A generic smartphone-based sensing and processing system applied to epileptic seizure monitoring

Roberts, F.J.M.

Award date:
2013

Technische Universiteit
Eindhoven
University of Technology

ACTLab,
Signal and Processing Systems,
The Department of Electrical Engineering
5T746

A generic smartphone-based sensing and processing system applied to epileptic seizure monitoring

Advisors:
Dr. Oliver Amft
Msc. Gabriele Spina
by F.J.M. Roberts [0632676]

Date: December 13, 2012

Contents

1 Introduction

Smartphone-based activity monitoring and behaviour recognition are key for various applications in patient care and support. Multiple sensor locations are essential to acquire a wide range of activities. Thus, external sensors need to be used in addition to the smartphone-integrated sensors. Smartphones and additional external sensors could yield an unobtrusive long-term monitoring, data interpretation, and feedback platform, applicable for a wide range of chronic diseases. Current smartphone-based sensing and processing systems are addressing single applications mostly, and lack features that are essential for recording and annotation, hence for recognition algorithms development. Existing home monitoring systems use for instance video cameras or dedicated devices that need to be worn. Such systems have a large impact on cost, installation, maintenance effort, and complexity of the data processing. While wearable systems (including smartphones) are ubiquitous, they introduce challenges. Such as limited battery-life and the ability to deploy them in a wide range of environments, which result in the user requirements for a generic smartphone-based sensing and processing system. This project targets to realise a configurable architecture that supports different sensor input, processing, and output components, implemented on Android. The concept is inspired by existing frameworks, e.g. CRNT, FUNF.
The developed architecture will be analysed by a case study. For this purpose, epileptic seizure monitoring is targeted. Epilepsy is a chronic illness with a large negative impact on the quality of life. Due to the high variance of patients and their seizures, it is difficult to evaluate their medication and overall progress after leaving a controlled hospital environment. Monitoring the seizure rate outside the hospital is a key information to aid medication and to evaluate treatment progress. In this project the focus is on epileptic seizure detection, not restricted to lab environment. Similar work has been done before, but suffers from high rate of misclassification's.

2 Related work

During this project main topics that have been researched are epileptic seizure detection using accelerometers and framework design for smartphones. Previous work regarding epileptic seizure detection will be discussed in Section 6.1. This section will focus on the other smartphone sensing frameworks that have been researched.

2.1 Frameworks

2.1.1 CRNT

The Context Recognition Network Toolbox (CRNT) is a software framework developed facilitate rapid prototyping of activity recognition applications. CRNT has several standard modules for reading sensors, processing data and supplying output data. By creating a JSON configuration file users of CRNT can create an application without having an in-depth knowledge of a programming language. If the user requires a functionality which is not supported it is possible to extend CRNT by writing custom modules.
CRNT has been ported to many different platforms over the years and CRNT based applications have been deployed in the field [1].
CRNT was also ported to Android in April 2011 by Jakob Weiger for his bachelor thesis. Google's NDK (Native Development Kit) was used to port the C++ code of the toolbox to the Android platform citeWeigert2011. An Android application using the CRNT port was created called CRN Toolbox Center (CRNTC). With this application the user could start the toolbox by selecting a configuration file located on the sdcard of the smartphone amongst other functionality.
In this set-up the CRN toolbox retains it strong functionality and expandability however the CRNTC application is fairly limited and various metrics of it's operation have not been researched. The original CRNTC application only supports the internal sensors, data can only be output by using CRNT to write data to the sdcard or wireless transmission and the general architecture was not specifically designed and implemented to allow for the addition of new modules. Metrics such as energy consumption and the ability to process data on-line have not been researched [2].

2.1.2 FUNF

The FUNF Open Sensing Framework is an extensible sensing and data processing framework for mobile devices, developed at the MIT Media Lab. The core concept is to provide an open source, reusable set of functionalities, enabling the collection, uploading, and configuration of a wide range of data types [3].

Like CRNT, users of the FUNF framework also specify a JSON configuration file which specifies the probes (modules in FUNF) that will be used to gather and analyse data. FUNF also offers several ways of outputting the data such as encrypted database storage but also sending it over WiFi. Since energy is constrained on smartphones the framework reduces energy consumption. Unlike CRNT, FUNF only runs on Android based smartphones. Out of the box there are a wide range of data logging possibilities for instance movement and location but also app and social media usage [4].
Funf is very suitable for gathering large amounts of data from a smartphone, however funf offers only limited support for processing this information on the smartphone.

2.1.3 OSAS

The Open Service Architecture for Sensors (OSAS) is an event-based programming system for sensor networks designed by the System Architecture and Networking (SAN) group in TU/e which can be used for programming heterogeneous Wireless Sensor Networks (WSN) on a high abstraction level. OSAS was partially developed inside the Wireless Accessible Sensor Populations (WASP) project. The OSAS platform allows sensors to be reprogrammed over the network by translating a single program for the entire network into a set of configuration messages and bytecode [5].
Within the SAN group the OSAS system is being used as test-bed and research vehicle for several projects. One of those projects is the VITRUVIUS project. The VITRUVIUS project aims at exploring the underlying key consequences for the architecture of Body Sensor Networks (BSN) and the handling of information about the individual's body. The connection of the privacy-sensitive "body state" to a rapidly evolving landscape of services, is the key theme of the project [6].
For the VITRUVIUS project the OSAS system has been ported to Android.
The OSAS framework only makes programming the sensor nodes in a sensor network easier but still requires a regular amount of coding in order to create an application that handles all the sensor data.

2.1.4 SENSE-OS

The SENSE Observation System was developed by The Sense company in Rotterdam. SENSE-OS consists of two components. The first is the Sense platform and can be downloaded for Android smartphones and iOS devices such as the iPhone or iPad. The second is the CommonSense platform which is cloud based. The Sense platform on the mobile device sends data into the cloud where it can be collected, interpreted and shared. The platform also offers some third-party sensor support via Bluetooth and ANT [7].
Like the FUNF framework, SENSE-OS is able to acquire data however the processing is done in the cloud and processing algorithms have to be written with a proprietary API.

3 Architecture design

3.1 Requirements

Based on the initial task-description, related literature, similar frameworks and various interviews conducted with people working within the ACTlab group a number of requirements have been derived.

3.1.1 Functional requirements

1. **Sensor input components**. The final design should represent different sensors as components.
2. **Output component**. The final design should have components to visualize, store or send data.
3. **Processing components**. The final design should have components to process input data.
4. **Extensibility with additional functionality**. The final design should be (easily) extensible with new components.
5. **ANT support**. Ability to support the ANT wireless protocol.
6. **User annotation** Users should be able to annotate data directly on their devices.
7. **Sensitive files not accessible by subjects**. Files such as configuration and log files should not be directly accessible by users.
8. **Ability to add GUI components**. The final design should allow the addition of GUI components.

3.1.2 Quality requirements

1. **Energy efficient**. Energy usage should not cripple the ability to record data for longer periods of time.
2. **Minimum coding effort**. By using a configuration file there is little programming needed to create a sensing application.
3. **Scalable**. The final implementation should still work when a relatively large number of components are used.

4. **Documentation**. Documentation on how to create a sensing application with the final design should be available
5. **Privacy**. The final implementation should guarantee a certain level of privacy of its users.
6. **Data is MATLAB compatible**. The generated data should be easily imported in MATLAB.

3.2 Comparison

The following section compares all frameworks discussed in Section 2.1 with the requirements from Section 3.1. Table 1 indicates whether a framework either already supports a certain requirement or if it is possible to extend the framework with the requirement in mind.

	FUNF	CRNT	SENSE-OS	OSAS
Sensor input components	+	+	+/-	+
Output components	+	+	+/-	-
Processing components	+	+	-	-
Extensible	+	+	+/-	+/-
ANT support	+	+/-	+	+
Energy efficient	+	+/-	+	+
Minimum coding effort	+	+	+	-
User annotation	-	+	+	+
Scalable	-	+	+	+
Sensitive files not accessible	+	+	+	+
Documentation	+/-	-	+	+/-
Privacy	+	+/-	+/-	+
Ability to add GUI components	+	+	-	+
MATLAB compatible output	-	+	-	+

Table 1: Framework comparison table. See Section 3.1 for the explanation of the table fields.

Based on this comparison and assessing all the requirements and use-cases, a choice was made to use the Android CRN Toolbox. The toolbox fits the following functional requirements from Section 3.1.1: 1,2,3 and 4. Next to the functional requirements the following quality requirements are met: 2,3 and 6. The toolbox also runs natively on Android making it very efficient [2].

However CRNTC only supports the internal sensors of the smartphone and the architecture is not designed to be extended upon. Also CRNTC has no capabilities to output and visualize data on it's own and is reliant on some functionality of CRNT or other Android applications such as the SimpleGraph application [2].

3.3 Use cases

Based on the functional and quality requirements several use cases have been derived. The use cases are specified in the UML format.

3.3.1 Add sensor component to framework

pre-condition:
Smartphone supports sensor hardware.
trigger:
A new sensor component is needed for framework.
guarantee:
Sensor component is added to framework.
scenarios:
1: Programmer writes component to decode sensor data.
2: Component is compiled and added to the framework.
3: Programmer adds documentation.

3.3.2 Add processing component to framework

pre-condition:
none.
trigger:
A new processing component is needed for input data.
guarantee:
Processing component is added to framework.
scenarios:
1: Programmer writes component which can analyse sensor data.
2: Component is compiled and added to framework.
3: Programmer adds documentation.

3.3.3 Add output component to framework

pre-condition:
none.
trigger:
A new output component is needed.

guarantee:
Output component is added to framework.
scenarios:
1: Programmer writes component which can extract features from input data.
2: Component is compiled and added to the framework.
3: Programmer adds documentation

3.3.4 Add user-interface component to application

pre-condition:
A framework application exists
trigger:
A user-interface is needed.
guarantee:
a user-interface component is connected to the application.
scenarios:
1: User interface is created with various GUI elements.
2: GUI elements are connected with input and output components.

3.3.5 Create data-gathering application

pre-condition:
All necessary components are implemented.
trigger:
Researcher wishes to gather data from subjects.
guarantee:
Data gathering application is created.
scenarios:
1: Sensor input components are added.
2: An output component is added.
3: Application is deployed/used.
4: Data is annotated off-line.
alternative:
1.1: Sensor input components are added.
1.2: An annotation component is added.
1.3: Add data merge component.
4.1: Data is annotated on-line.
alternative:
1.1: Sensor input components are added.
1.2: Processing component is added to synchronize sensor data

3.3.6 Train data offline

pre-condition:
Data is available.
trigger:
Researcher wishes to use data for online processing.
guarantee:
A file is created with trained weights/parameters/coefficients.
scenarios:
1: User annotated data is imported in MATLAB.
2: Data is trained in MATLAB.
3: Training file is generated by MATLAB script.
4: File is uploaded to smartphone.
5: Processing component uses file for classification.
alternative:
1.1: Data is imported in MATLAB.
1.2: Data is annotated by researcher.

3.3.7 Create data processing application

pre-condition:
All necessary components are implemented.
trigger:
Researcher wishes to process data on-line.
guarantee:
Data processing application is created.
scenarios:
1: Sensor input components are added.
2: Processing components are added.
3: Location of training file is specified.
3: Output components are added.
4: Application is deployed/used.
alternative:
3.1: Unsupervised learning is used, no training file needed.

3.3.8 Annotate data

pre-condition:
Framework application exists with user annotation.
trigger:
User wishes to annotate action.
guarantee:
Data is annotated.
scenarios:
1: User annotates action.
2: Input data is annotated.
3: After action is done, user annotates with idle action.
alternative:
4: User adds comment for researcher.

3.3.9 Use-case diagram

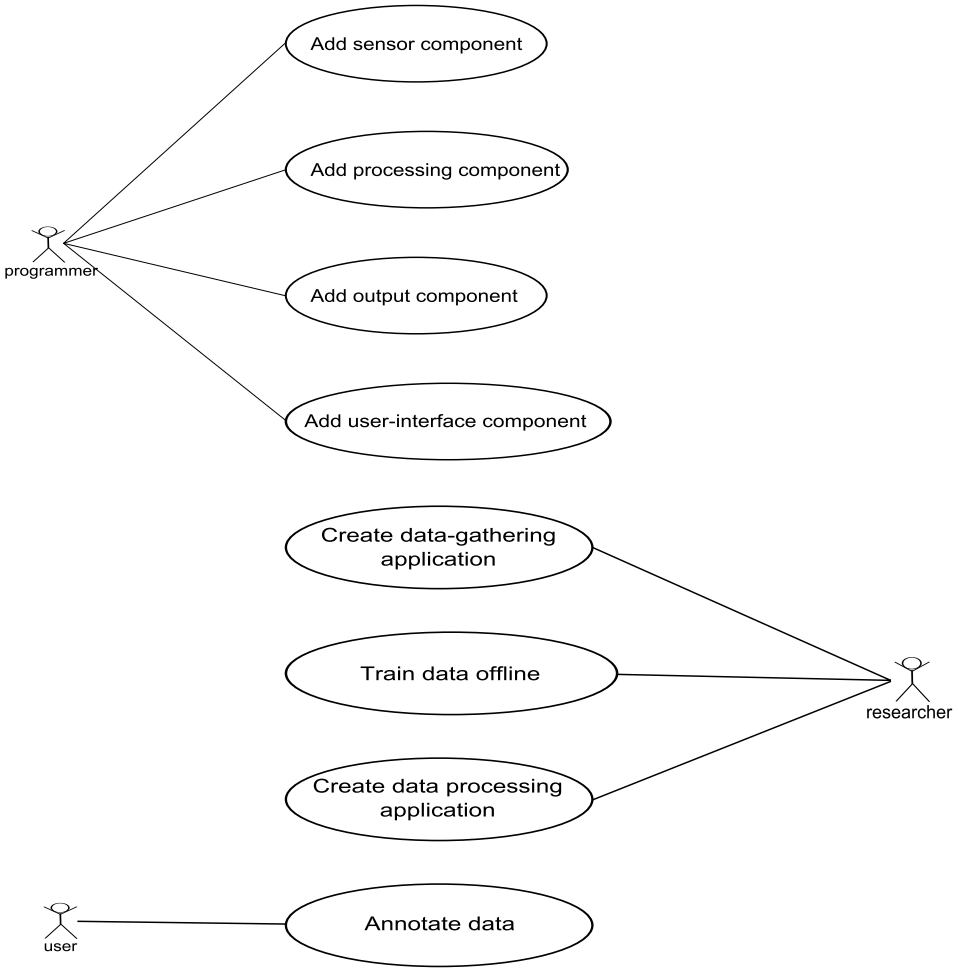

Figure 1: Use case diagram

3.4 Design

The CRN Toolbox library is integrated in an Android app called CRN Toolbox Center (CRNTC). Next to the toolbox there are several other modules which provide for instance accelerometer sensor data, GPS data and a user interface. The CRNTC app uses a Model View Controller (MVC) design pattern.

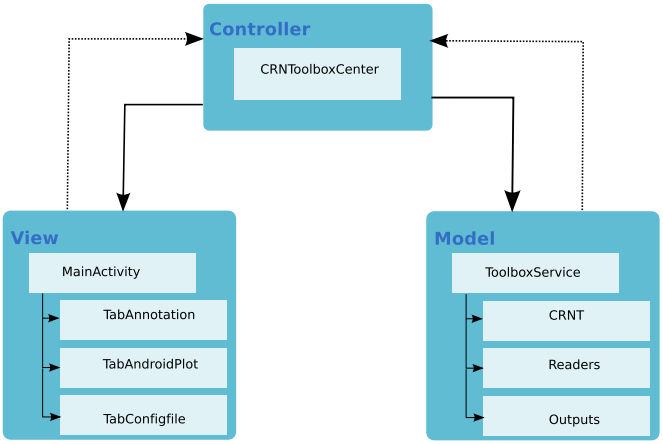

Figure 2: Simplified overview of the CRN Toolbox Center concept. All the different screens are part of the view part of the architecture. The CRNToolboxCenter class can be considered as the controller class, providing communication between the crnt library and the graphical user interface.

The MVC design pattern decouples the graphical user interface from the part of the architecture that actually does the computations on the data. The data processing part (model) is connected to the graphical user interface (view) by an abstract controller class. This design pattern is often used in smartphone applications due to a large amount of differing display sizes and types. Although the CRN Toolbox meets several important requirements others are not met namely: ANT support (functional requirement 5), User annotation (functional requirement 6) , Ability to add GUI components (functional requirement 8), Energy efficient (quality requirement 1), Documentation (quality requirement 4) and Privacy (quality requirement 5). In order to meet all the requirements several more modules are added to the CRNTC app, such as a user interface for annotation and ANT sensor support.

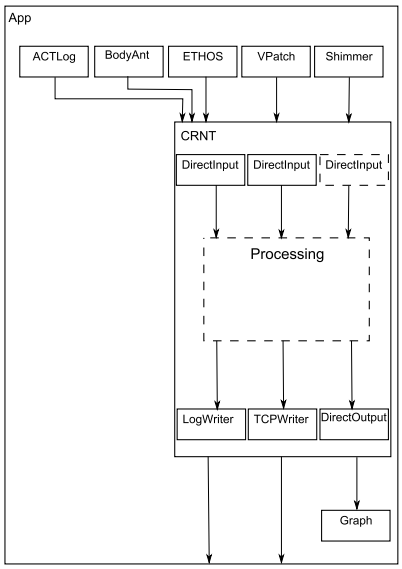

Figure 3: Architecture overview. The top row shows the software module representations of the various sensors that can be used in an application created with the framework. Their data is send to the CRN toolbox for processing or storage. The CRN toolbox is capable of sending the data over a wireless connection or store it on the sdcard of the smartphone. The toolbox can also output the data to an output component of the CRNCTC+ framework for visualization purposes for instance.

A single configuration file will be read which determines the modules that should be instantiated. This configuration file also contains the settings for the library. In order to receive data from the toolbox directly a DirectOutput module needs to be added to the toolbox library.

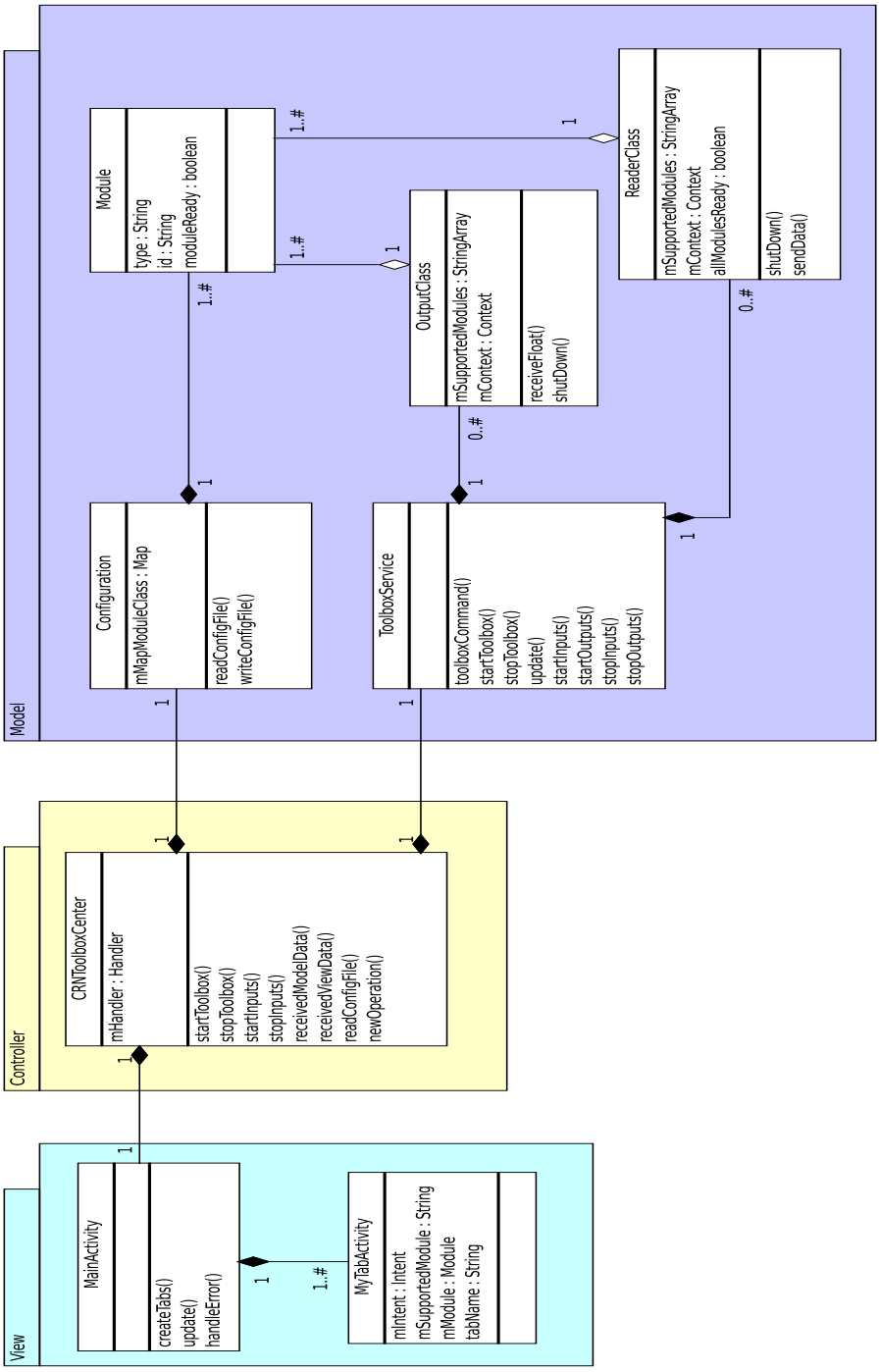

Figure 4: Class diagram of the CRNTC app with several new classes added to meet all the requirements

3.4.1 Description of classes

- **CRN Toolbox Center** This is the central class in the design and facilitates the communication between the GUI elements, the CRN library and the other modules. It is implemented as a singleton class meaning that during the lifetime of the application there is only one instance of this class.
  Within the MVC design pattern this class together with the Toolbox Service class can be regarded as the controller.
- **Toolbox Service** The Toolbox Service class communicates directly with the CRN toolbox. It is also responsible for creating all the appropriate Reader classes based on the configuration file.
- **MainActivity** The MainActivity class is responsible for all the different GUI elements.
- **ConfigObject** Reads in and analyses configuration files. It also instantiates the appropriate modules based on the configuration file.
- **ReaderClass** A ReaderClass object usually reads raw sensor data from hardware in, or connected to the smartphone. ReaderClass objects manage one or more modules which can interpret the raw sensor data.
- **Module** Module classes represent the various components in the CRNCT+ system. At the bare minimum they can contain information about the component. In case of sensor components for instance their modules also contain a method for decoding the raw sensor data from the reader classes. Module classes are instantiated based on the components described in the configuration file.

3.4.2 Design challenges

During the design of the new CRNTC app (CRNTC+) several challenges were identified that are critical to realise the system functionality during the implementation phase.

1. **Configuration file** In order to determine which modules should be used outside of the CRN Toolbox the CRNTC+ app needs to read a configuration file. Since the CRN Toolbox also reads in a configuration file it would be easier for the user if both information could be embedded in the same file. However without modifications the CRN Toolbox is not able to read parts of the configuration file that are meant for the external modules. Several precautions should be taken into account in order to make sure that the CRN Toolbox gets the configuration file data it can read. For instance a temporary CRNT configuration file can be created which can be read by the library.

2. **DirectOutput module** The CRN Toolbox uses DirectInput modules to communicate with modules from outside the library to receive input from sensors. To send output data the CRN Toolbox can use tcp/udp connections or write log files to the smartphone's sdcard. Nonetheless it could be useful to have a direct data transfer through DirectOutput modules. Since the architecture will be running on smartphones energy consumption is a problem and constant communication over tcp or udp might drain the battery quicker. Making use of DirectOutput modules would require changing the CRNT library.
3. **User interface** The CRNTC app uses tabs to switch between different views. This is very useful when a user wishes to annotate data but also wants to see if the sensors work by means of some sort of visualization. On the other hand if the user wishes to use many different views the top tab bar might no longer be readable because there is not enough space left to display all the icons.
4. **Ease of adding extensions** Although adding modules to the CRN Toolbox is relatively easy, adding new modules or GUI screens to the CRNT+ app might be a bit more difficult. This is because subtle changes might have to be made to numerous other modules in the app.

3.5 Architecture metrics

In order to rate the performance of the resulting CRNTC+ app several metrics will be defined based on some of the requirements in Sections 3.1.

1. **ANT support:** ANT support will be measured in the number of different ANT devices that are being supported by the framework. At least the following need to be supported:
   - BodyANT
   - ETHOS
2. **Extensibility:** The ease of creating a new module for the framework will be analysed by counting the number of steps and the complexity per step, needed to add a module to the framework. The number of steps for the following use-cases, stated in Section 3.3 and listed below for convenience, will be analysed:
   - Add sensor component to framework
   - Add output component to framework
   - Add user-interface component to application
3. **Energy efficiency:** Several real-world sensing and processing applications will be compared with each other. Although the energy usage changes depending on the setup of the application, a day's worth (about 12 hours) of operation would be the ideal situation.
4. **Scalability:** Scalability will be defined as the ability to add sensors and modules while at the same time the framework application will still be able to function in a desired fashion. To measure the scalability ANT type sensors are incrementally added. The ability to function in a desired fashion will be measured by the following criteria:
   - CPU usage
   - Measurement time
   - Sensor jitter
5. **Documentation** There should be documentation explaining every use-case from Section 3.3 in detail.

4 Implementation

4.1 ANTReader

The implementation of the ANTReader is based on the ANT+ for Android source code provided by Dynastream Systems [8].

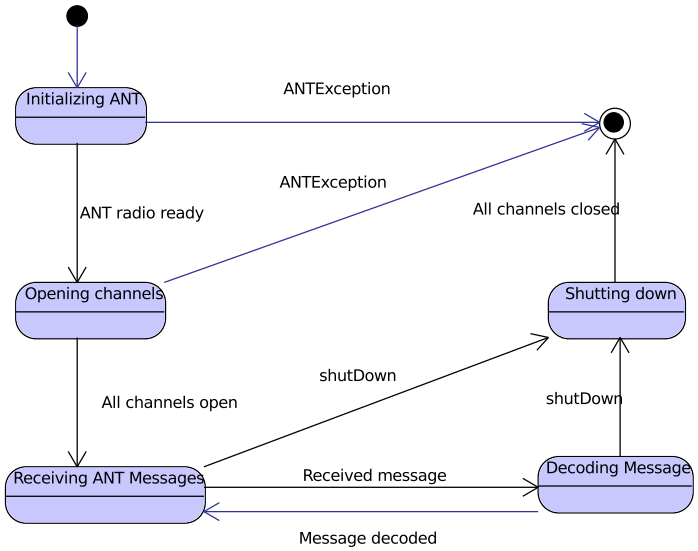

Figure 5: The ANTReader will first try to initialize the ANT radio. When successful the channels as defined by the modules of the ANTReader will be opened. When all channels are successfully opened the ANTReader is ready to receive ANT data and decode the data to messages.

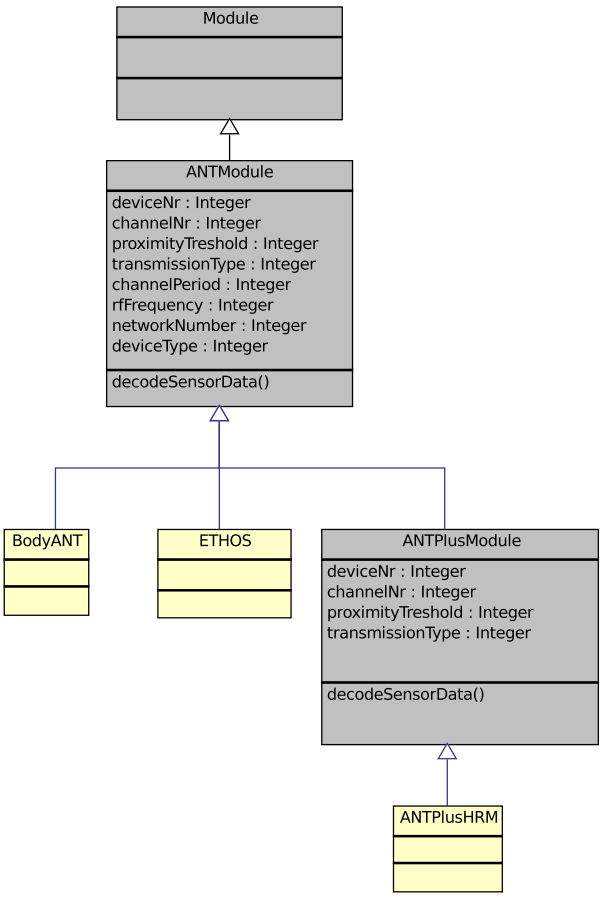

Figure 6: All ANTModule objects have a decodeSensorData method which gets called by the ANTReader when it receives raw ANT data from a channel that matches the channelNr attribute of the ANTModule. The grey classes are abstract classes and cannot be used directly

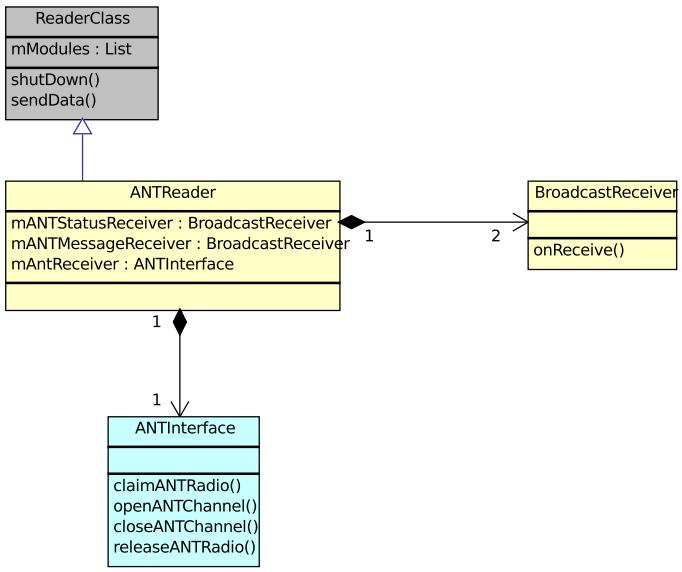

Figure 7: Whenever new ANT data is available the onReceiver method of the ANTMessage BroadcastReceiver is called. This data is then decoded by the appropriate ANTModule. The ANTInterface object serves as an interface between Android and the ANT radio and is supplied by Dynastream systems. The ANTInterface object is able to initialize the ANT radio and open ANT channels.

4.2 BluetoothReader

The BluetoothReader manages several BluetoothModules. Each BluetoothModule is responsible for connecting for a Bluetooth device and subsequently managing the connection. It is possible to use an external library, supplied by the manufacturer of the Bluetooth device, to establish and manage the connection. However a custom implementation can still be implemented if desired.
The following methods of the BluetoothModule are called by the BluetoothReader:

- **initConnection**
- **connect**
- **startStreaming**
- **shutDown**

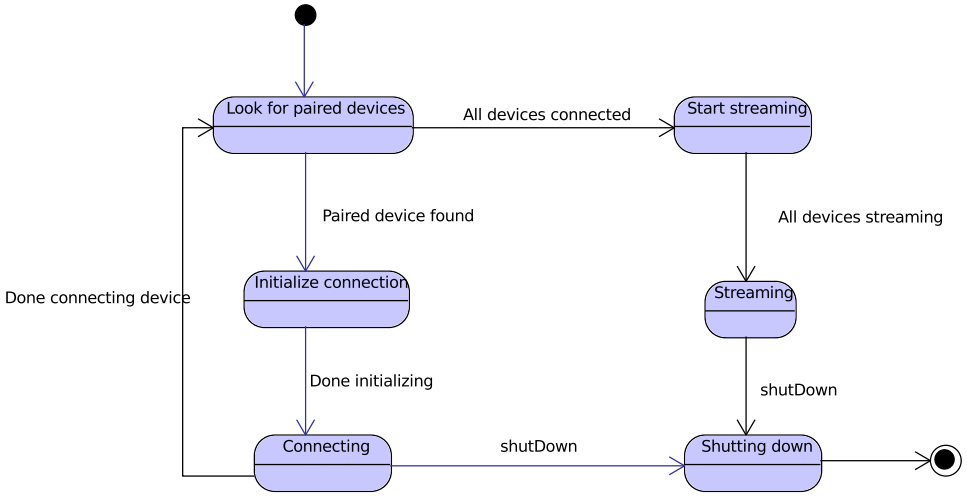

Figure 8: The BluetoothReader will attempt to connect to all the Bluetooth devices represented by the BluetoothModule objects obtained from the configuration file. After all devices are connected the BluetoothReader will tell all the modules to start streaming.

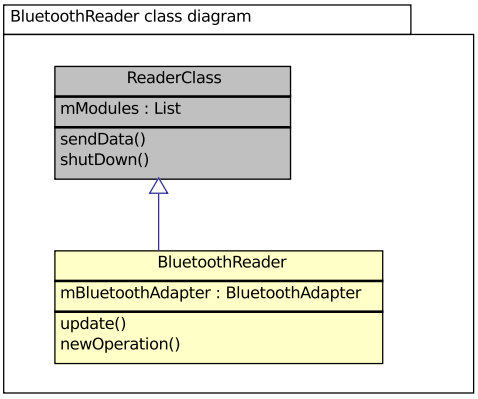

Figure 9: The update function is called whenever a BluetoothModule has new data ready. The BluetoothReader will then use the sendData function to transmit the data to the CRN toolbox.

4.3 Internal communication

4.3.1 Observer pattern

Communication between many of the classes of the CRNTC+ is implemented using the Observer pattern. With the Observer pattern there is one observer who observes one or more observables. When one of the observables has something to report it will notify the observer with the new data.

4.3.2 NotifyObject

The *NotifyObject* is used to transmit messages within the CRNTC+ such as error messages and state changes. The *NotifyObject* consists of a message type field and a data field which can contain an arbitrary object.

4.3.3 DataPacket

The *DataPacket* object is used to transmit data to and from the CRN toolbox. The object is loosely based on the DataPackets used by the CRN toolbox. The *DataPacket* object contains of an id and a values field. The id corresponds to the id of the module or CRN tooblox task that send the data in the values field. The values field is an array of the datatype Double.

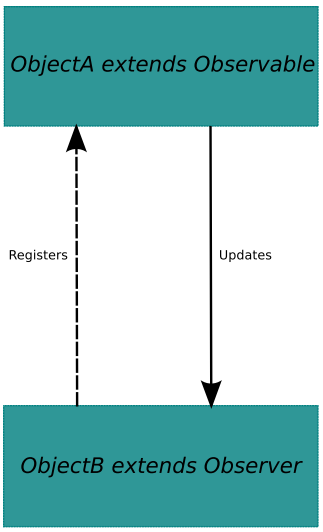

Figure 10: An observer registers one or more observable objects which will in turn update the observer with new information when this is available.

4.4 Dynamic class loading

4.4.1 GSON library

All modules specified in the JSON configuration file are instantiated at runtime. This is done through the GSON library. Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects of which the source-code is not available [9].
To create a working Java instance of a class based on a JSON representation the GSON library needs a class definition. All the class definitions of the supported modules are stored in a *Map* together with a String key which represents the type of the module. When a configuration file is being loaded the type field of the configuration file modules are checked and if they match to a key in the Map the module is instantiated by using the class definition that is coupled to that key.

4.4.2 Reflection

Because the behaviour of the CRNTC+ application can be configured by loading different configuration files, objects need to be instantiated at runtime. The GSON library takes care of this for the modules however the correct ReaderClass, OutputClass and MyTabActivities

should also be instantiated at runtime. To achieve this Java reflection is used. Java's Reflection API's makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

5 Framework analysis

In Section 3.5 several metrics were proposed on which to evaluate the framework. In this section the methods and the results of the evaluations will be discussed. All tests regarding the behaviour of the framework on a smartphone have been performed with a Sony Ericsson Xperia Active since it supports both the ANT protocol and Bluetooth.

Figure 11: Sony Ericsson Xperia Active.

5.1 ANT support

ANT is a practical wireless sensor network protocol running in the 2.4 GHz ISM band. Designed for ultra- low power, ease of use, efficiency and scalability, ANT easily handles peer-to-peer, star, tree and fixed mesh topologies. ANT provides reliable data communications, flexible and adaptive network operation and cross-talk immunity. ANT protocol stack is extremely compact, requiring minimal micro-controller resources and considerably reducing system costs [10].

5.1.1 BodyANT

The BodyANT sensor is a wireless sensor node design which has a miniature outline of 20x10x3mm. It uses the ANT wireless protocol to transmit data. With a single coin cell battery it can last almost 5 days of sending accelerometer data at 32Hz. The BodyANT is designed with off-the-shelf parts which makes it relatively inexpensive [11].

Accelerometer X	Accelerometer Y	Accelerometer Z	Battery temperature	Battery voltage	Counter
10 bits	10 bits	10 bits	8 bits	8 bits	8 bits

Table 2: Sensor readouts of the BodyANT sensor

The CRNTC+ implementation of the BodyANT sensor can read out the following information.

To decode the sensor data the implementation from the CRN toolbox has been used. In order to test the performance of the BodyANT sensor on the CRNTC+ an incrementing number of BodyANT sensors have been connected to a CRNCT+ application running on a Sony Ericsson Xperia Active smartphone running Android version 2.3.4.

BodyANT tests

The first test uses only one 1 BodyANT which will transmit data for about an hour. The number of BodyANT sensors is incremented up till 6. After the tests that data was analysed for energy consumption and data loss. The data loss was calculated based on the counter values (see Table 2) that is transmitted by the BodyANT sensor. The counter is incremented whenever the BodyANT sends a message [11]. The counter starts at 0 and in incremented up until 255 after that the counter is reset to 0 once again. A lost message can be detected in the resulting data by looking for differences greater than 1 in the succesive counter values unless the counter is reset.

Nr of sensors	BodyANT 1 data loss	BodyANT 2 data loss	BodyANT 3 data loss	BodyANT 4 data loss	BodyANT 5 data loss	BodyANT 6 data loss
1	0.03 %	NA	NA	NA	NA	NA
2	0.08 %	0.02 %	NA	NA	NA	NA
3	0.06 %	0.04 %	0.41 %	NA	NA	NA
4	0.02 %	0.03 %	0.17%	0.16 %	NA	NA
5	0.10 %	0.01 %	0.01%	0.01 %	0.31 %	NA
6	0.22 %	0.20 %	0.20%	0.17 %	0.22 %	0.22 %

Table 3: Data loss per BodyANT sensor

Nr of BodyANT sensors	Total average data loss
1	0.03 %
2	0.05 %
3	0.17 %
4	0.095 %
5	0.88 %
6	0.205%

Table 4: Total average data loss

To test the effect of multiple BodyANT sensors on the energy usage of the smartphone, the battery was measured. In Android the battery is measured in levels, where level 100 is fully charged and level 0 indicates battery depletion.

Nr of sensors	Battery level after 1 hour
1	96%
2	95%
3	94%
4	93%
5	92%
6	91%

Table 5: The number of battery levels that have dropped during 1 hour of measurements.

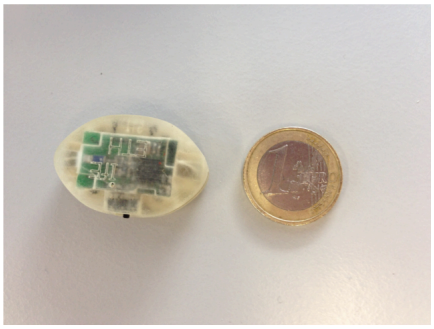

Figure 12: BodyANT wireless sensor node.

5.1.2 ETHOS

The ETHOS is a miniature attitude heading and reference system. It has a unit size of $2.5cm^3$ which makes it ideal for wearable use. Data can be stored on an internal sd-card

and/or transmitted wirelessly using the ANT protocol. With a sampling rate of 128Hz the ETHOS sensor can record for about 10 hours. Like the BodyANT sensor the ETHOS sensor is also designed with off-the-shelf parts [12].

The CRNTC+ implementation of the ETHOS sensor can read out the following information.

Gyroscope X	Gyroscope Y	Gyroscope Z	Accelerometer X	Accelerometer Y	Accelerometer Z
16 bits	16 bits	16 bits	16 bits	16 bits	16 bits

Magnetometer X	Magnetometer Y	Magnetometer Z	Counter
16 bits	16 bits	16 bits	8 bits

Figure 13: ETHOS attitude heading and reference system.

Further tests with the ETHOS where not possible due to the issues with the firmware of the ETHOS sensor which prevented ANT data transmission when not connected to a USB cable.

5.1.3 ANT enabled Heart rate monitors

Several ANT enabled Heart Rate Monitors (HRM) have been implemented such as the Suunto belt and the Adidas MiCoach ANT+ HRM belt. However as of writing the framework was unsuccessful in connecting with both of the devices. The Suunto belt needs a 128 bit network key which was unavailable at the time. The MiCoach HRM uses ANT+ which promotes greater interoperability amongst similar devices from different manufacturers [8]. Like the Suunto belt no successful connection has been established, this might be due to

defective hardware. Since both devices have no external feedback (lights, sound) it has hard to determine whether they are actually working.

5.1.4 Results

Both the BodyANT and the ETHOS sensors are supported by the framework. Several tests have been performed with the BodyANT sensor to test its effect on the battery of the smartphone and the data-loss that occurs when increasing the number of BodyANT sensors. Sadly due to ETHOS firmware problems it wasn't possible to perform similar tests as were done with the BodyANT. However it has been shown that the framework is capable of reading and decoding data from at least one ETHOS sensor.
To meet the requirement for ANT support as defined in Section 3.5 at least the BodyANT and ETHOS should be supported. Both the BodyANT and the ETHOS sensor can be connected to applications created with the framework. This implies that the metric set for ANT support has been met. Nonetheless it would be desirable to do more tests with the ETHOS sensor once its technical issues are solved.

5.2 Extensibility

Extending the framework is essentially described by three use cases:

- **Add sensor component** as described in Section 3.3.1
- **Add output component** as described in Section 3.3.3
- **Add user interface component** as described in Section 3.3.4

The ease of extending the framework will be determined by the number of steps needed that are described by the documentation, the minimum number of lines of code that are necessary in order to make an extension and possible other complexities such as creating icons and changing Android XML files. These minimal number of lines of code allow a module and all of the classes that are needed by this module, to be instantiated. Any other functionality is dependent on the wishes of the developer. See Appendix A for a more detailed description on the specifics of adding components.

Step	Minimum lines of code	Other complexities
1. Extend Module class	3	Further subclassing
2. Extend ReaderClass class	13	Depending on sensor
3. Add Module class definition	1	none
4. Add ReaderClass class definition	1	none

Table 6: Add sensor component

Step	Minimum lines of code	Other complexities
1. Extend OutputModule class	7	none
2. Extend OutputClass class	13	none
3. Add Module class definition	1	none
4. Add OutputClass class definition	1	none

Table 7: Add output component

Step	Minimum lines of code	Other complexities
1. Extend GUIModule class	12	none
2. Extend MyTabAcitivity class	13	retrieve GUIModule
3. Create icons	0	none
4. Create layout xml file	3	Depends on GUI structure
5. Create drawable xml file	6	none

Table 8: Add user interface component

5.2.1 Results

Although the actual complexity of extending the framework is dependent on the functionality the programmer wishes to add, it can be observed from Tables 5.2, 5.2 and 5.2 that adding a new GUI element requires the highest minimal effort. Not only does it require the most lines of code , it is also necessary to create at least one icon image. The reason for the added complexity is the manner in which Android handles the life cycles of Activities. Activities represent a screen on a smartphone with which users can interact. These Activities are the most important part of an Android application and their construction and destruction are handled by the Android framework. This is not the case for sensor and output components.
The number of steps needed to add a new sensor component or output component is 4. To add a sensor component 18 lines of code are needed and to add a new output component 22 lines of code are needed. Adding a new GUI element requires 5 steps and 34 lines of code. Even with the slight increase in complexity of adding GUI elements it can still be concluded that the framework won't get in the way when adding new functionality. Especially when taking in to consideration that the Android Software Development Kit (SDK) is a framework which introduces some necessary steps and lines of code that need to be implemented.

5.3 Energy efficiency

In order to test the usability of the framework a case study was performed based on epileptic seizure detection. During the various aspects of this case study two applications

where created, one for recording seizure data and one for detecting whether a seizure had taken place. For both applications the energy consumption of the smartphone has been monitored. For more information regarding this case study see Section 6.
During the development of the framework several students of the Ubiquitous Computing and Activity Recognition course have used the framework to create a fitness monitoring application. The sensor setup was completely different with respect to the setup used in the previously described case study. Like the case study the energy consumption of the fitness monitoring application has also been monitored and it will be presented in this section along side the energy consumption of the case study.

Scenario	Sensor setup	Duration	Battery depletion
Recording 1	4 Shimmers sending at 100Hz IMU1Shimmer: acc, gyro, mag ECG1Shimmer: acc, ecg ECG2Shimmer: acc, ecg ECG3Shimmer: acc ACTLog annotation	38 minutes	13%
Recording 2	4 Shimmers sending at 100Hz IMU1Shimmer: acc, gyro, mag ECG1Shimmer: acc, ecg ECG2Shimmer: acc, ecg ECG3Shimmer: acc ACTLog annotation	10 minutes	4%
On-line classification 1	1 Shimmer sending at 100Hz ECG1 Shimmer: acc Features: Variance X,Y,Z 200 training samples KNN classification On-screen graph	15 minutes	4%

Table 9: Energy consumption overview for the epilepsy case study. The frequencies denote the rate at which data is transmitted to the smartphone. During both of the recordings the screen of the smartphone was constantly in use for annotation purposes.

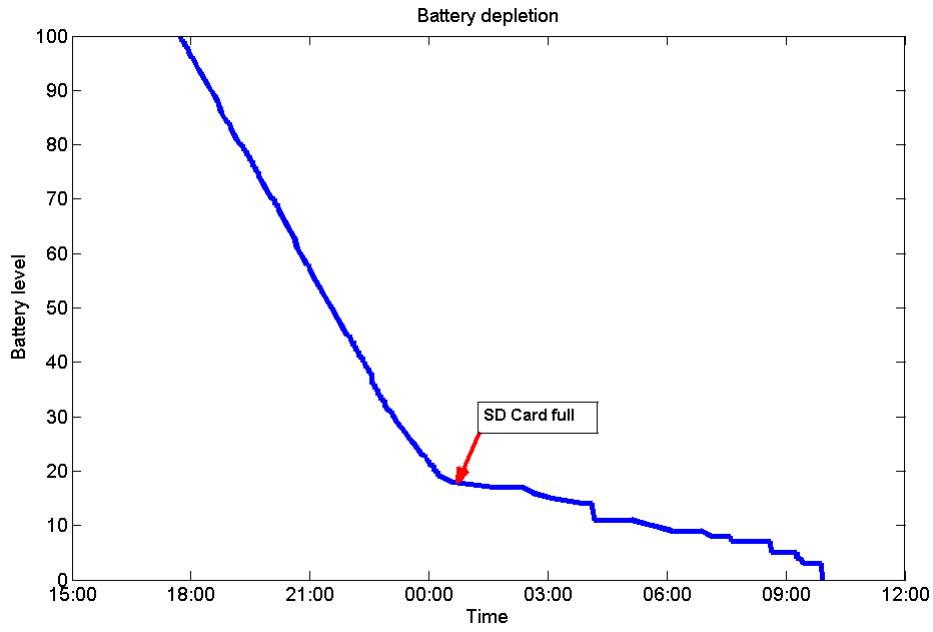

Figure 14: Graph of the depletion of the battery using the same configuration as recording 1 and 2 from Table 9. For this test the smartphone was fully charged and left running over-night. During this the test the screen was not active.

Scenario	Sensor setup	Duration	Battery depletion
Recording 1	1 BodyANT 32Hz, 1 Zephyr HRM 2Hz	10 minutes	1%
Recording 2	1 BodyANT 32Hz, 1 Zephyr HRM 2Hz	10 minutes	1%
Recording 3	1 BodyANT 32Hz, 1 Zephyr HRM 2Hz	11 minutes	1%
Recording 4	1 BodyANT 32Hz, 1 Zephyr HRM 2Hz	10 minutes	1%

Table 10: Energy consumption overview for the fitness monitoring project. The frequencies denote the rate at which data is transmitted to the smartphone. During this test the screen of the smartphone was turned off.

5.3.1 Results

Day-long recordings (about 12 hours) should be the ideal goal. The battery level depletion during recordings in Table 9 are quite significant. The reason for this can be ascribed to three factors:

1. Constant writing to the sd-card, see Figure 14.
2. Decoding of data sent over Bluetooth at 100Hz per sensor.

3. The constant usage of the screen for annotation purposes.

With this setup, usage and hardware it won't be possible to do day-long recordings without recharging the smartphone. However for the epilepsy case-study described in Section 6 there was more than enough power left to do all the necessary recordings.
Table 5.3 shows the results for the fitness monitoring project. For this project a less demanding sensor setup was chosen and the energy depletion levels are far less than those shown in Table 9. With the setup used during the fitness monitoring project day-long measurements should be possible.

5.4 Scalability

To further test the scalability several Shimmer sensors were used. Shimmer is a small wireless sensor platform that can record and transmit physiological and kinematic data in real-time. Designed as a wearable sensor, Shimmer incorporates wireless ECG, EMG, GSR, Accelerometer, Gyro, Mag, GPS, Tilt and Vibration sensors. [13]. The sampling frequency and the type of sensors can be set by adjusting their respective parameters in the configuration file. This makes the Shimmer an ideal platform for scalability tests.
Two metrics were analysed: the increase in energy consumption and the decrease in ability to process data.
During the testing phase two types of Shimmer hardware configurations were available. An ECG and an Inertial Motion Unit (IMU). The ECG unit has several ports which can be used to connect to electrodes. The IMU has a gyroscope and magnetometer. All unit types have an accelerometer. For this test all Shimmer units were configured to use only the accelerometer.

5.4.1 CPU Usage

The effect of incrementally adding sensors on the CPU was measured by using the *traceview* tool from the Android SDK. With this tool the threads running on the CPU can be visualized to see how much CPU time each thread takes.

Test nr	Nr of Shimmers	Shimmer configuration
1	1	Shimmer1: Accelerometer, 200Hz, calibrated data
2	2	Shimmer1: Accelerometer, 200Hz, calibrated data Shimmer2: Accelerometer, 200Hz, calibrated data
3	3	Shimmer1: Accelerometer, 200Hz, calibrated data Shimmer2: Accelerometer, 200Hz, calibrated data Shimmer3: Accelerometer, 200Hz, calibrated data
4	4	Shimmer1: Accelerometer, 200Hz, calibrated data Shimmer2: Accelerometer, 200Hz, calibrated data Shimmer3: Accelerometer, 200Hz, calibrated data Shimmer4: Accelerometer, 100Hz, calibrated data
5	4	Shimmer1: Accelerometer, 200Hz, calibrated data Shimmer2: Accelerometer, 200Hz, calibrated data Shimmer3: Accelerometer, 200Hz, calibrated data Shimmer4: Accelerometer, 200Hz, calibrated data

Table 11: Configurations for the scalability tests

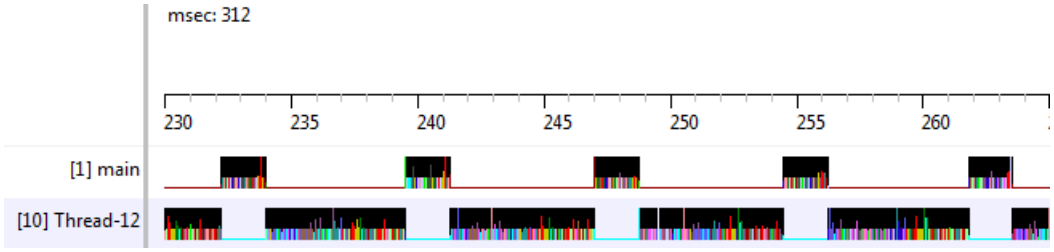

Figure 15: Trace view of the scalability test 1. [1] main shows the main thread. One of the main tasks of this thread is updating the GUI. [10] Thread-12 is the thread that is handling the Shimmer Bluetooth connection.

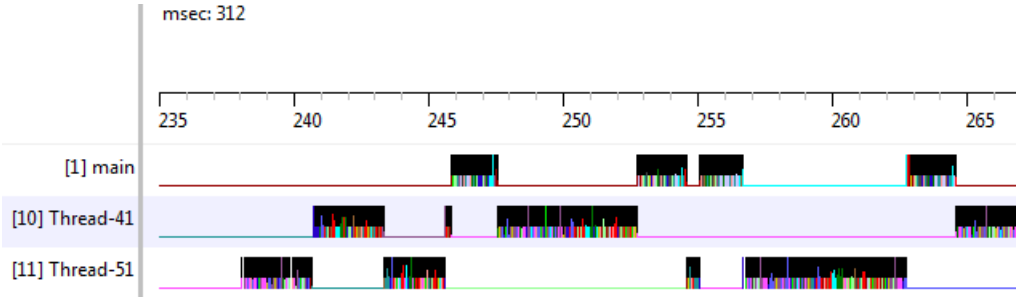

Figure 16: Trace view of the scalability test 2. [10] Thread-41 and [11] Thread-51 show the Shimmer threads that are handling the Bluetooth connections.

Figure 17: Trace view of the scalability test 3. [11] Thread-68, [12] Thread-78 and [13] Thread-88 show the Shimmer threads that are handling the Bluetooth connections.

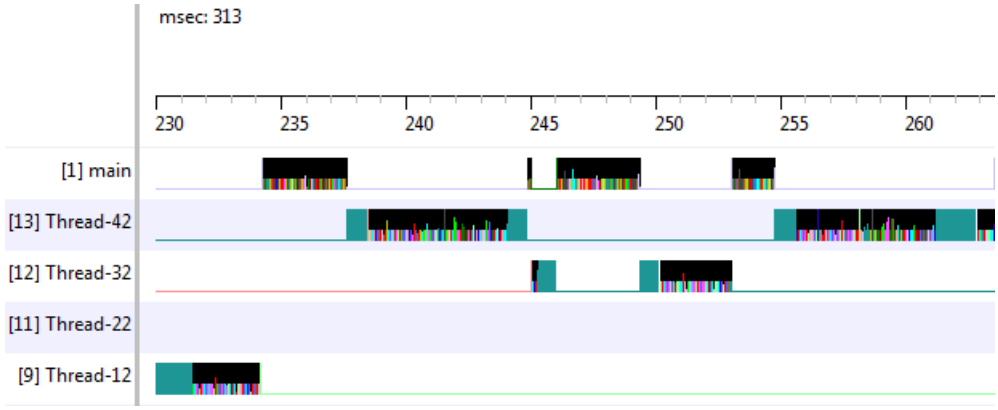

Figure 18: Trace view of the scalability test 4. [13] Thread-42, [12] Thread-32, [11] Thread-22 and [9] Thread-12 show the Shimmer threads that are handling the Bluetooth connections.

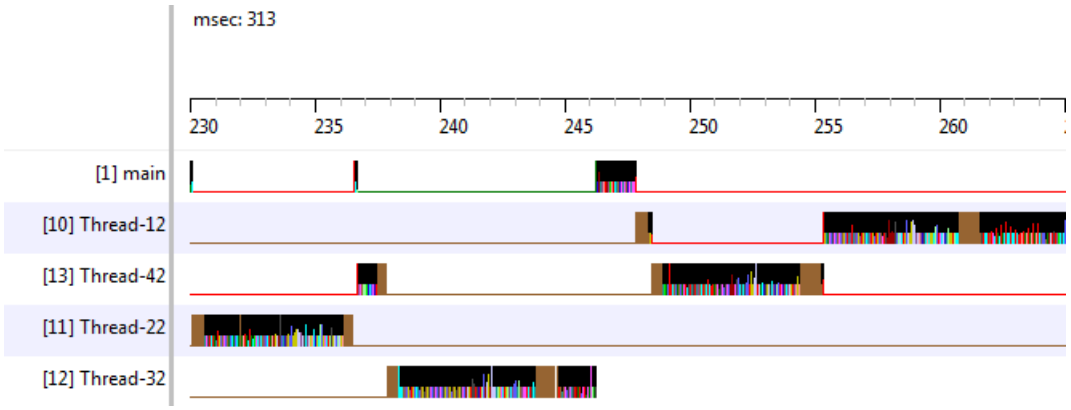

Figure 19: Trace view of the scalability test 5. [10] Thread-12, [13] Thread-42, [11] Thread-22 and [12] Thread-32 show the Shimmer threads that are handling the Bluetooth connections.

During test 4 and 5 (Figure 18 and 19) the responsiveness of the GUI was significantly reduced. During test 5 the GUI was almost completely not responding. A decrease in GUI responsiveness can also be seen from the trace views. The CPU time for the main thread decreases as more sensors are added.

5.4.2 Energy consumption

During the scalability test the energy usage of the smartphone was also measured.

Test nr	Battery level after 1 hour
1	93%
2	90%
3	84%
4	84%
5	98%

Table 12: The battery level of the smartphone after one hour of measurements with an increasing number of Shimmer units. Tests 4 and 5 have conflicting results compared with tests 1,2 and 3.

Table 12 shows that increasing the number of sensors decreases battery performance except for the final two tests. In Section 5.4.1 a significant drop in GUI responsiveness was observed. This could indicate that the main thread, which is responsible for updating the GUI, amongst

5.4.3 Results

As was expected the energy usage increases as more Shimmer sensors are added to the setup. When the total number of incoming sensor transmissions is greater than 600 per second the GUI starts to show noticeable lag, to the point that the battery readouts are no longer accurate due to the main thread not receiving enough cpu time. This limit of 600Hz should be taken into account when creating applications that use one or more Shimmer sensors.

5.5 Documentation

See Appendix A for the documentation of the project. Assessing the quality of documentation proved to be difficult. Literature search did not find any satisfying results to measure the quality of software documentation.

6 Epileptic seizure detection

To test the CRNTC+ it needs to be evaluated in a case study. The main goals of the case study are the following:

- Gathering data.
  Is the framework able to accurately record and annotate data in a real world situation?
- Analysis of the data.
  Can the data acuired in the first stage be analysed by existing tools such as MATLAB and the ACTLog toolbox?
- Implementation of seizure monitoring application.
  Is it possible to create an application with the framework that can accurately detect events on-line?

For this case study epileptic seizure detection was chosen. Epilepsy is a chronic illness with a large negative impact on the quality of life. Due to the high variance of patients and their seizures, it is difficult to evaluate their medication and overall progress after leaving a controlled hospital environment. Monitoring the seizure rate outside the hospital is a key information to aid medication and to evaluate treatment progress. In this case study the focus lies on epileptic seizure detection, not restricted to lab environment. Similar work has been done before, but suffers from high rate of misclassification.

6.1 Previous work

In [14] Lockman et al use a single accelerometer sensor device which is attached to either a wrist or ankle. The patients that were used experienced eight seizures in total during testing. Seven seizures were detected, however the number of false positives was relatively high. The work only aimed at detecting tonic-clonic seizures.
In the work of Nijsen [15] five 3D accelerometers to detect several types of epileptic motor seizures are used. Two on each wrists, two on each ankles and one on the sternum. All patient measurements were done at night when the patients are sleeping. Even though this should minimize other motions that can be confused as seizures that Positive Predictive Value (PPV) was still below the goal of 50% (one in two detections of a seizure is correct).
Hildeman [16] achieves good results by using three 3D accelerometer sensors, two on the wrists and one on the chest. Next to that a unsupervised learning algorithm is used. These good results are only achieved for tonic-clonic seizures, even with lots of background activity. Strictly tonic seizures are hard to distinguish from every day activities.

6.2 Data acquisition

6.2.1 Sensor setup

The following setup was used to gather the data:

- Sony Ericsson Xperia active smartphone
- One Inertial Motion Shimmer sensor attached to the wrist of the dominant arm.
- One ECG Shimmer sensor placed on the upper left arm. Electrodes placed on the chest of the subject are connect to the ECG modules through wires running along the shoulder. The device also detects motion of the upper left arm.
- One ECG Shimmer sensor placed on the abdomen. This device is connected to a respiration band placed around the chest. The device also detects motion of the abdomen area.
- One ECG Shimmer sensor placed on the upper right arm. This device detects motion of the upper right arm.
- One inertial motion ETHOS sensor attached to the wrist of the non-dominant arm.

All Shimmer sensors used a sampling frequency of 100 Hz. The ETHOS sensor used a sampling frequency of 128 Hz.

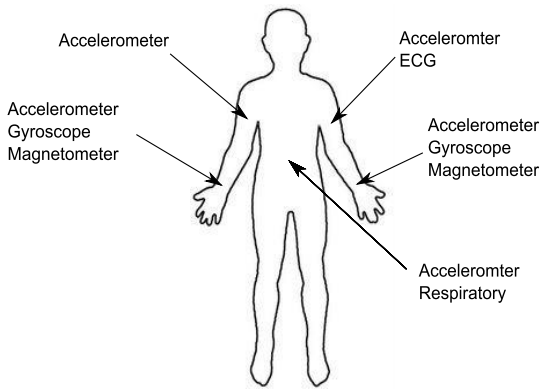

Figure 20: Schematic of the sensor placement.

6.2.2 Activity script

In order to approach a real world situation an activity script was written which contains several everyday activities intertwined with several seizures.

Waking up

- Lying on a bed (2 minutes)
- Getting dressed (putting on 2 socks and 2 shoes with laces while sitting, putting on a vest while standing) (2 minutes)
- Stand up (1 minutes)
- Walking
- Scratching (5 times)
- Walking
- Drinking from a glass
- Cutting food with knife (5 times)
- Walking
- Brush teeth (2 minutes)
- Seizure (duration determined by actor)
- Rest (5 minutes)

Morning gymnastics

- Push-ups (determined by the actor)
- Rest (5 minutes)

Walking to work

- Walking (5 minutes)
- Shaking hands (5 times)
- Using a computer mouse (determined by the actor)
- Typing on a keyboard (determined by the actor)
- Using a computer mouse (determined by the actor)
- Typing on a keyboard (determined by the actor)
- Seizure (duration determined by actor)
- Rest (5 minutes)

End of the day

- Walking (5 minutes)
- Unloading and putting away groceries (taking a basket filled with groceries: 2 cans of beans, 2 cartons of milk, 2 bottles of water) (2 minutes)
- Folding 8 towels (2 minutes)
- Vacuum cleaning (5 minutes)
- Seizure (duration determined by actor)
- Rest (5 minutes)
- Cutting food (determined by the actor)
- Doing de dishes (no water) (5 minutes)
- Watching television (2 minutes)
- Seizure (duration determined by actor)

6.2.3 Data recording

Annotations were performed by using the ACTLog annotation GUI which is part of CRNTC+.
Because of practical limitations of the recording area not all activities of the script could be performed and all seizures had to be performed while either lying on a bed or sitting in a chair. In total 10 different activities were performed and 5 different seizures simulated.

Participant	Activities performed	Seizures	Total Duration
1	Lying in bed Getting dressed Scratching Drinking from a glass Brushing teeth Sit-ups Shaking hands (2x) Using mouse Using keyboard Folding towels	Myoclonic Tonic (2x) Tonic-clonic (3x) Clonic (2x)	38 minutes
2		Myoclonic Tonic-clonic (3x)	10 minutes

Table 13: Participant overview

6.3 Data analysis

The first step in analysing the data is integrating the data in the ACTLab toolbox for MATLAB. The ACTLab toolbox offers functionality to analyse the data within MATLAB, for instance: annotation, feature extraction, classification etc.

6.3.1 Annotation

Although the data is already annotated during the recordings it is possible that mistakes were made. To correct possible mistakes or to add annotations for greater accuracy the Marker tool of the ACTLab toolbox can be used. Marker is able to detect the annotations already gathered during the recording stage. In this way possible annotation errors can be adjusted with minimal effort.

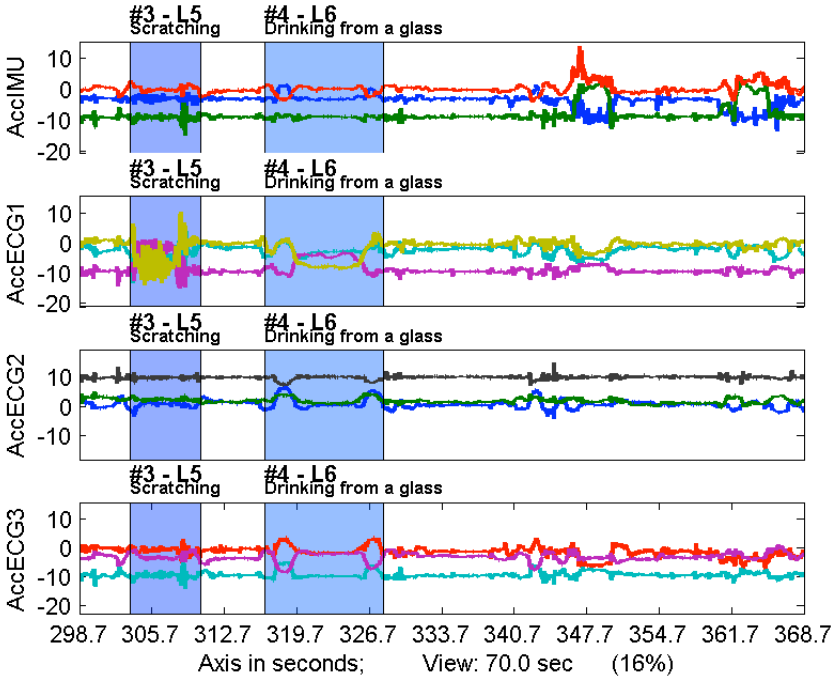

Figure 21: Two activities annotated in Marker. The two activities shown are scratching and drinking from a glass.

6.3.2 Feature selection

In order to perform the classification several features should be selected. Since actors were used instead of actual epilepsy patients their heart-rates would only increase during the

seizure. With real patients the ECG data could be of great significance [17]. The main goal would be to classify an event as either a seizure or a non-seizure using motion data, mean and variance of raw data were chosen as starting features.

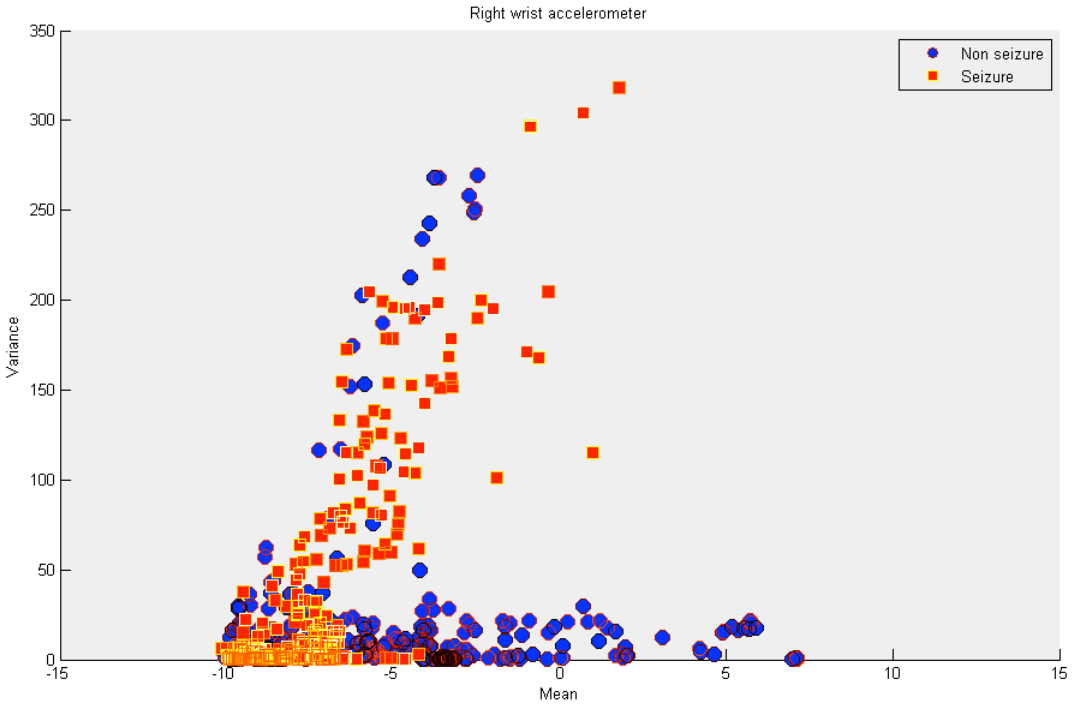

Figure 22: Features (mean and variance extracted from the data of the right wrist of the participant.

In Figure 22 the blue dots denote the non-seizures and the red dots the seizures. The data were recorded from the right wrist of the participant. Although the seizures often have a high variance several non-seizures also show this behaviour.

Figure 23 shows the mean variance plot of the upper left arm of the participant. In this plot the distinction between seizure and non-seizure is more apparent. The non-seizure points rarely have a variance higher than 50, on the other hand the seizures display a high variance just like in Figure 22. Although we can see from the Figure there still is some overlap. A reason for this difference could be that at the wrist much higher acceleration is measured during everyday activities than the upper arm. However during a seizure all four sensors registered high acceleration values, regarding of their placement. Another observation from Figure 23 is that the mean feature makes little difference in classifying between seizures and non-seizures.

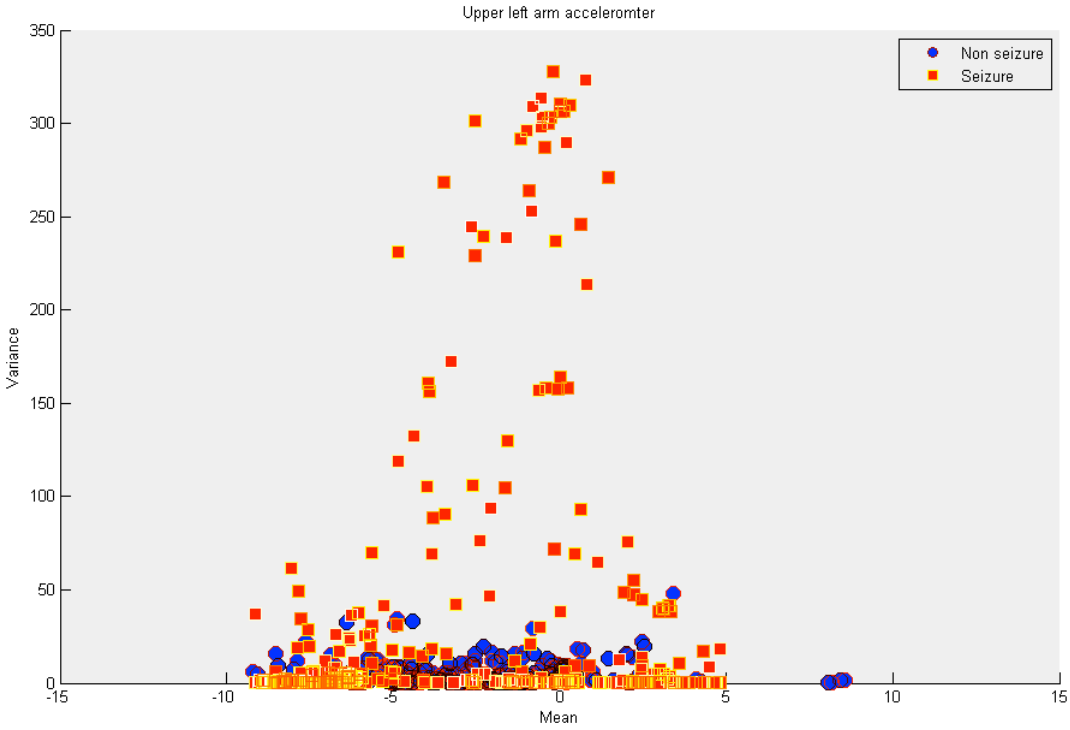

Figure 23: Mean variance plot showing accelerometer data of the upper left arm of the participant.

Classification tests

Based on above findings a set-up was chosen with two accelerometers on the upper arms and on the wrist of the dominant arm. The variance of the all the axis of the accelerometer sensors are used for classification. From the data of participant 1, 500 random samples per class (1000 in total) are chosen for training. The classification is tested against all the labelled data, both seizure and non seizure. In total 75332 samples.
All the tests in this paragraph were performed on a 2GHz Intel Core 2 Duo mac mini.

Lower right arm accelerometer Upper left arm accelerometer Upper right arm accelerometer	Variance on all axis	500 random samples per class	75332 samples from all the data of participant 1

Table 14: Overview of the first classification test

	Non-seizure	Seizure
Non-seizure	87%	13%
Seizure	25%	75%

Table 15: Confusion matrix of the first classification test. Using three accelerometer sensors and testing against labelled data.

This initial results (Table 15)from the first test show good results but the data used for testing only contains samples either labelled as seizure or non-seizure. Since the end goal is on-line classification a better test would be to use all the data for testing, including data which is not labelled.

Sensors	Features	Training data	Testing data
Lower right arm accelerometer Upper left arm accelerometer Upper right arm accelerometer	Variance on all axis	500 random samples per class	226791 samples from all the data of participant 1

Table 16: Overview of the second classification test. Using three accelerometer sensors and testing against all the data acquired during the recordings

	Non-seizure	Seizure
Non-seizure	64%	36%
Seizure	26%	74%

Table 17: Confusion matrix of the second classification test.

From Table 17 a degradation in performance can be observed. Especially the number of non-seizures classified as seizures has increased. For the ground truth of the unlabelled data the non-seizure class is used since the participant did not perform any seizures that have not been labelled. So the number of samples belonging to the non-seizure class is greatly increased. Which is explains why the increase in the non-seizure misclassifications. The next step is to look at the performance of a single sensor. During the feature selection the upper left arm accelerometer showed some good results.

Sensors	Features	Training data	Testing data
Upper left arm accelerometer	Variance on all axis	500 random samples per class	226791 samples from all the data of participant 1

Table 18: Overview of the classification test using only the upper left arm

	Non-seizure	Seizure
Non-seizure	59%	41%
Seizure	28%	72%

Table 19: Confusion matrix for the classification using only the upper left arm accelerometer

The results in Table 19 show an even greater decrease in performance for the non-seizure classifications.
For the next test the upper right arm accelerometer is used.

Sensors	Features	Training data	Testing data
Upper right arm accelerometer	Variance on all axis	500 random samples per class	226791 samples from all the data of participant 1

Table 20: Overview of the classification test using only the upper right arm

	Non-seizure	Seizure
Non-seizure	63%	37%
Seizure	24%	76%

Table 21: Confusion matrix for the classification using only the upper right arm accelerometer

The upper right arm accelerometer shows a slight increase in performance compared to the upper left arm. To continue on this improvement the the lower right arm was used for the next test.

Sensors	Features	Training data	Testing data
Lower right arm accelerometer	Variance on all axis	500 random samples per class	226791 samples from all the data of participant 1

Table 22: Overview of the classification test using only the lower right arm

	Non-seizure	Seizure
Non-seizure	62%	38%
Seizure	19%	81%

Table 23: Confusion matrix for the classification using only the lower right arm accelerometer

The lower right arm has the highest performance of all the single sensors when it comes to Detecting seizures. Also the number of misclassifications is comparable to the results of all the sensors in Table 16.
To improve the results of the lower right arm the gyroscope data is used. Since this data was also recorded and could further improve the classification.

Sensors	Features	Training data	Testing data
Lower right arm accelerometer and gyroscope	Variance on all axis	500 random samples per class	226791 samples from all the data of participant 1

Table 24: Overview of the classification test using only the lower right arm accelerometer and the gyroscope

	Non-seizure	Seizure
Non-seizure	65%	35%
Seizure	21%	79%

Table 25: Confusion matrix for the classification using only the lower right arm accelerometer and gyroscope

The results did not improve with respect to the result of only using the accelerometer of the lower right arm presented in Table 23.
The most likely set-up for an on-line detection would be an accelerometer placed on the lower right arm. This contradicts the initial findings that suggested the upper arm accelerometer should give better results since they experience lower variance levels for the non-seizure class.

On-line classification tests

In order to test the feasibility of on-line detection on a smartphone the same setup as in Table 24 was used on a smartphone. However due to the lack of processing power on the smartphone compared with the desktop computer used in the previous paragraph, many data samples were lost due to the internal CRNT buffers overflowing. To counter this the number of samples used per class was reduced.

Sensors	Features	Training data	Testing data
Lower right arm accelerometer	Variance on all axis	100 random samples per class	226791 samples from all the data of participant 1

Table 26: Overview of the classification test running on a smartphone with a reduced training set.

	Non-seizure	Seizure
Non-seizure	55%	45%
Seizure	22%	78%

Table 27: Confusion matrix for the classification using only the lower right arm running on the smartphone with a reduced number of training samples

The number of non-seizure events classified as seizures has increased significantly in Table 27. So the number of samples used in the training set has a considerable effect on the classification result.

Figure 24 shows a tonic-clonic seizure annotated in the data. Although high acceleration values have been measured in the middle section the first and last sections also show values where there is almost no motion. By removing these parts from the annotation and using

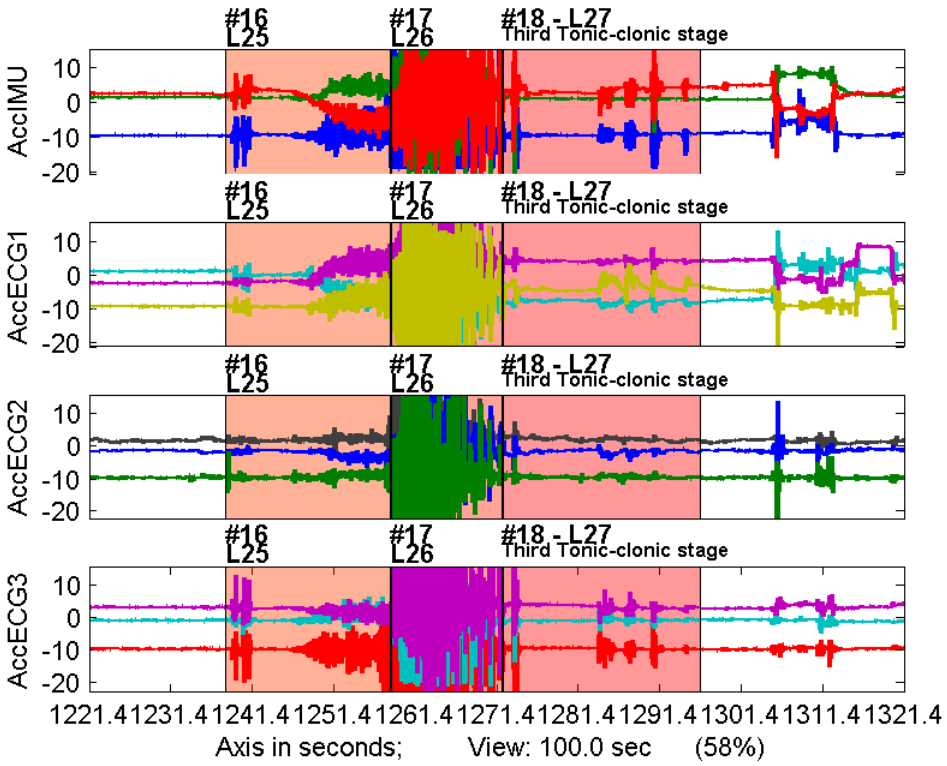

Figure 24: Tonic-clonic seizure annotation. The middle section shows a large amount of acceleration across all sensors. The first and last stage also show areas where there is almost no acceleration.

the resulting data for training the performance should increase because there should be less overlap in the low variance areas of the seizure and non-seizure classes.

Sensors	Features	Training data	Testing data
Lower right arm accelerometer	Variance on all axis	100 random samples per class. Improved annotation	226791 samples from all the data of participant 1

Table 28: Overview of the classification test running on a smartphone with a reduced training set.

	Non-seizure	Seizure
Non-seizure	78%	22%
Seizure	14%	86%

Table 29:

Table 29 shows that even with a relatively low number of samples good classification results can be achieved by using an annotation that omits uninteresting values in the signal.

6.4 Results

- **Is the framework able to accurately record and annotate data in a real world situation?**
  Using the framework an application was created using multiple sensors that was able to record the data during two sessions which lasted 48 minutes in total. The annotation of during the recordings was performed on the device and added to the data.
- **Can the data acuired in the first stage be analysed by existing tools such as MATLAB and the ACTLog toolbox?**
  The ACTLog toolbox was able to read the annotations and display them in the MATLAB annotation tool. This saves up a lot of work since it is no longer needed to manually find the events that need to be annotated with external annotation methods such as video and written down start and stop times.
- **Is it possible to create an application with the framework that can accurately detect events on-line?**
  Although a sensor set-up has been found that performs well with a large amount of non-labelled data. This has not been tested on the smartphone yet. Since the smartphone is limited with respect to computation compared to desktop workstation it is to be expected that set-up needs to be changed in order for on-line detection on a smartphone to be feasible.

7 Conclusion

The goal of this project was to realise a configurable architecture that supports different sensor input, processing, and output components, implemented on Android. And the developed architecture should be analysed by a epileptic seizure detection case-study. To achieve these goals a modular design approach was chosen as can be seen in Section 3. During the implementation of the framework a number of different sensors, using different protocols, have been added to the framework:

- BodyANT
- ETHOS
- Shimmer

Next to sensors several other modules have also been implemented:

- AndroidPlot to display information on the screen in a graph
- ACTLog annotation tool.

In Section 5 several tests were performed that show that the framework can easily be extended and scales well when more sensors are added. With the modules already implemented and the information presented in Section 5 it should already be possible to create applications for recording and analysis purposes.
The epileptic seizure case-study showed that the framework could be used to create an application that could be used for recording and simultaneously annotate the data. The acquired data was analysed and a model for an on-line detection set-up was derived.

References

[1] David Bannach, Paul Lukowicz, and Oliver Amft. Rapid prototyping of activity recognition applications. *Pervasive Computing*, 7(2):22–31, April-June 2008.

[2] Jakob Weigert. The context recognition network toolbox on android, April 2011.

[3] Funf media website. http://funf.media.mit.edu/about.html.

[4] Nadav Aharony, Wei Pan, Cory Ip, Inas Khayal, and Alex Pentland. Social fmri: Investigating and shaping social mechanisms in the real world. *Pervasive and Mobile Computing 7 (2011)*, 7:643–659, 2011.

[5] Osas website. http://www.win.tue.nl/san/wsp/.

[6] Vitruvius project website. http://vitruvius-project.com/.

[7] Sense website. http://www.sense-os.nl/.

[8] Ant website. thisisant.com.

[9] Gson library website. https://code.google.com/p/google-gson/.

[10] Dynastream Inovations, 228 River Avenue, Cochrane, Alberta, Canada T4C 2C1. *ANT Message Protocol and Usage*, rev 4.2 edition, 2010.

[11] Gerhard Trster Martin Kusserow, Oliver Amft. Bodyant: Miniature wireless sensors for naturalistic monitoring of daily activity. In *BodyNets 2009*, 2009.

[12] Holger Harms, Oliver Amft, Rene Winkler, Johannes Schumm, Martin Kusserow, and Gerhard Troester. Ethos: Miniature orientation sensor for wearable human motion analysis. In *IEEE SENSORS 2010 Conference*, 2010. .

[13] Shimmer-research website. http://www.shimmer-research.com.

[14] Juliana Lockman, Robert S. Fisher, and Donald M. Olson. Detection of seizure-like movements using a wrist accelerometer. *Epilepsy and Behaviour*, 20:638–641, 2011.

[15] Tamara Nijsen. *Accelerometry based detection of epileptic seizures*. PhD thesis, Eindhoven University of Technology, 2008.

[16] Anvers Hildeman. Classification of epileptic seizures using accelerometers. Master's thesis, Chalmers University of Technology, 2011.

[17] Jean Gotman Maeike Zijlmans, Danny Flanagan. Heart rate changes and ecg abnormalities during epileptic seizures: Prevalence and definition of an objective clinical sign. *Epilepsia*, 43:847–854, 2002.

A CRNTC+ Manual

Technische Universiteit
Eindhoven
University of Technology

CRNTC+ Manual

by Frank Roberts

Date: November 30, 2012

Contents

1 Introduction

This document serves as a manual for the CRNTC+ Android application which was developed as part of a master thesis. The CRNTC+ Android application provides means to quickly develop a prototyping application for sensing systems.
The CRNTC+ application is an extended version of the CRN Toolbox Center application developed by the University of Passau. [1] The CRN Toolbox Center application was a first attempt of using the Context Recognition Network (CRN) toolbox [2] on an Android smartphone. The CRN toolbox allows rapid prototyping of sensing applications. It was originally developed for Linux type systems but has since been ported to many other operating systems. [2]
This manual will document how to use the application but also how to extend it with for instance new sensor types, user interaction screens etc.

2 System requirements

- OS: Android 2.2 or greater
- ANT support: The following smartphones offer ANT communication support(**Warning! Not all listed smartphones support Android 2.2 or higher!**):
  - HTC Rhyme
  - Sony Xperia arco S
  - Sony Xperia S
  - Sony Xperia ion
  - Sony Live with Walkman
  - Sony active
  - Sony Xperia arc
  - Sony Xperia arc S
  - Sony Xperia mini
  - Sony Xperia mini pro
  - Sony Xperia neo
  - Sony neo V
  - Sony Xperia pro
  - Sony Xperia ray
  - Sony Xperia X8
  - Sony Xperia X10 mini
  - Sony Xperia X10 mini pro
- Software (Optional):If you wish to expand the CRNTC+ applicatio you will need Eclipse helios with the Android SDK installed. Possibly the Android NDK version 5b or greater is also needed if the CRN library needs to be rebuilt.

3 CRNTC+

3.1 Configuration files

The behaviour of the CRNTC+ application can be altered by using a configuration file. Such a file is written in JSON and can describe which sensors are used, if annotation is used, the use of some output of the sensor data, how data should be stored etc. A typical configuration file consists of several 'modules' that should be used by the application and a section containing a separate configuration for the CRN toolbox. This is because, as was mentioned in Section 1, that CRNTC+ uses the CRN toolbox which also is configurable with configuration files.
The CRN toolbox also uses several modules (strictly speaking these are called 'tasks' within CRN documentation) which can be connected in various ways with each other. This way a network of modules can be constructed to gather, analyse and output data from various sensors.

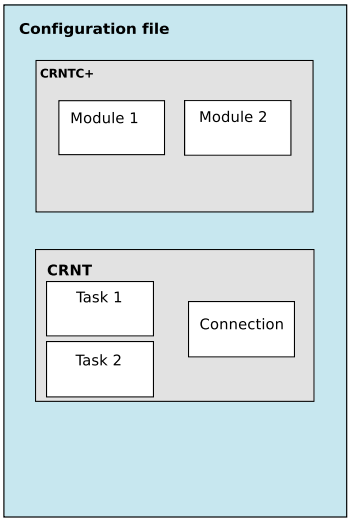

Figure 1: The general model of a configuration file. The CRNTC+ part contains several modules which can represent: sensors, outputs, GUI's etc. The CRNT part is just a standard CRNT configuration file and contain all the usual tasks and their connections.

3.1.1 Modules

The behaviour of the CRNTC+ application is defined by modules. The modules contain certain information on how a component should behave. For instance a sensor module could specify the sampling rate and can decode raw sensor data. An output module that writes data to a file could specify the name of the file and some other text formatting specifics. Modules usually contain very little actual functionality besides perhaps the ability to decode or encode raw data.

- Sensor modules: contain information about how the sensor should connect to the hardware and other information such as reading rates, number of channels etc.
- GUI modules: contain information that needs to be displayed on screen for a user.
- Output modules: contain information on how to connect and process information from the CRN toolbox

Every module has to have at least the following two items:

- Type: the type is used to add the module in the right place within the application.
- ID: the ID should be unique and is used to connect the module with a corresponding module in the CRN toolbox.

3.1.2 CRN toolbox configuration

The CRN toolbox uses a configuration file to specify it's behaviour. A fully CRN toolbox compatible configuration file can be specified in the CRNTC+ configuration file. In this context the CRN toolbox functions as an advanced processing module within the CRNTC+ application. Next to processing, the toolbox also has several ways of outputting data by writing it to the sdcard directly or by sending data back to the CRNTC+ application.
In Listing 1 the CRN toolbox configuration file is specified in the *"CRNT"* section. In order to communicate with the CRN toolbox so called DirectInputReaders and DirectOutputWriter tasks of the CRN toolbox are used. These tasks will be explained in the following sections.

4 Using the application

4.1 Writing the configuration file

As was mentioned in Section 3.1 a configuration file consists of two parts. One part details all the modules used by CRNTC+ these are usually the sensors and GUI components. The other part describes how the CRN toolbox should operate with the data generated by the sensors and or GUI components.
All modules should have at least a type and a unique id specified. Make sure to use the .json extension otherwise the CRNTC+ application will not be able to read the file.

```
{
  "CRNT": {
    "tasks": [
    {
      "type": "DirectInputReader",
      "id": "ShimmerInput"
    },
    {
      "type": "LoggerTask",
      "id": "output",
      "encoder": {"type": "TimestampedLinesEncoder"},
        "name": "sdcard/Shimmer9EFCOutput"

    }
    ],
    "connections": [
    {
      "type": "Connection",
      "sourceTask": "ShimmerInput",
      "sourcePort": 0,
      "destTask": "output",
      "destPort": 0
    }
    ]
  },
  "CRNT+": {
    "modules": [
    {
      "id": "ShimmerInput",
      "type": "Shimmer",
      "address": "00:06:66:46:9E:FC",
      "accelerometer": true,
      "gyroscope": false,
      "magnetometer": false,
      "ecg": true,
      "egm": false,
      "heart": false,
      "gsr": false,
      "strain": false,
      "samplingrate": 250

    }
    ]
  }
}
```

Listing 1: Configuration file example

4.2 Create data-gathering application

The following steps explain how to create a simple data gathering application that stores log files on the smartphone's sdcard.

1. **Add sensor components.**
   In order to create a simple data-gathering application one or more sensor needs to be added to the configuration. A sensor can be added in the CRNTC+ section of the configuration file. When using a certain sensor type make sure to correctly fill in the type name in the type field of the module in the configuration file. Also make sure that the ID value is unique.

2. **Add annotation (Optional)**
   To let users annotate data the ACTLog module can be used. ACTLog was initially developed as an independent application but has been integrated into CRNTC+. Like CRNTC+ ACTLog also uses a configuration file to determine how it should operate. An ACTLog module in CRNTC+ works almost exactly the same as the configuration file for the standalone application. The only exception being that activities in the module should have a unique number. For more information regarding ACTLog see it's manual. [3]

3. **Add an AndroidPlot module (Optional)**
   Sometimes it can be useful to visualize the sensor data so that it is possible to check whether the sensors produce sensible data. To visualize data the AndroidPlot library is used. [4] Make sure that the number of channels set in the configuration file matches the channels produced by the DirectOutputWriter task that is connected to the AndroidPlot module being used.

4. **Set up CRNT**

   - DirectInputReader
     The connection between the modules in the CRNTC+ part and the CRN toolbox is achieved by specifying one or more DirectInputReader modules in the CRN toolbox configuration. The number of readers depends on the number of sensor modules selected in the previous step. Make sure that the ID name of the modules is the same as the ID names of the DirectInputReaders.

   - LoggerTask
     To write data to the sdcard a LoggerTask needs to be added to the CRN toolbox configuration. Next to the type and ID two other things need to be specified. First is the type of encoding. The second is the name of the log file. Make sure to add *sdcard/* before the file name since the sdcard is the only place the user of smartphone can write data to.
     It is possible to create multiple LoggerTask modules if multiple sensors where created in the previous step.

- Connections The final step of creating the configuration file is adding the correct connections between the DirectInputReader modules and the LoggerTask modules.

5. **Starting the application** Open the configuration file tab and tap the button to open a configuration file. Navigate to where the configuration file is stored on the sdcard and open it. Press the start button to run the selected configuration file.

4.3 Create data-processing application.

On of the most important functionalities of the CRNTC+ application is the ability to analyse and process data from sensors connected to the smartphone.

1. **Add sensor components.**
   In order to create a simple data-processing application one or more sensor needs to be added to the configuration. A sensor can be added in the CRNTC+ section of the configuration file. When using a certain sensor type make sure to correctly fill in the type name in the type field of the module in the configuration file. Also make sure that the ID value is unique.

2. **Set up CRNT.**
   To process data the CRNT needs to be configured in such a way that it accepts data from the sensors selected in the previous step, process this data and optionally output the analysed data for further processing. [5]

3. **Add an AndroidPlot module (Optional)**
   Sometimes it can be useful to visualize the sensor data so that it is possible to check whether the sensors produce sensible data. To visualize data the AndroidPlot library is used. [4] Make sure that the number of channels set in the configuration file matches the channels produced by the DirectOutputWriter task that is connected to the AndroidPlot module being used.

5 CRNTC+ Design

The application is designed as a Model View Controller (MVC) pattern. The CRNTC+ application was based on the CRN Toolbox Center which was an Android application designed to use the CRN toolbox. [1] The CRNTC+ design is much more extensive and extendible however. This section will explain some of the design concepts of CRNTC+.

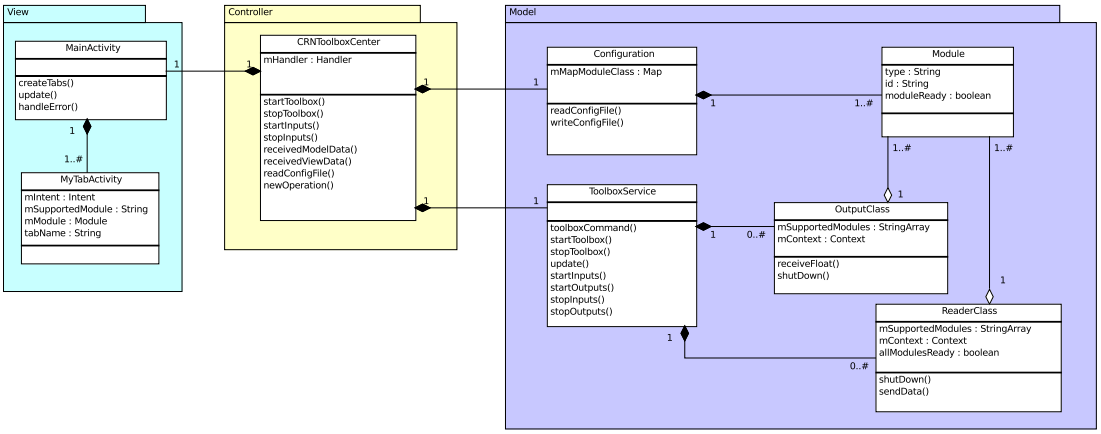

Figure 2: Class diagram of the CRNTC+

5.1 CRNToolboxCenter

This is the central class in the design and facilitates the communication between the GUI elements, the CRN library and the other modules. It is implemented as a singleton class meaning that during the lifetime of the application there is only one instance of this class. Within the MVC design pattern this class together with the Toolbox Service class can be regarded as the controller.

5.2 ToolboxService

The Toolbox Service class communicates directly with the CRN toolbox. This class can communicate with the CRN toolbox directly. Another responsibility of this class is creating and maintaining all the Reader and Output classes.

5.3 Modules

The behaviour of the CRNTC+ application is defined by modules. The modules contain certain information on how a component should behave. For instance a sensor module could specify the sampling rate and can decode raw sensor data. An output module that writes data to a file could specify the name of the file and some other text formatting specifics. Modules usually contain very little actual functionality besides perhaps the ability to decode or encode raw data. Although it is possible to create modules that do offer a substantial amount of functionality. The Bluetooth reader described in Section 7.2 uses modules that offer more complex functionality.
All modules extend the *ModuleClass* class. Most modules do not directly extend *ModuleClass*, usually there are one or more parent classes. For instance modules that represent an ANT type sensor extend the *ANTModule* class which states that all child classes should implement a method to decode raw sensor data sent over a wireless ANT connection. All

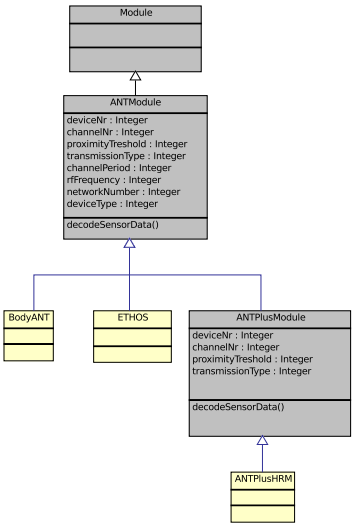

Figure 3: The grey class are abstract and cannot be instantiated. The yellow classes are actual modules. The ANTModule and ANTPlusModule share some attributes but not all. The missing attributes in ANTPlusModule are defined as constants since these are always the same for all ANT+ compatible devices.

modules have a type and an id. These fields are read-only so only the getters are defined. Modules are dynamically created from the json configuration file. All the standard fields are assigned values based on the configuration file. Values which should not be influenced by the configuration file should be marked with the *transient* keyword.

5.4 Readers

Reader classes are responsible for receiving raw sensor data. Each Reader object supports one or more modules which represent the sensor types that can decode the raw data. All Readers extend the *ReaderClass* class. When the raw sensor data is processed the reader will pas on the data to the ToolboxService where it will be sent to the CRN toolbox for further processing. The internal implementation of a Reader is left relatively open-ended since the handling of different communication protocols, internal sensors etc. varies widely in Android and the developer should have enough freedom to handle this variety.
All Reader classes should have special constructor and a shutdown method which is called when the CRN toolbox is stopped. All classes that extend *ReaderClass* should define a string array constant called: "mSupportedModules" which contains all the type names of the supported modules.

5.5 Outputs

Output classes can read and interpret data from the CRN toolbox. This data can then be stored on the local sdcard, send to a server or sent to a GUI component for visualization. Output components use their supported modules to encode data, for instance to add a timestamp or to transform it into a human or machine readable string.
All Output classes should have a special constructor and a shutdown method which is called when the CRN toolbox is stopped. All classes that extend *OutputClass* should define a string array constant called: "mSupportedModules" which contains all the type names of the supported modules.

5.6 GUI Tabs

To visualize certain aspects of CRNTC+ tabs on the main screen are used. These tabs can visualize data or send user commands to the CRNTC+. All tabs that are added to CRNTC+ extend the *TabActivity* class. Contrary to the Reader and Output components tabs only support one module instance and type. It is possible for Output components and GUI components to share a module type.

5.7 Internal communication

5.7.1 Observer pattern

Communication between many of the classes of the CRNTC+ is implemented using the Observer pattern. With the Observer pattern there is one observer who observes one or more observables. When one of the observables has something to report it will notify the observer with the new data.

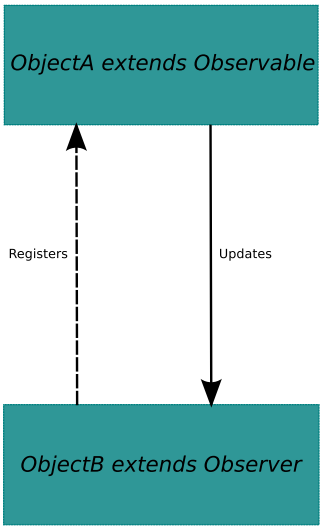

Figure 4: An observer registers one or more observable objects which will in turn update the observer with new information when this is available.

5.7.2 NotifyObject

The *NotifyObject* is used to transmit messages within the CRNTC+ such as error messages and state changes. The *NotifyObject* consists of a message type field and a data field which van contain an arbitrary object.

5.7.3 DataPacket

The *DataPacket* object is used to transmit data to and from the CRN toolbox. The object is loosely based on the DataPackets used by the CRN toolbox. The *DataPacket* object contains of an id and a values field. The id corresponds to the id of the module or CRN

tooblox task that send the data in the values field. The values field is an array of the datatype Double.

5.8 Dynamic class loading

5.8.1 GSON library

All modules specified in the JSON configuration file are instantiated at runtime. This is done through the GSON library. Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects of which the source-code is not available. [6]
To create a working Java instance of a class based on a JSON representation the GSON library needs a class definition. All the class definitions of the supported modules are stored in a *Map* together with a String key which represents the type of the module. When a configuration file is being loaded the type field of the configuration file modules are checked and if they match to a key in the Map the module is instantiated by using the class definition that is coupled to that key.

5.8.2 Reflection

Because the behaviour of the CRNTC+ application can be configured by loading different configuration files, objects need to be instantiated at runtime. The GSON library takes care of this for the modules however the correct ReaderClass, OutputClass and MyTabActivities should also be instantiated at runtime. To achieve this Java reflection is used. Java's Reflection API's makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

6 Extending the application

One of the most important aspects of CRNTC+ is it's extensibility. Several new components can be added. Such as new sensor types but also completely new sensor protocols as long as they are supported by the smarthpone hardware. It is also possible to add new GUI screens which can display data or allow some sort of interaction with other components.

6.1 Add input component

Adding new modules is one the most important tasks in the CRNTC+ application. Modules contain information on how certain components should behave. For sensors this can be sampling rates, number of channels, channel frequency and more. All modules can also specify how to extract raw sensor data and how to decode it.

1. **Extend ReaderClass class**

```
package de.uni_passau.fim.esl.crn_toolbox_center.readers;

import java.util.List;

import android.content.Context;
import de.uni_passau.fim.esl.crn_toolbox_center.modules.ModuleClass;

public class MyReader extends ReaderClass{

  /**
   * All Readers should define this String array constant! It should contain the
       type names of the
   * ModuleClasses that are supported by this Reader.
   */
  public static final String[] mSupportedModules = {"MyModule1", "MyModule2", "
      MyModule3"};
  /**
   * ReaderClass constructor
   * @param context Application context. Can be useful to get access to hardware
       sensors
   * @param modulesToUse List containing all modules that are supported by this
       reader. List is determined by the
   * mSupportedModules String array constant.
   */
  public MyReader(Context context, List<ModuleClass> modulesToUse) {
    super(context, modulesToUse);
    // Reader constructor
  }

  @Override
  public void shutDown() {
    // Shutdown all modules here

  }

}
```

Listing 2: Skeleton code for a ReaderClass

2. **Extend ModuleClass class**
   The ModuleClass is the base class for all modules. It can be directly extended but

usually a intermediary abstract class is extended which is more tailored to the needs of a specific Reader, Output or GUI Tab. See Section 8 for some examples.

3. **Add the ReaderClass and ModuleClass class definitions**

```
1   public void onCreate() {
2       super.onCreate();
3
4       mReaderClasses.add(SensorReader.class);
5       mReaderClasses.add(ANTReader.class);
6       mReaderClasses.add(LocationReader.class);
7       mReaderClasses.add(CRNTReaderTCP.class);
8       mReaderClasses.add(SystemStatusReader.class);
9       mReaderClasses.add(RandomReader.class);
10      mReaderClasses.add(BluetoothReader.class);
11
12      mOutputClasses.add(FileWriterOutput.class);
13      mOutputClasses.add(GraphOutput.class);
14      mOutputClasses.add(FileWriterOutputSFTP.class);
15
16      mReaderClasses.add(MyReader.class); //Your ReaderClass goes here
17      mOutputClasses.add(MyOutput.class); //Your OutputClass goed here
18
19  }
```

Listing 3: The class definitions of the ReaderClass are added in the constructor of the ToolboxService class. The class definitions are added in a map which couples a String to a class definition.

```
1   public Configuration() {
2
3       mModules = new ArrayList<Module>();
4
5       mGson = new GsonBuilder().setPrettyPrinting().create();
6       mUsedPorts = new LinkedList<Integer>();
7
8       //Module class definitions for Gson parsing
9       mMapModuleClasses.put("SmartphoneAccelerometer", SmartphoneAccelerometer.class)
            ;
10      mMapModuleClasses.put("SmartphoneMagnet", SmartphoneMagnet.class);
11      mMapModuleClasses.put("SmartphoneOrientation", SmartphoneOrientation.class);
12      mMapModuleClasses.put("SmartphoneLight", SmartphoneLight.class);
13      mMapModuleClasses.put("SmartphoneProximity", SmartphoneProximity.class);
14      mMapModuleClasses.put("SmartphoneLocation", SmartphoneLocation.class);
15      mMapModuleClasses.put("ACTLogAnnotation", ACTLog.class);
16      mMapModuleClasses.put("BodyANT", BodyANT.class);
17      mMapModuleClasses.put("SimpleGraphTCP", SimpleGraphTCP.class);
18      mMapModuleClasses.put("BatteryStatus", BatteryStatus.class);
19      mMapModuleClasses.put("ETHOS", ETHOS.class);
20      mMapModuleClasses.put("SuuntoBelt", SuuntoBelt.class);
21      mMapModuleClasses.put("RandomGenerator" , RandomGenerator.class);
22      mMapModuleClasses.put("Shimmer", Shimmer.class);
23      mMapModuleClasses.put("FileWriter", FileWriter.class);
24      mMapModuleClasses.put("ANTPlusHRM", ANTPlusHRM.class);
25      mMapModuleClasses.put("SimpleGraph", SimpleGraph.class);
26      mMapModuleClasses.put("AndroidPlot", AndroidPlot.class);
27      mMapModuleClasses.put("FileWriterSFTP", FileWriterSFTP.class);
28      mMapModuleClasses.put("ZephyrHxM", ZephyrHxM.class);
29
30      //Your Module goes here
31       mMapModuleClasses.put("MyModule", MyModule.class);
32      //Your GUIModule goes here
33       mMapModuleClasses.put("MyGUIModule", MyGUIModule.class);
34      //Your OutputModule goes here
35       mMapModuleClasses.put("MyOutputModule", MyOutputModule.class);
36  }
```

Listing 4: The class definitions of the modules are added in the constructor of the ConfigObject class. The class definitions are added in a map which couples a String to a class definition.

6.2 Add processing component

Most of the data processing (feature extraction, classification, etc.) is done by the CRN toolbox running within the CRNTC+ application. For more information on how to use CRNT and on how to create a processing component for CRNTC+ see [5].

6.3 Add GUI component

GUI components are displayed as tabs in the main screen of CRNTC+. GUI tabs only support one module at a time in contrary to Reader and Output components. GUI components can be used to let the user have more control over CRNTC+ or visualize data.

1. **Extend the GUIModule class**

```
package de.uni_passau.fim.esl.crn_toolbox_center.modules;

import de.uni_passau.fim.esl.crn_toolbox_center.DataPacket;

public class MyGUIModule extends GUIModule{

  /**
   * Process float array and returns a DataPacket. Used for transmitting data to the
       GUI.
   * @param data float array that needs to be process.
   */
  @Override
  public DataPacket processDataToDataPacket(float[] data) {

    return null;
  }

  /**
   * Process float array.
   * @param data float array that needs to be processed.
   */
  @Override
  public float[] processDataToFloat(float[] data) {

    return null;
  }

}
```

Listing 5: Skeleton code for a GUIModule class.

2. **Extend the TabActivity class**

```
package de.uni_passau.fim.esl.crn_toolbox_center.view;

import java.util.Observable;

import android.os.Bundle;
import de.uni_passau.fim.esl.crn_toolbox_center.modules.MyGUIModule;

public class TabMyInterface extends MyTabActivity{

  /**
   * All TabActivity classes should define this String constant!. It contains the
   * type name of the GUIModule that is supported by this TabActivity.
   */
  public static final String mSupportedModule = "MyGUIModule";

  private MyGUIModule mMyGUIModule;

  /**
   * The onCreate method is called by Android when a becomes visible for the first
      time
   * or right after it has been destroyed by Android.
   *
   */
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mMyGUIModule = (MyGUIModule) initTabActivity(MyGUIModule.class); //Call this
        method to receive the module information.
  }
  /**
   * Called when new information for this TabActivity is available.
   * @param observable the object sending the information.
   * @param data
   */
  public void update(Observable observable, Object data) {

  }

  @Override
  public void shutDown() {
    //Shutdown module here

  }

}
```

Listing 6: Skeleton code for a TabActivity class.

3. **Add TabActivity to the AndroidManifest** To make sure the Android OS can find the new MyTabActivity it should be added to the AndroidManifest.xml file. The AndroidManifest.xml file is located in the root of the project folder.

```
    <activity android:name=".view.TabStatus" android:label="@string/app_name" />
    <activity android:name=".view.TabConfigFile" android:label="@string/app_name" />
    <activity android:name=".view.TabDirectInput" android:label="@string/app_name" /
        >
    <activity android:name=".view.TabControlPort" android:label="@string/app_name" /
        >
    <activity android:name=".view.TabCategories" android:label="@string/app_name" />
    <activity android:name=".view.Activity_screen" android:label="@string/app_name"
        />
    <activity android:name=".view.TabSimpleGraph" android:label="@string/app_name" /
        >
    <activity android:name=".view.TabAndroidPlot" android:label="@string/app_name" /
        >
    <activity android:name=".view.TabMyInterface" android:label="@string/app_name" /
        > <!-- Your new MyTabActivity goes here -->
    <!-- Tools -->
```

Listing 7: The AndroidManifest.xml file

4. **Create layout XML file** The layout xml file should be placed in the res/layout folder of the project.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <!-- Your layout goes here -->
</LinearLayout>
```

Listing 8: Skeleton code of an Android layout file

5. **Add the 'drawable' components** Two icons need to be created for the tab button. One for when the tab is not selected and one for when it is. The images should be 32 by 32 pixels wide and stored in the .png format. To make sure the correct icon is displayed when the tab is either selected or de-selected another xml file needs to be specified.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android" >
<!-- your selection code goes here -->
</selector>
```

Listing 9: Skeleton code of an Android selector file

Both the icons and the selector file should be placed in the res/drawable folder of the project.

6. **Add the GUIModule and TabActivity class definitions**

```
1   public Configuration() {
2
3       mModules = new ArrayList<Module>();
4
5       mGson = new GsonBuilder().setPrettyPrinting().create();
6       mUsedPorts = new LinkedList<Integer>();
7
8       //Module class definitions for Gson parsing
9       mMapModuleClasses.put("SmartphoneAccelerometer", SmartphoneAccelerometer.class)
            ;
10      mMapModuleClasses.put("SmartphoneMagnet", SmartphoneMagnet.class);
11      mMapModuleClasses.put("SmartphoneOrientation", SmartphoneOrientation.class);
12      mMapModuleClasses.put("SmartphoneLight", SmartphoneLight.class);
13      mMapModuleClasses.put("SmartphoneProximity", SmartphoneProximity.class);
14      mMapModuleClasses.put("SmartphoneLocation", SmartphoneLocation.class);
15      mMapModuleClasses.put("ACTLogAnnotation", ACTLog.class);
16      mMapModuleClasses.put("BodyANT", BodyANT.class);
17      mMapModuleClasses.put("SimpleGraphTCP", SimpleGraphTCP.class);
18      mMapModuleClasses.put("BatteryStatus", BatteryStatus.class);
19      mMapModuleClasses.put("ETHOS", ETHOS.class);
20      mMapModuleClasses.put("SuuntoBelt", SuuntoBelt.class);
21      mMapModuleClasses.put("RandomGenerator" , RandomGenerator.class);
22      mMapModuleClasses.put("Shimmer", Shimmer.class);
23      mMapModuleClasses.put("FileWriter", FileWriter.class);
24      mMapModuleClasses.put("ANTPlusHRM", ANTPlusHRM.class);
25      mMapModuleClasses.put("SimpleGraph", SimpleGraph.class);
26      mMapModuleClasses.put("AndroidPlot", AndroidPlot.class);
27      mMapModuleClasses.put("FileWriterSFTP", FileWriterSFTP.class);
28      mMapModuleClasses.put("ZephyrHxM", ZephyrHxM.class);
29
30     //Your Module goes here
31      mMapModuleClasses.put("MyModule", MyModule.class);
```

```
32      //Your GUIModule goes here
33       mMapModuleClasses.put("MyGUIModule", MyGUIModule.class);
34      //Your OutputModule goes here
35       mMapModuleClasses.put("MyOutputModule", MyOutputModule.class);
36  }
```

Listing 10: The class definitions of the GUIModules are added in the constructor of the ConfigObject class. The class definitions are added in a map which couples a String to a class definition.

```
1       /**
2        * Called when the activity is first created. Starts the model and creates
3        * the tabs. Registers this Activity as Observer to the model.
4        */
5       @Override
6       public void onCreate(Bundle savedInstanceState) {
7
8           super.onCreate(savedInstanceState);
9
10          // start the model
11          new CRNToolboxCenter(this);
12
13          mTabActivities.add(TabCategories.class);
14          mTabActivities.add(TabSimpleGraph.class);
15          mTabActivities.add(TabAndroidPlot.class);
16          //Your MyTabActivity goes here
17          mTabActivities.add(TabMyInterface.class);
18
19          // make the tabs
20          setContentView(R.layout.tabhost);
21          createTabs();
22
23          CRNToolboxCenter.getInstance().addObserver(this);
24          IntentFilter filter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
25          batteryLogger = new LogObject(LogObject.LOG_BATTERY, "sdcard/CRNTC+
                BatteryLog");
26          registerReceiver(batteryReceiver, filter);
27      }
```

Listing 11: The class definitions of the MyTabActivities are added in the onCreate method of the MainActivity class.

6.4 Add output component

Output components can read data directly from the CRN toolbox and can be used in conjunction with GUI components but they can also be used in a standalone fashion. For instance an output component which writes data directly to a file or to a server.

- **Extend the OutputClass class**

```
1  package de.uni_passau.fim.esl.crn_toolbox_center.outputs;
2
3  import java.util.List;
4
5  import android.content.Context;
6  import de.uni_passau.fim.esl.crn_toolbox_center.modules.Module;
7
8  public class MyOutput extends OutputClass{
9    /**
10    * All Outputs should define this String array constant! It should contain the
         type names of the
11    * OutputModuleClasses that are supported by this Reader.
12    */
13   public static final String[] mSupportedModules = {"MyOutputModule1", "
         MyMOutputodule2", "MyOutputModule3"};
14
15   /**
16    * ReaderClass constructor
17    * @param context Application context. Can be useful to get access to hardware
         sensors
18    * @param modulesToUse List containing all modules that are supported by this
         reader. List is determined by the
19    * mSupportedModules String array constant.
20    */
21   public MyOutput(Context context, List<Module> modulesToUse) {
22     super(context, modulesToUse);
23   }
24
25   @Override
26   public void shutDown() {
27     // Shutdown all modules here
28
29   }
30
31
32
33 }
```

Listing 12: Skeleton code of an OutputClass object.

- **Extend the OutputModule class**

```
1  package de.uni_passau.fim.esl.crn_toolbox_center.modules;
2
3  public class MyOutputModule extends OutputModule {
4
5    /**
6       * Function that should processes data retrieved from a DirectOutputWriter task
           of the CRNT
7     * @param data retrieved from the CRNT.
8       * @return float array of processed data.
9       */
10   @Override
11   public float[] processDataToFloat(float[] data) {
12
13     return null;
14   }
15
16 }
```

Listing 13: Skeleton code of an OutputModule object.

- **Add the OutputClass and the OutputModule class definitions**

```
1    public void onCreate() {
2        super.onCreate();
3
4        mReaderClasses.add(SensorReader.class);
5        mReaderClasses.add(ANTReader.class);
6        mReaderClasses.add(LocationReader.class);
7        mReaderClasses.add(CRNTReaderTCP.class);
8        mReaderClasses.add(SystemStatusReader.class);
9        mReaderClasses.add(RandomReader.class);
10       mReaderClasses.add(BluetoothReader.class);
11
12       mOutputClasses.add(FileWriterOutput.class);
13       mOutputClasses.add(GraphOutput.class);
14       mOutputClasses.add(FileWriterOutputSFTP.class);
15
16       mReaderClasses.add(MyReader.class); //Your ReaderClass goes here
17       mOutputClasses.add(MyOutput.class); //Your OutputClass goed here
18
19   }
```

Listing 14: The class definitions of the OutputClass are added in the constructor of the ToolboxService class. The class definitions are added in a map which couples a String to a class definition.

```
1   public Configuration() {
2
3      mModules = new ArrayList<Module>();
4
5      mGson = new GsonBuilder().setPrettyPrinting().create();
6      mUsedPorts = new LinkedList<Integer>();
7
8      //Module class definitions for Gson parsing
9      mMapModuleClasses.put("SmartphoneAccelerometer", SmartphoneAccelerometer.class)
           ;
10     mMapModuleClasses.put("SmartphoneMagnet", SmartphoneMagnet.class);
11     mMapModuleClasses.put("SmartphoneOrientation", SmartphoneOrientation.class);
12     mMapModuleClasses.put("SmartphoneLight", SmartphoneLight.class);
13     mMapModuleClasses.put("SmartphoneProximity", SmartphoneProximity.class);
14     mMapModuleClasses.put("SmartphoneLocation", SmartphoneLocation.class);
15     mMapModuleClasses.put("ACTLogAnnotation", ACTLog.class);
16     mMapModuleClasses.put("BodyANT", BodyANT.class);
17     mMapModuleClasses.put("SimpleGraphTCP", SimpleGraphTCP.class);
18     mMapModuleClasses.put("BatteryStatus", BatteryStatus.class);
19     mMapModuleClasses.put("ETHOS", ETHOS.class);
20     mMapModuleClasses.put("SuuntoBelt", SuuntoBelt.class);
21     mMapModuleClasses.put("RandomGenerator" , RandomGenerator.class);
22     mMapModuleClasses.put("Shimmer", Shimmer.class);
23     mMapModuleClasses.put("FileWriter", FileWriter.class);
24     mMapModuleClasses.put("ANTPlusHRM", ANTPlusHRM.class);
25     mMapModuleClasses.put("SimpleGraph", SimpleGraph.class);
26     mMapModuleClasses.put("AndroidPlot", AndroidPlot.class);
27     mMapModuleClasses.put("FileWriterSFTP", FileWriterSFTP.class);
28     mMapModuleClasses.put("ZephyrHxM", ZephyrHxM.class);
29
30    //Your Module goes here
31     mMapModuleClasses.put("MyModule", MyModule.class);
32    //Your GUIModule goes here
33     mMapModuleClasses.put("MyGUIModule", MyGUIModule.class);
34    //Your OutputModule goes here
35     mMapModuleClasses.put("MyOutputModule", MyOutputModule.class);
36  }
```

Listing 15: The class definitions of the OutputModules are added in the constructor of the ConfigObject class. The class definitions are added in a map which couples a String to a class definition.

7 Readers

7.1 ANTReader

The implementation of the ANTReader is based on the ANT+ for Android source code provided by Dynastream Systems [**?**].

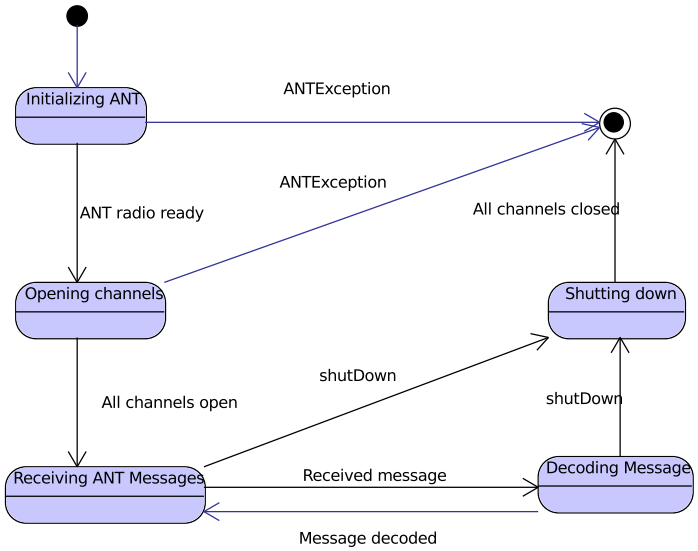

Figure 5: The ANTReader will first try to initialize the ANT radio. When successful the channels as defined by the modules of the ANTReader will be opened. When all channels are successfully opened the ANTReader is ready to receive ANT data and decode the data to messages.

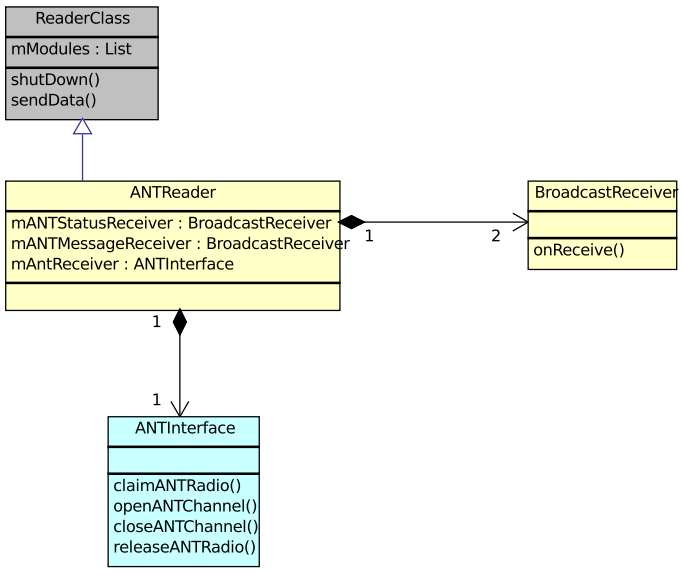

Figure 6: Whenever new ANT data is available the onReceiver method of the ANTMessage BroadcastReceiver is called. This data is then decoded by the appropriate ANTModule. The ANTInterface object serves as an interface between Android and the ANT radio and is supplied by Dynastream systems. The ANTInterface object is able to initialize the ANT radio and open ANT channels.

7.2 BluetoothReader

The BluetoothReader manages several BluetoothModules. Each BluetoothModule is responsible for connecting for a Bluetooth device and subsequently managing the connection. It is possible to use an external library, supplied by the manufacturer of the Bluetooth device, to establish and manage the connection. However a custom implementation can still be implemented if desired.
The following methods of the BluetoothModule are called by the BluetoothReader:

- **initConnection**
- **connect**
- **startStreaming**
- **shutDown**

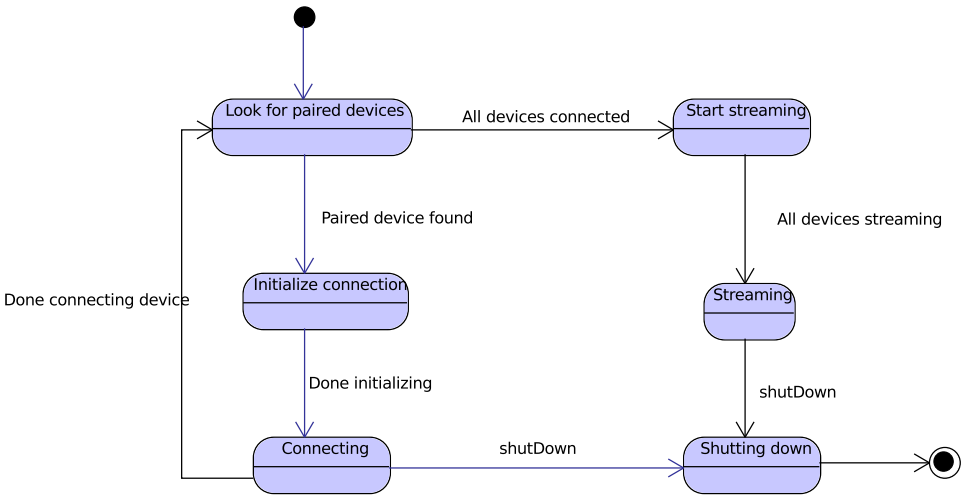

Figure 7: The BluetoothReader will attempt to connect to all the Bluetooth devices represented by the BluetoothModule objects obtained from the configuration file. After all devices are connected the BluetoothReader will tell all the modules to start streaming.

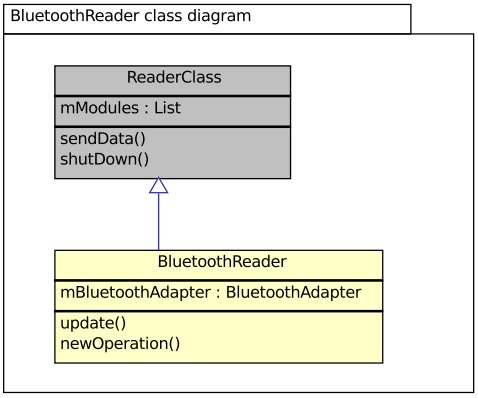

Figure 8: The update function is called whenever a BluetoothModule has new data ready. The BluetoothReader will then use the sendData function to transmit the data to the CRN toolbox.

8 Modules

This section will detail some of the configuration file parameters for the modules.

8.1 ANTModule

```
1  int deviceNr;     //REQUIRED: Unique 16 bit device number. 0 for wildcard search
2  int channelNr;      //REQUIRED: Unique arbitrary channel number
3  int proximityTreshold;  //REQUIRED: Number between 1 and 7 to specify the search area
4  int transmissionType; //REQUIRED: 8 bit field that defines the transmission. 0 for
       wildcard search
```

Listing 16: The configuration file parameters displayed are used by all ANTModule type classes.

8.2 Shimmer

```
1  int samplingrate;   //OPTIONAL: Sampling rate of the Shimmer:
       0,10,50,100,125,166,200,250,500 or 1000Hz Default: 0
2  boolean accelerometer;  //OPTIONAL: Activate accelerometer. Default: false
3  boolean gyroscope;    //OPTIONAL: Activate gyroscope. Default: false
4  boolean magnetometer; //OPTIONAL: Activate magnetometer. Default: false
5  boolean ecg;      //OPTIONAL: Activate ECG. Default: false
6  boolean emg;      //OPTIONAL: Activate EMG. Default: false
7  boolean gsr;      //OPTIONAL: Activate GSR. Default: false
8  boolean anex_a7;    //OPTIONAL: Activate anex a7. Default: false
9  boolean anex_a0;    //OPTIONAL: Activate anex a0. Default: false
10 boolean heart;      //OPTIONAL: Activate heart. Default: false
11 boolean strain;     //OPTIONAL: Activate strain. Default: false
12 boolean calibrated;   //OPTIONAL: Use calibrated data
13 int accRange;     //OPTIONAL: 0: (+-1,5G), 1: (+- 2G) , 2: (+- 4G) or 3: (+- 6G)
14 int gsrRange;     //OPTIONAL: 0: (10-56 kOhm) ,1: (56-220 kOhm), 2: (220-680 kOhm)
15             //      3: (680-4700 kOhm ) or 4: (auto range);
```

Listing 17: Configuration file parameters for the Shimmer device.

8.3 AndroidPlot

The AndroidPlot module displays a graph on the screen of the smartphone

```
1  int nChannels;    //REQUIRED: Number of data channels that should be read from crnt.
2  int maxRange;   //OPTIONAL: Max range of the Y-axis of the graph
3  int minRange;   //OPTIONAL: Min range of the Y-axis of the graph
4  int nPoints;    //OPTIONAL: Number of points on the X-axis
5  String xName;   //OPTIONAL: Label for the X-axis
6  String yName;   //OPTIONAL: Label for the Y-axis
```

Listing 18: Configuration file parameters for the AndroidPlot device. Since AndroidPlot also extends OutputModule the nChannels parameter should also be defined. This parameter should match the number of channels that the DirectOutputWriter task uses.

8.4 ACTLog

For a detailed description of the ACTLog parameters and other information see [3].

9 Known issues

9.1 User feedback

When certain critical events happen within CRNTC+ such as errors with certain sensors or an error in the configuration file the user should be notified. Although there are some messages and warnings already this could be improved.

9.2 DirectOutputWriter support

As of writing the DirectOutputWriter support is constrained. This is due to the fact that the crnc.cpp file, which facilitates communication between the CRN toolbox and the CRNTC+ framework, is not complete and contains some errors. Due to some of these it errors it is not possible to use more than one OutputModule class (only one DirectOutputWriter Task in CRNT) at the same time.

References

[1] Jakob Weigert. The context recognition network toolbox on android, April 2011.

[2] David Bannach, Paul Lukowicz, and Oliver Amft. Rapid prototyping of activity recognition applications. *Pervasive Computing*, 7(2):22–31, April-June 2008.

[3] Frank Roberts. *Documentation for the ACTLog project*. ACTLab, 2011.

[4] Androidplot website. http://androidplot.com/wiki/Home.

[5] Crnt tutorial. http://wiki.esl.fim.uni-passau.de/index.php/Toolbox_Tutorial.

[6] Gson library website. https://code.google.com/p/google-gson/.

B Recording plan

Technische Universiteit
Eindhoven
University of Technology

Recording plan

by Frank Roberts

Date: October 27, 2012

Contents

1 Introduction

This document was written as part of a master thesis project. The goal of the project was to develop a generic smartphone-based sensing and processing platform. The resulting application called Context Recognition Network Toolbox Center Plus (CRNCT+) should be evaluated in a case study. For this purpose, epileptic seizure monitoring is targeted. Epilepsy is a chronic illness with a large negative impact on the quality of life. Due to the high variance of patients and their seizures, it is difficult to evaluate their medication and overall progress after leaving a controlled hospital environment. Monitoring the seizure rate outside the hospital is a key information to aid medication and to evaluate treatment progress. In order to interpret seizure events correctly, daily routine information could be used for separating daily activities and actual events. In this project we focus on clonic epileptic seizure detection, not restricted to lab environment. Similar work has been done before, but suffers from high rate of misclassification. To deal with this issue, wrist-worn motion sensor (ETHOS) will be merged with smartphone motion data and a heart-rate sensor.
This document will describe the required hardware and placement of the hardware in Section 3. Section 4 will discuss the measurement plan (measurement time, number of participants etc.). Finally in Section 5 the operation of the CRNTC+ application will be explained.

2 General goal

The main goal of this project is evaluate the CRNTC+ platform by implementing an epileptic seizure monitoring application. This goal can be subdivided in three parts:

1. Gathering data of epileptic seizures.
   Will be evaluated by the recording time between recharges, accuracy of measured data and the up-time of the application.
2. Analysis of the data.
   Will be evaluated on the ease of using the resulting data with external mathematical tools for analysis.
3. Implementation of seizure monitoring application.
   Will be evaluated on the ability to actually differentiate epileptic seizures from other daily activities.

The success of each of the above steps will be analysed after all steps are complete.

3 Recording hardware setup

3.1 Requirements

- Sony Xperia Active with Android 2.3 and the CRN Toolbox Center+ application installed.
- 1 Shimmer Inertial Motion Unit (IMU)
- 1 ETHOS Inertial Motion Unit
- 3 ECG Shimmer sensors
- 3 ECG patches for shimmer
- 1 respiration belt
- 3 USB to Micro USB charging cables

3.2 Sensor placement

3.2.1 IMU Shimmer Sensor

The IMU1 will measure the motion at the wrist by using the accelerometer, gyroscope and magnetometer. Attach the IMU1 Shimmer sensor with the appropriate wrist band to the dominant arm of the participant.

3.2.2 ECG1 Shimmer sensor

The ECG1 will be used to measure the ECG values and the motion of the upper left arm of the participant. Attach three electrodes on the left side of the chest. See Picture 3.2.2. Attach the ECG1 with the correct strap to the upper left arm. Connect the top right electrode to the ECG1 port labeled LA. Connect the lower left electrode to the RA port. Finally connect the lower right electrode to the LL port. The wires can be attached to the shoulder with medical tape.
To test if the electrodes are correctly attached use the ECG1ECGTest.json configuration file. This should display a graph with the ECG data.

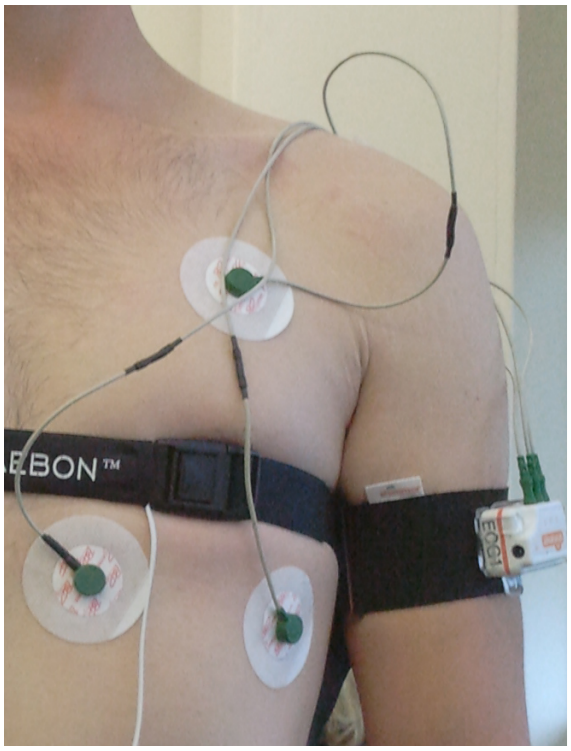

Figure 1: ECG1 Placement

3.2.3 ECG2 Shimmer sensor

The ECG2 will be used to measure the respiration values and the motion of the chest of the patient. Attach the respiration band around the chest. See Picture 3.2.2. Attach the ECG2 with the correct strap to the lower abdomen. Connect the respiration band plugs to the ECG2 ports based on their colours.

To test if the respiration band is correctly attached use the ECG2ECGTest.json configuration file. This should display a graph with the respiration data.

3.2.4 ECG3 Shimmer sensor

The ECG3 will be used to measure the motion of the upper right arm of the participant. Attach the ECG3 to the upper right arm using an appropriate strap.

3.2.5 ETHOS Inertial motion sensor

Turn on the ETHOS sensor and wait till the small red led light starts blinking. Attach the ETHOS to the non dominant wrist of the participant with an appropriate strap.

3.2.6 Smartphone

The smartphone will be connected to the sensors using Bluetooth. This means that wearers of the sensors need to be relatively close to the smartphone. The smartphone will also be used to do the annotations.

4 Study design

This is an observational study to measure physical epileptic seizures. The participants will be actors at the Kempenheaghe expertise centre. The participants will perform various everyday activities which simulate everyday live. During these activities several epileptic seizures of different types will be simulated. The measurements will be done by the CRNTC+ application. See Section 5 for more information on CRNTC+.

- Daytime measurements.
- 1 day measurement duration.
- 2 participants.
- Participants will perform a 'script' containing several every day activities for about 30 to 45 minutes.
- 2 Orientation sensors for detecting motion in the wrist areas. Samples at 100-128 Hz.
- 1 Heart-rate sensor for measuring changes in heart-rate and upper left arm motion at 100Hz.

- 1 Respiration sensor for measuring respiration and chest motion.
- 1 Motion sensor for measuring upper left arm motion.
- Measurement of daily activities to make a distinction between an actual seizure and other activities.
- Annotation will be done with the ACTLog annotation ability of the CRNTC+ application.
- Data will be written to the sdcard of the smartphone for later analysis. One of the inertial motion sensors will also store it's data on an internal sdcard.

4.1 Activity script

Waking up

- Lying on a bed (2 minutes)
- Getting dressed (putting on 2 socks and 2 shoes with laces while sitting, putting on a vest while standing) (2 minutes)
- Stand up (1 minutes)
- Walking
- Scratching (5 times)
- Walking
- Drinking from a glass
- Cutting food with knife (5 times)
- Walking
- Brush teeth (2 minutes)
- Seizure (duration determined by actor)
- Rest (5 minutes)

Morning gymnastics

- Push-ups (determined by the actor)
- Rest (5 minutes)

Walking to work

- Walking (5 minutes)

- Shaking hands (5 times)
- Using a computer mouse (determined by the actor)
- Typing on a keyboard (determined by the actor)
- Using a computer mouse (determined by the actor)
- Typing on a keyboard (determined by the actor)
- Seizure (duration determined by actor)
- Rest (5 minutes)

End of the day

- Walking (5 minutes)
- Unloading and putting away groceries (taking a basket filled with groceries: 2 cans of beans, 2 cartons of milk, 2 bottles of water) (2 minutes)
- Folding 8 towels (2 minutes)
- Vacuum cleaning (5 minutes)
- Seizure (duration determined by actor)
- Rest (5 minutes)
- Cutting food (determined by the actor)
- Doing de dishes (no water) (5 minutes)
- Watching television (2 minutes)
- Seizure (duration determined by actor)

5 Study protocol

The general procedure of a recording is as follows:

1. Start recording for the first time.
2. Recording may be stopped for privacy, medical reasons.
3. Annotation takes place within the application with the ACTLog annotation tool.
4. Recordings will take place for about 45 to 60 minutes.
5. Recharging of equipment.

When the application stops unexpectedly the application can be easily restarted. The following sections will explain how to start, stop and test whether the application is running correctly in more detail.

5.1 Start recording (first time)

1. **Start application** Start up the application by tapping it's icon.

Figure 2: Icon of the CRNTC+ application

2. **Select a configuration file** Navigate to the configuration file screen by pressing it's icon in the top of the screen. Next tap the 'Browse...' button and select the file called 'Recording.json'. The file that was selected last will be stored.
3. **Start the toolbox** Tap the button called 'Start toolbox with config file' and the application will start the toolbox. An icon should be displayed in the top left corner of the screen indicating that the toolbox has started. Note that all sensors should be switched on before recording is started! When all sensors are working and sending data the application will briefly display "All Readers Ready!". See Section 3.2 for instructions on how to turn on the sensors.
   To actually start the recording press and hold the 'Nothing' annotation option.

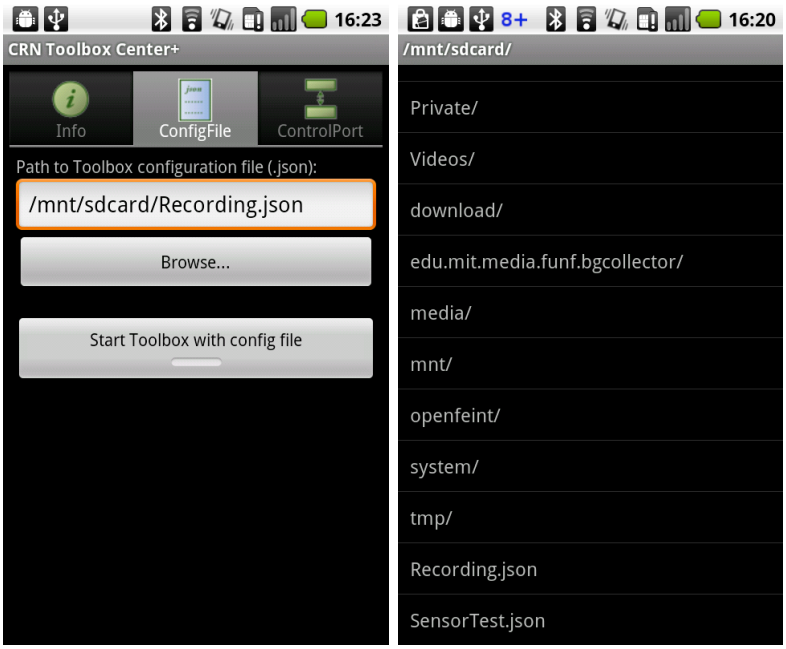

Figure 3: The Configuration file screen and the file browser. By pressing the 'Browse...' button a configuration file can be loaded into the application. By pressing the 'Start Toolbox with config file' the recording of sensor data can be started and stopped.

5.2 Stop recording

Navigate to the configuration file tab and tap the 'Toolbox is running - touch to stop' button to stop recording. See Figure 3.

5.3 Test sensors

1. **Start application** Start up the application by tapping it's icon. See image 1.
2. **Select a configuration file** Navigate to the configuration file screen by pressing it's icon in the top of the screen. Next tap the 'Browse...' button and select the file called 'ECG1Test.json'. See image 2. There now should be several more tabs added to the top of the screen.

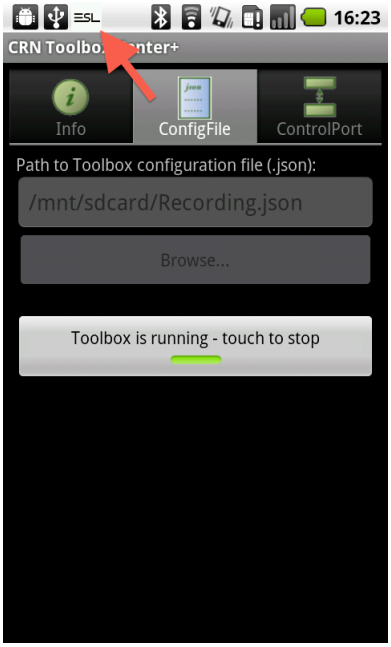

Figure 4: The red arrow indicates the icon that is displayed when recording is started.

3. **Start the test** Tap the button called 'Start toolbox with config file'. Navigate to one of the graph tabs at the top of the screen. If the sensors are working several graph lines should be visible on the screen. Try moving the Shimmers around and see whether the graphs change based on the motion.

5.4 Annotation

For monitoring epileptic seizures during sleep a combination of EEG and video annotation are the gold standard. During daytime measurement of every day activities both of these methods prove to be problematic. EEG values can be influenced by every day activities and for video recording a direct line of sight is always needed.
Since actors will be used to simulate seizures and perform various activities the built-in ACTLog tool will be used for annotation.

1. **Select ACTLog**
   While a recording is started press the ACTLog tab.
2. **Annotation**
   Tap and hold a category (seizure or activity) for about one second. A second screen is shown with several options. Right before an activity or seizure is performed tap and hold an option to annotate the recording data. When the activity or seizure has ended tap and hold the 'nothing' option to mark the end of the activity or seizure.

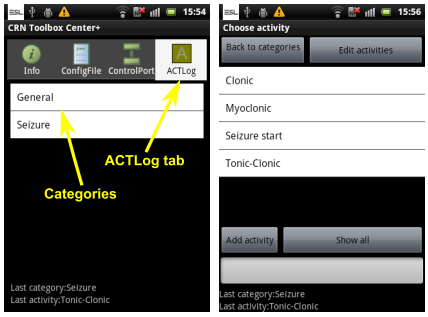

Figure 5: ACTLog annotation screens.

3. **Mistakes**
   In case data has been annotated by mistake tap and hold the 'Mistake' category on the first ACTLog screen for about one second. On the second screen tap and hold the mistake option to mark the last annotation as a mistake.

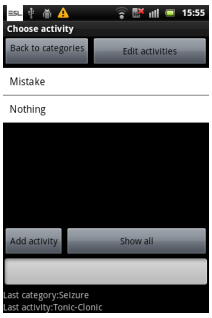

Figure 6: The 'General' category of annotations can be used to signal that a mistake has been made when during annotation.

5.5 Recharging equipment

After each recording session of about 8 hours some of the hardware equipment needs to be recharged.

1. **Smartphone**
   The smartphone can be recharged by using the supplied USB to Micro USB cable. See Figure **??**. The battery icon in the top-right corner of the screen of the smartphone will change to indicate that the smartphone is charging.

Figure 7: Make sure the battery icon looks like this when charging.

2. **Shimmer sensors**
   The Shimmer sensors can all be charged using the charging dock.