Technical University of Denmark

Development of iOS Data Sensing Framework

IMM-B.Eng-2013-47

Delivered by:

Mathias Hansen, s093478

DTU Informatics
Department of Informatics and Mathematical Modelling

Supervisors: Jakob Eg Larsen & Sune Lehman

Delivery: April 4, 2013
The report contains 75 pages

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-B.Eng-2013-47

Summary (English)

The goal of this project is to implement a data sensing platform for iOS. The platform should collect data from the various sensors on the iPhone 5 such as the accelerometer, battery and other components.

The collected data should be stored and transferred securely to a remote storage, allowing further analysis to be performed by researchers.

The project is based on previous research by the same author, conducted in the course: "Special course: Research Concerning Development of a Data Sensing Platform for iOS"[3].

The project has been running from January 1st 2013 to April 4th 2013, in which time period the student has worked full-time.

It was possible to fully implement the project scope, focusing heavily on the core requirements and features as well as extensibility.

Summary (Danish)

Målet med dette projekt var at implemente en data sensing platform til iOS. Platformen skulle opsamle data fra sensorer på iPhone 5 såsom accelerometer, batteri og andre komponenter.

Det opsamlede data skulle gemmes og overføres på en sikker måde til en form for ekstern lagring, hvilket gør det muligt for forskere at analysere det opsamlede data fra et centralt sted.

Projektet er baseret på tidligere research af samme forfatter, i kurset: "Special course: Research Concerning Development of a Data Sensing Platform for iOS"[3].

Projektet har kørt fra d. 1. Januar 2013 til d. 4. April 2013, i hvilken periode den studerende har arbejdet fuldtid på projektet.

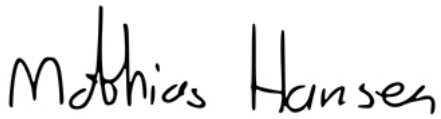
Det var muligt at implementere det fulde projekt efter de fastsatte specifikationer, med ekstra fokus på kerne krav og features samt med vægt på mulighed for at gøre det nemt at tilføje udvidelser.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfillment of the requirements for acquiring a B.Sc. in IT.

The thesis deals with development of an iOS Data Sensing platform.

Washington D.C., April 3rd 2013



Mathias Hansen

Acknowledgements

I would like to thank the following people for help, guidance and support during research and development of this thesis project.

- Michele Walk
- Sune Lehmann
- Jakob Eg Larsen
- Søren Ulrikkeholm
- Arkadiusz Stopczynski
- Per Boye Clausen
- Jeppe Kronborg Nielsen
- Terkel Brix

Table of Contents

List of Figures

Listings

Chapter 1

Introduction

This chapter will give a short introduction of Sensible DTU and Funf, in order to give an overview of the background for the project.

This is followed by a description of the project and introduction to some general concepts and terminology used in the report.

About Sensible DTU 1.1

The Sensible DTU project seeks to map how social networks are created and how individuals are interacting in real time.

This is primarily done by anonymously tracking social relations for undergraduate students at the Technical University of Denmark, using as many data sources as possible. These includes Facebook data, phone calls, text messages as well as detecting when students meet each other. Other supporting data is also collected such as location information and general device data.

Such an experiment was rolled out in the summer of 2012, spanning several hundred participants and using Android devices equipped with custom software based off of the Funf framework.

Sensible DTU is led by associate professors Jakob Eg Larsen and Sune Lehman. Several students, professors and researchers are involved with Sensible DTU, including researchers from sociology, anthropology and other social sciences.

About Funf 1.2

The Funf framework is a modular data sensing and processing framework for Android, developed at the MIT Media Lab. The framework provides core functionalities needed to collect, store, configure and upload a wide range of data types.

Funf was originally developed for academic research and provides privacy protection measures for all built-in probes to make sure that no human-readable data can be linked to a specific personal identity[2].

Project description 1.3

The goal of this thesis is to develop a framework and an app for iOS which collects data for a wide range of sensors such as accelerometer data, location, social media data, phone calls, battery information, etc. The data is to be used with the Sensible DTU[1] project.

The framework should handle collection and normalization of the data and have a structure that makes it possible to add new data sources later on.

The project is inspired by Funf Open Sensing Framework which provides the same functionality on the Android platform.

The development is based on research conducted in the course: "Special course: Research Concerning Development of a Data Sensing Platform for iOS"[3] and should thus be implemented based on the results of the project. This also means that this project will not cover individual probe analysis and other parts that have already been covered by the previously documented research.

Data should be saved locally on the device and synchronized in intervals with an online backend/data storage. This should happen in a secure way that does not infringe with the users privacy. It is desirable to develop a minimal implementation of the backend server for testing purposes as part of this project as well.

In addition, the app should visualize some of the collected data so the user can get an overview of the stored data that has been collected from their device.

Concepts and terminology 1.4

To better explain and describe the different concepts in the app, several terms have been established. These are defined and portrayed in the following sections.

1.4.1 Probe

A probe refers to a individual piece of software that is built to collect a certain type of data. An example is a battery probe.

A battery probe would be designed to collect battery information such as whether the device is charging and what the current charge level is. The probe should then return this collected information when other parts of the overall system request it.

1.4.2 Data batch

When a probe has collected data from its source, there are usually several different values returned such as in the above example with a battery probe, where both charging status and battery level would be collected.

This collection of probe values is called a data batch. Throughout the report, there will be mentions of collecting and storing data batches, meaning a collection of probe data as a whole.

1.4.3 Local storage

Local storage refers to storing files persistently on the iOS device. When local storage is mentioned, it is usually a reference to storing the collected data on the device so it can be retrieved again at a later time.

Local storage is the opposite of remote storage, which refers to uploading and storing files on a remote server.

Chapter 2

Analysis

From the analysis of the project scope and overall concept it was possible to create a rich picture that provides a visual overview of the actors and components involved in the project, as well as their individual relationships.

The rich picture as seen on Figure 2.1 on the facing page is based directly off of the project description.

It can be seen that the mobile device serves a central role in the project as it is directly connected to most of the other component. It is also notable that security and anonymization is an important aspect both regarding data transfer and data storage.

Judging by the complexity and many different components it has been decided to separate the project into several sub-projects.

It can be seen that a separate server component is necessary for remote data storage. This can be built as a sub-project independent from the other parts of the system.

The mobile device itself is a central and complex part of the project, so it would be relevant to split this part in to a separate app project and library project thus keeping visualization and user interaction separate from the actual data collection process.

This results in three sub-projects: The OpenSense Collector app, The OpenSense Library/Framework and the backend server.

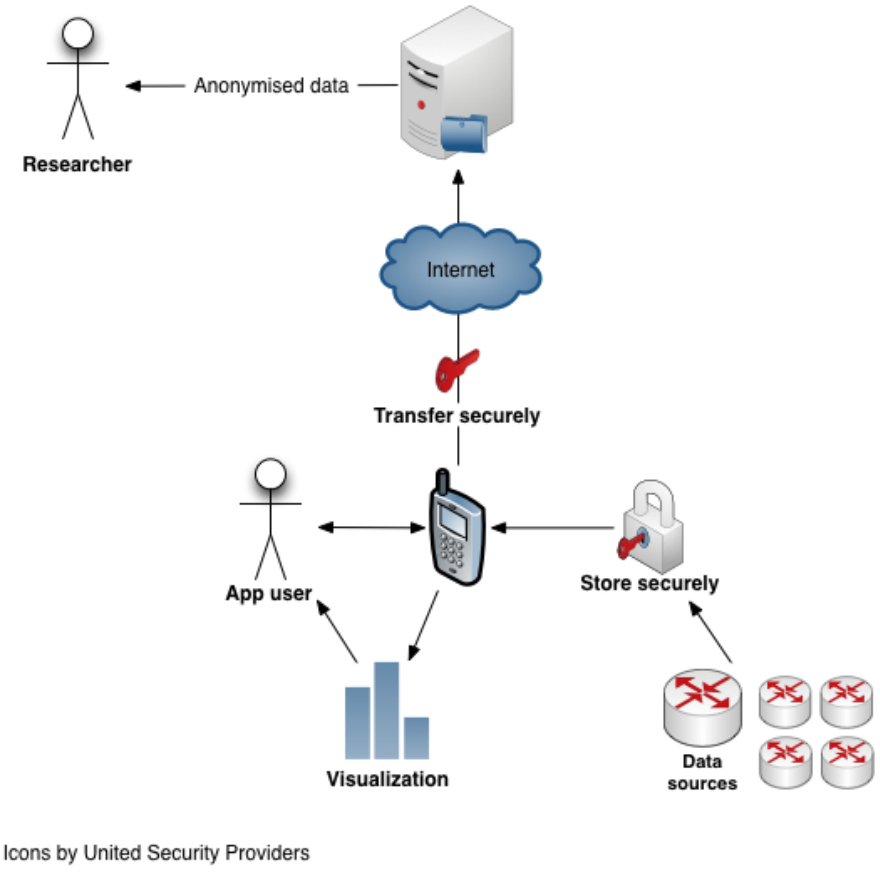


Figure 2.1: Rich picture based on the project introduction

Requirements 2.1

Based on the initial analysis and project seperation process, a set of general requirements have been determined for the library, server, app and overall scope. This assists prioritizing and planning in the project development process.

Requirements have been divided into three priority-based main groups: Necessary, Desirable and Nice to Have, ranging from highest to lowest priority.

2.1.1 OpenSense Collector app

Necessary

- It should be possible to start/stop the collector process from the app
- The collector process should be able to run in the background

Desirable

- The user should be able to access and view the raw collected data
- The user should be able to see a visual representation of *some* of the collected data

Nice to Have

- The user should be able to see a visual representation of *all* of the collected data

2.1.2 OpenSense Library

Necessary

- Data must be collected from several sources and stored locally
- Data must be securely stored

Desirable

- Collected data should be periodically transferred to a remote server

Nice to Have

- It should be easy to add additional data sources

2.1.3 Backend server

Necessary

- Server should be able to handle data upload & storage

Desirable

- Communication protocol should use open and widely recognized standards
- Server should be able to provide app with latest configuration file

Nice to Have

- Should be highly scalable (e.g. handle thousands of users)

2.1.4 Overall

Necessary

- Code base should be built for iOS 6
- Code base should be built for iPhone 5
- Code base should use `ARC`

Desirable

- It should always be possible for a moderator/researcher to access the locally-stored data and decrypt it manually

Nice to Have

- It should be possible to update the app configuration remotely
- A lot of implemented data sources

Timeline 2.2

As one of the first steps of the project planning process, a Gantt chart depicting the project timeline was created (as seen on Figure 2.2). This allowed for a more complete overall picture of the project and enabled realistic time management of the many different parts of this project.

First, several milestones were defined to set specific goals for the project. These were the minimum viable product (MVP) for the app and library as well as a feature-freeze and the final project hand-in.

It was decided to spend the first weeks focusing on programming and implementing the project, starting with the framework followed shortly by the app. A large portion of time was allocated for documenting the process with quality insurance and testing in between.

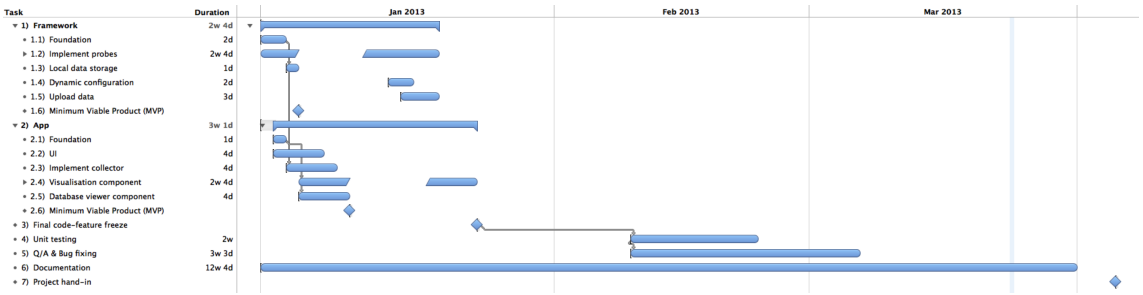


Figure 2.2: Project timeline

A larger version of the timeline is available in the Appendix, section B on page A-7.

Chapter 3

Design

Overview 3.1

From the beginning, it was decided that the overall project should be divided into several sub-projects. This makes it easier to replace individual parts and maintain a properly separated overall structure.

The sub-projects, as seen in Figure 3.1 on the next page are the OpenSense Library, OpenSense Collector app and OpenSense server.

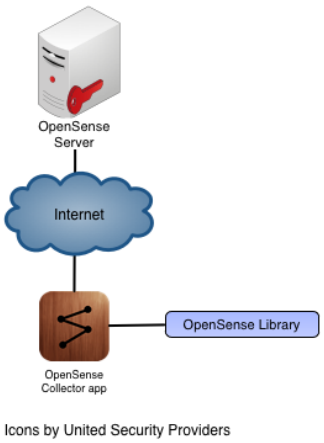


Figure 3.1: Design overview

The library handles the collection, storage and upload of data probe and the collector app implements the library and acts as an interface to the user for browsing and visualizing collected data. The server receives the collected data and stores it for later analysis (not covered by this project).

This chapter covers the design process for each of these separate projects.

Library 3.2

The Open Sense framework has been designed as a separate static library. This allows it to be reused in other projects as well as isolating the framework-specific code to its own project.

By using a static library, all routines and classes are copied into the app target at compile time. As such, the library source code is fully embedded in the final app build.

3.2.1 Library design

The library is designed with a main class called `OpenSense`, which functions as the entry point for all operations. It is designed as a singleton class, which ensures that only a single instance of the class is ever created.

The `OpenSense` class handles general tasks that affect the whole library. This includes probe management and the process of starting and stopping data collection as well as device registration and data uploading.

In addition to the `OpenSense` class, the following classes also serves a primary purpose in the library.

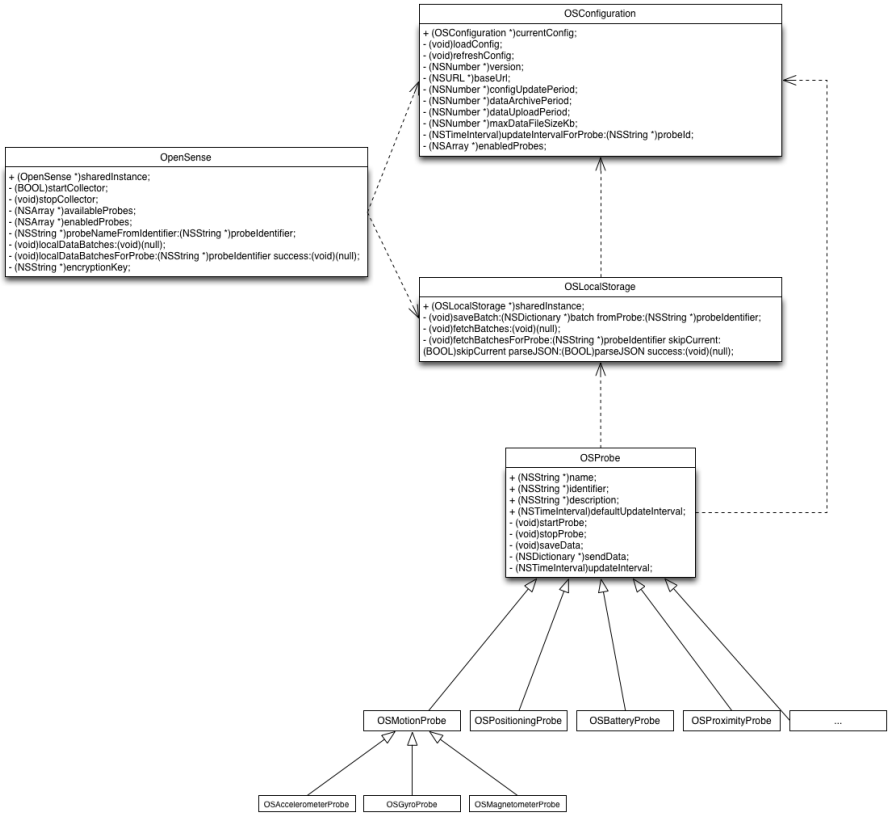


Figure 3.2: Simplified class diagram of the OpenSense library

`OSConfiguration`

This is the configuration manager. This singleton class handles parsing of the current JSON config file and access to all the config properties, such as enabled probes and the config update interval.

The configuration manager also contains the core functionality for downloading and storing new config files.

The currently active configuration can always be accessed by calling `[[OSConfiguration currentConfig] propertyName]` and is thus easily accessible from other parts of the library.

`OSLocalStorage`

The purpose of this class is to store and retrieve local probe data collected through the library in a safe and secure manner.

The local storage manager also manages rotation of data files, which ensures that the local data files won't grow too large.

`OSLocalStorage` also follows the singleton pattern, which allows straightforward access from other classes to save and retrieve data.

`OSProbe`

This is the superclass to all probes. The `OSProbe` cannot be initialized by itself and should thus be considered an abstract class.

The `OSProbe` class defines meta data for the probe as static functions. This includes probe name, identifier (in reverse domain name notation[1]), description and default update interval.

It also includes operations commonly used with most probes such as update-interval scheduling and methods to handle starting and stopping of the individual probes as well as invoking the probe data storage operation.

A full overview of the OpenSense library classes and their relations can be seen in Figure 3.2 on the preceding page.

3.2.2 Project structure

To allow for development of the library and the app at the same time, Xcode is configured with two separate projects.

Figure 3.3: Nested projects in Xcode

The library project is then configured as a nested project to the app projects, as seen in Figure 3.3.

This enables several options to make the full build process automatic.

Linking the library First, the Open Sense library needs to be linked as a binary from the OpenSense Collector app target

Build target order The app scheme then needs to be modified to make sure that the library is always built first.

[1] A naming convention commonly used for technical identifiers, example: dk.dtu.imm.sensible.dtu

Header search path The library saves its headers in a pre-determined `includes` folder when the project is built. In order to make the header files visible to the app, the following search path needs to be added: `$SRCROOT/../OpenSense/build/includes/`

3.2.3 Extensible probes

An important requirement for the project was to make the process of adding additional probes as straightforward as possible.

This has been realized by designing an `OSProbe` super class, as shown in Figure 3.4, that defines the foundation and protocol for separate probes. Every probe needs to implement this class and its methods.

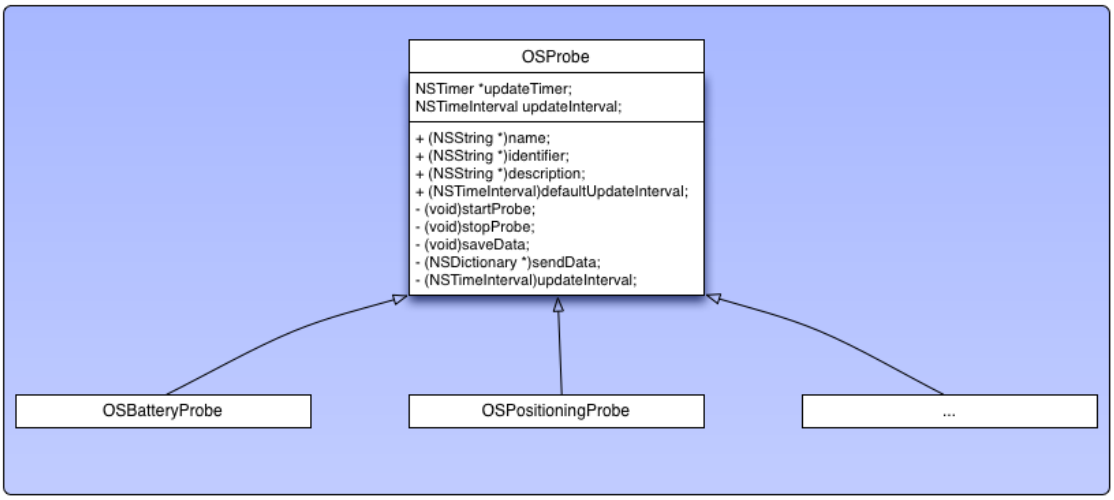


Figure 3.4: Class diagram depicting `OSProbe` relationship as a superclass

The `sendData` method is called when data needs to be collected and saved in local storage. The probe can either invoke the method itself or configure an update interval to poll for updates. The data should be returned as a `NSDictionary` with a key for each data point that needs to be stored.

In addition, the individual probes need to redefine the static methods that describe the probe:

`+ (NSString*)name` The name of the probe, e.g. "Battery"

`+ (NSString*)identifer` A unique identifier for the probe in reverse domain name notation, e.g. dk.dtu.imm.sensible.battery

`+ (NSString*)description` A description of the probe and what it collects

`+ (NSTimeInterval)defaultUpdateInterval` If a positive number is defined, the `saveData` method will be called with this interval in seconds. If the method is

invoked by the probe, the `kUpdateIntervalPush` constant should be returned instead.

If all of these methods are not overwritten, it will cause an assertion to fail and thus end up raising a `NSInternalInconsistencyException` at runtime. It is unfortunately not possible to detect this at compile time, as Objective C does not support virtual classes or methods due to its inherit architecture.

It is also possible for probes to implement the optional `startProbe` and `stopProbe` methods so that they are notified when a probe is started and stopped. However, it is important to note that the super class function must be called after these methods have been executed, by invoking `[super startProbe]` or `[super stopProbe]`.

Collector App 3.3

The OpenSense Collector app interfaces directly with the OpenSense library. The app's functionality can be divided into three main groups:

Collector status This part of the app is also the most crucial. It allows the user to view the current status of the OpenSense collector as well as starting/stopping the data collection process. The current status includes amount of data entries collected and duration.

Figure 3.5: Collector status

Probe visualization This utilizes locally-stored data to visualize selected probes, enabling the user to see the collected information at a glance.

Collected data This provides a raw view into the locally collected and stored probe data, giving a detailed view of every single bit of information that is stored and transferred.

3.3.1 Implementing the library

The Open Sense library is designed to work fully independently from the collector app, enabling development of alternative collector apps with other intents or purposes. It even enables implementation of the OpenSense framework in an existing iOS app.

To use the library, a default configuration file must be created first. The configuration file is in the widely adopted `JSON` format and is loosely based on the Funf[2] configuration format.

The most important part of the configuration is the `baseUrl` property which lets the app know where further configuration updates should come from and where data should be uploaded. When the library is first run within an app, a new configuration file will automatically be downloaded.

An example of such a default configuration can be seen below:

Listing 3.1: Example default config.json

```
{
  "name": "defaultconfig",
  "version": 1,
  "dataUploadOnWifiOnly": true,
  "baseUrl": "http://data.opensense.org",
  "configUpdatePeriod": 21600,
  "dataArchivePeriod": 300,
  "dataUploadPeriod": 7200,
  "maxDataFileSizeKb": 1024,
  "dataRequests": {
    "dk.dtu.imm.sensible.battery": [
      {
        "interval": -1
      }
    ]
  }
}
```

After the configuration file has been added to the project, the library needs to be added. This can either be done by including the project as a sub-project as seen in section 3.2.2 on page 12, by compiling the library as a static library and adding header files, or by simply copying the source files into the app project.

Several frameworks are required by the library: `CoreMotion`, `CoreLocation`, `MobileCoreServices`, `SystemConfiguration`, `Security` and `Foundation`. They need to be added to the project as well.

The `OpenSense` class can then be accessed to control the library functionality as previously seen in the class diagram on Figure 3.2 on page 11.

The relationship between the app and the library can be seen on Figure 3.6 on the following page.

The app will primarily use the `OpenSense` class as an entry point to access all features, but will also be able to indirectly handle `OSProbe` instances using, e.g., the `- (NSArray*)availableProbes` method.

[2] See section 1.2 on page 2

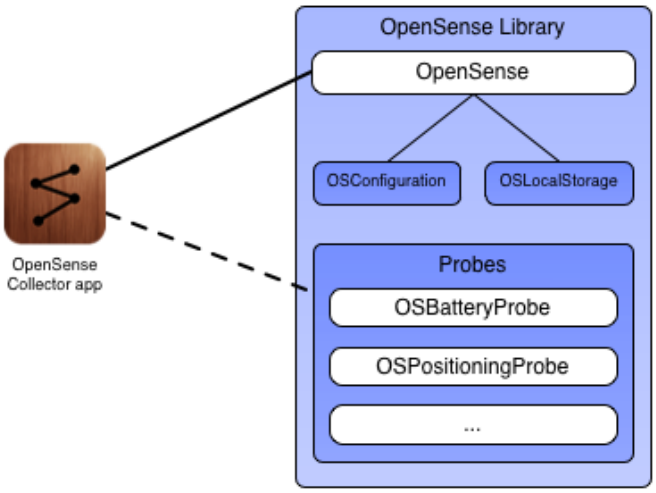


Figure 3.6: Using the OpenSense library from the OpenSense collector app

Simple usage of the library would just involve invoking `[[OpenSense sharedInstance] startCollector]` or `[[OpenSense sharedInstance] stopCollector]` using the singleton instance of `OpenSense`.

Note that `startCollector` returns a `BOOL`, depicting whether the collecting process was started. If the collecting process could not be started, it is most likely because a valid encryption key from the server has yet to be generated. If such, the app should retry again a few seconds later.

Server 3.4

A simple backend server has been implemented for testing purposes. This allows the whole data-flow of the app to be tested and tried.

It has been decided to build the server component as a `JSON`-based `RESTful` webservice over `HTTP(S)` [6]. This is a widely adopted standard that allows easy access and implementation from many different programming languages and interfaces.

Successful requests to the webservice will always return the `200 OK` HTTP status code. In case of a client error, the HTTP status code will be in the `4XX` range and return a `JSON` document with an `error` property. An example response of such a request can be seen in Listing 3.2.

Listing 3.2: Example error message

```
1 {
2   error: "Missing device_id parameter"
3 }
```

Three endpoints have been designed to support the app architecture.

3.4.1 `POST` /register

This endpoint is used to register a new device.

Parameters

device_id A unique device id, ideally a 128-bit UUID.

The device id is passed as a parameter and a server-generated encryption key is returned.

The app is responsible for safely storing the key and use it to encrypt data locally. It should not possible to retrieve the key again from the server later on.

Listing 3.3: Example response for `GET` /register

```
1 {
2   key: "G@Qf2a9F4EcFV1V%Gvs72Mw65nQG2WL05pJFJUQ4"
3 }
```

3.4.2 `GET` /config

This returns the current app configuration. The format is the exact same as the default app config shown in Listing 3.1 on page 15.

No parameters are passed, as the config is globally the same for all devices.

3.4.3 `POST` /upload

This upload endpoint handles storage of collected data.

Parameters

device_id The unique id of the users device.

file_hash `MD5` hash of the uploaded data.

data Array-based `JSON` document with collected data that is going to be stored.

The device id is passed so the server knows the origin of the data set and the actual `JSON` document is uploaded via the `data` parameter. As the `HTTP` protocol does not have any built-in mechanisms for integrity checking, it is important to pass a `MD5` file hash so the server can verify the uploaded data. If the integrity check fails, a `400 Bad Request` response will be returned, as seen in Listing 3.4 on the next page.

Listing 3.4: Example response error for `POST /upload`

```
{
  "error":"Integrity check failed"
}
```

Correctly uploaded data will return a `200 OK` response with a body, as seen in Listing 3.5.

Listing 3.5: Example successful response for `POST /upload`

```
{
  "status":"ok"
}
```

A more detailed explanation of the implementation of this is available in section 4.8 on page 37 in the Implementation chapter.

Data flow 3.5

An important part of the overall project architecture is designing and determining the way data is carried through the system.

It was initially decided to mimic and reuse the flow from the Funf platform. This was however eventually decided against for various reasons. The reason for this change is described in greater detail in a memo in the appendix, section C on page A-9.

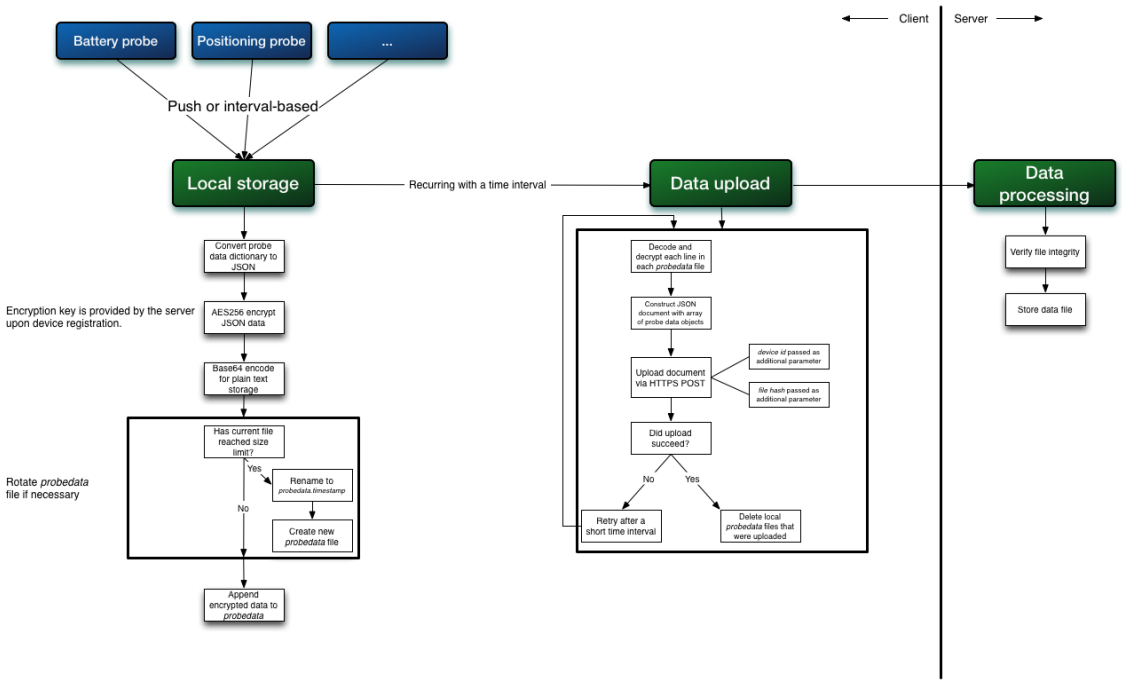


Figure 3.7: Data flow diagram

The current data flow has been designed to use open and widely adopted standards whenever possible, as well as keeping the data as secure as the requirements[3] allow.

The process can be divided into four main areas, as marked with blue and green boxes on Figure 3.7 on the facing page.

3.5.1 Step 1: Data collection (probes)

Data is collected using the currently active probes. This is marked as the blue boxes on Figure 3.7 on the preceding page. The "..." should be interpreted as additional possible probes. The specific data that a probe has collected is held in a key/value dictionary format.

The dictionary is sent to the local storage handler with a specific time-interval or pushed on-demand depending on the probe and its current configuration.

3.5.2 Step 2: Local storage

As memory is a limited and fragile resource (especially on mobile devices), it is important to store the collected data right away. This is marked as the first green box on Figure 3.7 on the facing page.

First, the dictionary is serialized by converting it to `JSON` format. This is done to meet the goal of using widely adopted standards, and also allows for good readability and easy storage & handling.

The `JSON` document may contain sensitive information, so before it is stored it is necessary to encryption the data. This is done using industry-standard `AES256` encryption with a pre-defined key[4]. Newer iPhone devices also support hardware-accelerated AES encryption[5].

The data is stored in a rolling log file format, which means that it is appended to a file which is rotated periodically to avoid an unmanageable large file. So before storing the entry, the size of the current probe data file is checked and a new one is created if necessary.

Lastly the probe data is appended to the current file. Data entries are stored in `ASCII` format, individually separated by the newline (`\n`) character. As the encryption algorithm returns binary data, it is necessary to `Base64` encode the entry first to get an `ASCII` representation of the encrypted data.

[3] As defined in section 2.1 on page 6
[4] See section 4.8.3 on page 40 for more information
[5] `http://images.apple.com/ipad/business/docs/iOS_Security_Mar12.pdf`

3.5.3 Step 3: Data upload

The data upload process is invoked within a specific time-interval defined by the current app configuration, as seen on the second green box on Figure 3.7 on page 18.

Before the data can be uploaded it must be prepared and modified from a format suited for storage to a format suited for uploading.

All archived `probedata` files are considered for uploading, the currently used file is however skipped to avoid any conflicts. As the `HTTP` protocol already supports encrypted data transfer on top of the `SSL/TLS` protocol, the locally stored data is decrypted first so it can be uploaded it in a standards-complaint way.

This is done by `Base64` decoding and decrypting each line in each considered file as seen in the looping operation on the figure. These individual `JSON` entries are then concatenated together to form an array-based `JSON` document.

An attempt is then made to upload the constructed document. If the upload times out or fails because of a server issue or inadequate internet connectivity, it will be retried again next time the data upload process is invoked. If it succeeds, the local data files will be deleted.

3.5.4 Step 4: Data processing

The last step takes place on the server. When the data has been fully uploaded, the server performs a `MD5` hash sum check on the data to verify its integrity. If the integrity check succeeds, the server can store the file in relation to the supplied device ID.

Chapter 4

Implementation

4.1 Logging

Logging is an important part of the library development and debugging process. This allows the developer to easily receive detailed information about the current actions being performed while the app is running.

Cocoa includes the `NSLog(...)` function for logging output to the console. However, it is not ideal for substantial use for several reasons. When the app is shipped, it is important to disable logging for performance and security reasons but this is not directly possible using `NSLog`. The default log function does also does not provide any information about the origin of the log entry.

These problems are solved by creating a custom `OSLog(...)` function that takes the same parameters as `NSLog`.

By taking advantage of the C preprocessor features, the function macro is dynamically defined during compile-time. The definition can be see in Listing 4.1.

Listing 4.1: `OSLog(...)` in `Common.h`

```
12 #ifdef DEBUG
13 #define OSLog(...) NSLog(@"%s %@", __PRETTY_FUNCTION__, [NSString stringWithFormat:
       __VA_ARGS__])
14 #else
15 #define OSLog(...) do { } while (0)
16 #endif
```

The `DEBUG` macro is defined automatically for debug builds by setting up a preprocessor macro for the build target, as seen on Figure 4.1.

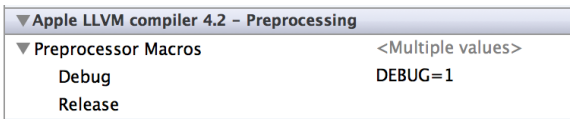


Figure 4.1: Setting preprocessor macros in Xcode

If the `DEBUG` symbol is present, the `OSLog` macro is defined to display output via the `NSLog` function, formatted with a prepended method name using the magic constant `__PRETTY_FUNCTION__`.

This will display easily identifiable log output like the following:
`-[OSProbe stopProbe] Positioning stopped`

```
-[OSProbe stopProbe] Battery stopped
-[OSProbe stopProbe] Proximity stopped
```

However if `DEBUG` constant is not defined, the `OSLog` macro will be defined as the bogus code `do {} while(0)` which essentially does nothing. This is needed to make calls to `OSLog` still work but not generate any output.

As the `OSLog` macro is used in many classes, it has been put it in a separate header file called `Common.h`. This file is automatically prepended to all source files by using the Xcode build systems' prefix header. In the library, this file is called `OpenSense-Prefix.pch`.

The special prefix header file is also usually pre-compiled to speed up the usual compile time.

Implementing the singleton pattern 4.2

As previously mentioned in the design chapter, all primary classes in the library follow the `singleton` pattern to prevent them for being instantiated more than once.

This is implemented using the `dispatch_once(...)` method in Grand Central Dispatch (GCD). An example of this implementation can be seen in Listing 4.2.

Listing 4.2: Singleton implementation for the `OpenSense` class

```
30 + (OpenSense*)sharedInstance
31 {
32     static dispatch_once_t pred = 0;
33     __strong static id _sharedObject = nil;
34     dispatch_once(&pred, ^{
35         _sharedObject = [[self alloc] init];
36     });
37     return _sharedObject;
38 }
```

First a `static` predicate object is created, this is used by `GCD` to distinct between dispatch calls. Next, another `static` variable is created to hold the instance of the class.

By using the `dispatch_once(...)` method, `GCD` guarantees that the class will only be instantiated exactly once[6]. This also makes the class instantiation inherently thread safe: If the dispatch function is called simultaneously from multiple threads, it will wait synchronously until the block has completed.

In all cases, the current instance of the class will then be returned.

[6] Grand Central Dispatch (GCD) Reference, 2013, http://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html

OpenSense 4.3

4.3.1 Device registration

When the `OpenSense` class is first initialized and the `- (id)init` method is called, the keychain is queried to verify that an encryption key is stored for the app. If that is not the case, the device registration process is initiated. An excerpt can be seen i Listing 4.3.

Listing 4.3: The `- (void)registerDevice` method in `OpenSense.m`

```
57 - (void)registerDevice
58 {
59     // Make sure that registration can not be called multiple times at once
60     if (registrationInProgress) {
61         return;
62     }
63     registrationInProgress = YES;
64
65     // Make HTTP request to register the device
66     AFHTTPClient *httpClient = [[AFHTTPClient alloc] initWithBaseURL:[OSConfiguration
           currentConfig].baseUrl];
67
68     NSDictionary *params = [NSDictionary dictionaryWithObjectsAndKeys:
69                             [[UIDevice currentDevice] uniqueGlobalDeviceIdentifier], @"
                                device_id",
70                             nil];
71     NSMutableURLRequest *request = [httpClient requestWithMethod:@"POST" path:@"/
           register" parameters:params];
72
73
74     AFJSONRequestOperation *operation = [AFJSONRequestOperation
           JSONRequestOperationWithRequest:request success:^(NSURLRequest *request,
           NSHTTPURLResponse *response, id JSON) {
75
76         // If a key was provided, store it in the keychain
77         if ([JSON objectForKey:@"key"]) {
78
79             NSError *error = nil;
80             if (![STKeychain storeUsername:@"OpenSense" andPassword:[JSON objectForKey:
                   @"key"] forServiceName:@"OpenSense" updateExisting:NO error:&error]) {
81                 OSLog(@"Could not store encryption key: %@", [error
                       localizedDescription]);
82             } else {
83                 OSLog(@"Device registered with key: %@", [JSON objectForKey:@"key"]);
84             }
85         }
86
87         registrationInProgress = NO;
88     } failure:^(NSURLRequest *request, NSHTTPURLResponse *response, NSError *error, id
           JSON) {
89         UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"Could not
               register device" message:@"The device could not be registered, please try
               again later." delegate:nil cancelButtonTitle:nil otherButtonTitles:@"OK",
               nil];
90         [alertView show];
91
92         registrationInProgress = NO;
93     }];
94     [operation start];
95 }
```

First, the `registrationInProgress` variable is used to make sure that method will not be executed while an operation is already in progress.

A request is then made to the webservice using the device's unique identifer in order to perform the actual device registration. The server will then recognize the device as being registered and return a unique encryption key that the app should use for local data storage.

If the connection failed or the server is having issues, the user is notified using an `UIAlertView`. The process will be initiated again when the user starts the collector process, so there is thus no reason to retry again automatically.

If an encryption key was successfully returned, it is stored in the iOS Keychain (the preferred way for secure storage on iOS[7]). It is important to note that the encryption key should never be changed. This is to prevent stored data from being corrupted in case of a lost key, and to make it easy to decrypt stored data for any reason.

4.3.2 Collector process

The `OpenSense` class acts as the main process manager for starting and stopping the collection process. This process is invoked by calling `- (BOOL)startCollector` or `- (void)stopCollector` on the class.

Starting the collection process

Before the collection process can be started, it is ensured that it isn't already running, it is also a requirement that the device has been correctly registered. This ensures that the server knows that the device exists and that the app has a valid encryption key for secure local storage.

As seen in Listing 4.4 the keychain is queried to detect whether the device registration process needs to be started. This is the same check as when the `OpenSense` class is first initialized, except that the method will return `NO` if a device registration is initiated.

Listing 4.4: Excerpt from `- (BOOL)startCollector` in `OpenSense.m`

```
99      // Make sure that the collector process is not already running
100     if (isRunning) {
101         return NO;
102     }
103
104     // Make sure that the encryption key is available
105     NSError *error = nil;
106     if (![STKeychain getPasswordForUsername:@"OpenSense" andServiceName:@"OpenSense"
            error:&error]) {
107         [self registerDevice];
108         return NO;
109     }
110
111     // Update state information
112     isRunning = YES;
```

[7] Keychain Services Programming Guide, 2013, http://developer.apple.com/library/ios/documentation/Security/Conceptual/keychainServConcepts/01introduction/introduction.html

```
113     startTime = [NSDate date];
114
115     // Start all probes
116     activeProbes = [[NSMutableArray alloc] init];
117     for (Class class in [self enabledProbes])
118     {
119         OSProbe *probe = [[class alloc] init];
120         [activeProbes addObject:probe];
121         [probe startProbe];
122     }
123
124     // Start timers
125     uploadTimer = [NSTimer scheduledTimerWithTimeInterval:[[[OSConfiguration
            currentConfig] dataUploadPeriod] doubleValue] target:self selector:@selector(
            uploadData:) userInfo:nil repeats:YES];
126
127     configTimer = [NSTimer scheduledTimerWithTimeInterval:[[[OSConfiguration
            currentConfig] configUpdatePeriod] doubleValue] target:self selector:@selector(
            refreshConfig:) userInfo:nil repeats:YES];
```

After it has been determined that the collection process can start, two state variables are set, which lets the class know that the process has been started and at what time.

Using the `- (NSArray*)enabledProbes` method, the app then loops through all of probes that have been enabled by the configuration. It then initializes a new instance of the class and calls their individual `- (void)startProbe`[8], and also builds a new array called `activeProbes` for these instances.

This new array serves two purposes: It holds the actual instances of the probes rather than referencing the class, it also distinguishes between actually running probes and probes that should be enabled. This is important as a configuration update may make `activeProbes` and `enabledProbes` differentiate from each other.

The last step is to start the upload and config update timers which handle periodically uploading stored data and periodically updating the local app configuration.

Stop the collection process

Stopping the collection process works very similar to starting the process, except that the actions are reversed.

As seen in Listing 4.5, the method first checks if the collection process is in fact running. If not, it returns and does not proceed further.

Listing 4.5: Excerpt from `- (void)stopCollector` in `OpenSense.m`

```
138     if (!isRunning) {
139         return;
140     }
141
142     for (OSProbe *probe in activeProbes)
143     {
144         [probe stopProbe];
145     }
146     activeProbes = nil;
```

[8] This method is described in greater detail in section 4.6 on page 33

```
147
148     isRunning = NO;
149
150     // Stop timers
151     [uploadTimer invalidate];
152     uploadTimer = nil;
153
154     [configTimer invalidate];
155     configTimer = nil;
```

Next, it loops through `activeProbes` and calls `- (void)stopProbe` on each of them. This allows the individual probes to gracefully shut down and stop collecting data.

Because we are using `ARC`, `activeProbes` is set to `nil` to let the compiler know that we are done using the object. This will ensure that the object is correctly released from memory at run-time.

The `isRunning` state variable is set to `NO`, so the class knows that we are not collecting data anymore and lastly the `- (void)invalidate` method is called on active timers to let `NSTimer` know that the timer should be stopped. As with the `activeProbes` the individual timer variables are also set to `nil`.

Configuration 4.4

The configuration manager resides in the `OSConfiguration` class.

This class is responsible for parsing and loading config files, determining default property values, and downloading and persisting updated configurations.

4.4.1 Loading the configuration

When the class is first initialized, the `- (void)load` method will be called. This method will parse and load the newest configuration into memory.

A selection of the implementation can be seen in Listing 4.6. Any downloaded config file will be stored in the root of the app's documents directory, so the method will first look there.

If a file was not found, it will use the default configuration from the app bundle as a fallback. This will most likely occur when the app has just been installed.

Listing 4.6: Excerpt from `- (void)load` in `OSConfiguration`

```
42 - (void)load
43 {
44     // First, try to load config file from the documents directory
45     NSString *documentsPath = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
           NSUserDomainMask, YES) objectAtIndex:0];
46     NSString *filePath = [documentsPath stringByAppendingPathComponent:@"config.json"];
47
```

```
48      if (![[NSFileManager defaultManager] fileExistsAtPath:filePath])
49      {
50          // Try to load local json config file instead
51          filePath = [[NSBundle mainBundle] pathForResource:@"config" ofType:@"json"];
52      }
53
54      NSData *data = [NSData dataWithContentsOfFile:filePath];
55
56      if (data)
57      {
58          // Parse json as dictionary
59          NSError *error = nil;
60          config = [NSJSONSerialization JSONObjectWithData:data options:kNilOptions error
                :&error];
61
62          if (!config) {
63              OSLog(@"Could not parse config: %@", [error localizedDescription]);
64
65              // Delete config.json in the documents directory if possible
66              [[NSFileManager defaultManager] removeItemAtPath:[documentsPath
                    stringByAppendingPathComponent:@"config.json"] error:nil];
```

With the path to the most recent `config.json`, the configuration file can now be loaded into memory and parsed using the built in `NSJSONSerialization` class. If the file was successfully parsed it will return an `NSDictionary` which is stored in the `config` variable in the class scope.

If the file could not be successfully parsed, it is most likely because a corrupted `JSON` file was released. We then need to delete the locally stored `config.json` to prevent the method from trying to load the file again and thus causing an infinite corruption loop.

4.4.2 Updating the configuration

The configuration manager is built to allow updates using a periodic timer that downloads the newest configuration file.

The `OpenSense` class is responsible for calling the configuration managers `- (void)refresh` method which invokes the update process.

Listing 4.7: The `- (void)refresh` method in `OSConfiguration`

```
71  - (void)refresh
72  {
73      // Try to download json config file
74      AFHTTPClient *httpClient = [[AFHTTPClient alloc] initWithBaseURL:[OSConfiguration
            currentConfig].baseUrl];
75
76      NSString *documentsPath = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
            NSUserDomainMask, YES) objectAtIndex:0];
77      NSString *filePath = [documentsPath stringByAppendingPathComponent:@"config.json"];
78
79      NSMutableURLRequest *request = [httpClient requestWithMethod:@"GET" path:@"/config"
            parameters:nil];
80      AFHTTPRequestOperation *operation = [[AFHTTPRequestOperation alloc] initWithRequest
            :request];
81      operation.outputStream = [NSOutputStream outputStreamToFileAtPath:filePath append:
            NO];
82
83      [operation setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *operation, id
            responseObject) {
```

```
84          [[OSConfiguration currentConfig] load]; // Reload the config file
85      } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
86          // Ignore failure. Will be retried later anyways.
87      }];
88
89      [operation start];
90  }
```

As seen in Listing 4.7 on the preceding page, the refresh method works by making a `HTTP(S) GET` request to the `/config` endpoint on the webservice.

It is then stored in the apps' documents directory, overwriting the existing `config.json` if any. The output is streamed to the file, which means that the whole response does not necessarily have to be loaded into memory before the file can be saved.

If the download was successful, the load method will be called as described in section 4.4.1 on page 27. This will parse the new configuration and load the new settings into memory.

If the request fails because of a server or internet connection issue, we ignore the failure as the refresh method will be called later on anyways because of the nature of periodic updating.

4.4.3 Accessing configuration properties

If a configuration is loaded, it is possible to query the values of the different configuration properties.

This is done using simple `getter` methods that returns the appropriate value using either the value set in the loaded configuration dictionary or using a hardcoded default value as a fallback. An example getter can be seen in Listing 4.8.

Listing 4.8: The `dataUploadPeriod` getter method in `OSConfiguration`

```
124 - (NSNumber*)dataUploadPeriod
125 {
126     if (!config)
127         return nil;
128
129     return [config objectForKey:@"dataUploadPeriod"] ? [config objectForKey:@"
            dataUploadPeriod"] : [NSNumber numberWithLong:kDefaultConfigUpdatePeriod];
130 }
```

The default constants are defined at the top of `OSConfiguration.m` as seen in Listing 4.9. This allows for an easy overview and interchangeability. The values are only used if no value is provided for the property in the current configuration file.

Listing 4.9: Default configuration constants

```
12 #define kDefaultVersion                0
13 #define kDefaultDataArchivePeriod      3 * 60 * 60 // 3 hours
14 #define kDefaultDataUploadPeriod       6 * 60 * 60 // 6 hours
15 #define kDefaultConfigUpdatePeriod     1 * 60 * 60 // 1 hour
16 #define kDefaultDataUploadOnWifiOnly   NO
17 #define kDefaultMaxDataFileSizeKb      2048 // 2mb
```

Utility functions are also provided, such as
`- (NSTimeInterval)updateIntervalForProbe:(NSString*)probeId` that returns the update interval for a specific probe using the current configuration.

As seen in Listing 4.10, this method will query the `dataRequests` property in the configuration for a probe with the specified identifier, if the probe does not exist, the method will return −1.

Listing 4.10: The `updateIntervalForProbe` method in `OSConfiguration`

```
140 - (NSTimeInterval)updateIntervalForProbe:(NSString*)probeId
141 {
142     if (!config)
143         return -1;
144
145     // Get probe data from config
146     NSArray *probeData = [[config objectForKey:@"dataRequests"] objectForKey:probeId];
147
148     // Check if probe and DURATION key exists first
149     if (!probeData || [probeData count] <= 0)
150         return -1;
151
152     NSDictionary *firstProbeData = [probeData objectAtIndex:0];
153
154     if (![firstProbeData objectForKey:@"interval"])
155         return -1;
156
157     return [[firstProbeData objectForKey:@"interval"] doubleValue];
158 }
```

If the probe was found in the configuration, the method will try to find the interval key and return the value of it as a `double`. If the key does not exist, it will also default to returning −1.

Another utility method is `- (NSArray*)enabledProbes` which works similary, but returns an array of the identifiers of enabled probes instead.

Local storage 4.5

The `OSLocalStorage` class receives data from the active probes and persists it to the device in a safe, secure and reliable manner. It also handles reading the data again at a later point.

The seeds of data received from probes are internally known as batches or data batches.

4.5.1 Saving batches

When a data batch is generated from a probe, the `saveBatch:fromProbe:` method is called. The arguments is a dictionary with the collected data and the identifier for the responsible probe.

As seen in Listing 4.11, some additional data is first added to the initial data dictionary. This is the probe identifier and the current timestamp in a `yyyy-MM-dd HH:mm` format.

Listing 4.11: Excerpt from `- (void)saveBatch:fromProbe:` in `OSLocalStorage`

```
43      NSMutableDictionary *jsonDict = [[NSMutableDictionary alloc] initWithDictionary:
            batchDataDict];
44      [jsonDict setObject:probeIdentifier forKey:@"probe"];
45      [jsonDict setObject:[dateFormatter stringFromDate:[NSDate date]] forKey:@"datetime"
            ];
46
47      // Convert to JSON data
48      NSError *error = nil;
49      NSData *jsonData = [NSJSONSerialization dataWithJSONObject:jsonDict options:
            kNilOptions error:&error];
50      if (!jsonData) {
51          OSLog(@"Could not serialize JSON data: %@", [error localizedDescription]);
52          return ;
53      }
54
55      // Encrypt data
56      NSData *encryptedJsonData = [jsonData AES256EncryptWithKey:[OpenSense
            sharedInstance].encryptionKey];
57      NSString *encryptedJsonDataStr = [encryptedJsonData base64Encoding];
58
59      // Rotate probe data file ifneedbe
60      [self logrotate];
61
62      // Append data to file
63      [self appendToProbeDataFile:encryptedJsonDataStr];
64
65      // Post "batch saved" notification
66      [[NSNotificationCenter defaultCenter] postNotificationName:
            kOpenSenseBatchSavedNotification object:jsonDict];
```

The `NSJSONSerialization` class is then used to serialize the dictionary to a `JSON` object. In case of a serialization error, we have to abort the save operation and drop the data. This will however most likely only happen in case of a programming error by the probe author.

The `JSON` object is then encrypted using `AES256` with the local device-specific encryption key and Base64 encoded for plain-text storage.

Before storing the encrypted data the `-(void)logrotate` method is called, this operation is explained in the following section.

After invoking the log rotation method, the encrypted data needs to be appended to the existing `probedata` file, this is done by calling `- (void)appendToProbeDataFile:` which is an internal helper method simply built for this purpose. A line break is added after the encrypted content, to distinguish each data batch from each other.

Finally, a notification is sent using the built-in notification center, this can be used by the app to automatically register and display a newly saved data batch.

Also note that the decrypted `jsonDict` is sent with the notification for easy access to the data that has been submitted, by using `NSNotificationCenter` the notification will

only be sent within the app sandbox[9] meaning that third-party apps will not be able to gain access to the collected data batch.

4.5.2 Log rotate

The log rotation method is an important part of the batch saving flow. The functionality and name of this method is heavily inspired by the log rotation concept, commonly found in unix-like operating systems.

The method ensures that the `probedata` file does not grow to large by archiving dated data into separate files.

An excerpt of the method can be seen in Listing 4.12, covering the main functionality. The file size of the current probe data is measured and compared against the `maxDataFileSizeKb` property in the configuration. Note that the file sizes are measured in bytes, this means that the configuration property needs to be multiplied by 1024 to be converted from kilobytes to bytes as well.

Listing 4.12: Excerpt from `- (void)logrotate` in `OSLocalStorage`

```
88      long fileSize = [[[NSFileManager defaultManager] attributesOfItemAtPath:currentFile
            error:nil][NSFileSize] longValue];
89      long maxFileSize = ([[OSConfiguration currentConfig].maxDataFileSizeKb longValue] *
            1024L);
90
91      if (fileSize > maxFileSize) { // If file is to big
92          OSLog(@"Rotating log file");
93
94          NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
95          [dateFormatter setDateFormat:@"yyyy-MM-dd'T'hh-mm-ss"];
96
97          NSString *newFileName = [NSString stringWithFormat:@"probedata.%@", [
                dateFormatter stringFromDate:[NSDate date]]];
98          NSString *newFile = [dataPath stringByAppendingPathComponent:newFileName];
99
100         // Rename current probedata file to probedate.CURRENT_DATETIME
101         NSError *error = nil;
102         if (![[NSFileManager defaultManager] moveItemAtPath:currentFile toPath:newFile
                error:&error]) {
103             OSLog(@"Could not rename file %@ to %@ - %@", currentFile, newFileName, [
                    error localizedDescription]);
104         } else {
105             [[NSFileManager defaultManager] createFileAtPath:currentFile contents:nil
                    attributes:nil]; // Create a new probedata file
106         }
107     }
```

If the probe data file size have reached the maximum limit, the currently active file is renamed to have the current timestamp appended. This could for example look like this: `probedata` → `probedata.2013-02-01T14-23-21`.

If the renaming succeeded, a new, empty `probedata` file is created.

[9] Posting a Notification, 2013, http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Notifications/Articles/Posting.html

OSProbe 4.6

4.6.1 Implementing inheritance

The `OSProbe` class is a polymorphic super class that defines the overall traits for all implemented probes. Programatically, it should be considered an abstract class as it can not be instantiated directly, it is however not possible to strictly enforce this at compile-time because of limitations in the Objective-C programming language.

It has been decided to overcome this by doing some extra run-time sanity checks. As seen in Listing 4.13, the constructor for `OSProbe` is using `NSStringFromClass` to get the name of the current instance of the class. An assertion is then used to verify that the app did not try to initialize the super class by itself.

Assertions are typically not included in *release* builds, this means that these extra sanity checks are only performed in development builds, helping the developer avoid obvious mistakes without running extra, unnecessary operations in a production environment. In the OpenSense Collector app and the OpenSense Library the `NS_BLOCK_ASSERTIONS` preprocessor macro is defined for release builds to let the compiler know that assertions should be stripped out.

If an assertion fails, it will throw an exception and output a message like the following to the console:

```
*** Assertion failure in -[OSProbe init], /.../OSProbe.m:21
```

Listing 4.13: The init method for `OSProbe`

```
17 - (id)init
18 {
19     if (self = [super init]) {
20         NSString *instanceClassName = NSStringFromClass([self class]);
21         NSAssert((![instanceClassName isEqualToString:@"OSProbe"]), @"This is an
               abstract class and should never be instantiated directly");
22     }
23
24     return self;
25 }
```

The same type of sanity checks is implemented to enforce that inherited probe classes are implementing the name, identifier and other meta-data methods. An example of this, enforcing implementation of the `+ (NSString*)name` method can be seen in listing 4.14.

Listing 4.14: The `+ (NSString*)name` in `OSProbe`

```
27 + (NSString*)name
28 {
29     NSAssert(NO, @"This is an abstract method and should be overridden");
30     return nil;
31 }
```

This works using the normal mechanics of inheritance, as this method is automatically called when the inherited class does not have a definition for it. It however also expects

the probe-implemented method to not call the `super` method of the instance, as the assertion would then obviously fail.

4.6.2 Handling interval-based updates

Determining update intervals

As previously described in the design chapter in section 3.2.3 on page 13, the individual probes are either pushing new data when it is available or being pulled for updates periodically. This behavior is determined by the `defaultUpdateInterval` probe property, which either returns the `kUpdateIntervalPush` constant (with a value of −1) for pushing data or a positive integer, depicting the number of seconds between pulls.

The current value of this property is determined by the `updateInterval` method, as seen in Listing 4.15. We try to get the update interval for the probe using the current configuration, if that however is not provided, the default probe-specific update interval is used instead.

Listing 4.15: Excerpt from `- (NSTimeInterval)updateInterval` in `OSProbe`

```
96 - (NSTimeInterval)updateInterval
97 {
98     // Get update interval from config
99     NSTimeInterval configInterval = [[OSConfiguration currentConfig]
           updateIntervalForProbe:[[self class] identifier]];
100
101    // If config did not provide an update interval, use the default probe interval
           instead
102    if (configInterval == kUpdateIntervalUnknown)
103    {
104        [[self class] defaultUpdateInterval];
105    }
106
107    return configInterval;
108 }
```

Implementing interval-based updates

The crucial update interval value is used when the probe is first started with the `- (void)startProbe` method.

The start operation will take care of creating a repeating `NSTimer` as seen in listing 4.16. This is done by calling the previously mentioned `updateInterval` method and verifying that an actual update interval time period has been set. The timer is only created if that is the case.

The `NSTImer` invokes the `- (void)saveData` method on the main thread. This method takes care of asking the probe for a new data batch and passing it on to the local storage. The timer is running on the main thread, as it is not expected that the data batch collection process should perform any time-consuming operations mainly because of the frequency of which the process can be invoked.

Listing 4.16: The - (void)startProbe method in OSProbe

```
51 - (void)startProbe
52 {
53     NSTimeInterval timerUpdateInterval = [self updateInterval];
54
55     OSLog(@"%@ started with %f update interval", [[self class] name],
           timerUpdateInterval);
56
57     if (timerUpdateInterval != kUpdateIntervalPush && timerUpdateInterval !=
           kUpdateIntervalUnknown)
58     {
59         updateTimer = [NSTimer scheduledTimerWithTimeInterval:timerUpdateInterval
               target:self selector:@selector(saveData) userInfo:nil repeats:YES];
60     }
61     else
62     {
63         updateTimer = nil; // The probe is pushing data instead
64     }
65 }
```

Likewise the - (void)stopProbe method makes sure to invalidate and remove the timer if it has been started.

It is therefore important that individual probes are following the standard Objective C convention by calling [super startProbe] and [super stopProbe] in their individual inherited implementations.

Uploading data 4.7

The local probedata files are first loaded using the OSLocalStorage class. This method call is asynchronous using an Objective C block as the callback, which also means that the initial call to uploadData is backgrounded and asynchronous.

An array of batches is passed when the callback block is executed, and per the parameters set to the fetchBatchesForProbe:skipCurrent:parseJSON:success method the batches are returned in a decrypted, unparsed JSON format.

We need to convert these individual data batches into a full array-based JSON document that can be uploaded to the server. One solution could be to parse the individual items to construct a new, full document, it has however been deemed more efficient to construct it using simple string operations instead.

First, the individual data batches are converted from NSData objects to actual strings, and then concatenated using comma separation as seen in Listing 4.17. This is simply done by using NSString's stringByAppendingFormat: followed by substringToIndex: to remove the last two characters (comma and a space). The data is then wrapped with square brackets to have the correct syntax of an array, to finalize construction of the full document.

Listing 4.17: Concatenating data batches in - (void)uploadData:(id)sender

```
226         for (NSData *lineData in batches) {
```

```
227             NSString *lineStr = [[NSString alloc] initWithData:lineData encoding:
                    NSUTF8StringEncoding];
228
229             if (lineStr) {
230                 jsonFile = [jsonFile stringByAppendingFormat:@"%@,", lineStr];
231             }
232         }
233
234         // We don't need to upload anything if no valid data was found
235         if ([jsonFile length] <= 0) {
236             return;
237         }
238
239         // Remove the last comma
240         jsonFile = [jsonFile substringToIndex:[jsonFile length] - 1];
241
242         // ...and add array brackets
243         jsonFile = [NSString stringWithFormat:@"[%@]", jsonFile];
```

It is now possible to assemble the three parameters that needs to be supplied for the `/upload` endpoint (as previously seen in section 3.4.3 on page 17). The `file_hash` parameter is defined by the `MD5` hash of the `JSON` document string, the `device_id` as determined by the `UIDevice` category helper and the actual `data`.

Listing 4.18: `HTTP POST` parameters for data uploading

```
252         NSDictionary *params = @{
253             @"file_hash": jsonFileHash,
254             @"device_id": [[UIDevice currentDevice] uniqueGlobalDeviceIdentifier],
255             @"data": jsonFile
256         };
```

An asynchronous `AFJSONRequestOperation` is then created and started to handle the actual data uploading, as seen in Listing 4.19. A callback is set for *success*, *failure* and *progress*.

Upload progress and failure is just logged to the console via `OSLog(...)`, it is not deemed necessary to visualize the uploading progress for the user and in case of an upload failure, we will just fail silently and try again next time the method is called.

It is verified that the upload succeeded by making sure that the returned `JSON` response has the **status: ok** parameter set.

Listing 4.19: The `HTTP` request operation in `- (void)uploadData:(id)sender`

```
262         AFJSONRequestOperation *operation = [AFJSONRequestOperation
                JSONRequestOperationWithRequest:request success:^(NSURLRequest *request,
                NSHTTPURLResponse *response, id JSON) {
263
264             if ([JSON objectForKey:@"status"] && [[JSON objectForKey:@"status"]
                        isEqualToString:@"ok"]) {
265                 OSLog(@"Data succesfully uploaded!");
266
267                 // Determine file path
268                 NSString *documentsPath = [NSSearchPathForDirectoriesInDomains(
                            NSDocumentDirectory, NSUserDomainMask, YES) objectAtIndex:0];
269                 NSString *dataPath = [documentsPath stringByAppendingPathComponent:@"
                            data"];
270
271                 // Find files in data directory
272                 NSArray *probeDataFiles = [[NSFileManager defaultManager]
                            contentsOfDirectoryAtPath:dataPath error:NULL];
```

```
273                 for (NSString *file in probeDataFiles) {
274                     if ([file hasPrefix:@"probedata"] && ![file isEqualToString:@"
                            probedata"]) {
275                         [[NSFileManager defaultManager] removeItemAtPath:file error:nil
                                ]; // Delete file
276                     }
277                 }
278             } else {
279                 OSLog(@"Could not upload collected data");
280             }
281         } failure:^(NSURLRequest *request, NSHTTPURLResponse *response, NSError *error,
                id JSON) {
282             OSLog(@"Could not upload collected data");
283         }];
```

If the upload was successfull and verified, we know for certain that the server now has the data stored. We can then delete local `probedata` files that are no longer used.

Note that the current implementation considers all probe files except the currently active one to be deleted. In rare circumstances where a `- (void)logrotate` operation is performed while data is being uploaded, this may result in deletion of the newly archived file that has not been saved remotely yet.

This problem could be resolved in the future by not allowing the log rotation method to be executed while an data upload operation is in progress, or by having `OSLocalStorage` keep track of the files that are used when the `fetchBatchesForProbe:skipCurrent:parseJSON:success` method is called so only appropriate files will ever be deleted.

Server 4.8

It has been decided to implement the backend server using Node.js, which is an event-driven, asynchronous software system based on `JavaScript`.

As the server was only implemented as a mock-up for testing purposes, it was possible to rapidly prototype the basic implementation using very few built-in Node modules. It is also fully stand-alone and doesn't require a separate web server, making the backend very portable and simple to run.

The full implementation of the server listener can be seen in Listing 4.20 on the next page. When a request is made to the server, the `url` module is used to parse the full request url in to separate path and query string parts. This is needed to identify the endpoint as well as the supplied data for the request.

If the request is made with the `POST` method, the data parameters needs to be retrieved first. This is done by using the `data` event to receive data chunks and the `end` event to determine when all data has fully been received.

In all cases, the `handleRequest(...)` function can now be called, to perform the appropriate action regardless of the `HTTP` request method that was used. The

querystring (qs) module is used to parse the query string or received post data in to easily-accessible JavaScript object notation.

If any unhandled error occurs, the try/catch block will make sure that a graceful response will be sent, using the appropriate 500 Internal Server Error code.

Listing 4.20: Creating the HTTP server listener, excerpt from server.js

```
11 var server = http.createServer(function (req, res) {
12     try {
13         // Parse request URL
14         var uri = url.parse(req.url);
15         var path = uri.pathname;
16
17         // If POST request, wait till' all data is received.
18         if (req.method === "POST") {
19             var data = "";
20
21             req.on("data", function(chunk) {
22                 data += chunk;
23             });
24
25             req.on("end", function() {
26                 handleRequest(path, qs.parse(data), res);
27             });
28         } else {
29             handleRequest(path, qs.parse(uri.query), res);
30         }
31     } catch (err) {
32         // In case of any exceptions, just output a 500 Internal Server Error status
               code
33         res.writeHead(500, {"Content-Type": "text/plain"});
34         res.end("Internal␣Server␣Error");
35     }
36 });
```

4.8.1 Input validation

To make the overall implementation easier, testable and stable it was decided to implement thorough input validation.

This is implemented in the handleRequest(...) function, for all endpoints as the first logic statement before any other operations.

An excerpt from the input validation for the /upload endpoint can be seen in Listing 4.21 on the facing page.

If a parameter is missing or is in an invalid format, a JSON response is generated, with a meaningful error message. The 400 Bad Request status code is also set in accordance with the HTTP protocol standards[6], letting the user know that invalid input was specified in the request.

The same method is used to generate a response if the integrity check of the uploaded data failed.

Listing 4.21: Validation for the `/upload` endpoint, excerpt from `server.js`

```
83          if (!data.device_id) { // Make sure that a device id is specified
84              res.writeHead(400, {"Content-Type": "text/json"});
85              res.end(JSON.stringify({
86                  error: "Missing device_id parameter"
87              }));
88          } else if (!data.file_hash) { // Make sure that the file hash is specified
89              res.writeHead(400, {"Content-Type": "text/json"});
90              res.end(JSON.stringify({
91                  error: "Missing file_hash parameter"
92              }));
93          } else if (data.file_hash.length != 32) { // Make sure that the file hash
                has correct length
94              res.writeHead(400, {"Content-Type": "text/json"});
95              res.end(JSON.stringify({
96                  error: "file_hash must be exactly 32 characters"
97              }));
98          } else if (!data.data) { // Make sure that the data is specified
99              res.writeHead(400, {"Content-Type": "text/json"});
100             res.end(JSON.stringify({
101                 error: "Missing data parameter"
102             }));
103         } else {
104             // Check integrity of uploaded data
105             var hash = crypto.createHash('md5')
106                 .update(data.data)
107                 .digest("hex");
108
109             if (hash != data.file_hash) {
110                 res.writeHead(400, {"Content-Type": "text/json"});
111                 res.end(JSON.stringify({
112                     error: "Integrity check failed"
113                 }));
114             } else {
```

4.8.2 Integrity checking

As previously mentioned, the `HTTP` protocol does not implement any integrity checking itself. It has therefore been decided to add the `file_hash` parameter to the `/upload` endpoint, containing a simple `MD5` hash of the uploaded data.

It has been decided to use the `MD5`-hashing algorithm as it allows input of an arbitrary size with a static output size of 128-bits (or 32 characters as a hexadecimal representation), it is also widely used and implemented on most software platforms.

`MD5` is no longer considered a secure hashing algorithm [4], that is however primarily due to collision vulnerabilities which is not a concern when strictly using it for integrity verification.

By hashing the `JSON` document, two representations of the data gets sent to the server. This allows the server to hash the document as well (as seen in Listing 4.21 on line 105-107) and compare the generated hash with the already supplied one, making it almost certain that the data has been uploaded fully uncorrupted.

4.8.3 Key generation

To meet the data recoverability requirements[10], it has been decided to generate and store the data encryption key server-side when a new device is registered.

This allows the server to keep track of devices and their encryption keys, so data can be manually retrieved and decrypted if needed.

The key generation function can be seen in Listing 4.22. A pre-defined list of seed characters are first defined, this list contains alphanumeric characters as well as some special characters.

Using a `for` loop, each character in the encryption key is selected randomly from the seed character list.

The `Math.random()` function returns a pseudo-random decimal number between 0 and 1, by multiplying that with the amount of seed characters a index within the correct range will be returned. By calling `Math.floor(...)` on the value, the decimal-part will effectively be removed, leaving a simple integer for the final index value.

The `do/while` loop on line 167-169 ensures that selected characters do not repeat, by generating a random index until the current index is not the same as the previous one.

Listing 4.22: Key generation function, excerpt from `server.js`

```
160 function generateKey() {
161     var validCharacters = '
            abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789@#$%';
162
163     var key = '';
164
165     for (var i = 0; i < 40; i++) {
166         var index;
167         do {
168             index = Math.floor(Math.random() * validCharacters.length);
169         } while (i != 0 && validCharacters[index] == key[key.length - 1]); // Prevent
                repeating characters
170
171         key += validCharacters[index];
172     }
173
174     return key;
175 }
```

4.8.4 Security

In this test implementation the server is not operating over `HTTPS`, allowing all traffic to be monitored and sniffed for debugging purposes.

In a production environment it is necessary to use a secure connection in order to protect the users data when it is being uploaded.

[10] As defined in section 2.1 on page 6

A good solution would be to generate a self-signed SSL certificate and distribute the public certificate with the OpenSense Collector App, in addition to enabling validation of the HTTPS connection in the library using this certificate.

The Node.js server could easily be modified to serve a secure connection by using the https module instead of http as well as specifying the SSL key and certificate.

A self-signed SSL-certificate solution would arguable be more secure than a normal HTTPS solution via a CA as such a solution would require trusting several third-parties. The self-signed solution is only doable because of this unique situation where we have full control of both the app and the server.

OpenSense Collector App 4.9

The collector app primarily acts as an interface on top of the OpenSense library. This means that only a few key features are implemented directly within the app.

4.9.1 Backgrounding

Being able to collect and process data in the background is a really critical feature. This part must be handled by the app, and is therefore implemented in the AppDelegate of the collector app.

To easily support all types of data sources, it has been decided to run the app continuously in the background when the collector is running. Other possible solutions have been discussed in a previous paper on this matter [3].

When the app is no longer in the foreground, the - (void)applicationDidEnterBackground: delegate method will be called as seen in Listing 4.23. We then check if the collector is running and create an infinite-running background task if that is the case.

It is necessary to keep the app alive, for the OpenSense library to continue running. This is done by launching a *dummy* thread with an infinite loop as seen in line 51-54.

Listing 4.23: Backgrounding, implemented in the AppDelegate

```
44 {
45     if ([[OpenSense sharedInstance] isRunning])
46     {
47         bgTask = [[UIApplication sharedApplication]
               beginBackgroundTaskWithExpirationHandler:expirationHandler];
48
49         // Start the worker task
50         dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
               ^{
51             while (true)
52             {
53                 [NSThread sleepForTimeInterval:10.0f]; // Free up the CPU
54             }
55         });
```

```
56      }
57  }
```

4.9.2 Visualization

It is possible to add visualization to the OpenSense Collector app, giving the user a graphical overview of data collected from a specific source.

As an example, a map has been added for the positioning probe, giving the user a better overview than the raw `latitude/longitude` coordinates that the "Collected data" tab provides.

The collected data is often very differentiated, so it has been decided that each probe should have their own separate visualization view controller.

Handling visualizers

The `Probes` tab that lists visualizers is handled by `ProbesViewController`. The `-(NSArray*)availableProbes` method in `OpenSense` is called when the controller is loaded, to store and display a list of all probes available probes.

When a probe is tapped we try to initialize the corresponding visualization view controller. This is done by constructing the appropriate name of the visualization class, as seen in Listing 4.24. Then the `NSClassFromString` method is used to retrieve the corresponding `Class` struct.

Listing 4.24: Dynamically instantiating a visualizer view controller in `ProbesViewController`

```
82
83      NSString *probeName = [[probes objectAtIndex:[indexPath row]] name];
84      NSString *className = [NSString stringWithFormat:@"%@ProbeDataViewController",
            probeName];
85      Class dataViewControllerClass = NSClassFromString(className);
86
87      if (dataViewControllerClass)
88      {
89          dataViewController = [[dataViewControllerClass alloc] init];
90      }
```

If it was possible to retrieve a valid `Class` struct, the actual class can now be dynamically instantiated and the visualizer can be shown. If it was not possible to instantiate a class with that name, the method just `return`s and no visualizer is shown for the probe.

Implementing a visualizer

As mentioned in the previous section, the visualization class has to conform to a specific naming scheme to be automatically available.

This naming scheme is as follows: *Name-of-the-probe* `ProbeDataViewController`.
As such, the visualizer for the positioning probe is called `PositioningProbeDataViewController`.

It is fully up to the individual class how the graphical visualization is implemented. The data for a given probe can be retrieved using `- (void)localDataBatchesForProbe:success:` in the `OpenSense` class as seen in Listing 4.25.

Listing 4.25: Loading probe data to be visualized

```
41    [[OpenSense sharedInstance] localDataBatchesForProbe:@"dk.dtu.imm.sensible.
          positioning" success:^(NSArray *batches) {
42        collectedData = batches;
43
44        [self showData];
```

4.9.3 Collected data

The app also provides a raw look into the actual data that has been collected, this is handled by the `CollectedDataViewController` which inherits from `UITableViewController`.

The `OpenSense` class has a full interface to all methods that the app would want to access. When the collected data controller is loaded, the `OpenSense` instance is called, to fetch locally stored data. This request is internally forwarded to the `OSLocalStorage` class and the data is then returned asynchronously using the accompanying `block` as seen in Listing 4.26.

Listing 4.26: Fetching local data batches in `CollectedDataViewController.m`

```
35    [[OpenSense sharedInstance] localDataBatches:^(NSArray *fetchedBatches) {
36        batches = [[NSMutableArray alloc] initWithArray:fetchedBatches];
37        [self.tableView reloadData];
38
39        // Remove loading view
40        [loadingView removeFromSuperview];
41        loadingView = nil;
42    }];
```

As the data needs to be loaded from the disk and then decrypted, the operation may take several seconds to complete. This is handled by the controller by showing loading indicator while the batches are being retrieved.

When the data batch array is returned, it is stored in the class scope and the `UITableView`'s reload method is called to invoke the UI update process. Finally, the loading indicator is removed again.

As mentioned earlier in section 4.5.1 on page 30, the notification center is used to send a `kOpenSenseBatchSavedNotification` when new data is stored. The collected data controller is listening for this notification, and adding new items to the top of the list dynamically as seen in Listing 4.27.

Listing 4.27: Adding a new data batch dynamically to the list in `CollectedDataViewController.m`

```
51 - (void)batchesUpdated:(NSNotification*)notification
```

```
52 {
53     NSDictionary *batch = [notification object];
54     [batches insertObject:batch atIndex:0];
55     [self.tableView reloadData];
56 }
```

Chapter 5

Test

Because of the complexity of the project and having a lot of moving parts, it was important to perform appropriate testing. This helps verifying and proving that the software works properly.

A lot of functional testing and field testing has been performed along the way. This has been done by constantly evaluating the newest version of the app and using it day-to-day, allowing the app to collect data and using the app to show the collected data as well.

This helped catching any glitches or bugs in the development process, and ensured great stability from the beginning.

It was however also necessary to perform some more systematical testing, namely unit testing which is described in this chapter.

Overview 5.1

Unit testing is a type of white-box testing where individual parts of the code base is systematically tested. This is often done by writing test cases and using a unit testing framework to run the test cases periodically.

The test cases interacts with pieces of the source code and `assert`'s that values are correctly set and that methods return the expected values.

In this particular project the `OCUnit` framework is used with Apple's `SenTestingKit` to perform the unit tests.

`SenTestingKit` defines several macros such as `STAssertTrue(...)`, `STAssertNil(...)`, `STAssertEqualObjects(...)` among others. The unit test results will give an overview of how many of these assertions were evaluated as `true`.

A test will be considered fully passed, if all of the assertions have been evaluated as `true`.

Test cases 5.2

It has been decided to focus unit testing on the library, namely the retrieval and storage process of local data as this is a very crucial and complicated part of the project.

This has been accomplished by creating the `LocalStorageTests` class which extends from the `SenTestCase` class from the `SenTestingKit` framework.

As with all test cases, the class has a `- (void)setUp` method which gets called before the test is executed and a `- (void)tearDown` method which gets called after. Before the test is executed, the path to the `Documents` directory is defined, as it is used throughout the test process.

Two test methods has been defined, that tests different ways of retrieving the stored data. These are described in the following sections.

Before each test method is run, the `Documents` directory is being cleared, to make sure that each of them are individually isolated and does not share any persisted data.

5.2.1 `testFetchAll`

First a new data batch is created using the `-(void)saveBatch:fromProbe` method in `OSLocalStorage`. Then `NSFileManager` is used to verify the filesystem to make sure that the correct folders and files exists.

The `- (void)fetchBatches:` is then used to retrieve the stored data again. It is however important to note that the fetch method is executed asynchronously, which makes it necessary to create a `semaphore` to make sure that the test method does not finish executing before the callback `block` has been called and fully tested.

In the callback `block`, the data batch and the properties of the batch are verified to make sure that they are set to the expected values.

5.2.2 `testFetchSingle`

As with the first test, a data batch is created with some pre-defined values.

The `-(void) fetchBatchesForProbe:skipCurrent:parseJSON:success:` method in `OSLocalStorage` is then used to retrieve the data batch again. This is differs from the first test as this is only fetching data from a specific probe.

First, we try to fetch data from a probe that does not exist, to make sure that no data batches are returned. Afterwards, the data batches are fetched with the same method but using the correct probe name and ensuring that a valid data batch is correctly returned.

As in the first test, `semaphores` are used to ensure that the test method doesn't complete until all callbacks have been processed.

Results 5.3

The goal of the unit test cases is ultimately that they always are passing. A screenshot of a full test can be seen on Figure 5.1.

Figure 5.1: Output from a successful unit test

The `OpenSenseTests` test suite is executed which involves setting up the `LocalStorageTests` class and running the test methods. Both test methods ends with the `passed` status, meaning that all assertions was evaluated as passed.

The last line in the test output indicates that 2 tests were executed with 0 errors, making the full test suite completed and passed.

Chapter 6

Conclusion

A full data sensing platform has been developed and tested, implementing the full data flow as depicted in Figure 3.7 on page 18. This includes collecting data, storing it securely on the local device and transferring it to a remote location.

A significant amount of time was spent developing the architecture and platform strategy, with less focus on actually implementing probes and visualizers but making it easy to extend the code base later on.

In the midst of the implementation process, it was realized that the initial plan to closely follow the `Funf` framework structure was no longer viable. This changed the development path significantly but did not cause any overall scope changes or delays.

The initial timeline, as seen in the Gantt chart on Figure 2.2 on page 8, was followed tightly. The implementation work was focused on in the first third of the project period, with the documentation process requiring full attention for the remainder of the project period.

All of the requirements categorized as "Necessary" and "Desirable" in the requirements overview in section 2.1 on page 6 has been fully met. Most of the "Nice to Have" requirements have been met aswell.

By basing this thesis project on previous research by the same author[11], it was possible to avoid a lot of common mistakes in the implementation process and make informed decisions from the start.

The final project outcome is a fully implemented framework that can be used with existing or new iOS apps, including an accompanying backend that allows the collected data to be stored remotely.

[11] "Special course: Research Concerning Development of a Data Sensing Platform for iOS"[3]

Third-party libraries

The full project was designed and developed with the use of the following third-party tools and libraries.

Name	URL	Author(s)
Test Pilot SDK	https://testflightapp.com/sdk/	TestFlight App, Inc.
AFNetworking	https://github.com/AFNetworking/AFNetworking	Gowalla
UIDevice+IdentifierAddition & NSString+MD5Addition	https://github.com/gekitz/UIDevice-with-UniqueIdentifier-for-iOS-5/	Aurora Apps
STKeychain	https://github.com/ldandersen/STUtils/blob/master/Security/	Buzz Andersen / System of Touch
NSData+AESCrypt	https://github.com/alexeypro/EncryptDecrypt/	Jim Dovey, Jean, Dave Winer, Kyle Hammond, Michael Sedlaczek
iOS App Icon Template	http://appicontemplate.com	Michael Flarup
GLYPHICONS	http://glyphicons.com	Jan Kovařík

Bibliography

[1] Sensible DTU. (danish) sensible dtu. http://www.sensible.dtu.dk, 2013.

[2] Funf. About funf. http://funf.org/about.html, 2013.

[3] Mathias Hansen. Research concerning development of a data sensing platform for ios. Technical report, DTU Informatics, 2012.

[4] Internet Engineering Task Force (IETF). Updated security considerations for the md5 message-digest and the hmac-md5 algorithms. http://tools.ietf.org/html/rfc6151, 2013.

[5] Apple inc. ios developer library. http://developer.apple.com/library/ios/navigation/, 2012.

[6] Network Working Group / W3C. Hypertext transfer protocol – http/1.1. http://tools.ietf.org/html/rfc2616, 2013.

Appendix

Table of contents

Abbreviations

AES256 Advanced Encryption Standard (with a 256-bit key)

ARC Automatic Reference Counting

ASCII American Standard Code for Information Interchange

Base64 A group of binary-to-text encoding schemes

CA Certificate authority

GCD Grand Central Dispatch

IDE Integrated Development Environment

JavaScript An interpreted scripting language, primarily used in web-browsers

JSON JavaScript Object Notation

MD5 MD5 Message-Digest Algorithm

MVP Minimum Viable Product

RESTful Representational State Transfer

singleton A design pattern that restricts the instantiation of a class to a single object.

SSL/TLS Secure Sockets Layer (SSL) & Transport Layer Security (TLS)

UUID Universally Unique Identifier

Xcode IDE developed by Apple for developing software for OS X and iOS

Appendix A

FUNF data flow issues memo

Introduction A.1

This short note outlines problems with data handling in the FUNF Framework that promotes bad practices or makes it difficult/impossible to provide compatibility with a new platform (iOS). It also provides solutions on how these issues can be improved.

Local storage and encryption A.2

FUNF stores collected data in `SQLite` database files. Databases are periodically archived to prevent them from growing too large.

The archival process uses a custom implementation that basically concatenates binary data and encrypts it using single DES encryption algorithm with `MD5` for hashing. Both `DES`[12] (even triple-DES) and `MD5`[13] has long been known to be very insecure and should only be used for legacy systems.

As archives only are encrypted, the currently used `SQLite` database is stored without any form of encryption or security.

By using a custom archiving format it makes it very cumbersome to provide compatibility with other platforms. This would be okay if there was a legitimate reason, but as the current solution both gives a lot of overhead (storing multiple database files) and is insecure by implementation it does not make a lot of sense to bring all these burdens to another platform.

[12] Security Implications of Using the Data Encryption Standard (DES), 2013, http://www.rfc-editor.org/rfc/rfc4772.txt

[13] Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms, 2013, http://tools.ietf.org/html/rfc6151

A.2.1 Proposed solution

Collected data should continued to be stored in a SQLite database but instead use an encryption layer on the driver-level that offers industry standard protection. A good solution to this could be SQLCipher[14] which is an open source extension to SQLite that uses 256-bit AES encryption and works with both Android and iOS. Never iPhone devices even support hardware accelerated AES encryption[15]. There also does not seem to be a reason to split the data into multiple databases as big files is not deemed to be an issue. The only thinkable reason would be that Android SD Cards formatted as FAT32 has a file size limit of $(2^32) - 1$ bytes[16] due to limitations in the file system.

Data format A.3

The database structure for local storage is laid out in two tables: file_info and data.

The file_info table provides information about the current database file including a unique uuid of the device that generated the file as well as creation time, this seems mostly like useful and relevant data.

The data table holds collected data, received from various probes. For each received batch a probe identifier, timestamp and JSON object is stored. The JSON data objects are not well designed and does not seem to follow any consistent conventions or standards. For example: Some keys are lowercase, some are uppercase and others has mixed casing. The probe identifier and timestamp is saved both in the database row and JSON object and thus promotes duplicate data for every single data batch.

In addition, storing JSON in a SQL database is not considered to be good practice unless there is a good reason for it (which could not be found).

Also, the FUNF config is stored in the database every time it is refreshed (even though it hasn't changed), this adds a lot of unnecessary data that later needs to be uploaded to the backend.

A.3.1 Proposed solution

Storing the data in a format that fits the relation database paradigm and structure would be a much neater solution. A proposed database schema could look like the following snippet.

[14] SQLCipher, 2013, http://sqlcipher.net

[15] Deploying iPhone and iPad - Security Overview, 2013, http://images.apple.com/ipad/business/docs/iOS_Security_Mar12.pdf

[16] Limitations of the FAT32 File System in Windows XP, 2013, http://support.microsoft.com/kb/314463

```
1  CREATE TABLE batches (
2    pk INTEGER PRIMARY KEY  AUTOINCREMENT  NOT NULL ,
3    created DATETIME,
4    probeIdentifier TEXT
5  );
6
7  CREATE TABLE batch_data (
8    pk INTEGER PRIMARY KEY  AUTOINCREMENT  NOT NULL ,
9    batchId INTEGER,
10   key TEXT,
11   value TEXT
12 );
```

Every time a probe has new data, an entry will be created in the batches table, each data entry would then be stored in the `batch_data` table with `batchId` as a foreign key.

Inconsistent sensor data A.4

Because of the nature of two entirely different operating systems, the probes will collect very different data that is not always able to be normalized. An example is the battery probe which on iOS returns a battery level and battery state (`UNPLUGGED`, `CHARGING`, etc.), on Android the battery state probe returns a number for the battery state instead, because the enum that holds the state has not been correctly normalized.

A.4.1 Proposed solution

Storing the collected probe data as readable objects with well-formed names and values. The normalization can then happen on the server side for the data that needs to be compared across the platforms. - Alternatively the collected iOS data would have to be stored exactly as the Android data and as such for example use the enum integer values from Android when storing the battery state.

Data upload A.5

In FUNF, data is uploaded to the backend using the existing archives. This means that the uploaded data will have a lot of overhead such as database schema, metadata and the vast amount of config updates (as described in the “Data format” section). It also means that the data would be uploaded uncompressed.

In addition, the uploading connection would also be insecure and prone to attacks when uploading over an unsecured http connection, considering the weak encryption used on the archives (as described in the “Local storage and encryption” section).

A.5.1 Proposed solution

The data should be uploaded in a universal format that is easily parsed by the server and supported by many platforms. This could for example be `JSON` or `XML` which is easily compressible using e.g. gzip over `HTTP`. Encryption could be handled by using `HTTPS` with a self-signed certificate and distributing the client certificate with the framework or app, this allows the library to easily validate the connection.

Appendix B

Timeline

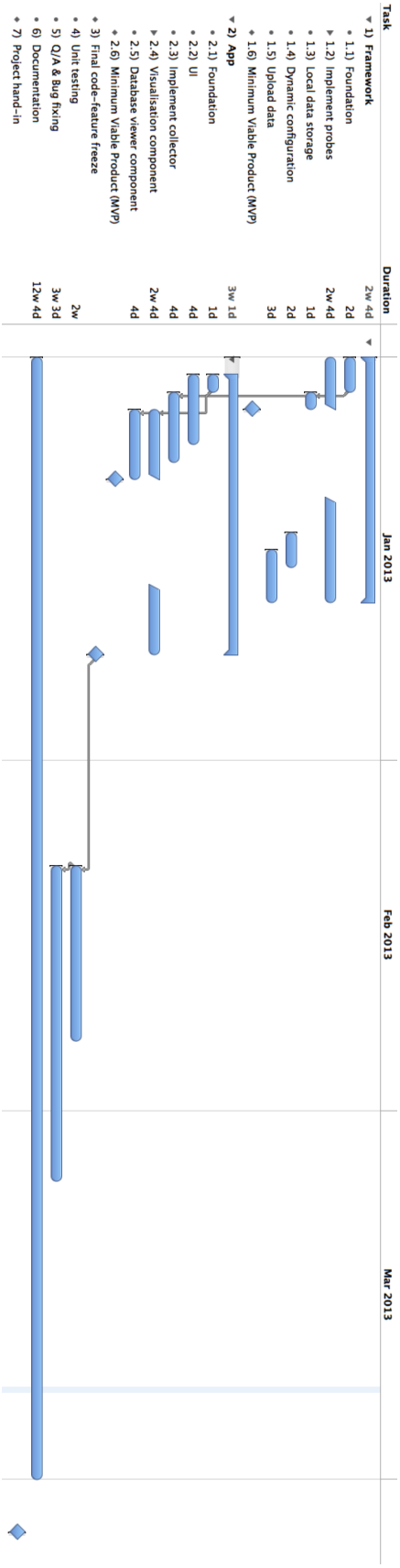


Figure B.1: Project timeline

Appendix C

OpenSense Collector app

Figure C.1: Status tab, depicting current collection status

Figure C.2: Probes tab, listing the different available probes

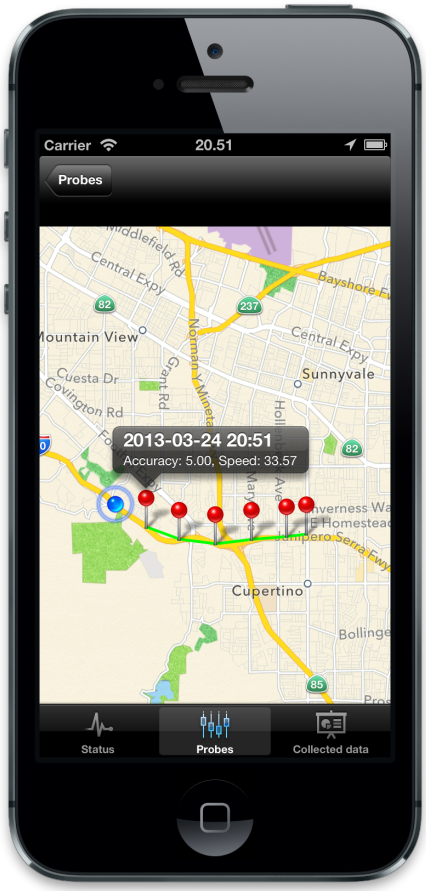


Figure C.3: Visualizing the data collected using the positioning probe

Figure C.4: Collected data tab, showing a raw list of data batches that have been collected

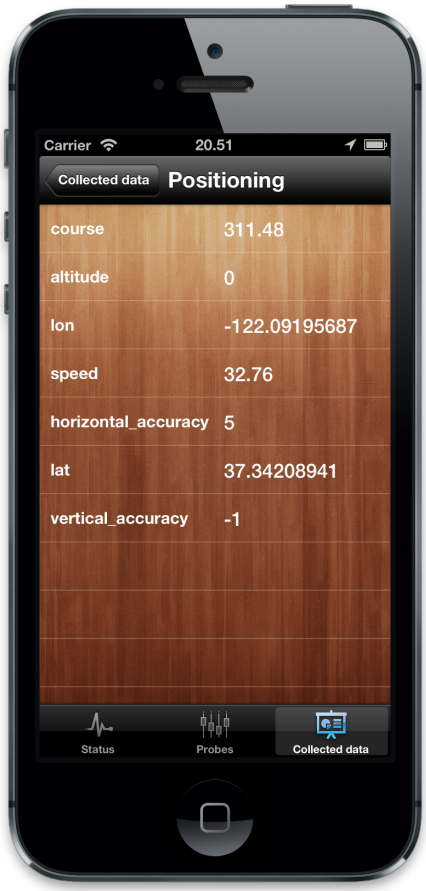


Figure C.5: Showing all collected data for an individual data batch