Technical University of Denmark

Special Course

Research Concerning Development of a Data Sensing Platform for iOS

Delivered by:

Mathias Hansen, s093478

DTU Informatics
Department of Informatics and Mathematical Modelling

Special Course

Supervisors: Jakob Eg Larsen & Sune Lehman
Delivery: December 11, 2012
The report contains 26 pages

Table of Contents

List of Figures

Listings

Chapter 1

Introduction

This project seeks to examine what opportunities and limitations the iOS platform provides for data sensing. This includes all possible parameters such as accelerometer data, location using GPS, social media data, phone calls, battery info, etc.

A special focus will be given on which new possibilities the just-released iOS 6 will bring, compared to iOS 5. Alternative methods will be used if needed, covering usage of private frameworks or APIs or external resources such as existing Wi-Fi infrastructure.

The project should also discuss solutions for storage and uploading of collected data to be used for later processing and analysis. All subsequent processing is however not covered by this project.

Furthermore the project will focus on anonymizing and securely collecting data, keeping users privacy in mind. This covers (but is not limited to) data encryption, secure data transfer and anonymization of the uploaded data so the user is no longer identifiable.

This research is the base of a later implementation of the platform and therefore needs to enlighten all possible challenges and opportunities that could come up in the development phase.

Chapter 2

Analysis

Data sources 2.1

It has been decided to consider sensing data from all data sources that can be relevant for further studies and is supported by the newest iPhone device and iOS software – which at the time of writing is the iPhone 5 running iOS 6.1.

For most of the collectible data, it may be possible to extract the same type of information from different data sources. This is often necessary in order to get the most accurate data while also keeping power-efficiency in mind, this is especially true for *positioning* the user. Determining the user's position using GPS is pretty accurate but also moderately battery-intensive. Thus, it can be necessary to look at alternatives such as Wi-Fi or cell tower triangulation which is more power efficient but also less accurate.

This is also applies for *social* interaction, which can be determined from multiple sources such a the call log or Facebook interactions. Here it is important to use several sensors when determining the user's social circle as frequency and amount of interactions with friends can vary a lot by interaction type.

Because of this common structure, the data sources has been split into multiple categories, covering all necessary inputs. These categories are: *positioning*, *social*, *motion*, *environment*, *device info* and *device interaction*. In this study, the aim is to gather as much data as possible within each category with a special focus on having a high accuracy rate, whenever possible.

A full overview of the selected data sources can be seen on figure 2.1 on the facing page.

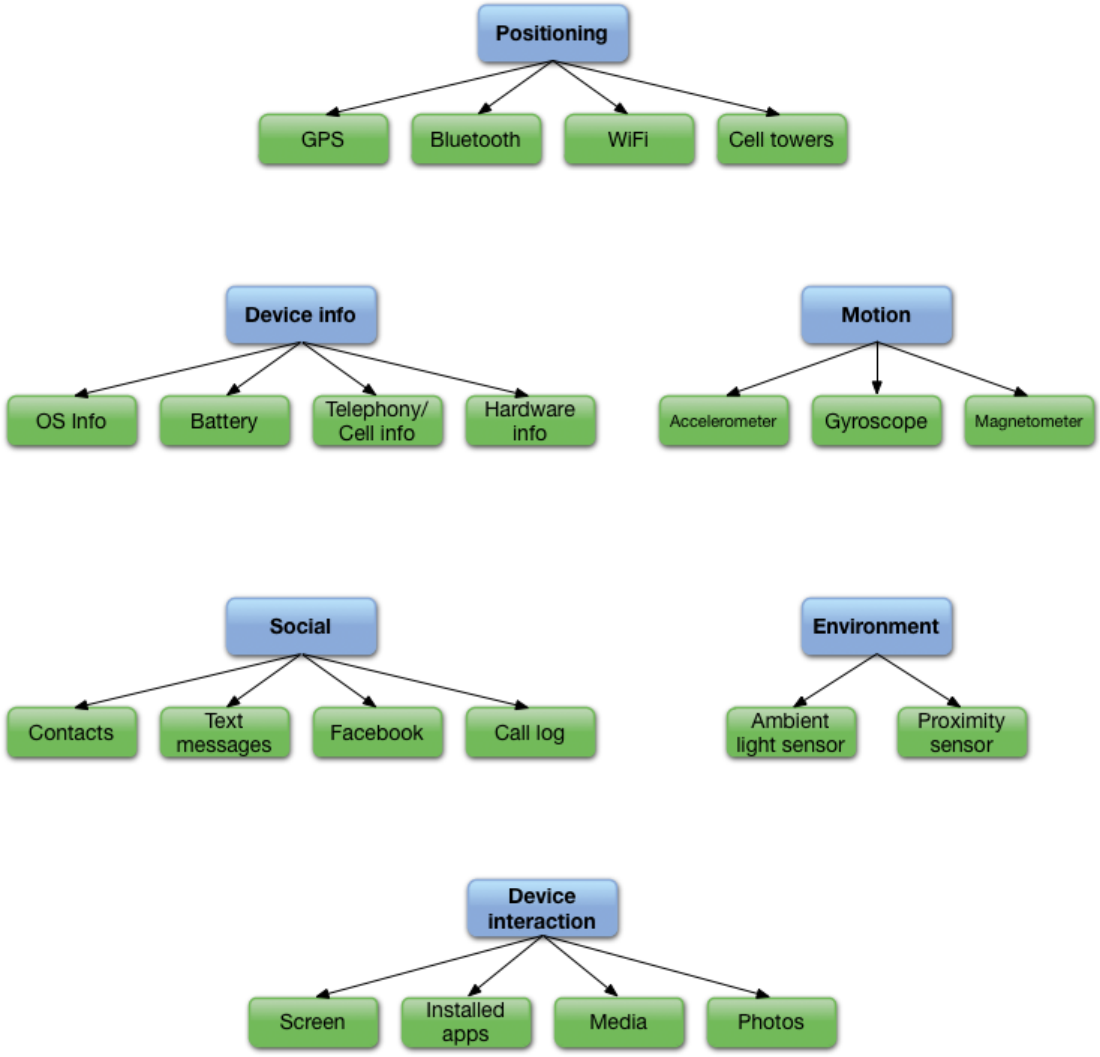

Figure 2.1: Overview of available data sources

Techniques & Scope 2.2

All data must be collected through a single iOS app and cannot rely on regular user interaction or other manual tasks. The application will be running in the iOS app sandbox, and will thus have several limitations predetermined based on the limitations of the iOS SDK.

The app cannot use any data sensing techniques that requires the user to jailbreak or otherwise modify their device. However, as the initial goal of the research project does not include App Store distribution, it is not necessary to use methods that adhere to the iOS App Store guidelines. This means that it is possible to use private frameworks or access data that usually wouldn't be allowed.

For some data categories it may even be necessary to use unconventional methods such as brute-forcing techniques or tricking the device by "lying" to the operating system.

Chapter 3

Implementation

This chapter discusses the actual techniques used to implement the general data sensing aspects of the app, enlightening what features the iOS SDK already supports and what techniques that needs to be used in order to circumvent the limitations of the iOS SDK.

Backgrounding 3.1

It is often not feasible to require the app to be running in the foreground in order to enable data sensing. Therefore, it is necessary to look at different techniques for sensing data in the background without any user interaction.

Background tasks can quickly become very battery-intensive and because of that, the iOS SDK imposes several limitations such as restricting the background time for apps as well as allowed CPU usage. The SDK however also provides features that allows the limited resources to be used more efficiently.

3.1.1 Infinite backgrounding

It is possible to keep the app infinitely in the background. This could be categorized as a lazy or inefficient method, but can also be necessary for some data sources where continuously polling is necessary.

When the user exits the app, a background thread can be started. The operating system will however close the running task after a 10 minute max execution time.

To circumvent this issue a background task is started with an asynchronous thread as seen on listing 3.1. An expiration handler is set as well, when the task is started. The expiration handler is called every time the background task expires, this allows the `block` to start the task over again and thus keep the thread running infinitely. The expiration handler implementation can be seen on listing 3.2. This approach is using techniques suggested by JackPearseless [2].

Listing 3.1: Starting the background task

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    bgTask = [[UIApplication sharedApplication]
        beginBackgroundTaskWithExpirationHandler:expirationHandler];

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        // Perform infinite-length work
    });
}
```

Listing 3.2: Expiration handler `block`

```
expirationHandler = ^{
    [[UIApplication sharedApplication] endBackgroundTask:bgTask];

    bgTask = UIBackgroundTaskInvalid;
    bgTask = [[UIApplication sharedApplication]
        beginBackgroundTaskWithExpirationHandler:expirationHandler];
};
```

3.1.2 Interrupt/Event-driven backgrounding

This is a more intelligent type of backgrounding that uses built-in features from the iOS platform to only perform background work when certain outside criteria are met. This allows the app to "wake up" when new data becomes available and only use CPU cycles when it is actually needed.

Event-driven backgrounding is however also limited to a certain number of services.

Location-based backgrounding

The `location` background mode was added in iOS 4 and allows an app to receive location updates in the background by either subscribing to significant location changes only[1] using the `startMonitoringSignificantLocationChanges` method in `CLLocationManager` or simply all location changes by setting a desired accuracy and distance filter and calling `startUpdatingLocation`.

Since iOS 5 it is also possible to define and monitor specific regions to receive a notification when a user approaches a predetermined location, this is also known as *geo fencing*. Region monitoring can be enabled using the `startMonitoringForRegion:` method.

For location-based backgrounding it is necessary to add a `UIBackgroundModes` key to *Info.plist* and add the `location` background mode as a value. This lets the operating system know that the app wants to receive location updates in the background.

Motion updates

It is possible to receive motion data from the device that is generated when the app is not in the foreground. This includes accelerometer and gyroscope readings (iOS 4) and magnetometer readings (iOS 5).

Motion updates are initiated by the `CMMotionManager` class using the method corresponding to the type of data that needs to be read (for example `startAccelerometer UpdatesToQueue:withHandler:`). An `NSOperationQueue` acts as a buffer for readings, making sure that background readings are delivered as soon as the app moves back to the foreground again. Listing 3.3 on the next page shows how the motion manager is used to subscribe to accelerometer updates with a specific interval and storing the sensor readings in a mutable array on the main thread.

As with location-based background it is still necessary to add the `UIBackgroundModes` key to *Info.plist*, for motion updates it however doesn't matter what background mode is chosen as app store distribution isn't considered initially for this project and app store

[1] iOS determines a significant location change as at least 500 meters with at least a five minute interval in between each update

guideline adherence thus is not required. For this implementation `location` or `voip` could for example be used as a supported background mode.

Listing 3.3: Listening for accelerometer updates using a background task

```
motionManager.accelerometerUpdateInterval = (double)1/5; // 5 Hz

[motionManager startAccelerometerUpdatesToQueue:accelerometerQueue withHandler:^(
    CMAccelerometerData* data, NSError* error) {
    dispatch_async(dispatch_get_main_queue(), ^{
        [accelerometerReadings addObject:data]; // Add reading to a mutable array
    });
}];
```

Positioning 3.2

The iOS platform provides a high-level interface for determining and following the user's position. To follow the users current location, a new instance of `CLLocationManager` is created and a desired location accuracy and distance filter is set. The location accuracy can be set to a value ranging from "Accurate to the nearest three kilometers." to the highest-level of accuracy. Location updates will only be provided if the accuracy meets the specified demand. The distance filter can be set to make sure that updates are only provided if the new locations delta distance is greater than the filter value.

`CLLocationManager` uses the devices built-in GPS as well as cell tower triangulation (if radio is available) and WiFi triangulation. By backing the location data by multiple sources it is possible to get a location fix as power efficient and fast as possible based on the desired location accuracy.

3.2.1 Using Bluetooth

If the goal is to detect nearby devices it is much more precise and efficient to utilize the Bluetooth LE compatible radio found in newer iOS devices (see table 3.1 for list of compatible devices). By using the `CoreBluetooth` framework it is possible to advertise and scan for services using the energy-efficient Bluetooth LE standard.

iPhone 4S and later	✓
iPod Touch 4th gen and later	✓
iPad 3rd gen and later	✓
iPad Mini	✓

Table 3.1: iOS Devices that supports Bluetooth LE

To advertise a service a device running iOS 6 is required. First, the service needs a predefined UUID, this UUID is unique to every type of service. The name of the service is then set, in this case the name is set to a predefined constant concatenated with a unique identifer, based on the device's network MAC address. This is done to have a consistent, unique identifier for each device, as the `CoreBluetooth` framework does not expose the

MAC addresses of Bluetooth devices. Finally, we call the `startAdvertising:` method on our instance of the `CBPeripheralManager` class. An example of this process can be seen on listing 3.4.

Listing 3.4: Advertising a Bluetooth LE service

```
- (void)startAdvertising
{
    if ([self isAdvertising])
        [self stopAdvertising];

    CBUUID *serviceUUID = [CBUUID UUIDWithString:kTransferServiceUUID];
    NSDictionary *advertisment = @{
        CBAdvertisementDataServiceUUIDsKey : @[serviceUUID],
        CBAdvertisementDataLocalNameKey: [NSString stringWithFormat:@"%@-%@",
            kServiceName, [[UIDevice currentDevice] uniqueGlobalDeviceIdentifier]]
    };
    [peripheralManager startAdvertising:advertisment];
}
```

For scanning, the `scanForPeripheralsWithServices:options:` method in `CBCentralManager` is used. It is possible to only look for specific services by specifying the pre-defined service UUID when calling this method. When devices that matches the criteria are discovered, the `didDiscoverPeripheral` callback in the delegate will receive the service info.

Listing 3.5: Scanning for a specific Bluetooth LE service

```
- (void)startScanning
{
    CBUUID *serviceUUID = [CBUUID UUIDWithString:kTransferServiceUUID];
    NSArray *services = [NSArray arrayWithObject:serviceUUID];
    NSDictionary *options = [NSDictionary dictionaryWithObject:[NSNumber numberWithBool
        :NO] forKey:CBCentralManagerScanOptionAllowDuplicatesKey];
    [centralManager scanForPeripheralsWithServices:services options:options];
}
```

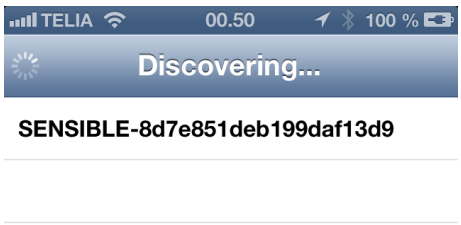

Figure 3.1: Example app that has found a nearby device

3.2.2 Using custom WiFi triangulation

When doing indoor positioning, WiFi has a clear advantage compared to GPS and cell tower triangulation. This is especially because a WiFi access point structure per-design is built to provide signal strength in all areas of a building.

For the DTU Lyngby campus, positioning using WiFi access points could be an ideal solution as the area is covered with official access points with known physical locations. This would also have a big advantage over Apple's official WiFi triangulation methods, as our data basis would be much more precise.

WiFi triangulation works by looking for nearby SSIDs and determining signal strength for each, by using this data it is possible to establish a fairly precise location of the device. Pre-iOS 5 it was possible to use the private `WifiManager` bundle to scan for wireless networks, in iOS 5 this bundle however was removed and replaced with the private `IPConfiguration` bundle which has similar features.

The new bundle however requires superuser privileges to be able to perform a WiFi scan, this means that it is not possible to use the framework with a device that hasn't been jailbroken and thus this technique does not meet the requirements for this project.

Motion 3.3

iOS devices contains various motion-related sensors that all are available for probing. A full list of compatible devices can be seen in table 3.2.

Device	Accelerometer	Gyroscope	Magnetometer
iPhone 4 and later	✓	✓	✓
iPhone 3GS	✓	–	✓
Original iPhone	✓	–	–
iPad Mini	✓	✓	✓
iPad 2 and later	✓	✓	✓
iPad (1st generation)	✓	–	✓
iPad Mini	✓	✓	✓
iPod Touch (4th generation)	✓	✓	–
iPod Touch (2nd and 3rd generation)	✓	–	–

Table 3.2: Motion sensor support for iOS devices

An accelerometer – also called a g-force sensor – measures gravity acceleration on all three axis, making it possible to determine what orientation the device has and how much is tilted in the 3d space. The gyroscope works much like an accelerometer but has some important subtle differences. While the accelerometer measures linear acceleration, the gyroscope only measures the change in rotation on an axis. Lastly, the magnetometer or digital compass measures the strength and direction of magnetic fields and can thus be used to find the users current heading.

The `CoreMotion` framework provides high-level access to all motion sensors using the `CMMotionManager` class. It is often useful to use multiple sensors together when developing motion-sensitive apps, to increase the reliability and efficiency of the readings. An example of using the framework can be seen in section 3.1.2 on page 7.

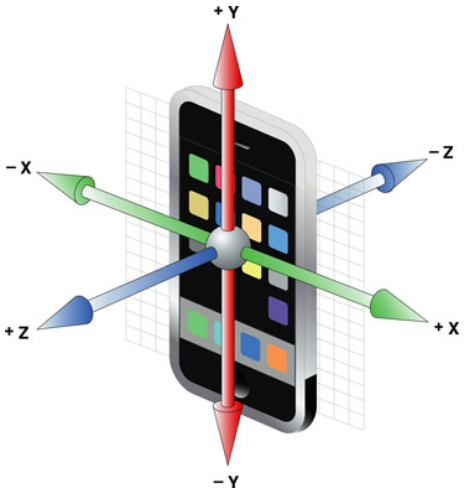

Figure 3.2: Orientation of accelerometer axis (Illustration © Apple Inc., 2010)

Environment 3.4

All iOS devices comes with an ambient light sensor that is internally used for adjusting the screens brightness if "Auto-Brightness" is enabled. This for example makes sure that the screen brightness automatically gets reduced in low-light situations.

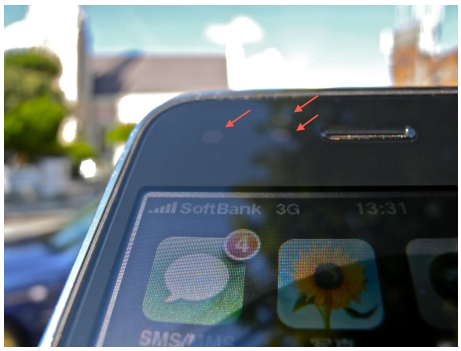

Figure 3.3: Location of ambient light (left) and proximity sensors (right) on iPhone 3GS (Illustration by raneko, 2009, http://www.flickr.com/photos/raneko/)

Most devices are also equipped with a proximity sensor (the two dots to the right in figure 3.3) that allows the operating system to turn off the display and touchscreen when the device is brought near the face during a call. This is done to save battery life and prevent inadvertent inputs from the users face or ears. A full list of device compatibility can be seen on table 3.3.

Device	Ambient light sensor	Proximity sensor
iPhone	✓	✓
iPod Touch	✓	–
iPad	✓	✓

Table 3.3: Environment sensor support for iOS devices

Ambient light sensor

This sensor was previously accessible through the private `IOKit` framework, which provided low-level I/O access to device hardware components. As such, it is no longer possible to access the ambient light sensor on a newer iOS device that is not jailbroken[3].

Proximity sensor

The proximity sensor returns a boolean state value that determines whether the device is close to the users face/ear or not. To monitor when the proximity state changes, the `setProximityMonitoringEnabled:` method in the global `UIDevice` instance is called. It is then possible to add an observer on *UIDeviceProximityStateDidChangeNotification*, as seen in listing 3.6. When a notification is received, it is then possible to probe the proximity state as shown in listing 3.7.

Listing 3.6: Probing the proximity sensor

```
[[UIDevice currentDevice] setProximityMonitoringEnabled:YES];
[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(
    proximityStateChanged:) name:@"UIDeviceProximityStateDidChangeNotification" object:
    nil];
```

Listing 3.7: Handling proximity sensor value changes

```
- (void)proximityStateChanged:(NSNotificationCenter *)notification
{
  if ([[UIDevice currentDevice] proximityState] == YES)
    NSLog(@"Device is close to user.");
  else
    NSLog(@"Device is not close to user");
}
```

Social 3.5

On a modern smartphone device, there are many ways of interaction socially, which opens up several opportunities for determining and mapping the users social network and relations. In this section, we are looking at some of the most important communication channels.

3.5.1 Contacts

With the users permission, it is possible to retrieve their full address book including creation and modification date for contacts. Contacts are accessed through the `AddressBook` framework.

It is also possible to read the creation and modification date of items in the address book, to determine which contacts that were recently created.

3.5.2 Text message & Call log

This information is stored in two separate SQLite databases, in */private/var/mobile/Library/SMS/sms.db* and */private/var/wireless/Library/CallHistory/call_history.db*.

Unfortunately, as all iOS apps are sandboxed, it is not possible to access files that live outside the sandbox. It is therefore not possible to access the text message database or call log with a device that has not been jailbroken.

3.5.3 Facebook

It can be very beneficial to look at the users social relations through a social network such as Facebook. Fortunately, it is fairly straightforward to allow users to connect through Facebook using the official Facebook SDK[2].

After setting up the Facebook SDK and linking to the Facebook library, the user can be authenticated by using `openActiveSessionWithReadPermissions:allowLoginUI:completionHandler:` in the `FBSession` class. In this method call it is also possible to request specific permissions in order to be able to retrieve the desired data.

When the user has authenticated and a session is established we can access the `accessToken` property for the active `FBSession` instance, in order to retrieve an OAuth access token that can be used by e.g. an external server to retrieve relevant Facebook data when needed.

Device info 3.6

The `UIDevice` class provides a singleton instance with `[UIDevice currentDevice]` that exposes several properties, describing a lot of relevant device info.

3.6.1 OS Info

name The user-specified device name, e.g. “Mathias Hansen’s iPad 3”

systemName The name of the operating system, seems to always be “iPhone OS”

systemVersion The installed version of iOS, e.g. “6.0”

[2]Facebook SDK for iOS, `https://developers.facebook.com/ios/`

3.6.2 Hardware info

model The current device model. Returns either "iPhone", "iPod touch" or "iPad"

userInterfaceIdiom The type of user interface that the current device uses. `UIUser InterfaceIdiomPhone` or `UIUserInterfaceIdiomPad`

3.6.3 Battery info

batteryMonitoringEnabled This boolean determines whether battery monitoring is enabled, this allows the app to be notified of changes to the battery state.

batteryLevel Returns a value between 0.0 (fully discharged) and 1.0 (100% charged), note that battery monitoring must be enabled, or else this property will return -1.0

batteryState The current charge state for the battery, returned as an enum constant. Can be any of: unknown, unplugged, charging, full.

3.6.4 Telephony/Cell info

Using the `CoreTelephony` framework, it is possible to access various information about the currently used cell carrier (if any) as well as monitoring active calls.

Carrier info can be retrieved by allocating an instance of the `CTTelephonyNetworkInfo` class and accessing the `subscriberCellularProvider` property, the returned `CTCarrier` object can then be queried to access various carrier data such as carrier name, carrier's country and network code, ISO country code and whether VOIP[3] is supported.

Current call info can be accessed by creating an instance of the `CTCallCenter` class and accessing the `currentCalls` property, which returns an `NSSet` of `CTCall` objects. Each call object contains the current call state (dialing, incoming, connected, disconnected) as well as an unique id for the cellular call. Unfortunately the id does not have a relation to the called id, and can thus not be used to track frequency of calls to specific phone numbers.

The `CoreTelephony` framework also provides some private methods that allows retrieval of the phone's IMEI[4], it has however not deemed possible to access this data without jailbreaking the device.

[3] Voice Over IP
[4] International Mobile Station Equipment Identity

Device interaction 3.7

3.7.1 Screen

It is possible to get properties of the device's screen by calling `[UIScreen mainScreen]`, the screen dimensions are then accessible through the `bounds` property.

Another interesting data point is the `scale` property, that determines whether the operation system scales the screen to provide high-resolution graphics. For now, this can mostly be used to determine whether the device is equipped with a retina display.

Newer iOS devices also supports external displays, the properties of these can be accessed through `[UIScreen screens]`.

3.7.2 Installed apps

It is not directly possible to get a list of installed apps through the operating system, it is however possible to use multiple techniques to get a pretty good data sample.

Most apps registers a url handler (e.g. `fb://` for Facebook). By using the `canOpenURL:` method in the `UIApplication` class, it is possible to detect whether the URL can be opened and thus whether the app is installed, as seen in listing 3.8.

Listing 3.8: Detecting if a given URL can be opened

```
BOOL isAppInstalled = [[UIApplication sharedApplication] canOpenURL:[NSURL
    urlWithString:@"fb:"]];
```

The downside to this method is however that it requires an existing database of app names mapped to URL handlers. It is possible to find existing databases available online, for example `http://handleopenurl.com` that also provides a `XML` API that can be used to scrape the needed data into a local database.

It is also possible to detect running processes (and therefore apps), by using the `sysctl` kernel function, the example shown in listing 3.9 outputs running processes with their process id's. Note however that this also display system processes and will thus have to be filtered for a clean data-output.

Listing 3.9: Detecting running processes, based on approach by Vladimir, `http://stackoverflow.com/a/4312705` (2012)

```
size_t size;
struct kinfo_proc *procs = NULL;
int status;

int mib[4] = { CTL_KERN, KERN_PROC, KERN_PROC_ALL, 0 };

status  = sysctl(mib, 4, NULL, &size, NULL, 0);
procs   = malloc(size);
status  = sysctl(mib, 4, procs, &size, NULL, 0);
```

```
if (status == 0) {
    if (size % sizeof(struct kinfo_proc) == 0) {
        int nprocess = size / sizeof(struct kinfo_proc);
        if (nprocess) {
            for (int i = nprocess - 1; i >= 0; i--) {
                NSString * processID = [[NSString alloc] initWithFormat:@"%d", procs[i
                    ].kp_proc.p_pid];
                NSString * processName = [[NSString alloc] initWithFormat:@"%s", procs[
                    i].kp_proc.p_comm];

                NSLog(@"%@: %@", processID, processName);

                [processID release];
                [processName release];
            }
            free(procs);
        }
    }
}
```

13159: GroupMe	13790: kbd	13793: AppleIDAuthAgent	18218: MobileSlideShow
18720: Facebook	19906: Maps	20046: xpcd	20107: Dropbox
20278: AppStore	20281: ubd	20306: appmap	20313: sociald

Table 3.4: Snippet of example output

A combination of these two methods, should give a fairly good overview of installed and used apps on the user's device.

3.7.3 Media/Photos

With the users permission, it is possible to access all the users photos/videos using the `AssetsLibrary` framework. Using the `MediaPlayer` framework it also possible to access the users iTunes library which holds access to music, podcasts and so forth.

For all assets it is possible to query the source files directly both also let the user pick items through a built-in view controller. For the iTunes library, this is the `MPMediaPickerController` and likewise the `UIImagePickerController` for photos/videos.

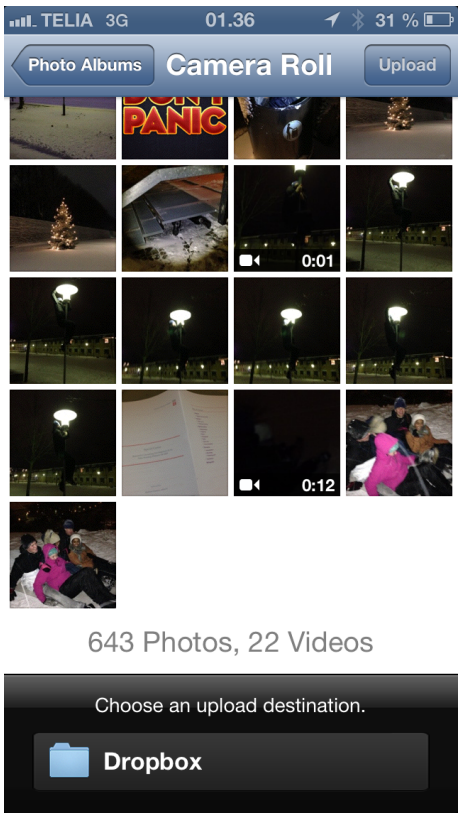

Figure 3.4: Example of an `UIImagePickerController` in the Dropbox app

Chapter 4

Discussion

Data storage 4.1

The collected data must be stored locally while waiting to be uploaded to an external web service. The most organized and convenient storage solution would be a SQLite database, stored in the app's private, sandboxed `Documents` folder. This database file would only be accessible to the app itself and to the user's computer when syncing through iTunes.

This type of storage should be secure enough to meet most privacy concerns, but if deemed necessary, it is also possible to encrypt the full SQLite database. A good solution to this is SQLCipher[5] which is an open source extension to the SQLite software that provides transparent 256-bit AES encryption of database files. A password keyphrase is then needed to read and write data in the database.

The recommended practice for defining a keyphrase would be to generate a long and secure passphrase with the first access to the database, and store it in the iOS keychain. This makes sure that the key is never stored within the app bundle or binary. A more secure, but less convenient approach, would be to let the user define a passphrase, and prompt for it when the database needs to be accessed.

[5]More information about SQLCipher can be found `http://sqlcipher.net/`

4.1.1 Anonymization

When working with a lot of data collected from a users device it is important to make sure that their privacy is not violated. A lot of personal information from the data will not be visible or relevant for data scientists and researchers that later needs to analyze the data, it is however still important to keep the integrity of the data so it is possible to create maps and connections for the collected information.

As such, it is important to look at ways to anonymize the collected data, already before storing it in the local SQLite cache. A good example for this would be the user's address book. The social media app: Path, sparked a lot of controversy earlier in the year, when a user discovered that the app was uploading the device's full address book without direct consent from the user[6].

For our purpose, a better solution would be to only look at data fields in the address book that is absolutely relevant to the research. This could for example be e-mail addresses and phone numbers. These can easily be hashed, so it is impossible to read the actual data but still keep track of e.g. the frequency of which a person occurs in people's address books.

The same technique can be used for e.g. installed apps or certain Facebook information.

Uploading data 4.2

Collected data should be uploaded to an external server in regular intervals. This makes sure that the data doesn't take up to much storage on the device, while keeping a steady data flow to the data-collecting backend.

Before uploading the data it is necessary to decrypt the SQLite database (if encryption has been used) and then normalize it to a format that is accepted by the server. It is also important to make sure that the data is securely uploaded, this is simply done by using the `https` protocol transferring. This both prevents a malicious person from sniffing the traffic and also secures the integrity of the data.

It should be noted that it is important to verify the `SSL` certificate on the server to make sure that no third-party has tampered with the connection. But it is still possible for a malicious person to exchange the `https` server with his own by creating his own Certificate Authority (CA) and using it to create a certificate for the server and then storing the CA's root certificate in the keychain on the iOS device.

This can be prevented by creating a self-signed certificate and distributing the password protected public certificate with the app app, and then validating the `https` connection using the stored certificate.

[6] Source: http://mclov.in/2012/02/08/path-uploads-your-entire-address-book-to-their-servers.html

Chapter 5

Conclusion

All relevant data sources on iOS have been examined and it has been determined that most of the sensors were fully available for probing.

It can also be concluded that the security on the platform has been tightened during the last several releases, causing a few major data points such as call log and text messages database to no longer be available in the newest iOS versions.

The iOS software is deemed to be a powerful and fairly supportive platform for data sensing, that meets most of the demands for data gathering and analyzation from a research perspective.

Taking figure 2.1 on page 3 from the Analysis chapter and the full research conducted in this project into consideration, a complete overview of accessible sensor can be seen on figure 5.1 on the next page.

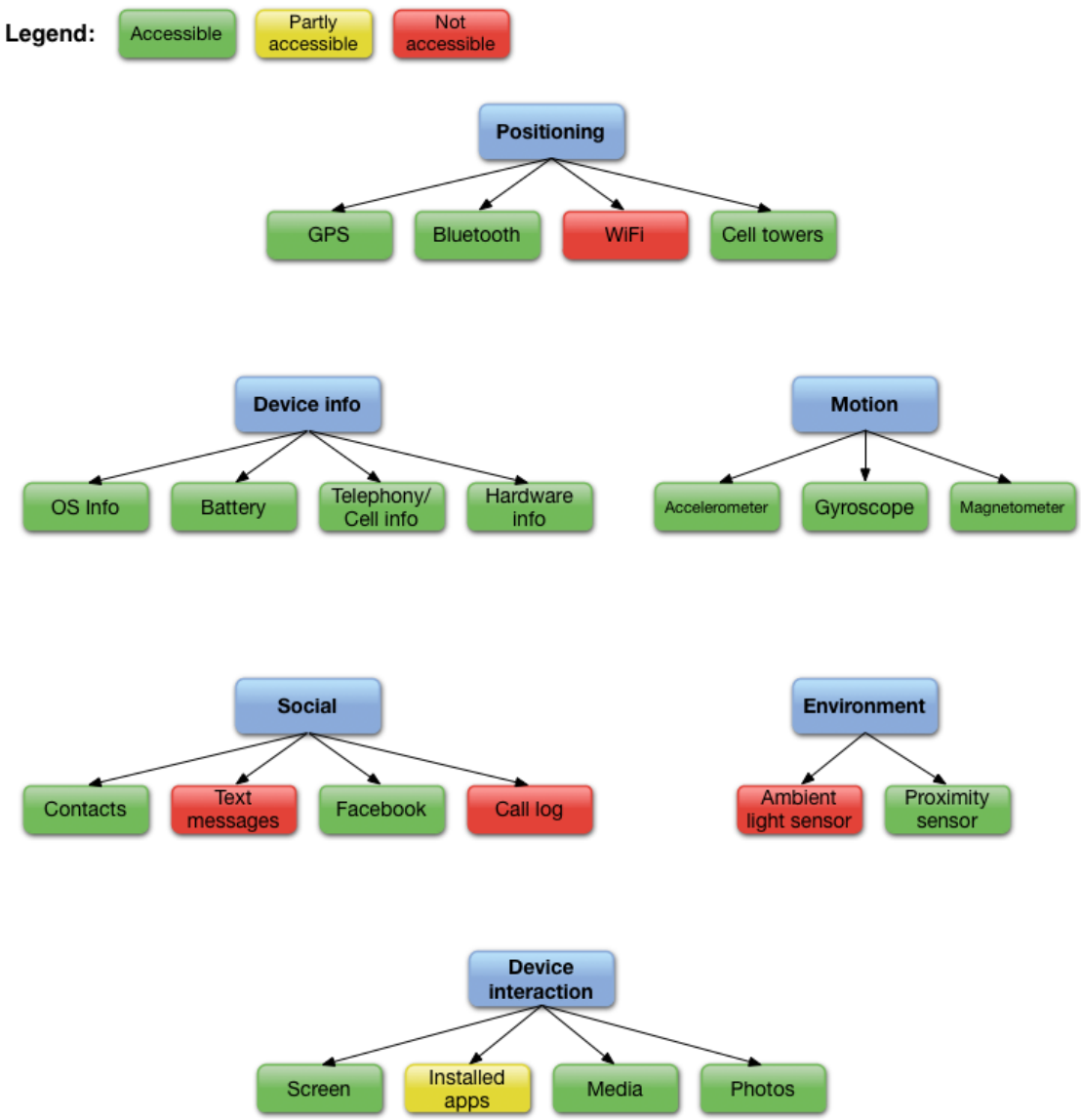

Figure 5.1: Overview of iOS data sources that is accessible

Bibliography

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

[2] JackPearseless. iphone - backgrounding to poll for events. http://stackoverflow.com/a/4808049, 2011.

[3] Mringwal, Conradev, Limneos, and KennyTM. Iohidfamily. http://iphonedevwiki.net/index.php/IOHIDFamily, 2012.