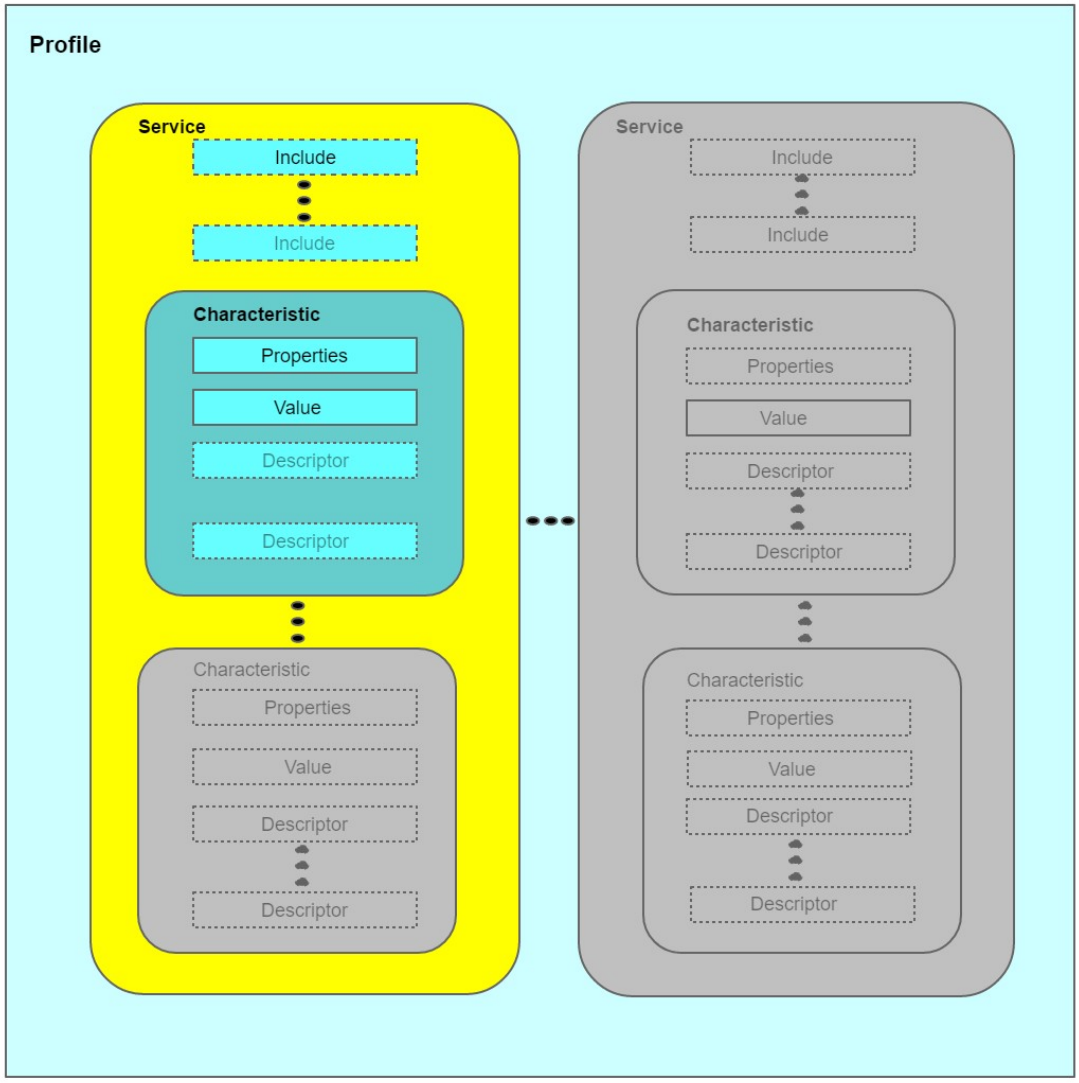

Figure 6: GATT-Based Profile Hierarchy (GAP)

2.3.2.1. GATT service

Service is a complete collection of data and its associated behaviors to act as a feature for a device [38, 39]. Services can include other services and set of characteristics that compose the service. GATT defines two types of services, primary and secondary. Primary services provide primary functionality of a device. Secondary services are intended to be included to primary services as their modifiers.

To ensure backward compatibility, new revision of service can only add new included services or characteristics. New revisions are forbidden to change earlier service behavior from definition. Included services are method for incorporating another service to GATT server definition as a part of service. Entire included

services become part of service including it, with any included services and characteristics. There are no limits for nesting services. Any included services exist also as an independent service.

2.3.2.2. GATT-characteristic

Characteristic is a value in service. Characteristic contains properties and configuration information about how value is displayed. By definition, characteristic contains declaration, properties and the value [38]. Characteristic may also contain descriptors or permit different configurations for GATT server to act when trigger-event happens in characteristic value.

2.4. Internet of Things

Internet of Things (IoT) as a concept begins of small, embedded devices capable of internet communication. Communicating radios can be integrated to everything that has attributes for observer to measure, control, or monitor [21]. Things in IoT need to have a physical appearance, the device must have functional communication to external word and be uniquely identified for both human and machines. Device should have enough processing power, depending on intended use from complex computations, managing networks, sensing physical phenomena, and so forth, depending on intended usage.

However, IoT as a term is wider than the embedded, internet capable sensors [21]. Term also applies to other networked *things,* required middleware and gateways. Communication system for IoT is called Web of Things (WoT) [23, 24]. Third integral part of IoT is applications and services derived from sensors and WoT. Concept of IoT relies on three main aspects [1]. Everything is *identifiable*, everything *communicates,* and everything *interacts.* While the paradigm remains the same, vision and orientation for IoT varies depending on perspective of interests [25].

Between sensors and gateways in IoT infrastructure is middleware layer. This layer abstracts the physical objects such as embedded sensors or RFID tags and composes their services ready for application layer. Instead of using one coherent and complex system, its' components can be decomposed into well-defined interfaces and protocols for each different service [25]. This is called Service oriented Architecture (SoA) and it is one of the most important architectural approaches building middleware to IoT.

This object abstraction is an attempt to fix one of the biggest limiting factors in IoT, the heterogeneity of devices [21, 25]. IoT networks are highly granular as devices are heterogeneous and they are operating in distributed network with plethora of smart devices. Everything can and should form spontaneous connections with every other smart thing nearby, depending on use case and user needs. This amount of connectivity is highly unpredictable and needs well-defined context to work properly.

Context is where the end-devices and sensors are operating. Being aware of context helps to resolve one of the big challenges in IoT: getting everything to communicate on a massive scale.

2.4.1. Context-awareness

Context-awareness using sensors is an ongoing effort at Oulu University [26]. Unifying the context-awareness would help IoT devices to get smarter and setting up ad-hoc networks would be more straightforward if devices are aware of their surroundings already.

Using intelligent API layer between smart-devices proxy such as MQTT [28] or AMQP [29] and user device such as smart phone or desktop, the granularity of service can be adjusted without user devices having to adapt or change service contracts. Allowing similar functionality as atomic services would provide in run-time, using earlier mentioned service combination (service assembler) or run-time contract decoupling, following components will compensate the lack of heterogeneous protocol support that is present in SoA approach. This service coordination layer offers well-scaling, lightweight microservices [27].

2.4.2. Microservices

Using microservices can be described as an intelligent WoT framework [27]. While thematically it's similar with SoA as they both offer architecture for services and reusing them, difference lies in how framework defines the services and micro-services being a lightweight solution compared to SoA. Micro-services take abstraction one step further than SoA, as it atomizes services based on services' size and unifies them based on services' interactions.

Micro-services in IoT will help with smart things' power consumption by atomizing the services. Services can be classified into functional and non-functional services. Functional services are used in smart things' core functionalities and are usually just literals that are exposed to external parties such as readers or web-applications, if they have access to proper services. Non-functional services are non-operational tasks that ensure reliable operation for smart thing such as authentication, logging and so on. Using these specific services via API will save smart thing's resources for intended use instead of having to go through whole stack of operations each time when for example making queries.

However, atomizing services will create redundant processing lag to and through the system, if querying device or transaction must move or process resources for different services. To solve this, it's proposed that service assembler is used for batching required services together and dispatching this new composite service when it's required [27]. This way service consumer doesn't have to spend extra time on making remote access calls. Coordinating these services in SoA is usually done by service orchestrator that processes multiple coarse-grained services to complete transactions. This can be done in IoT systems with less complex solution called

service choreographer. This executes a single service without a mediator. Queried service then calls second service and so on, creating a service chain. This increases the remote access call time as multiple services are queried, but only requires a single call as earlier mentioned service assembler ensures that request can be done with single remote call.

2.4.3. Bluetooth Mesh

Bluetooth mesh is a protocol built on top of BLE-core system specification [32]. This protocol transforms the star shaped BLE networking topology into distributed many-to-many networking mesh. Difference is displayed in figure 2. In mesh topology, smart phone requires only single Bluetooth antenna for communicating with entire network. Star topology needs one antenna per peripheral connection, and this is a big limiting in traditional BLE with sensor networks. Number of simultaneous connections depends on the smartphone model.

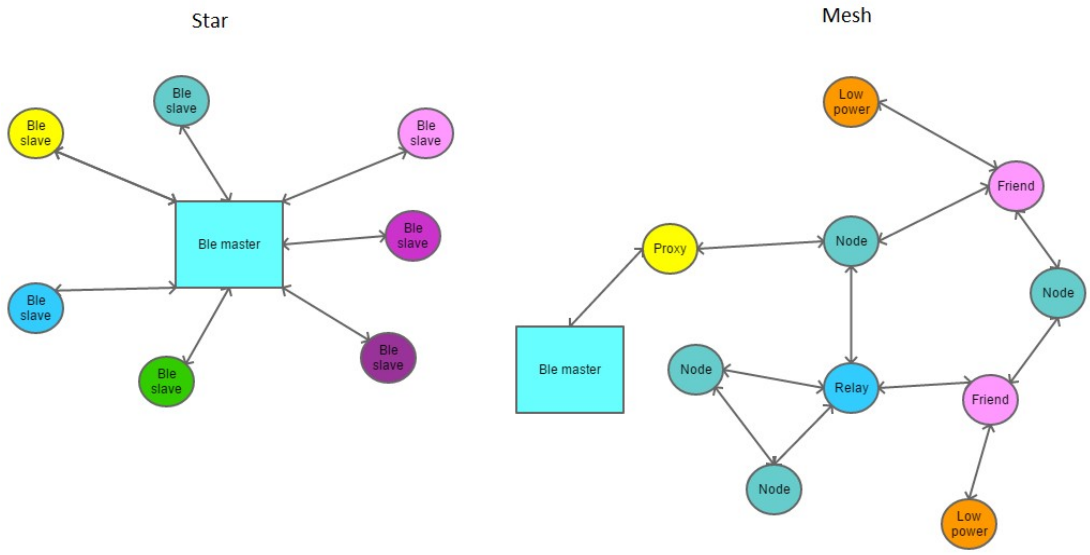

Figure 7: Star and Mesh topologies

In star topology, all the BLE peripherals are their own separate entities. They communicate to BLE master device such as smartphone through exposed services. In mesh, there are different node types, dedicated for their own features [32]. Mesh communicates to smartphone with a *proxy node*. Instead of having to communicate with every single node, mesh can be abstracted into composed microservices through proxy node. BLE communication is usually done over short distances (<10m) so to extend the range and scale, Bluetooth mesh uses *relay nodes* to scale and extend the size of network. *Friend node* is used as an additional message cache to support *low power nodes*. Low power nodes allow sleeping and they poll messages with friend nodes periodically.

Mesh communication is done with managed-flood transmission. Sending node transmit messages through broadcast channels to other nodes that can receive and relay messages. This makes mesh network modular and more robust compared to single-point-of-failure star topology.

Payload message exchange inside mesh network is done with publish-subscribe pattern [32]. Nodes that generate messages send the message into unicast address, virtual address or group address. Any node interested of these messages subscribe to these addresses.

Using Bluetooth mesh is similar to using microservices, as gatt-services can be created to specifically answer to individual user's needs. Services can be composed with necessary information so that user only has to create gatt profile once to communicate with whole sensor network.

3. DESIGN AND IMPLEMENTATION

Design goal is to create an Android library for collecting and storing data from BLE devices. As concept in Figure 8 presents, data is collected with sensors that are connected to Bluetooth capable controller chips. After initialization with embedded code, these chips can be controlled through GATT-profile.

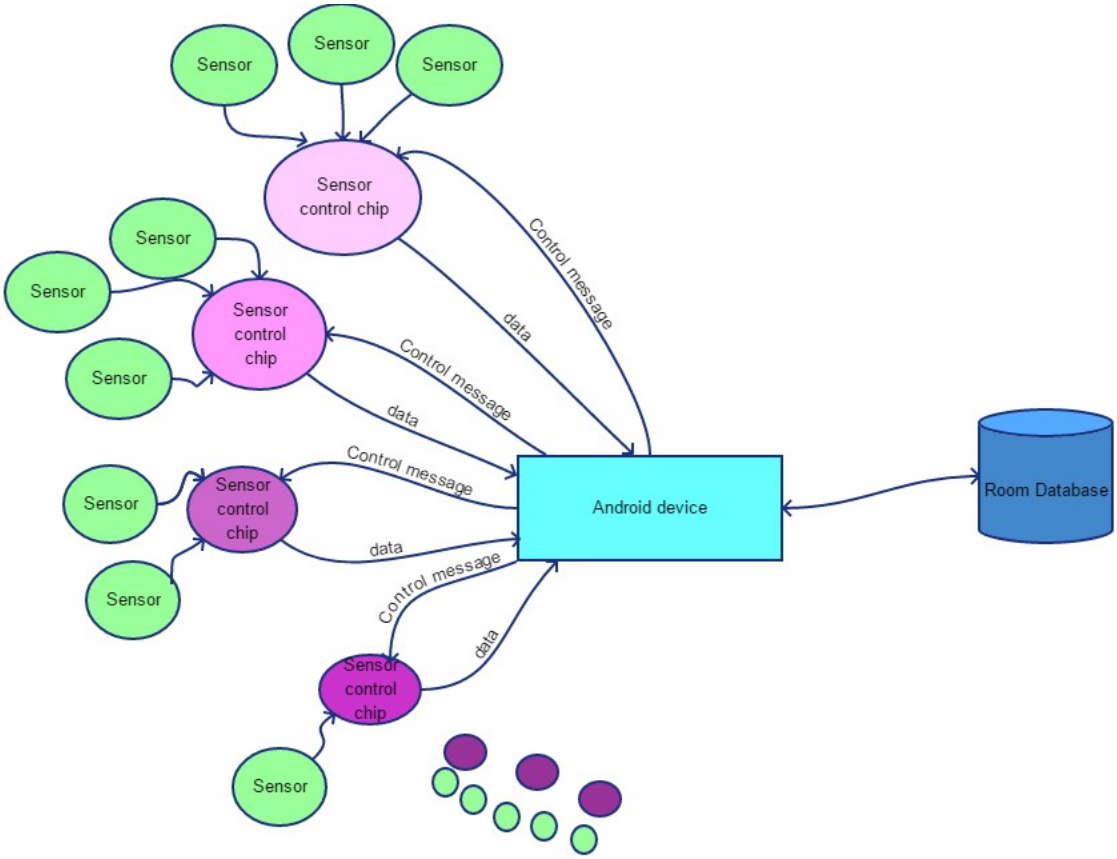

Figure 8: Concept

3.1. Architecture

Figure 9 presents system architecture.

i) `GattService` creates GATT profile for connected device. Methods for listing available services and characteristics for those services are from `android.bluetooth` – API. OnCharacteristicChanged(…) – method is bound to listen for changes in accelerometer's angle data characteristic. Same characteristic also displayed battery values from embedded chip. This data is sent to MainActivity with `LocalBroadcaster`.

ii) MainActivity displays angle data, GATT services and –characteristics. Feed can be saved into Room database with save –button, see chapter 5.2 *Classes and User Interface.*

iii) ScanActivity displays peripherals that responded to `android.bluetooth.le` – API's `ScanCallback()` – method. For convenience , only peripherals with local name are displayed, along with MAC address.

iv) Room database stores data entries as Java objects with entry ID, username, timestamp, data category and value for data. Entries are accessed with `DataDao` and Dao - methods are stored at `DataRepository`. Neither is visible on Figure 9, for more details on these see chapter 2.1.2 *Room Persistence Library*.

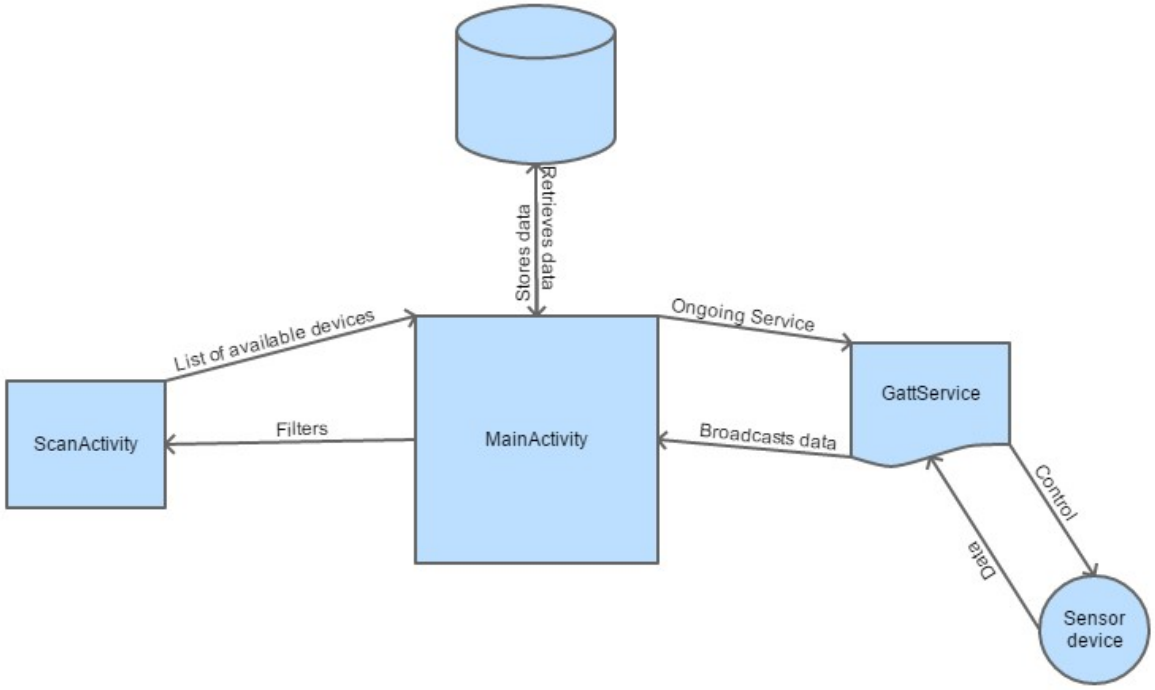

Figure 9: System architecture

System architecture is designed following Model-View-Controller (MVC) design pattern [48]. This compound pattern allows three components to be designed and configured separately. MVC-pattern is displayed in Figure 10.

i) *Model* contains and manages the data. Models in this app are dataEntry, characteristicEntry, serviceEntry and scanEntry.

ii) *Controller* facilitates data exchange between Model and View. Each model (data-, characteristic-, service-, and scanEntry) has its own dedicated controller.

iii) View acts as UI (User Interface) between user and the system. View handles all the user inputs and conveys them into Controller and receives data from Model.

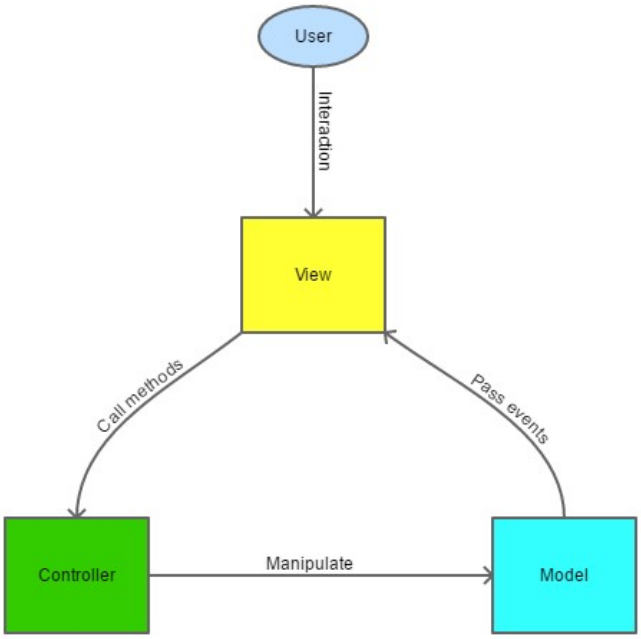

Figure 10: MCV pattern diagram

3.1.1. Permissions and adapters

Together with GPS and location services mentioned earlier, Android device's Bluetooth adapter must be turned on when application is running.

These methods are implemented at dedicated Activity and are not part of this thesis. `MainActivity.java` keeps track of active adapters with LocalBroadcasters that trigger when an adapter is not available [14]. For more information, see Chapter *5.1 Runtime permissions and active adapters*.

3.1.2. Concurrent threading

To keep the app responsive, Android does not allow UI-thread (i.e. main-thread) to do tasks that might take a long time, such as accessing directly into a database or an online resource. This can be done with worker threads that deliver the result then to UI-thread. Another example is Gattservice in Figure 9, it's an extension of Androids' Service component. These Services can perform long-running operations on background without affecting the app performance or requiring input or actions

from user. Services can send data back to Activity with for example using LocalBroadcasters. Broadcasters broadcasts Intents through Android internal messaging system [14].

This library uses these methods for passing data back and forth to worker threads and UI-thread:

i) `Activity.runOnUiThread(Runnable)`

ii) `View.post(Runnable)`

iii) `View.postDelayed(Runnable, long)`

Runnables are new tasks that are sent to main-looper via MessageQueue. This process is displayed in Figure 11. These methods are used for storing and retrieving data from database and updating UI as new data arrives from database or BLE sensor.

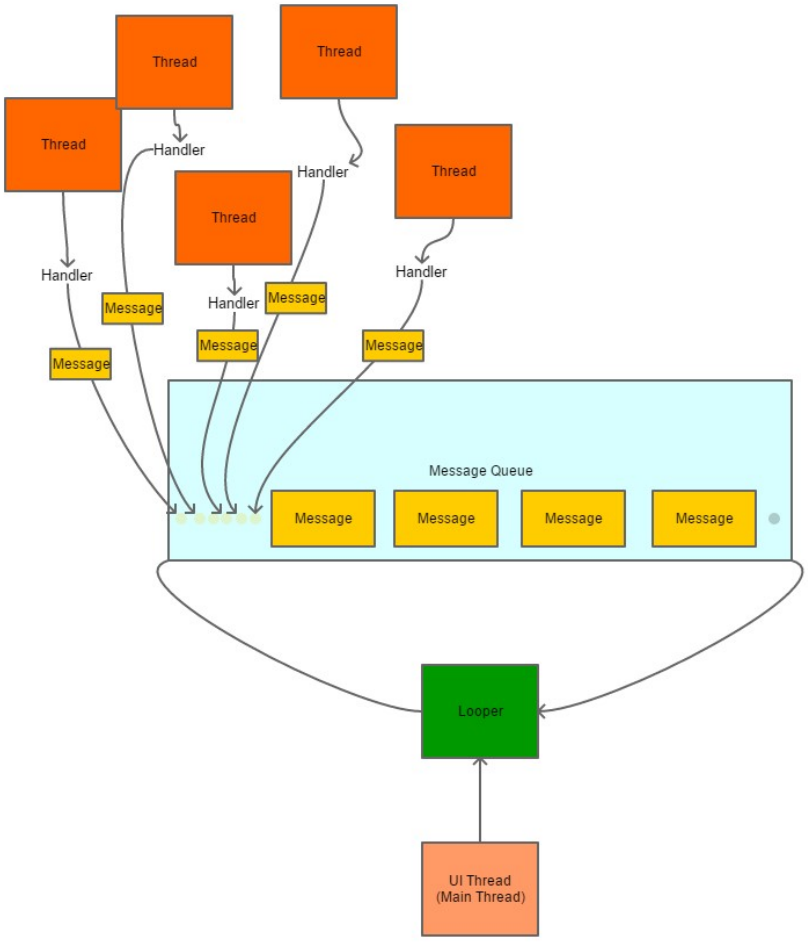

Figure 11: Android Handler and UI Communication.

3.1.3. GATT callback

BLE communication is done through master-device (smart phone) creating a GATT profile with peripheral-device (embedded sensor) [14]. Peripheral answers asynchronously with callback with connection state. `onConnectionStateChange` - method triggers and device's currently available services are listed [12]. Services that device offers can be queried for their respective characteristics and characteristic's specific values and descriptors. This architecture is displayed in Figure 6.

3.1.4. connectGatt – flags

When creating a GATT profile, developer can specify auto-connect flag. This flag helps device to reconnect GATT if connection is lost. However, during the testing this flag seemed to complicate establishing the connection in the first place. Without proper tools the actual cause for failing connection is impossible to detect [10]. With auto-connect flag set to true, sometimes master starts the connection and peripheral indicates established connection with Bluetooth led.

However, when observing Android Studio's Logcat feed, it was noted that onConnectionStateChange failed to trigger. Only fix was to re-activate the master's Bluetooth adapter and power off peripheral completely and try with fresh connection. Problem disappeared when implementing a custom reconnect - `Handler` and setting auto-connect flag to false. Reconnect - `handler` attempts to connect and on detected fail calls `gatt.close()` - method for controlled shut down for connection and re-initializes gatt-object back to null before trying to establish a new connection with gatt.

3.1.5. GATT profile naming conventions

Services and characteristics are given specific name through Bluetooth specification norm [13, 17]. Service and characteristic names are mapped with shortened UUIDs corresponding to Bluetooth Specification and UUIDs are parsed through name mapping to give user descriptive names. These are stored in static values at `uuids.xml` at Android Studio resources folder. Due the large number of UUIDs, the conversion to paste-able format was done with Python script.

As BLE – data packet size has limited to 20 bytes, developer should aim to optimize the use of UUIDs. For more information see chapter *6 Discussion*.

3.2. Development Environment

Development is done with Android Studio 3.3.1 with JRE 1.8.0_152_release-1248-b01 amd64-version and Gradle version 4.10.1.

Development BLE-chip was Bluno Beetle v1.1 with MOD-MPU6050 - Open Source Hardware Board MEMS 3-axis gyroscope and 3- accelerometer. Gyroscope data can be sent between 100ms and 1000ms bursts for testing purposes.

3.3. Deployment

App and library are shared at authors GitHub – page under apache 2.0 license [11].

4. EVALUATION

Thanks to the new prototype I was working with at RD Velho, evaluation for app and library was made with different BLE chip than initial implementation. Most recent architectural changes required slight modification to gatt-service connection and data-reading and sending triggers, but overall software architecture remained the same.

Data was got from the sensor using through BLE gatt - `BluetoothGattCallback.onCharacteristicChanged()` trigger method and sent to MainActivity via `LocalBroadcaster`. MainActivity then displayed the value and saved it to Room database. Same `onCharacteristicChanged()` - method was also triggered by battery – values changing. Results were monitored daily to ensure sensor uptime.

App performance was monitored, and crash results were logged into Fabric.io [30]. Fabric results are displayed in Figure 14.

Crashes were mostly due hasty implementation or experimenting with new features. Fixes were delivered as soon as fabric.io reported crash to ensure testing phase continuing with least amount of downtime.

Master device was an old Android smartphone from LG, running Android Marshmallow v. 6.0.1. Phone was kept in vicinity of test-chair and it was plugged to power supply, constantly charging it due GPS draining battery rapidly.

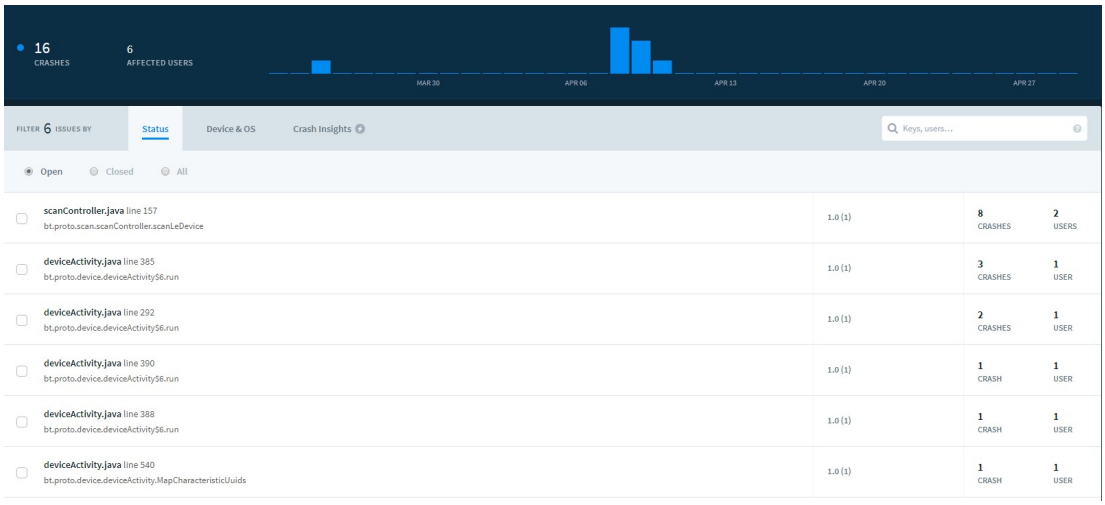

Figure 12: Fabric.io report

Table 1: Crash report explained

Class	Exception	crash count	reason
Scancontroller	NullPointerException	8	Bluetooth adapter was disabled during runtime
DeviceActivity	NullPointerException	3	Custom error for testing fabric.io
DeviceActivity	IndexOutOfBoundsException	2	Database search with too large row-number
DeviceActivity	IndexOutOfBoundsException	2	Attempting to display stored data when there was none
DeviceActivity	IndexOutOfBoundsException	1	Deprecated feature was left active to UI.

4.1. Evaluation setting

Battery-powered embedded chip with accelerometer sensor was hanged to office chair headrest. This sensor sent vertical angle data once per second for ten seconds. After not sensing movement for five seconds, sensor went to sleep. This setting allowed autonomous measuring for chair movement. Eight persons were used as test subjects. Each person was recorded for at least three days, together accumulating to total of 31 days. Sensor was stabile between 82° to 90°, depending on how the sensor-device moved inside cover when preparing the test-bench. Conceptual drawing is presented on Figure 15.

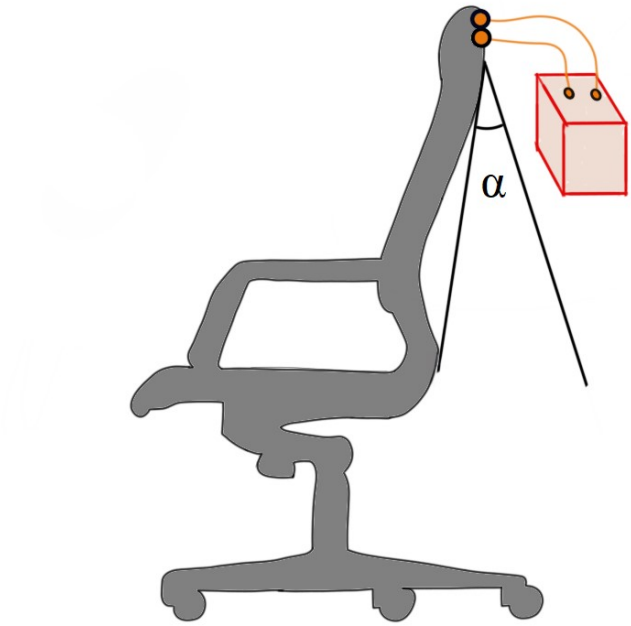

Figure 13: Evaluation setting

4.2. Results

Data was collected for 31 days total. Resulted raw data was stored into local Room database as Java objects with user, timestamp and value. Database was accessed with SQLite browser [46] and imported into an Excel csv-file.

Initial files varied from 400 to 18 000 lines. This data included a lot of redundant data such as database entry numbers, test-subject IDs' and sensor values, all in separate lines.

Data was cleaned up using Python. This resulted into 31 csv-files, one for each day. Files varied from 200 to 5300 lines, with only timestamp and sensor value per each line. Complete results are displayed in Table 2 and Figures 16 to 24.

In first set of graphs, X-axis displays the accumulated datapoints recorded for each day. Y-axis displays the angle values for each day.

Second set plots same days with hourly graphs. Daily hours are displayed on Y-axis from midnight to next midnight and X-axis is amount of datapoints for each day.

Table 2: Collected data

Day	Test-subject ID	Angle values	Stabile at	No. of datapoints	Day of the week	Result
May 19th	2	14°-94°	88°	2180	Tue	present
May 20th	3	15°-97°	87°	4028	Wed	present
May 21st	3	15°-89°	87°	712	Thu	semi-AFK
May 22nd	6	8°-99°	83°	4345	Fri	present
May 23rd	6	82°-85°	84°	501	Sat	AFK
May 24th	6	82°-85°	84°	203	Sun	AFK
May 25th	4	2°-97°	85°	3073	Mon	present
May 26th	2	5°-94°	83°	1486	Tue	present
May 27th	8	80°-83°	82°	159	Wed	AFK
May 28th	8	83°-89°	87°	1878	Thu	present
May 29th	8	70°-92°	87°	2304	Fri	present
May 30th	1	86°-88°	87°	482	Sat	AFK
April 1st	1	0°-98°	87°	2270	Mon	present
April 2nd	1	0°-99°	88°	2786	Tue	present
April 3rd	1	2°-98°	88°	2530	Wed	present
April 4th	2	5°-99°	88°	2025	Thu	present
April 5th	2	81°-91°	88°	421	Fri	AFK
April 8th	3	57°-99°	84°	4139	Mon	present
April 9th	4	82°-84°	83°	154	Tue	AFK
April 10th	4	70°-98°	84°	2508	Wed	present
April 11th	5	56°-98°	84°	5283	Thu	present
April 12th	5	90°-90°	90°	491	Fri	AFK
April 13th	5	90°-90°	90°	460	Sat	AFK
April 15th	6	36°-94°	85°	1300	Mon	present
April 16th	6	84°-86°	85°	594	Tue	AFK
April 17th	7	64°-92°	85°	2721	Wed	present
April 18th	7	83°-90°	85°	1570	Thu	present
April 19th	7	84°-86°	85°	403	Fri	AFK
April 20th	7	84°-86°	85°	372	Sat	AFK
April 21st	7	84°-86°	85°	398	Sun	AFK
April 22nd	7	82°-87°	85°	499	Mon	AFK

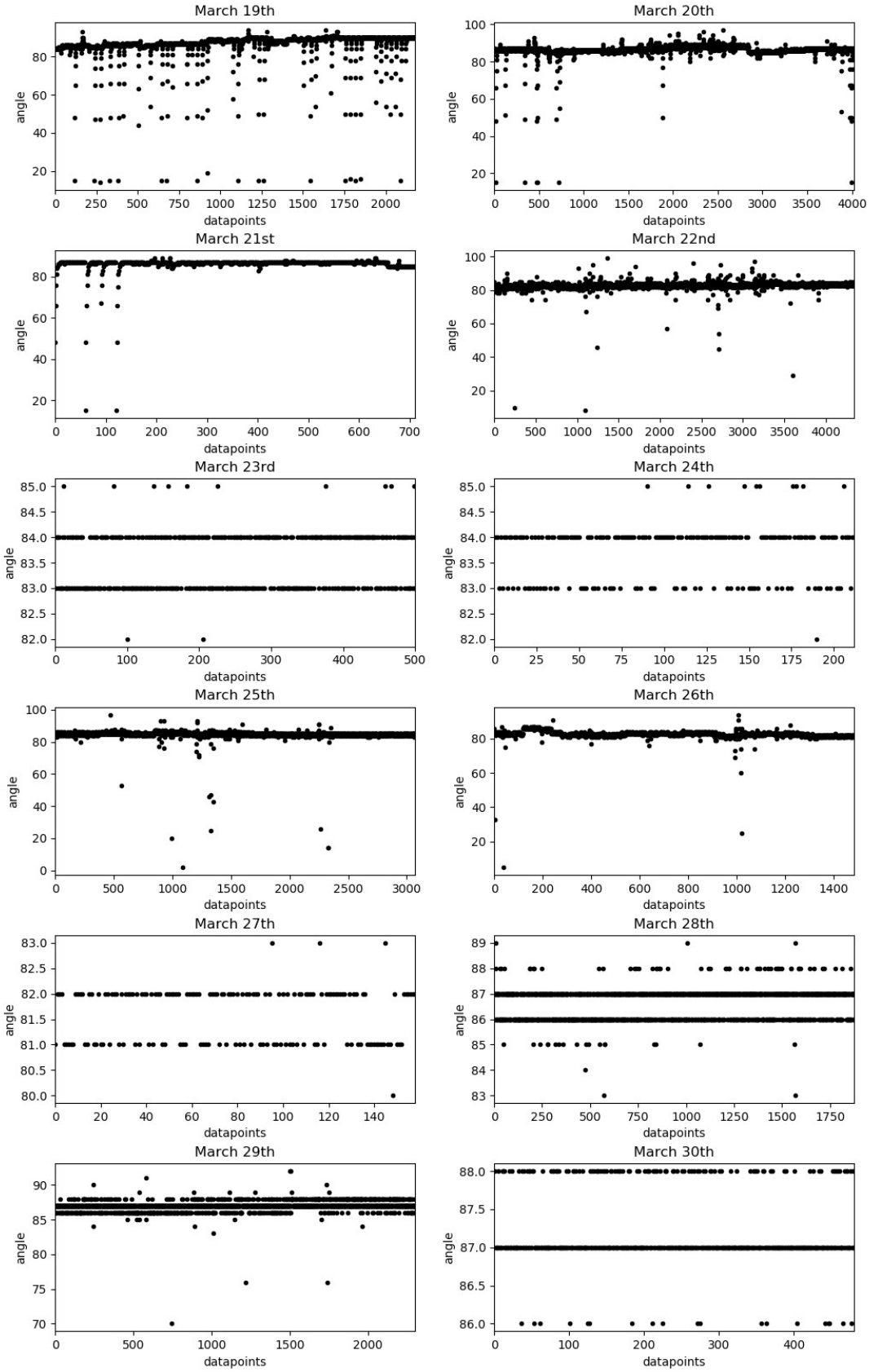

Figure 14: Results May 19th to May 30th

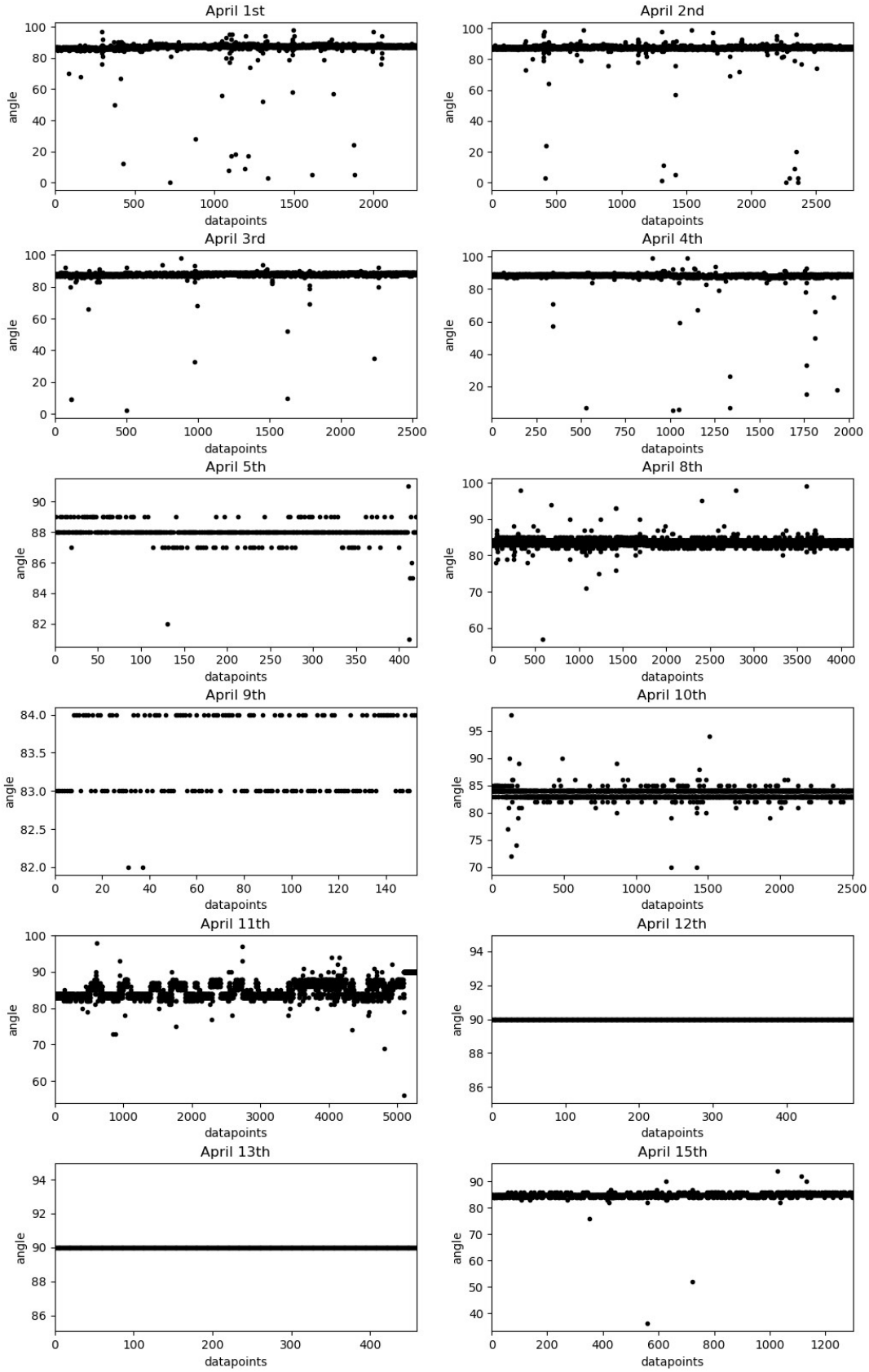

Figure 15: Results April 1st to April 15th

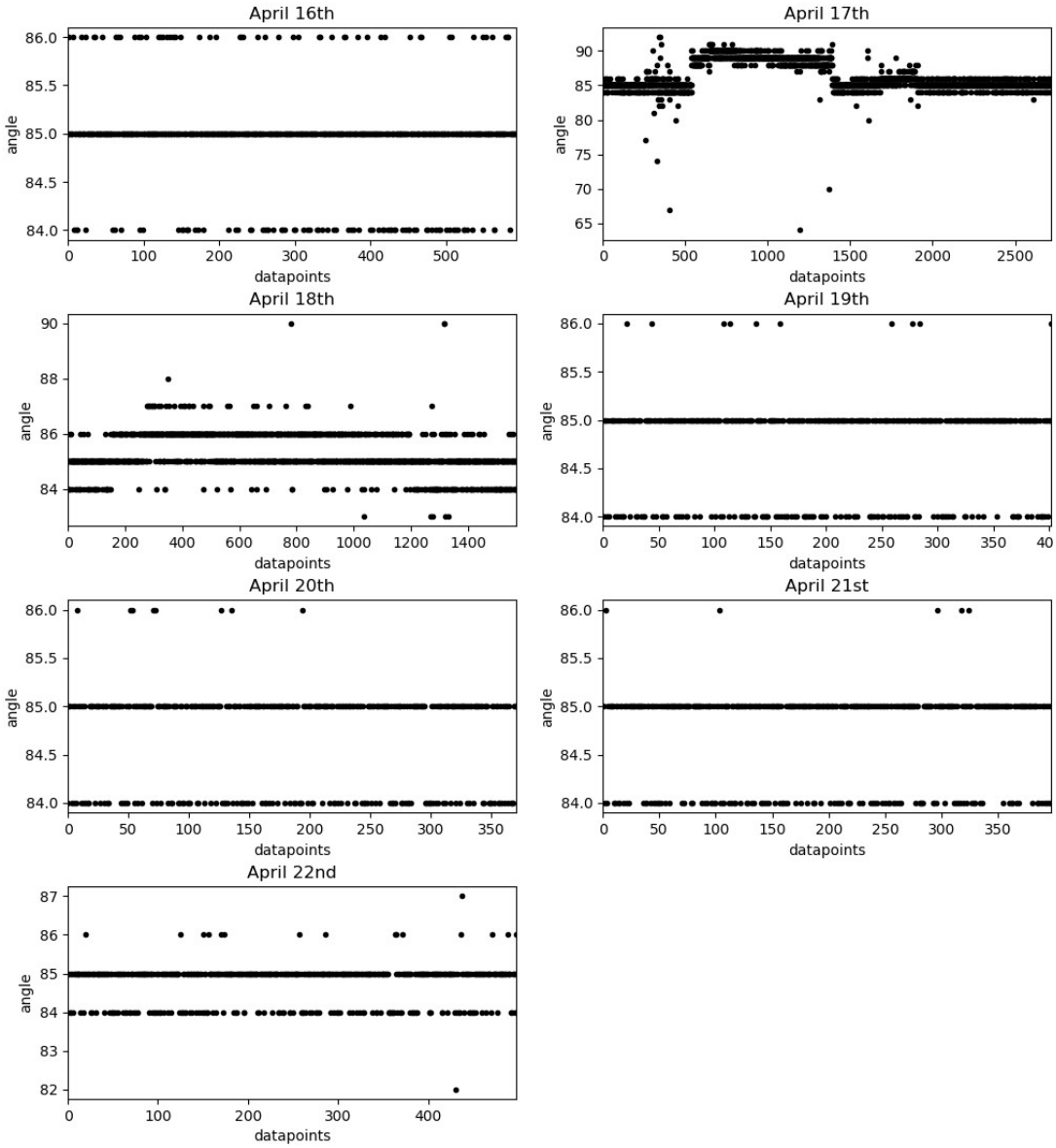

Figure 16: Results April 16th to April 22nd

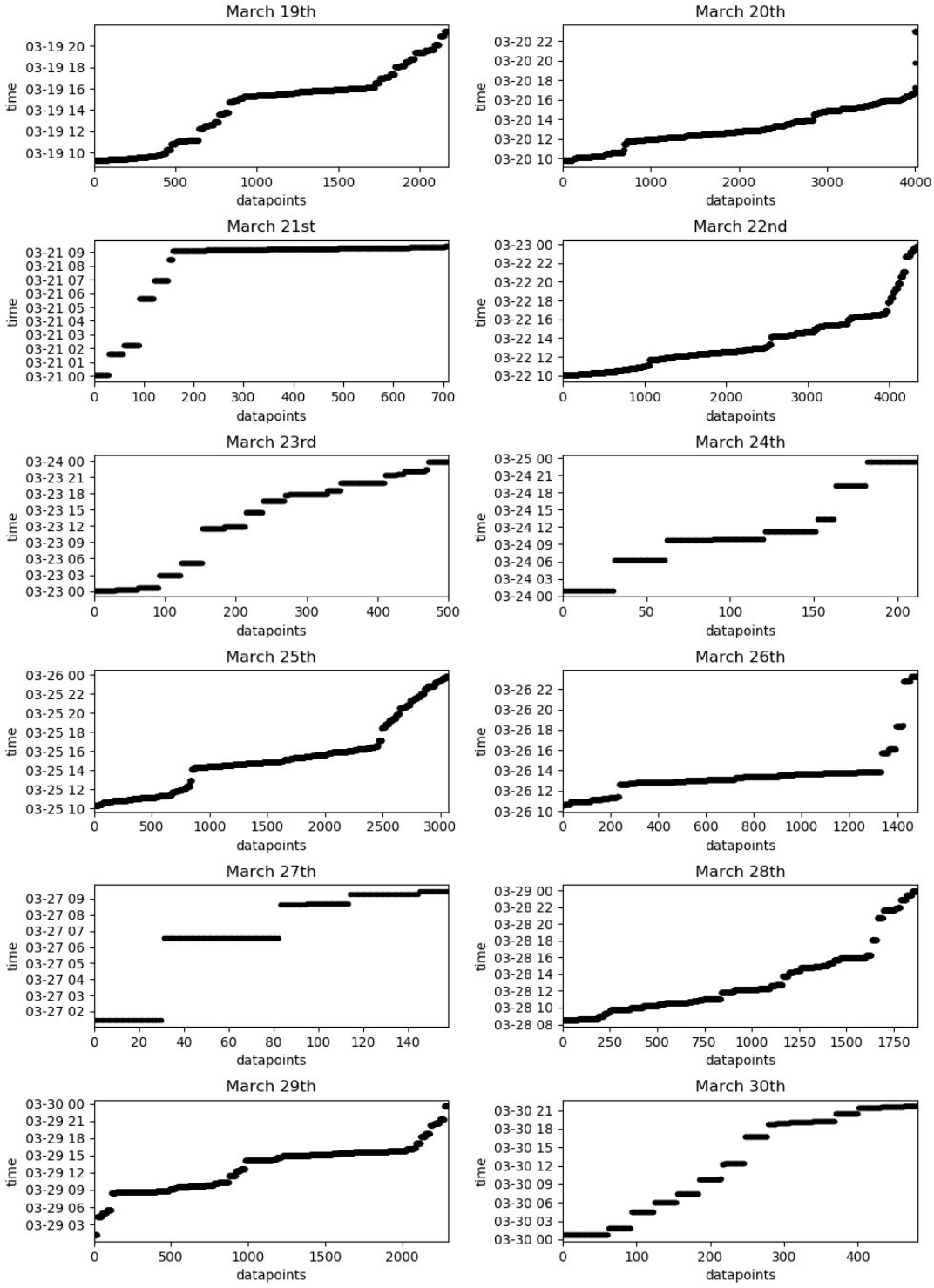

Figure 17: Time results March 19th to March 30th

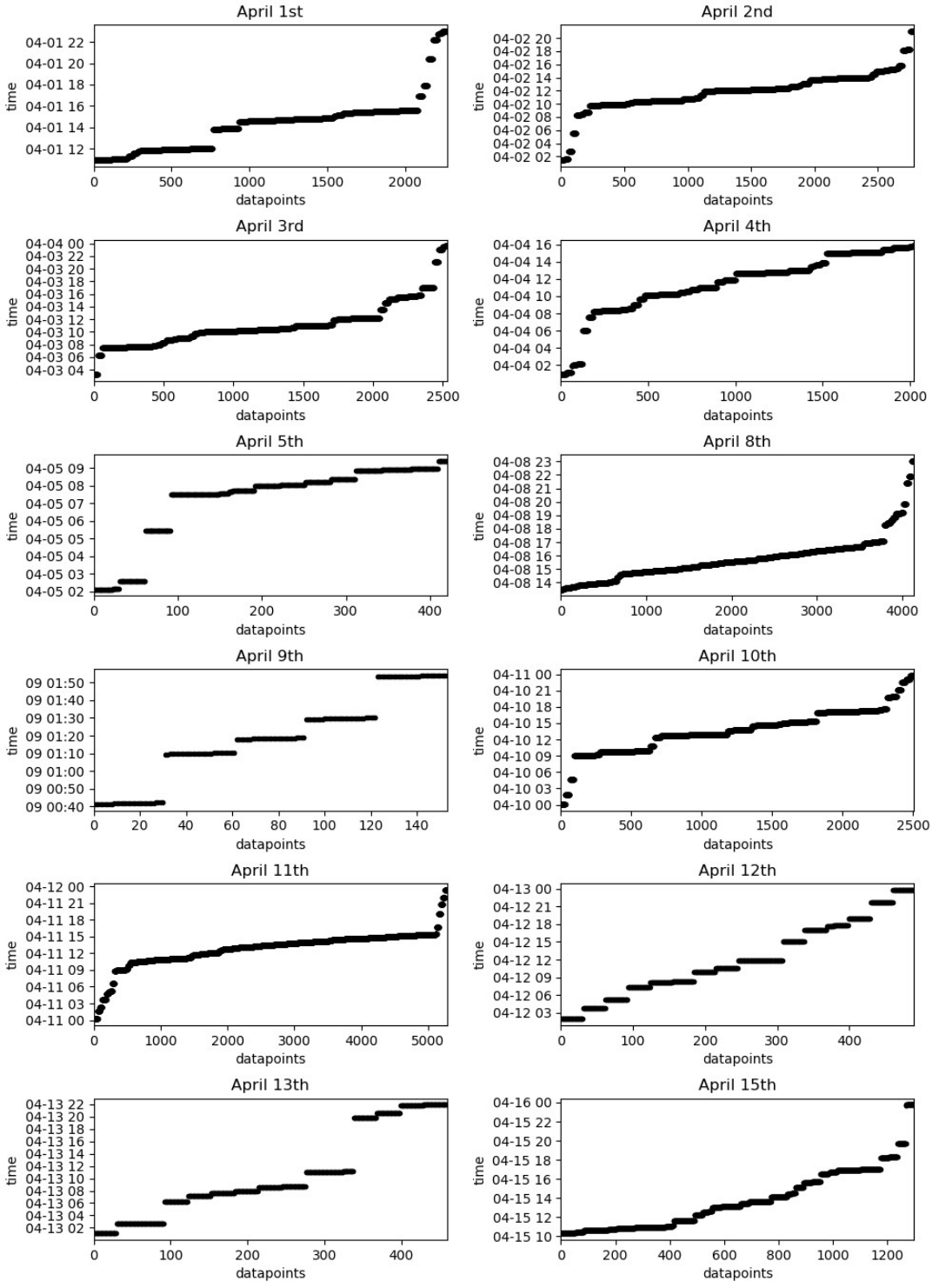

Figure 18: Time results April 1st to April 15th

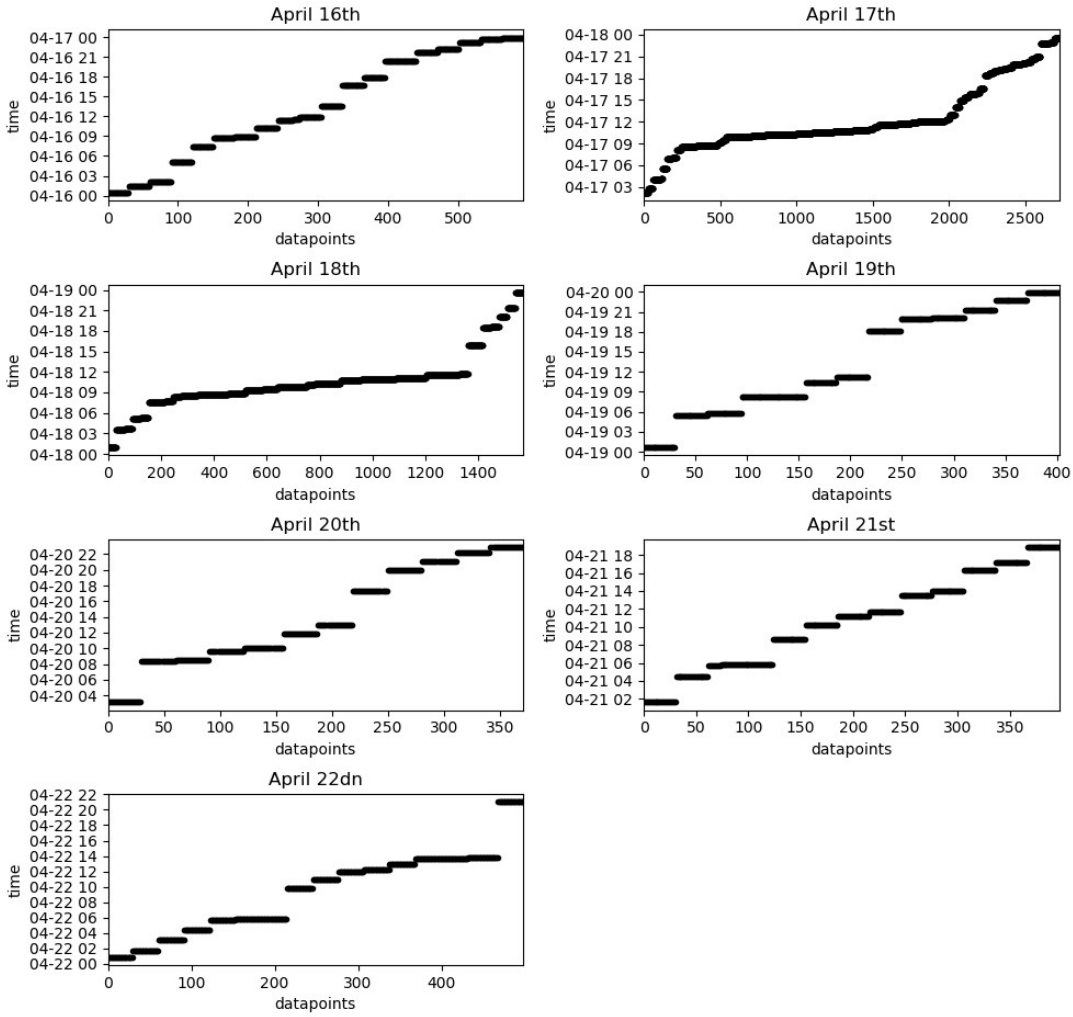

Figure 19: Time results April 16th to April 22nd

4.2.1. Result analysis

Table 2 shows two separable datasets:

i) present
ii) Away from keyboard (AFK).

These are distinguishable by deviation from median (sensor's stabile position) and number of datapoints collected per each day.

When test-subject was not sitting on test-chair, amount of datapoints was clearly on lower side (n<500) as displayed for example April 9th in Figure 20. Test-subject, while sitting on test-chair activates the sensor on movement. Even when person is sitting at test-chair only for a small part of workhours, such as April 18th on Figure 21, amount of recorded datapoints is three times bigger than on AFK - days (n>1500).

Graphs display multiple different conditions on test-chair. Discontinuing lines show when sensor was on either asleep or on transmitting. Short, symmetrical, and flat lines, especially on non-office hours, such as in Figure 21 April 22nd results - shows that sensor was activated periodically by a breeze from buildings air conditioning. Differences in AC-activated sequences come from test-chair being different parts at office and AC not working properly, as office-maintenance confirmed. Long, continuing lines at office hours display test-subject sitting on test-chair.

Lastly, sensor on test-chair being activated from proved useful on monitoring the system uptime. Using AC activation as a heartbeat, it could be verified that sensor was working even when there was not anyone present.

4.2.2. Closer look on singular days

Following single day charts, Figures 22 and 23 were chosen for their variance in test-subject data. Both cases display test-subject sitting on test-chair for most of the day and amount of datapoints is way over the n=500 AFK - mark.

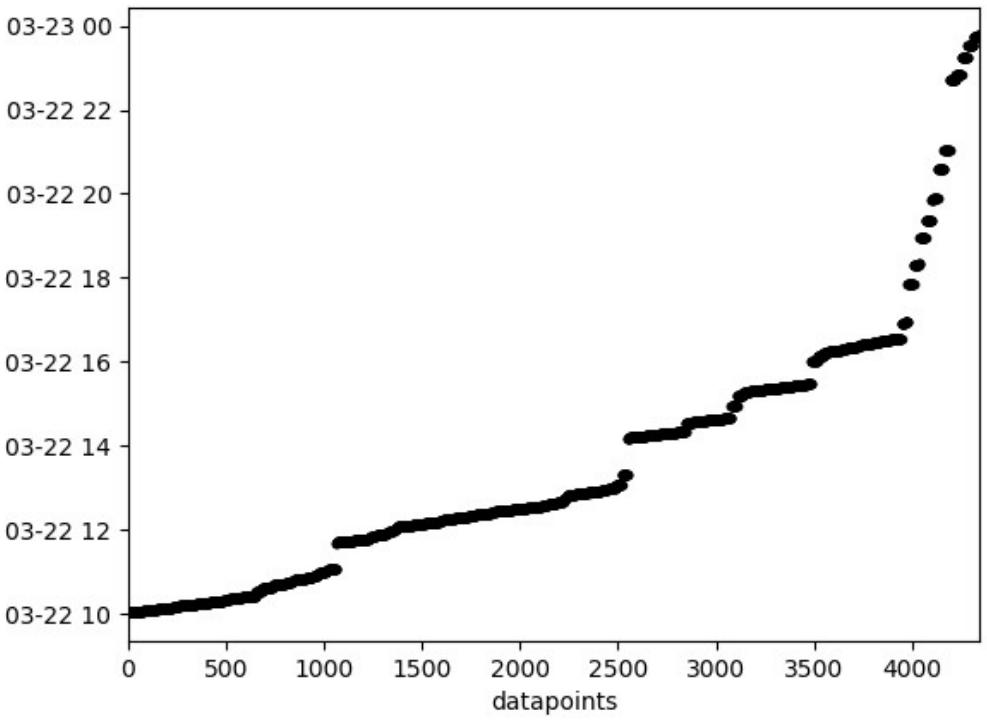

Figure 20: Results for March 22nd

Figure 22 displays results for March 22rd. This consisted of 4400 datapoints and test-subject sat in two-hour periods. Discontinuity in graph shows that test-subject was not sitting at test-chair constantly. Office has lunch at 11 o'clock and communal

coffee break at 14 o'clock. Both can clearly be seen from Figure 22 where graph discontinues.

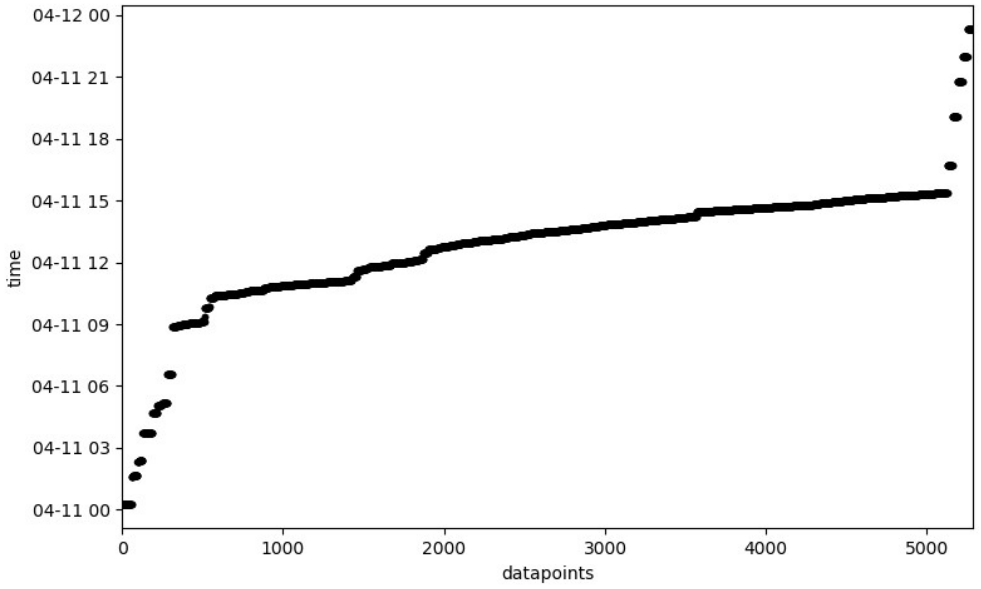

Figure 21: Results for April 11th

Figure 23 for April 11th was 5300 datapoints and graph displays the activation from AC from midnight to 6 AM. Continuous graph showed that test-subject sat through day from 9 AM to 15 PM without having any bigger breaks from sitting on test-chair.

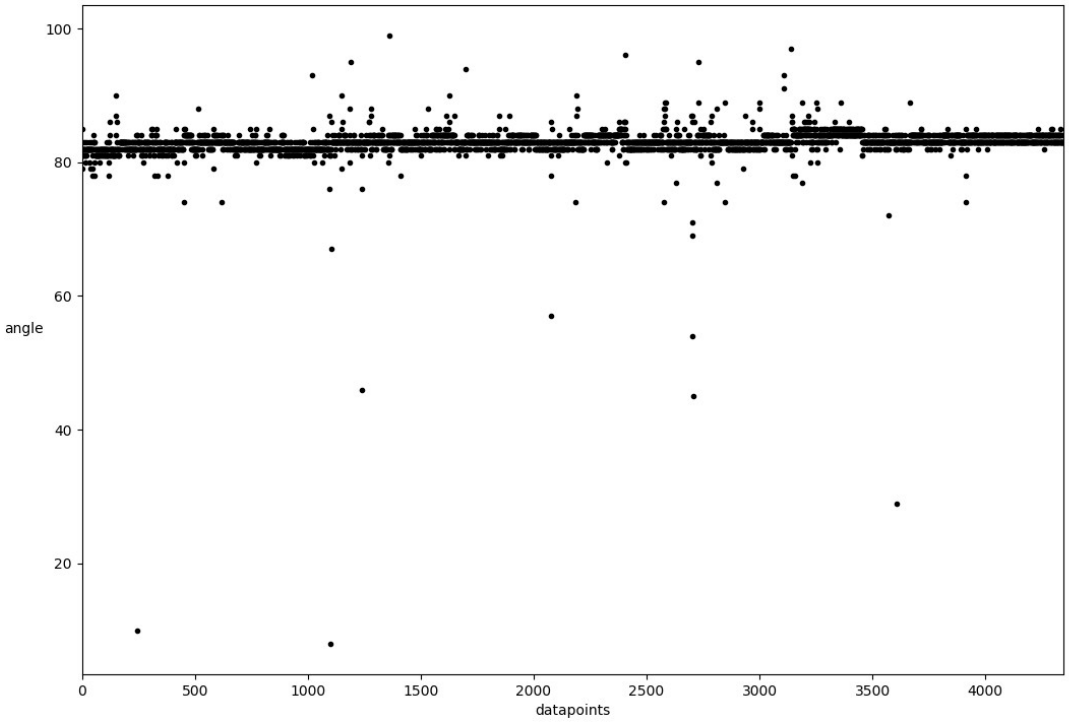

Figure 22: March 22nd angle data

Due the duty – sleep cycles and only 1 second interval for measuring the angle data, results shown in Figure 24 are not that interesting. When test-subject moved, sensor woke up. Angle data would be sent after one second and by then, most of the sensor's pendulum-swing movement was already done and sensor was already back to resting position, in about 84 degrees.

Accelerometer was part of a bigger system. Sensor could be calibrated for less sleep – hours and shorter intervals for sending the data, but this would have taken considerable toll on sensor battery. This was not possible due current setup. With this evaluation setup, device was running up to 12 days with single CR2032 lithium button battery. Some optimization can be done, see chapter *6 Discussion.*

4.3. Evaluation effects on participants

Engineers, designers and developers were somewhat suspicious when installing the sensor. They felt that they were monitored, and system might be used to evaluate their effectiveness. Few test-subjects wondered if they should be able to justify their time spent AFK during work hours.

However, after initial annoyance test had positive feeling. Biggest issue with test-subjects was the sensor-device cover swinging around and slapping to test-chair as test-subjects moved. Collected data proved to be interesting and we could evaluate if buildings' AC was on. In 2 cases, test-subjects felt that they are more motivated to stand while working after resulted graphs were displayed to them.

5. USAGE

After user pressing scan button displayed in Figure 25, all found devices are listed with their names and MACs displayed. User chooses the device and presses the connect button. After successful connection, the device's services are displayed. Characteristics under service are displayed when user clicks the service.

5.1. Run-time permissions and active adapters

App or library does not have features for managing permissions or adapters. User must manually enable these from Android system. More info about these see chapter *6 Discussion.*

5.2. Classes and User Interface

App has two Activities:

i) `ScanActivity.java`
This activity (View in figure ["MVC"]) displays results from `ScanController.java` (Controller in figure ["MVC"]). Found devices are put into `ScanEntry.java` (Model in figure ["MVC"]) and Recyclerview displays them into scrollable list for user. Sometimes scanner founds tens of devices, so results are limited to devices with local name, such as MLJSJJSS Demo and Smart Light in Figure 25. After device local name is device MAC address. Checkbox on far right indicates the one device app is attempting to connect with.

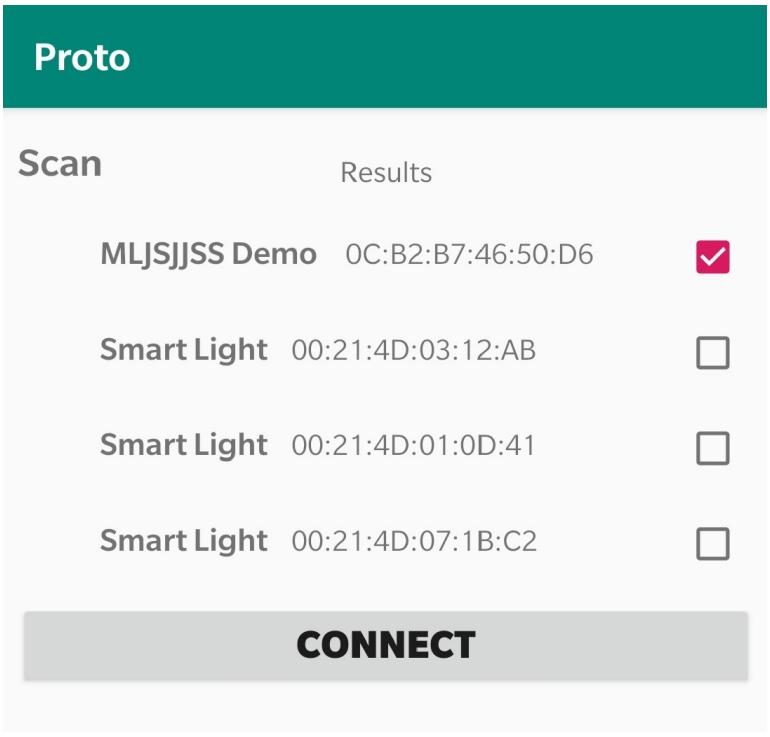

Figure 235: Scan Activity

ii) `DeviceActivity.java`

Device activity works architecturally similar fashion as `ScanActivity.java` but instead of scan results, it displays gatt-services and -characteristics. Names for these are parsed through `uuids.xml` for human readable form.

Line 2: 17,25 in Figure 26 is debug line specifically for Bluno beetle chip. Line angle 83 is last characteristic value sent from sensor chip. List saved – button lists database entries into Android logcat. Record button changes states from green (recording) to red (not recording). User button displays dialog for changing the username. Data is recorded with angle value, current user and timestamp.

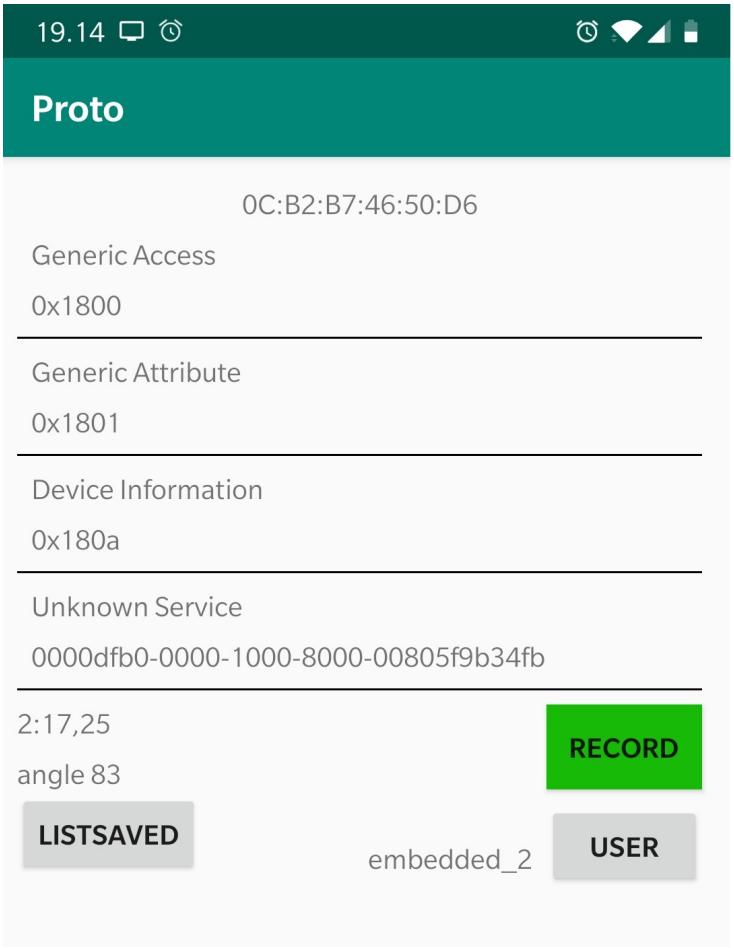

Figure 26: Device Activity

5.3. Real world examples

Good example for app with this sensor system would be hard to track with using just human senses. This can be for example for harmful noise-levels or detecting exposure and duration for harmful substances.

Apps such as this could be used for example as a reminder or warning system. Climate change is causing unprecedented heat weaves, which are dangerous to elderly people and other groups with weakened health. App could monitor air humidity and temperature in vicinity and remind user to stay hydrated. This could also be used with wearable tech and app could warn the user from staying in direct sunlight for too long and apply more sun cream.

Another type of warning could be from air quality meter. Emissions from traffic could be measured with sensors in street level. Nearby people could be warned through BLE advertisement channels if harmful particles exceed tolerance thresholds.

Useful system to people using cottages with fireplaces would be CO2 monitor in chimney. Sensor would send periodically information about CO2 levels and resident would be able to adjust airflow to fireplace for optimal burning rate.

All these examples could be done with singular, cheap sensor and slight modifications for app presented in this thesis.

6. DISCUSSION

App worked reliably during evaluation. However, being built as a POC, library only had most critical features to stay in scope for this thesis. Multiple development ideas rose during implementation and testing periods, but most were scrapped to keep library from straggling during POC-phase. Ideas are presented below, in chapter *6 Discussion.*

As mentioned in Chapter *2.3.1.1 UUIDs,* service- and character-identifiers take up big chunk of each sent BLE-data packet. Instead of using custom service- and character-identifiers such as in Figure 26 unknown service, developer could use existing 16-bit addresses instead mentioned in chapter *2.3.1.1 UUIDs*. Majority of the embedded-BLE chips battery usage is from transmitting data. Reducing the number of bytes sent by shortening the identifiers, will improve battery life considerably. However, this implementation could be considered cute and might lead into other problems such as mixing up Gatt-feeds and complicate gatt-implementation on both client and server.

Chapter *2.3.2.2 Gatt-Characteristic* mentioned that characteristics have descriptors, describing different properties for that characteristic. Currently app just displays them, but if app is used as debugging tool, characteristic descriptors could also be changed in `DeviceActivity.java`. However, this implementation would require additional user authentication and authorization, as current reader might be someone else than device owner. This is one of the challenges in BLE-systems as sensors can be left unattended. This implementation should be done separately for each use case. In source code, changing characteristic descriptors can be done in `GattService.java gattCallBack()` - method and setting each value separately for each descriptor.

App currently stores username to identify current test subject. Username could also be used to store different sensors to improve user experience when using the app. Currently evaluation only used one sensor that was moved around the office, so this feature was not needed. However, in the future using many sensors with many users simultaneously, database should store user specific sensors.

App could be automated to track user location with geofences [44]. Geofences are areas with specific GPS-coordinates. Concept is displayed in Figure 27. For example, when user enters geofence, app would attempt to connect to the sensor-device affiliated with those coordinates. This would be useful especially when user uses many separate sensors. This would also reduce the app's battery drainage, as GPS adapter and gatt-services could be turned off when user leaves the geofence.

Due the nature of work at RD Velho, activities handling permissions are not included in this thesis. Implementing them are straightforward as Internet is full of examples. Handing these at runtime is combination of Broadcasters and Managers to ensure graceful handling if permission or adapters are turned off during runtime. Handling is explained below.

Before asking for user to turn on GPS adapter, either `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` permission needs to be granted. After receiving permission for `LOCATION_SERVICE`, app can access `LocationManager`. This allows for user to be taken to GPS adapter page in settings to explicitly activate the adapter.

While Bluetooth adapter can be programmatically switched on from inside the app, activating GPS adapter needs to be done from Android own setting screen or from quick settings drop down menu. Often system gives a persistent notification if an app is using GPS.

Tracking Bluetooth is done with `BluetoothManager`. This manager provides an instance of `BluetoothAdapter`. `BluetoothAdapter` has constant `ACTION_STATE_CHANGED`, which changes if adapter state is switched.

GPS adapter state tracking is done with Android `location_service` and `LocationManager`. After getting user permission for location services, `requestLocationUpdates()` - method can be used for checking `GPS_PROVIDER` availability to check if provider's state changes, i.e. adapter is turned off during runtime. This periodical checking will affect smartphone's battery, so calling it should be optimized.

To take the sensor network even further, sensors could be installed into Bluetooth mesh network. Instead of using different gatt-profile for each sensor separately, the feed from entire mesh network could be communicated from Bluetooth mesh proxy-node (see chapter *2.4.3 Bluetooth Mesh).* This would require large amount of embedded chips and different architecture on gatt-server side.

Smartphone and app could be used as a gateway to store feed from sensors straight to cloud. Storing could be done with Google's HTTP library Volley [45]. Volley provides automatic scheduling for network requests so it wouldn't require much boilerplate code to implement REST-calls into gatt-triggers.

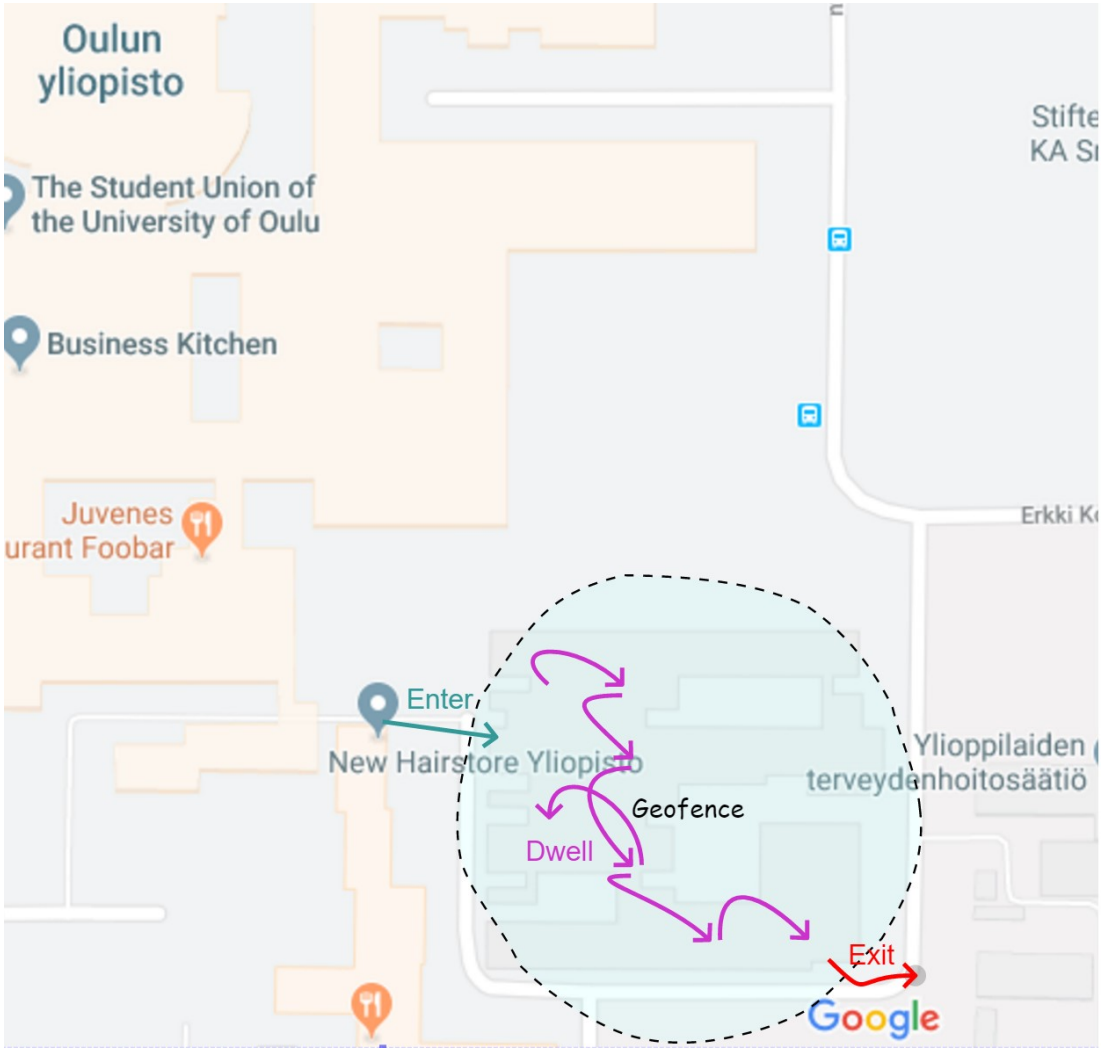

Figure 27: Geofence concept

7. CONCLUSION

Ongoing Internet of Things evolution requires embedded chips to be installed for things user wants to inspect, measure or command. Addressing each thing separately needs low-powered, scalable communication solution between communicating entities. Bluetooth Low Energy (BLE) is one of these solutions.

Android Operating System offers programming libraries to implement apps that are capable of BLE communication with discoverable devices. To effectively build apps in Android OS, developer is required to know how Android's memory management, internal messaging system and thread handling operates. Using BLE in apps will also require knowledge about BLE communication stack.

Problem is building a new system for each use case takes lot of boilerplate code and developer needs to understand a lot about Android system and BLE-stack to start developing. For example, user does not have complete control for Android's memory usage and system could abruptly terminate monitoring service if device runs out of memory. Android also restricts actions that might make UI unresponsive, such as retrieval of data from database or accessing online resources, when interacting method call happen directly from UI-thread.

This thesis offers a starting point as an Android app and separate Bluetooth Low Energy – library. Thesis also offers guidance for configuring the library for individual needs and gives remarks for improving the library.

App was tested for a month with fabrio.io reporting tool and eight people as test subjects. Test was concluded in office space and measured office chair movement using embedded accelerometer sensor. Sensor had Bluetooth radio which sent sensor movement into Android smartphone using BLE-stack. Sensor was acting as Gatt-server and smartphone was gatt-client. Results from sensor were stored into local Room database. These measurements are displayed in chapter *4 Evaluation*. Also, chapter 4 provides fabric.io report and analysis for results and most causes for fabric-reported crashes.

Results proved that app had good uptime and recorded data reliably through test period. Stored results provided meaningful set of data that accurately measured test-subjects presence in their desktop.

App worked reliably during evaluation. However, being built as a POC, library only had most critical features to stay in scope for this thesis. Multiple development ideas rose during implementation and testing periods, but most were scrapped to keep library from straggling during POC-phase. Ideas are presented above, in chapter *6 Discussion.*

8. REFERENCES

[1] "SIG INTRODUCES BLUETOOTH LOW ENERGY WIRELESS TECHNOLOGY, THE NEXT GENERATION OF BLUETOOTH WIRELESS TECHNOLOGY" [Online]. Available: https://www.bluetooth.com/news/pressreleases/2009/12/17/sig-introduces-bluetooth-low-energy-wireless-technologythe-next-generation-of-bluetooth-wireless-technology [Accessed: 28-Jan-19]

[2] "Bluetooth low energy overview" [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth-le [Accessed 29-Jan-19]

[3] "Room Persistence Library" [Online] Available: https://developer.android.com/topic/libraries/architecture/room [Accessed 29-Jan-19]

[4] "Mobile Operating System Market Share Worldwide" [Online]. Available: http://gs.statcounter.com/os-market-share/mobile/worldwide [Accessed: 29-Jan-19]

[5] "GATT Overview" [Online] Available: https://www.bluetooth.com/specifications/gatt/generic-attributes-overview [Accessed: 29-Jan-19]

[6] (Android) "Architecture" [Online] Available: https://bootlin.com/doc/training/Android/Android-slides.pdf [Accessed: 26-Oct-18]

[7] "Save data in a local database using Room" [Online] Available: https://developer.android.com/training/data-storage/room/ [Accessed: 28-Jan-19]

[8] "Phone, Tablet & PC Bluetooth 2018" [Online] Available: https://www.bluetooth.com/markets/phone-pc [Accessed: 25-Feb-19]

[9] "Android Handler Tutorial" [Online] Available: https://medium.com/@manishgiri/android-handler-tutorial-ccda6994f01c [Accessed: 25-Feb-19]

[10] "A short story about Android BLE connection timeouts and GATT internal errors" [Online] Available: https://blog.classycode.com/a-short-story-about-android-ble-connection-timeouts-and-gatt-internal-errors-fa89e3f6a456 [Accessed: 25-Feb-19]

[11] Author's GitHub page [Online] Available: https://github.com/tappoman/BTAndroid [Accessed: 25-Feb-19]

[12] " GATT Services" [Online] Available: https://www.bluetooth.com/specifications/gatt/services [Accessed 1-Mar-19]

[13] "Specifications" [Online] Available: https://www.bluetooth.com/specifications [Accessed 1-Mar-19]

[14] P. Dutson, *Android Development Patterns: Best Practices for Professional Developers.* ch. 4. Components p. 46-47, 2016.

[15] P. Dutson, *Android Development Patterns: Best Practices for Professional Developers.* 13. Optional hardware APIs p. 173, 2016.

[16] RD Velho homepage [Online] Available: https://www.rdvelho.com/en/ [Accessed 1-Mar-19]

[17] "GATT Characteristics" [Online] Available: https://www.bluetooth.com/specifications/gatt/characteristics [Accessed 2-Mar-19]

[18] J. C. Haartsen, "The Bluetooth radio system" *IEEE Personal Communications*, vol. 7, no. 1, pp. 28-36, 2000.

[19] C. Gomez, J. Oller, J. Paradells, "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology." *Sensors* (Basel, Switzerland), 12(9), pp. 11734-11753, 2012.

[20] M. Siekkinen, M. Hiienkari, J. K. Nurminen and J. Nieminen, "How low energy is Bluetooth low energy? Comparative measurements with ZigBee/802.15.4," *2012 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, Paris, pp. 232-237, 2012.

[21] D. Miorandi, S. Sicari, F. De Pellegrini, I. Chlamtac, "Internet of things: Vision, applications and research challenges." *Ad Hoc Networks* September 2012, Vol.10(7), pp.1497-1516, 2012.

[22] "About International Telecommunication Union (ITU)" [Online] Available: https://www.itu.int/en/about/Pages/default.aspx [Accessed 26-May-2019]

[23] J. A. Martins, "Hypermedia APIs for the Web of Things." *IEEE Access*, 5, pp. 20058-20067, 2017.

[24] D.D. Guinard, V. M. Trifa, "Building the Web of Things: With examples in Node.js and Raspberry Pi.", 2016.

[25] L .Atzori, A. Iera, G. Morabito, "The Internet of Things: A survey." *Computer Networks* 2010, Vol.54(15), pp.2787-2805, 2010.

[26] D. Ferreira, "AWARE: A mobile context instrumentation middleware to collaboratively understand human behavior." Oulu: University of Oulu. 2013

[27] O. Uviase, G. Kotonya, “IoT Architectural Framework: Connection and Integration Framework for IoT Systems” *ALP4IoT@iFM*, 2017.

[28] “Message Queuing Telemetry Transport” [Online] Available: http://mqtt.org/ [Accessed: 7-Jul-19]

[29] “Advanced Message Queuing Protocol” [Online] Available: https://www.amqp.org/ [Accessed: 7-Jul-19]

[30] “Build. Understand. Grow.” [Online] Available: https://get.fabric.io/ [Accessed: 7-Jul-2019]

[31] “2018 reform of EU data protection rules” [Online] Available: https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules_en [Accessed: 7-Jul-2019]

[32] “Mesh Networking Specifications” [Online] Available: https://www.bluetooth.com/specifications/mesh-specifications/ [Accessed: 7-Jul-2019]

[33] R. Faragher, R. Harle, “Location Fingerprinting With Bluetooth Low Energy Beacons.” *IEE Journal on selected areas in communications*, vol. 33, No. 11, November 2015, 2015.

[34] N. Gupta, “Inside Bluetooth Low Energy”, Second Edition (Mobile Communications), 2016.

[35] “Bluetooth Low Energy – It starts with Advertising” [Online] Available: https://www.bluetooth.com/blog/bluetooth-low-energy-it-starts-with-advertising/ [Accessed: 7-Jul-2019]

[36] “Generic Access Profile” [Online] Available: https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile/ [Accessed: 7-Jul-2019]

[37] “Assigned Numbers” [Online] Available: https://www.bluetooth.com/specifications/assigned-numbers/ [Accessed 7-Jul-2019]

[38] ”At the core of everything Bluetooth / Bluetooth Core Specification version 5.1” [Online] Available: https://www.bluetooth.com/specifications/bluetooth-core-specification/ [Accessed: 7-Jul-2019]

[39] ”Archived Specifications/ Bluetooth Core Specification version 4.0” [Online] Available: https://www.bluetooth.com/specifications/archived-specifications/ [Accessed: 7-Jul-2019]

[40] C. Gomez, J. Oller, J. Paradells, “Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology.” *Sensors* 01 August 2012, Vol.12(9), pp.11734-11753, 2012.

[41] ”Permissions overview” [Online] Available: https://developer.android.com/guide/topics/permissions/overview [Accessed: 7-Jul-2019]

[42] ”Activity” [Online] Available: https://developer.android.com/reference/android/app/Activity [Accessed: 7-Jul-2019]

[43] P. Dutson, “Android Development Patterns: Best Practices for Professional Developers.” ch. 4. Components p.48, 2016.

[44] “Create and monitor geofences” [Online] Available : https://developer.android.com/training/location/geofencing Accessed: 7-Jul-2019]

[45] “Volley overview” [Online] Available: https://developer.android.com/training/volley [Accessed: 7-Jul-2019]

[46] “DB Browser for SQLite” [Online] Available: https://sqlitebrowser.org/ [Accessed: 7-Jul-2019]

[47] B. Phillips, C. Stewart, B. Hardy, K. Marsicano, “Android Programming. The Big Nerd Ranch Guide”, 3rd Edition, 2017.

[48] P. Dutson, “Android Development Patterns: Best Practices for Professional Developers.” ch. 8. Application Design: Using MVC p. 118, 2016.

9. APPENDICES

Appendix 1. Dependencies for Android Studio

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-
layout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation
'com.android.support.test:runner:1.0.2'
    androidTestImplementation
'com.android.support.test.espresso:espresso-core:3.0.2'
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support:design:28.0.0'
    implementation 'com.android.support:recyclerview-v7:28.0.0'
    implementation 'android.arch.persistence.room:runtime:1.1.1'
    annotationProcessor
'android.arch.persistence.room:compiler:1.1.1'
}
```