University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Paweł Bedyński
Student no. 239764

Andood – an Android application

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz
Institute of Computer Science

June 2011

Supervisor's statement

Hereby I confirm that the present thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the present thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

My task was to design and implement an Android application that would enable users to interact with the existing web service called *Dood*. The dedicated application has a form of a desktop widget and supports OS versions from Android 2.1 onwards. The significant part was dedicated for designing and implementing a RESTful API which constitutes communication protocol between client applications and the web service. Andood application takes advantage of many solutions and techniques already pre-installed on the Android platform. It has been optimized according to the best practices recommended by Google in order to increase user experience and reduce power consumption. The application is part of a system that provides local businesses and their clients with a platform for instant, personalised and geo-localised advertisements.

Keywords

Android OS, Java, mobile applications, performance, design patterns

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classifcation

D. Software
D.2.11 Software Architectures
D.3.3. Language Constructs and Features
D.4.2 Storage Management
H.2.1 Logical Design

Title in Polish

Andood – aplikacja na platformę Android

Contents

Introduction . . . 5

1. The Android project . . . 9
- 1.1. Android revolution . . . 9
- 1.2. Application Framework – fundamentals . . . 12
  - 1.2.1. Manifest . . . 12
  - 1.2.2. Main application components . . . 13
  - 1.2.3. Process handling . . . 14

2. Andood – overview . . . 15
- 2.1. Use cases . . . 15
  - 2.1.1. Business owner's perspective . . . 16
  - 2.1.2. User's perspective . . . 18
- 2.2. Architecture . . . 18
  - 2.2.1. System architecture . . . 19
  - 2.2.2. Application's architecture . . . 19
- 2.3. Design principles and application's interface . . . 21
  - 2.3.1. Companion Widget . . . 22
  - 2.3.2. Action Bar . . . 23

3. Andood – business analysis . . . 25
- 3.1. Dood project . . . 25
- 3.2. Business model . . . 25
- 3.3. Market analysis . . . 27

4. Authentication and Security . . . 29
- 4.1. The Account Manager in Android 2.x . . . 29
- 4.2. The system of authentication in Andood . . . 30
- 4.3. Personal information protection . . . 32
  - 4.3.1. Location aware applications . . . 32

5. The Data Exchange Model . . . 33
- 5.1. Implementation of REST clients in Android . . . 33
  - 5.1.1. Solution I – *Services* . . . 34
  - 5.1.2. Solution II – *ContentProvider* API . . . 35
  - 5.1.3. Solution III – *ContentProvider* API and *SyncAdapter* . . . 36
- 5.2. SyncAdapters in Android 2.x platforms . . . 37
- 5.3. Dood API . . . 38
  - 5.3.1. Data model . . . 38

5.3.2. Session control 39
5.3.3. Object requests 39
5.3.4. Local cache support 40
5.4. Data synchronisation – model 41
5.5. Data synchronisation – action flow 42

6. Performance and power consumption issues 45
6.1. Performance guideline 45
6.1.1. Optimising large lists 46
6.1.2. Improving responsiveness 49
6.2. Reducing power consumption 50
6.2.1. Reducing power consumption in Andood 50

7. Application release and conclusions 53
7.1. Application release 53
7.2. Future work 53
7.3. Conclusions 54

Bibliography 57

A. CD Contents 59

Introduction

It has been about two and a half years since the first Android phone has been released to the public and less than one and a half years from the last major upgrade to platform 2.x. Within this short period of time Android has managed to overtake a significant part of the global smartphone market, becoming a clear leader in year-to-year growth. With a huge and incredibly fast growing number of developers and smartphones vendors it definitely can be regarded as an interesting platform for advertisers.

Dood was founded in 2009 by two students of the University of Warsaw (Łukasz Kidziński, Paweł Bedyński) and a student of the Warsaw University of Technology (Adam Kaliński). The whole system consists of the main web service and a number of supporting applications. *dood.pl* is a search engine of local businesses integrated with a social network (connected to Facebook). Among many actions users can rate businesses, write reviews and recommend them to their friends. The word *business* can mean anything from local restaurant or theatre to major financial institution. The only requirement is that it should have a permanent address and it should be publicly accessible.

The main objective of this work was to design and develop an application that would enable users to access some of the Dood service's functionality directly from mobile, Android-running devices. There were a couple of goals to achieve which, at the same time, could be regarded as architectural constraints:

1. The application had to use a publicly accessible API to communicate with the server. A one year old, previous version was available, nevertheless it had some important drawbacks which had to be resolved. Moreover, the existing data model could not be changed significantly with respect to the already deployed version of Dood's database, as many other applications relied on it.

2. The application had to fit in Dood's business model, possibly adding some new features in order to take a full advantage of mobile platforms capabilities.

3. The application had to be portable. The smartphones market is already quite fragmented so each decision of limiting the scope of target devices should be accurately justified.

4. The application had to be properly designed in terms of graphics and user interface. There are plenty of Android applications available online and certainly positive user experience and intuitiveness of the interface are the two factors which distinguish good ones from the others.

5. The application should make a full use of the pre-implemented solutions designed by Google developers and added to the Android platform. It turns out that plenty of problems which are typical for multilayered, RESTful application have already been solved, so another two objectives were to explore these solutions and potentially incorporate them within the new application.

6. The application had to be fast, responsive and – at the same time – secure. Naturally, due to technology constraints some compromises had to be made, nevertheless, diminishing the role of any of these three factors or disregarding them entirely was unacceptable. The goal was to explore the existing, pre-implemented solutions and use them wisely.

7. Finally, having in mind that the core part of the new application would be developed by only one developer, an important issue was to choose which functionalities of Dood's service were to be implemented first. The clear goal was to create a complete product that would be ready for release.

This paper introduces the Andood application with its basic architectural concepts, design principles and other technical aspects. All of them are presented in the context of related Google's best practices advised to Android developers. General ideas are followed each time by the outline of specific solutions implemented in the Andood application.

Chapter one provides background information about the Android platform itself. It focuses on fundamental components of the operating system from the point of view of application developers. Furthermore, it shows the most up-to-date smartphone market data with some statistics about the Android platform versions.

The second chapter presents an overview of Andood. It shows the connection of this application to other components of Dood's system architecture. Moreover, it covers the most common usecases from both user's and business owner's perspectives. Some of the design patterns and user interface solutions with their implementations are presented at the end of the chapter. The main aim of the chapter is to provide answers to the following questions: what users can do with this application, how they can use it and – finally – what are the key architectural elements that take part in these actions.

Andood's business model and its compliance with the Dood's business model are presented in chapter three. The short outline is followed by the discussion about potential competitors of the new application on the Polish market.

The remaining chapters (excluding the last one) give a more detailed insight into the application's components. They highlight the most important aspects of software development which translate directly to the overall performance and quality of the application. Every time the leading questions are: what are the options to solve any particular problem, are there any pre-implemented solution provided by Google developers and – finally – how can these concepts be implemented in Andood.

The fourth chapter focuses on authentication and security issues. Highlights of the new Android components introduced in the Android 2.x platform precede a list of solutions implemented in the Andood application. More information about law constraints related to location aware applications is delivered afterwards.

Details about the data exchange model are presented in chapter five. Andood is a RESTful client application and it takes full advantage of significant contribution of Google developers who have made great effort in preparing a list of best practices related to different data exchange models with remote services. A short overview about various architectural settings is followed by the presentation of the specific model implemented in Andood application. Some information about data model and special API used to communicate with a Dood server is presented in between to give a proper background for the discussion about process of synchronisation.

Chapter six is dedicated to performance issues and problems related to battery limitations. Because optimising performance is one of the most widely discussed matters in Android's

developers community, this chapter focuses only on tricks and mechanisms already implemented in Andood application (or already planned for the next release).

The last chapter brings forth some information about Andood's release plans as well as main *milestones* for the next upcoming release. This is followed by final conclusions and discussion about the project's future.

Chapter 1

The Android project

Android is an open source project developed by the Open Handset Alliance and held by Google Inc. It is often wrongly attributed to the operating system based on Linux kernel alone, but in fact it contains a middleware and a variety of additional applications. For these reasons it is more fair to say that Android is a software stack for mobile devices [AND]. All phones running the Android system come with a range of pre-installed applications like Maps, Google Search, Gmail or YouTube.

Users can easily download new applications (also called *apps*) directly from their mobile devices, or by using the Android Market official web site (over 200.000 apps are available there). Both paths provide a very seamless experience and require only a little user interaction. The system takes care of the whole installation process in the background including finding a path for the new application. Users are only asked if they agree on the application's permissions which basically means a set of actions that the new *app* will be permitted to perform (like accessing resources or device's sensors).

It is also important to mention that all applications within the Android system are *equal*. Regardless to whether it is a third party application or a core system application, they all run in the same environment and potentially have the same access rights to all phone resources. This way, for instance, new applications can easily replace the old ones if they offer similar functionality. Android is based on an event-driven mechanism so it is all a matter of listening to specific system broadcast messages and acting accordingly.

One of the most appreciated aspects of Android is its openness. The source code has been revealed to the public, enabling many developers around the world not only to have better understanding of what is happening in the background of the system, but also to actively contribute to the project. It makes Android more flexible, allowing new cutting-edge technologies to quickly incorporate in the system.

1.1. Android revolution

It has been a little over two years now from the moment the first Android phone has been released to the public. Within this short period of time Android has come a spectacular way from an early stage project with prototype devices to being the second most popular OS in the smartphone market in the world. Moreover, according to some recent surveys it has managed to dethrone Nokia's Symbian from its ten-year top position. Without any doubt, Android is now a global leader in terms of a year-to-year growth and still has great potential for future increase in sales. Table 1.1 shows the data collected in 2010 compared with 2009. Despite a significant growth in the whole market (70% year-to-year), Android managed to

increase its shares in 2010 from less than 4% to a little less than 23% selling approximately ten times more units.

Worldwide Smartphone Sales to End Users by Operating System in 2010 (Thousands of Units)

Company	2010 Units	2010 Market Share (%)	2009 Units	2009 Market Share (%)
Symbian	111,576.7	37.6	80,878.3	46.9
Android	67,224.5	22.7	6,798.4	3.9
Research In Motion	47,451.6	16.0	34,346.6	19.9
iOS	46,598.3	15.7	24,889.7	14.4
Microsoft	12,378.2	4.2	15,031.0	8.7
Other Oss	11417.4	3.8	10432.1	6.1
Total	**296,646.6**	**100.0**	**172,376.1**	**100.0**

Source: Gartner (February 2011)

Table 1.1: Worldwide smartphone sales by operating system (2010), *source: Gartner Research http://www.gartner.com/it/page.jsp?id=1543014*

Figure 1.1 shows market shares of the top three mobile operating systems across six months from June 2010. This period of time is very interesting as Google's biggest competitor in this market, Apple, has launched its new iPhone 4 on the 24th of June. Despite a massive marketing campaign Apple did not manage to get even close to the increase of Android which gained extra 14% of the US market within just six months.

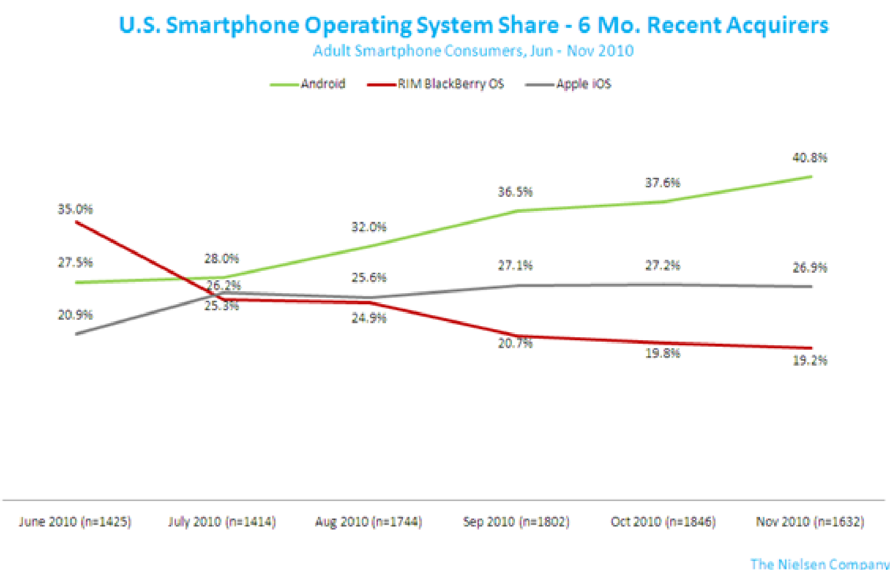

Figure 1.1: U.S. Smartphone Operating System Share (Jun-Nov 2010), *source: The Nielsen Company http://blog.nielsen.com/nielsenwire/online_mobile/apple-leads-smartphone-race-while-android-attracts-most-recent-customers*

Naturally, Android itself is not a single operating system. It comes in many versions with new major updates being released every half year or even more frequently. So far all new updates have kept a total backward compatibility but this is likely to become more

problematic as Google is launching a tablet optimized version 3.0 – *Honeycomb*. The most up-to-date chart (figure 1.2) from Android Developer [AND] web site shows a distribution of Android versions. Data has been collected from requests that came from mobile devices to the Android Market during 14 days before April the 1st 2011.

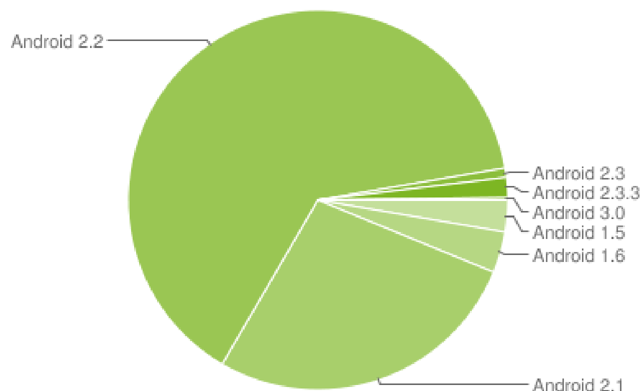

Figure 1.2: Android versions (April 2011), *source: Android Developers* http://developer.android.com/resources/dashboard/platform-versions.html

The recently released Android 3.0 (Honeycomb) and Android 2.3 (Gingerbread) with the following 2.3.3 quick update are hardly visible on the chart (2,7% together), nevertheless, 93% of Android devices are now running 2.x versions or newer. Naturally, these statistics apply only to devices which have made a successful connection to the Android Market. Hence it is not literally speaking the actual distribution of Android versions on the global market, however it definitely can be regarded as the long term trend, showing the speed of the upgrade process. This is a very important observation as many developers have decided to release applications that support only versions from 2.1 onwards. This is mainly because the Android 2.x platform offers new, easy to use, pre-implemented solutions that proved to be very useful in many applications. Andood *app* is one of them.

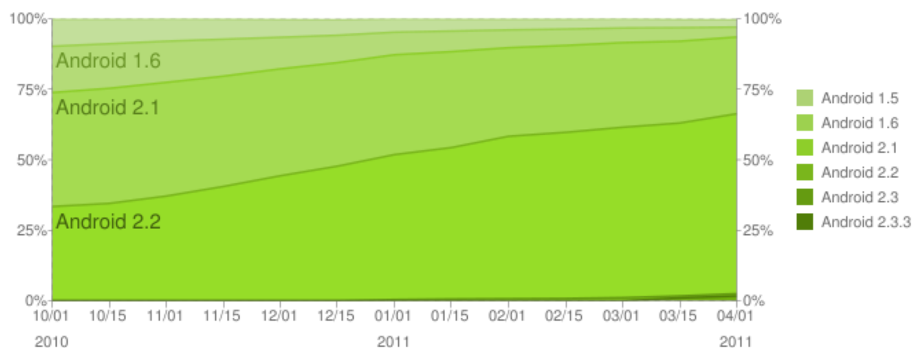

Figure 1.3: Android versions (April 2011), *source: Android Developers* http://developer.android.com/resources/dashboard/platform-versions.html

The next figure (1.3) shows how fast the process of replacing old version of Android with the new ones is. The vertical axis has accuracy of barely two weeks, but still, this little amount of time is quite enough to observe a significant change. There are at least two good reasons why it is happening so fast. The first one is that smartphone vendors want to release new phones regularly (and customers also buy them regularly) so frequent system updates create

such an opportunity. The second is that Android is an open source project, so if the device's hardware fulfills the new version's minimal requirements, it is just a matter of time to get an upgrade.

When designing a new Android application, one has to take under consideration not only a variety of platform versions but also a target device's specification. The most important feature is probably the screen resolution. Oppositely to Apple's iPhone, Android is not limited to a single device and Google does not manufacture its own phones. As it was mentioned beforehand, Android is a software stack and thus it can be installed on practically any device that satisfy a minimal set of requirements and it was meant to support a variety of resolutions or even screen orientations. The next chart (figure 1.4) shows that the vast majority of devices have either high or normal density screens, but it is good practice to support other options (especially as larger screens are likely to appear in the near future).

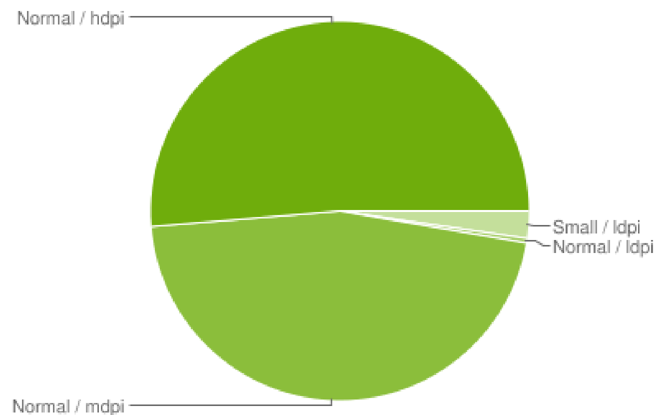

Figure 1.4: Android devices resolutions (February 2011), *source: Android Developers* http://developer.android.com/resources/dashboard/screens.html

1.2. Application Framework – fundamentals

Even though Android is a relatively new system, it comes with good documentation (some people question that) which can be found on the official developers page ([AND]). There is a number of publications and coursebooks for people who seek basic information and introduction to the system. Additionally, a very large community of developers is actively exchanging questions and answers on web services like Google Groups ([ADG]), Android Blog ([ADB]) or Stack Overflow ([ASO]). It does not make sense to quote big fragments of Android developers guide. Instead, this section covers the fundamental aspects of designing an Android application which are necessary in order to understand the following discussion about Andood *app*.

1.2.1. Manifest

Every Android application must have a single manifest file in a root directory (named *AndroidManifest.xml*). This file contains essential information about the application which is required by the system to run it. All new components have to be registered there along with permissions needed to perform actions requested by the application. The file is also used to inform the system if a developer wishes to expose the application's data to other *apps*. It specifies the application's reaction to various system events such as incoming calls or photo

capturing. Within the Manifest file developers can specify if they want one of their activities (described below) to become a *launch* activity. For the purpose of this paper applications which contain a launch activitity will be called *launch applications*. Such applications are added to the Android's menu screen so they can be launched independently from other apps.

1.2.2. Main application components

All Android applications contain zero or more of the following components:

- *Activity* – a single piece of user interface (UI). In a system it runs independently from other elements although users have a smooth experience of different activities appearing on the screen in a sequence. Activities usually contain some graphical elements (customised or taken from a pre-defined set) and constitute a specific action that users can perform like writing an email or selecting contact from a list.
- *Service* – a separate part of the application without a graphical representation (running in a background). Service itself does not start a separate thread because it runs in context of the activity or the broadcast receiver which has started it. However, it usually should use a working thread to perform some complex computation or lengthy I/O operations.
- *Content Provider* – a mechanism that allows applications to share data between each other or to simply persistently save some information. It is basically an interface which provides standard methods to access data like query, insert, update, delete. No specific type of data structure is imposed by the system, it can be a single file or an SQLite database. Content provider is uniquely identified by its *authority* and can contain many types of data objects (many tables in a database). The most common method of accessing an object is to query Content Provider with a specific *content uri* which has a generic form of:

  ```
  content://<authority>/..<type path>../<id>/
  ```

- *Broadcast Receiver* – a part of the application which responds to actions broadcasted by the system itself (dimming the screen, capturing a new photo) or by other parts of the application (the same one or not). Broadcast Receiver does not have its own graphical representation but it can initiate certain actions to keep users informed about its work. This could be done for instance by creating a notification in a status bar or updating a desktop widget.

As it was mentioned previously Android, is an event-driven system. These events are called *Intents*. Intent API is a very powerful mechanism that manages interactions between application components across the whole system. Developers do not have to write any additional lines of code in order to integrate their application with other ones as long as they know their set of supported Intents. Moreover, it does not make any difference whether you send an Intent to a separate application or another part of the same one – the mechanism is exactly the same. A good example is the Map application which supports Intents that contain a request to display specific geo location. Developers can simply broadcast this Intent whenever they want to show a map with a specific location. Additionally, should users have installed a different *app* which masks the functionality of the Map application (by filtering Intents), this new application would be used automatically instead. The Intent mechanism

allows developers to easily reuse different system components across the platform. They can focus on really innovative and unique functionality and simply add ('attach') new components like maps or navigation to their applications which makes them even more compelling and useful.

1.2.3. Process handling

One of the key issues when designing an operating system for mobile devices is memory management. Android is trying to do as much as possible in the background without bothering anybody, but at the same time there is remarkable space for alterations and modifications. Therefore, developers should at least be aware of the most basic aspects of process handling not to create applications that would be 'arrogant' or even 'hostile' to the system.

Very often Android has to deal with situations of memory shortage. The system has to reclaim resources in order to allow new processes to run. In such situations Android ranks all active processes according to the following order:

1. *Foreground process* – the process which hosts a foreground activity or a service that is bound to a foreground activity. It basically means the part of the system that the user is currently interacting with, so at any given time there are only few such processes. These processes are killed only in absolute critical situations as not terminating them will most likely cause lack of responsiveness (or even displaying error messages to the user).

2. *Visible process* – a process that does not have any active components, but still is visible to the user. A good example is a process which hosts an activity that launched another not-full-screen activity. Visible processes are considered important and will only get killed if the system cannot find enough memory for foreground processes.

3. *Service process* – a process which basically hosts a service and at the moment it is neither a *foreground* nor a *visible* process. Those processes however may be doing some important tasks for the, user like playing music in a background, or downloading some data from the Internet.

4. *Background process* – a process which handles an activity that is currently not visible. These processes are likely to get killed at any given time so activities should be prepared for that.

5. *Empty process* – a process without any of the application's components. The only reason why the system holds these processes alive is for caching purposes.

Developers should constantly be aware of the fact that their processes could get killed and reestablished by the system in the background even if they did not expect such a situation to happen in the first place. To allow a smooth navigation between different screens, Android introduces a number of helping methods to save and restore the state of activities. It is good practice not only to use these methods whenever there is a need to, but also to do it wisely. For instance, no lengthy operations should ever be performed within them. A good example is an activity where users can edit a new text message being interrupted by an incoming call. The screen is captured by the new activity, decreasing the priority of the old one. Therefore, the process which handles the message-editing activity is more likely to get killed. Users would probably expect to be able to continue editing the text of the message after finishing the call. However, this would be impossible if the text had not been persistently saved beforehand.

Chapter 2

Andood – overview

The Andood project was officially launched in September 2010 as one of the key components in the long term strategy of the LemoNET company. The fundamental requirement was to provide a platform for small and medium businesses that would enable their owners to almost instantly publish short and targeted advertisements. Moreover, the same platform could be used by anybody with the device running the Android system (version 2.1 and subsequent) to receive notifications about current offers according to the specified search filters. The application has a form of a desktop widget and can be placed directly on the Android's Home application. The widget receives its content from Dood servers using a RESTful API based on HTTP requests and JSON data format. Documentation of the API has been published on the Google Project web page ([GCW]) and can be freely used by all third party developers.

The current form of Andood application is just the first, but a very important stage in LemoNET plans related to the Android market. The widget was simply the most independent and properly defined module of the bigger, dedicated application which is expected to be released afterwards. It also addresses the majority of planned functionality in terms of the business model (please refer to section 3.2). Thanks to the design patterns imposed by the Android system as well as Andood's application's architecture, the process of developing and merging a widget with the bigger *launch* application (visible on the menu screen) should not be complicated and definitely will not require any substantial changes in the existing code.

2.1. Use cases

The main concept of Andood application was to design a system that would be both functional and easy to use. These principles apply to all situations of interaction with the system and can be categorised in two dual perspectives. The first one is a *business owner's perspective*. It can literally mean a legal owner of the business or simply anybody who has been delegated and granted privileges to access the system by the real business owner. Dood web service has its own system of authenticating a true owner and a special section of the web page where they can manage their business's profiles. The algorithm, however, is not a part of the core Andood application, so it will not be described here, although, it is important to point out that the system guarantees that an authorised *agent* of the business owner can interact with the system if he or she requests such *privileged* status. Second perspective is the *user's perspective*. It basically can be anybody with an Android device and Andood application installed on it. It is worth mentioning here that no registration is required prior to launching Andood, although users can, and are advised to log in if they have an active *Dood* account. Currently, application does not support registration from Android devices (except entering

the Dood website in a browser), but this feature is on a schedule for the next release. More information about authentication and security issues can be found in chapter 4.

The two perspectives however, do not comply altogether with generally accepted conventions of presenting use cases of software projects. This is why instead of a complex flowchart showing interaction with different layers of the application, in this section it is better to underline the simplicity of the system.

In the following sections and chapters the word *neighbourhood* will appear quite frequently. A *Neighbourhood* (or *My Location*, *Location*) is an object created by a server in response to the position revealed by the user and some other variables. It is an abstract user's *Location* which carries information about surrounding businesses and promotions. *Neighbourhood* does not define a separate territory nor is it defined by any itself. Oppositely, in can vary in terms of range and content depending on the user's preferences and search requests. The object itself is described in section 5.3. For the purpose of use cases it is only necessary to know that all of the following actions and states are possible: users can enter or leave a neighbourhood, the neighbourhood can expire (possibly containing data which is not relevant to the user), the business and/or promotion can become a part of a neighbourhood.

Similarly, the word *Promotion* is often used next to the word *Advert* or *Advertisement* (or replacing them) which can cause confusion. Out of these two it is Promotion which is the real database object (described in section 5.3). The word *Advertisement* is more likely to be used in a meta-context when one wants to point out that certain information enters the system for a given price. Consequently, *Promotion* creates a sort of a unified class which transports Advertisements from that moment onwards. At first it may seem quite counter-intuitive as not all advertisements follow some of the promotion's specification (like expiration date for instance, advertisements are not so prone to change within time). However, the system has to maintain its objects somehow, hence Promotion is much more preferable to handle with than Advertisement. Eventually, this subtle difference turns out to be more of the naming convention problem than a real blocking issue.

2.1.1. Business owner's perspective

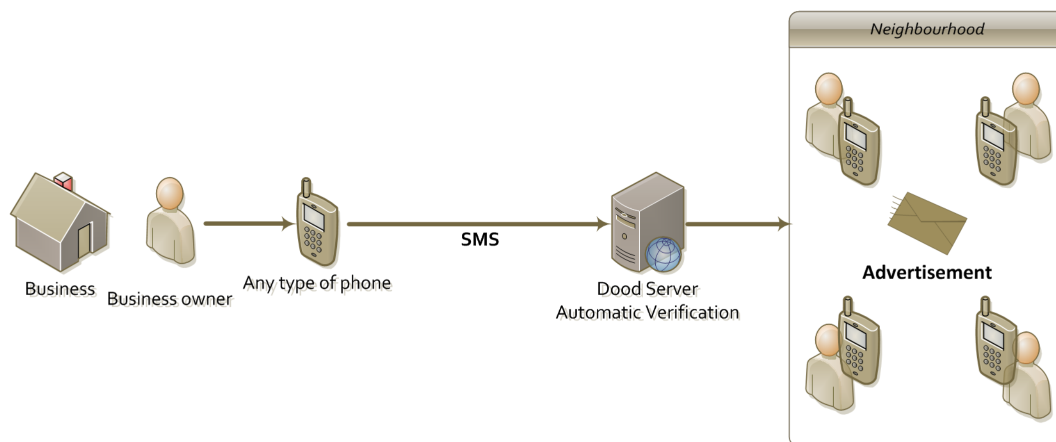

Figure 2.1: Use case 1: Business owner's perspective. Promotion message sent by phone.

Figure 2.1 depicts the first situation when business owner uses a personal device to create and publish a new advertisement (promotion). No restriction is made about the hardware components of this device (it does not have to be an Android device), the only requirement

is that it should be able to send a text messages (SMS). These messages, however, should be uniquely identified by a sender id (typically a phone number). Dood web service provides a functionality to pair the id of incoming messages with the specific businesses. This could be done by business owners themselves (business management section on Dood web site) or remotely by authorised employees of Dood who can perform such operation from their mobile devices or standard PCs. The latter option is a very important feature of the application used in marketing campaign when authorised sellers can show a live demo of sending and receiving a new advertisement (promotion) directly from their devices. However, adding pairs of device's and business's ids is not part of the Andood application, hence it is not described later on in this paper.

According to the flow presented in figure 2.1, the business owner creates an SMS message with a text of the advertisement (promotion) and sends it to the given number. The list of active numbers and prices per message can be found on the Dood web site. Each number corresponds with a different (but predefined) set of promotion options such as duration, range or maximum number of impressions. The message itself cannot be longer than 140 characters but it does not have to contain the business's name or address. It arrives at the Dood servers and is automatically checked for with a list of forbidden expressions (like vulgarisms). The algorithm of filtering forbidden expressions is publicly accessible via web service (Dood web site). Each user can insert a desired text and receive a boolean result (success or failure). Additionally, a dictionary of forbidden words and expressions is available for download. The incoming message which passes this test is automatically (without any human interaction) matched with the business and added to the system as a promotion (the text of the message becomes the title of the new promotion). It means that each advertisement can reach the end point of the path through the whole system (from business owners to users) within just milliseconds. If the incoming message text contains one or more of the forbidden expressions, it is transferred to a separate part of the system where it awaits for the acceptance of a Dood employee. He or she decides whether to approve the message according to the company rules (please refer to Dood.pl web site). The message can be rejected (such situation happens also when the system cannot match a message id with any business) and in that case it will not be added to the system. If the business has been correctly identified from the message id, a special notification should appear in the business owner's profile. Regardless the result of this process, the cost of the SMS message is not refunded.

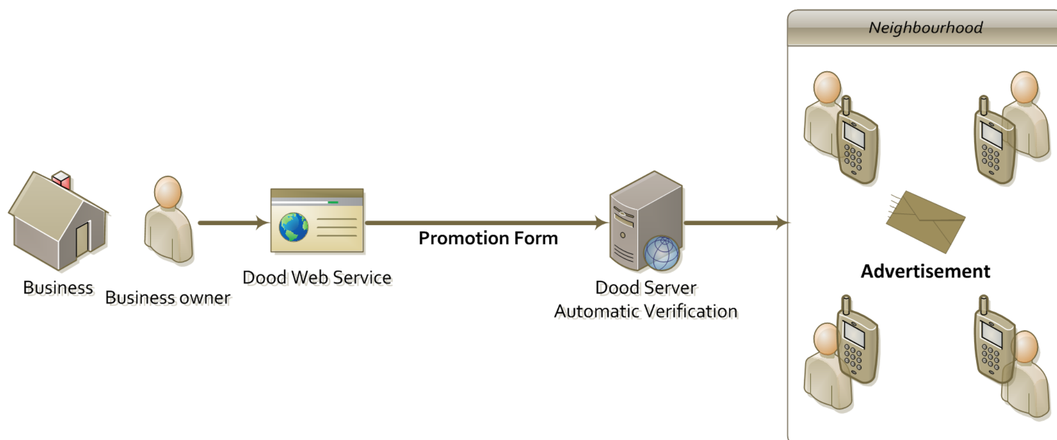

Figure 2.2: Use case 2: Business owner's perspective. Promotion message sent via web service.

In the second use case (figure 2.2) business owners use a special form in the management

section (available on Dood.pl web site) to create and add new promotions. Business owners are able to check whether their message passes the test for forbidden expressions before adding it to the system. Moreover, not only can they add a 140 characters long title of the promotion, but they can also provide the promotion's description (it applies to all active promotions, regardless of the method in which they have been created).

2.1.2. User's perspective

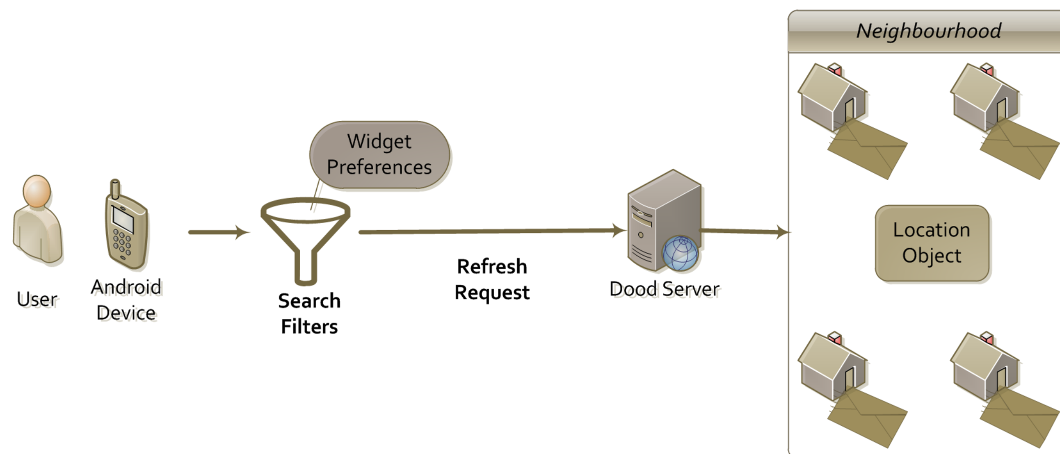

Figure 2.3: Use case 3: User's perspective. Refreshing widget data according to user's preferences.

The third use case describes an opposite situation. Following the figure 2.3 from the left-hand side, the first action is sending a refresh request by the Andood application. This can be forced knowingly by the user or executed remotely by the system in inexact (but constrained) time intervals according to the user's preferences and the current application's data. More details about *heart beat control* of the application can be found in chapter 6. After the application decides to send a refresh request the user's preferences are collected and attached. The specific parameters with their meanings can be found in section 5.3. These preferences are not only standard search filters. They can alter the response objects and by that means change application's behaviour (like indirectly manipulate refresh intervals). When the request reaches the Dood server an *ad hoc* *Location* object is created according to the given parameters and send back to the user. The moment the application receives the new *Location* object and refreshes the application can be called *entering a new neighbourhood*. Please refer to section 5.3 for more details about the *Location* object. For the time being it is enough to say that this object contains a list of selected *Promotions* and additional *Businesses* (basic information about recommended businesses that currently do not have any promotions).

2.2. Architecture

Andood application is only a part of a bigger service called Dood. Hence, when dealing with the architecture there are basically two perspectives in which the application can be depicted. First one is the *System's architecture* where the most important aspect is how the Andood application interferes with different components of a Dood system. Second perspective is the *Application's architecture* where only Andood application is presented with its different

modules and the way they communicate with each other. Another aspect of this perspective is how these modules fit within the Android system and its pre-defined, specialised components.

2.2.1. System architecture

The system architecture has been illustrated on figure 2.4. From the left-hand side an Android device (not necessarily a smartphone, it could be a tablet) with the Andood application already installed communicates with the Dood web service using a special RESTful API. The application sends HTTP requests with GET/PUT/POST/DELETE method headers and receives well formatted XML or JSON responses. REST objects can be exchanged both ways. Please refer to section 5.3 for more details about the API. The Dood web service is based on Django technology [DJO] and has been implemented by Łukasz Kidziński and Adam Kaliński. The Web service uses a standard Django model to instantiate objects and fetch them from a MySql database. A separate service (Content Delivery Network) is used to serve some static content like images, style sheet files (CSS) or Java Scripts.

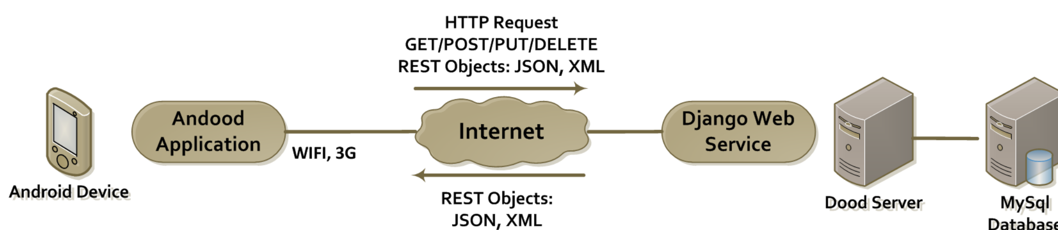

Figure 2.4: System Architecture

2.2.2. Application's architecture

This perspective presents the Andood app alone. As it is shown on figure 2.5, communication with a Dood server is replaced with a simple requests and responses of API. The whole application is divided into two separate and independently developed modules (Eclipse projects). The first one, called JCL (Java Client Library), is simply a set of standard Java packages and classes, and does not require any Android libraries. It is meant to be reusable in any Java project that wants to use a Dood API to communicate with the server (that is why it is a Maven project). The second module (called *Core Andood Application*) actually contains quite loosely connected components, but the one thing they all have in common is that they all use Android libraries, so they can only by used on Android devices.

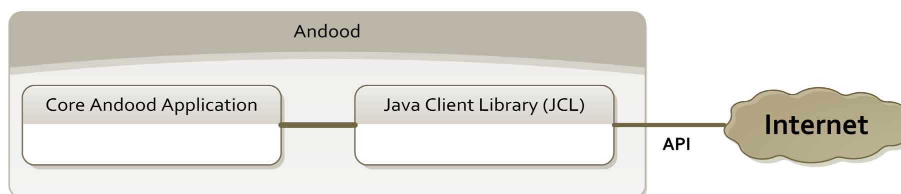

Figure 2.5: Application overview

The Java Client Library (figure 2.6) is written entirely in Java technology. It defines a *Session* class as well as an interface for the connection. The idea was to enable developers to use this library even if they do not want to connect to the specific Dood server. They can easily write their own implementation of *Connection* interface and set up a different server

(or other source of data) to supply its content according to the rules defined by the API. A default implementation of the *Connection* which connects directly to the Dood server has been supplied. JCL defines standard *Datatypes* classes for *Database Objects*. Each database object can parse itself from JSON format and encode itself back into it. The library contains the API's error and exception types and a bunch of other helper classes that are useful in transporting the objects through the connection. There is a number of JUnit tests to validate communication protocol with the server and proper decoding and encoding of the Database Objects.

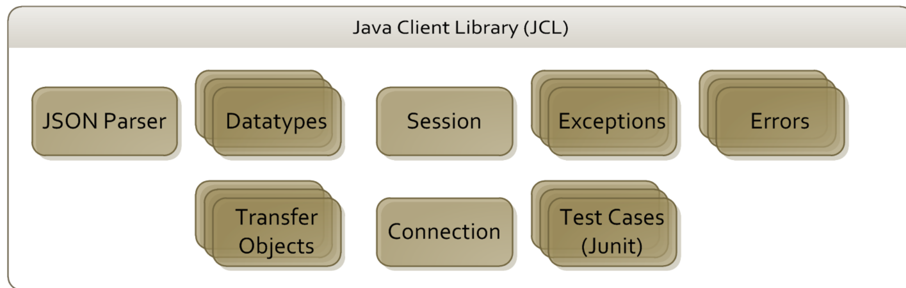

Figure 2.6: Java Client Library

The JCL library is used by the Core Andood Application, but – more importantly – it will be used by the *launch* Andood Application without any need for modifications. Only some new Database Objects will be added to the list (*Review*, *User* etc.).

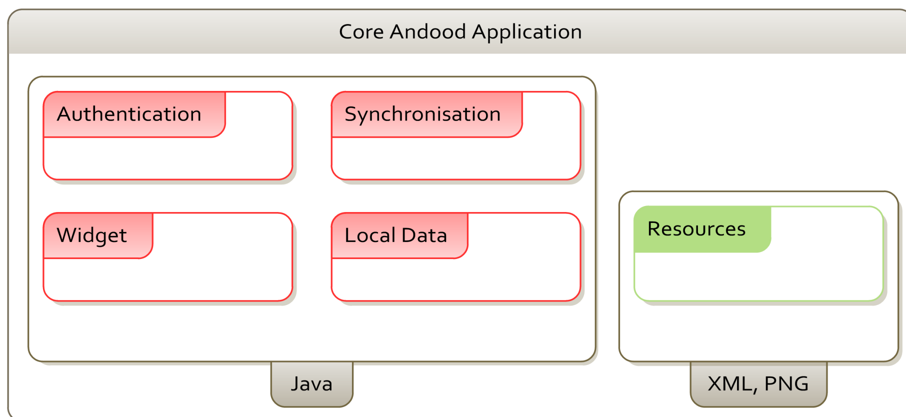

Figure 2.7: Core Android Application

Figure 2.7 shows the basic decomposition of the Core Andood Application. Thanks to the Android system architecture all of the Java modules can be treated as separate containers responsible for providing different functionality. There is some sort of a connection between any given pair, but only Synchronisation and Authentication modules are eligible to communicate with the JCL library. On the other hand only Widget and Authentication modules contain a presentation layer, thus they are *visible* to the user. All modules are written in Java technology with many XML files used for storing static resources.

The android platform enables developers to export many types of resources like layouts, strings, configuration of drawables or system components into XML files. This is not always obligatory but it definitely helps in maintaining the source code and making it more vivid. Moreover, developers who decide to follow this guideline (which is definitely one of the most

important best practices in developing Android applications) do not need to care (inside a code) so much about handling the different device's configurations (like density, size of screen etc.). Android automatically detects a predefined set of folders like *drawables-hdpi, drawables-mdpi, drawables-ldpi* and developers should only populate them with the appropriate versions of their resource files. The system will choose the best available file for them according to the device's configuration. This is a very powerful mechanism which is available right out-of-the-box and developers are strongly encouraged to make an extensive use of it. To give an example in the Andood application: creating a second language version was as easy as copying a single XML file and translating all of the strings inside of it into a new language. Naturally, this does not apply to the data which comes from the network. Currently, Dood servers do not contain any reviews or information about businesses in any other language than Polish, so despite changing the language into English some of the visible text will not be translated in the Andood application.

2.3. Design principles and application's interface

The design has become one of the key issues in developing Android applications. With the rapidly growing number of available apps, users expect them to be at the same time intuitive and elegant. It ultimately means that developers should constantly strive to find a balance between making their interface *clear* but rich and making it *simple* [IO1]. Simple is often perceived in a negative connotation as something not useful, not functional. On the other hand, rich interface is not always intuitive and not all of the users are willing to learn how to use new applications if it takes too much time. Android comes with a variety of ready-to-use UI components and it is highly recommended to use those 'blocks' in order to achieve consistency with the pre-installed Android applications. However, the system does not specify any unified layouts for third party applications to allow developers to change them according to their will.

From the time of the first release of Android system, Google's developers have been collecting ideas about different design patterns which they could recommend to the broad community. These are both genuine ideas of people who work for Google and suggestions about UI posted on different forums or already implemented in third party applications released on the Android Market. During the previous Google I/O conference which took place in San Francisco in May 2010 they have been presented on a session called *Android UI design patterns*. The design patterns can be divided into two separate groups. The first one is related to game development, hence it is not interesting from the point of view of this paper. The second one gathers ideas about developing standard application interfaces and the Andood application takes advantage of many of them.

The current form of the Andood application is not a full *launch* application so it does not require all of the interface elements presented in the session mentioned above. Consequently, at this point it implements two of the five design patterns presented by Google (Companion Widget and Action Bar). Nevertheless, apart from properly implementing the core functionality, the design was the second most important issue in developing the Andood application. The graphical elements (png and 9.png files) have been prepared by Filip Łysyszyn (http://lysyszyn.pl), who cooperates with the LemoNET company as a design and QA specialist. Layouts and all of the user interface interaction scenarios were discussed by both Filip Łysyszyn and the author of this paper but they were implemented only by the latter.

2.3.1. Companion Widget

The heart of the Andood application in this release is the widget. In the design patterns the widget is simply one of the entry points to the main (*launch*) application. The goal here is to display some of the application's content to make the home screen feel more custom and personalised. Figure 2.8 shows the template widget created by Google's developers for the purpose of the Google 2010 I/O talk.

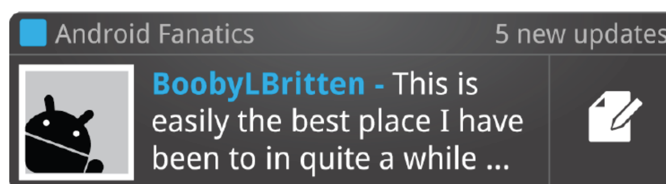

Figure 2.8: Companion Widget – Google [IO1]

The phone's home screen has been divided into a 4x4 grid, so each widget fits into the size of a rectangle with sides lengths varying from 1 to 4. The Google's exemplar (template) widget (figure 2.8) has a 4 by 1 size. However, instead of filling the available space entirely, it uses a NinePatch background to achieve a sightly oval shape at the edges. It definitely makes the widget look much more elegant. The template widget contains the name of the app, counter of new messages, some important contents and a single button which launches the appropriate action in the main application. It is a good practice not to implement the whole functionality inside a widget, but use it more like a rich and refreshable link to the main (*launch*) application.

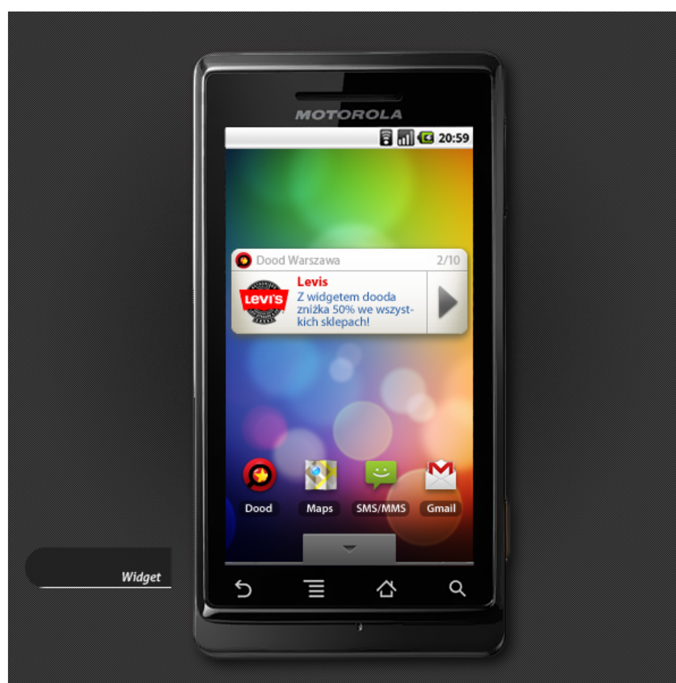

Figure 2.9: First sketch of Andood Companion Widget – © Filip Łysyszyn

In the case of the Andood application the original design pattern had to be slightly modified as the widget became a separate component in this release. However, except for

the simple link to the business or promotion activity which will replace the existing pop-up dialog, there will be no other significant changes. In fact, thanks to the Android's ability to manage interactions between activities by Intents, this modification should influence only a couple of lines in the existing code of Andood's application.

Figure 2.9 shows the first sketch of the Andood Widget prepared by Filip Łysyszyn. Over time some slight changes have been made in the layouts, nevertheless the final outlook (figure 2.10) is quite similar to the initial concept. The widget has a 4 by 1 size (proportions of screen size are kept in figure 2.9) so it is not "invasive" and it leaves a large part of the home screen free for the user to utilise. It consists of the *title bar* with Andood's icon, the name of the application, optionally the name of the current location (city) and a simple counter of promotions and businesses available at this location. On the right side there is a small, clickable icon which sends users to the preferences screen. The bottom panel consists of only two clickable elements. The first one is a *widget content container* which holds information about promotions or businesses currently displayed by the widget. It can also display an error message and change the *on click* action accordingly. The second element is either the *next* button (iterates over the list of promotions), or the *refresh* button (refreshes the widget's content) depending on the context of the application.

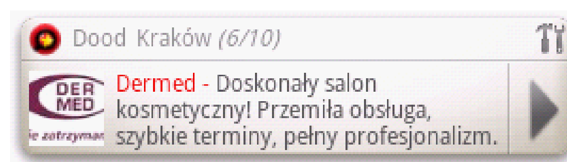

Figure 2.10: Andood Widget

2.3.2. Action Bar

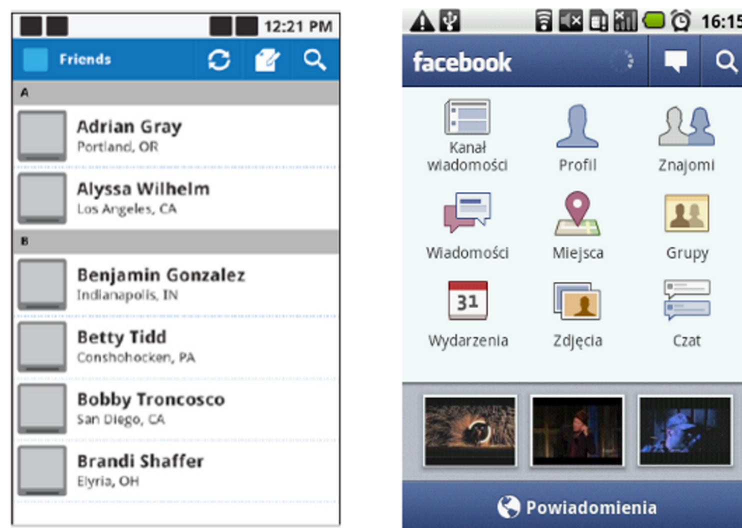

Figure 2.11: Google's template (left) and Facebook (right) ActionBars.

The second Google's design pattern already implemented in the Andood application is the Action Bar. Basically it is a panel at the top of the screen which consist of a logo (or name of the application) and up to three buttons with the most common actions. This layout

structure should preserve consistency across any application's activities and it should not be contextual (there is another design pattern for contextual actions – *Quick Actions*). It is supposed to replace the default Android Title Bar thus somehow *brand* the application. It should also follow certain UI conventions like clicking the logo (name of the application) should bring the user back to the main activity (like *Dashboard* – another element from the design patterns). A set of actions visible on the Action Bar are meant to represent the most common things that user can do within the application that are not context-dependent (like triggering 'search' or 'write message' activities).

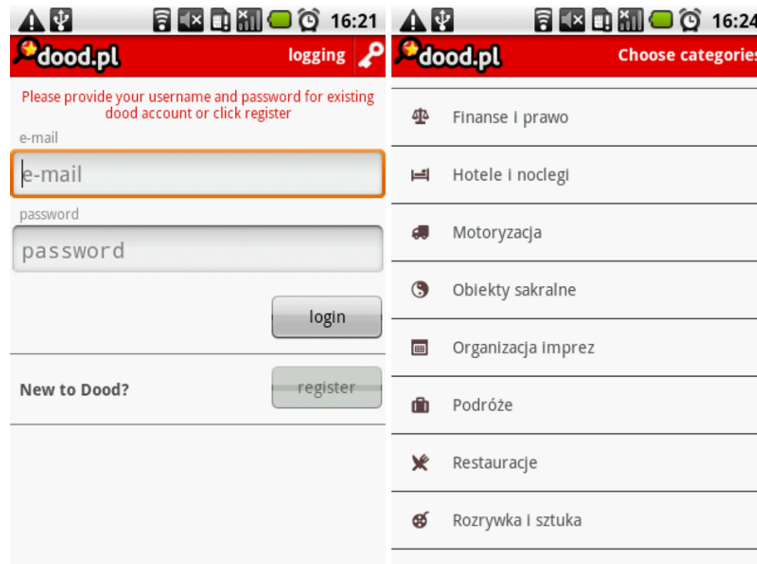

Figure 2.12: Action Bar Design Pattern in Andood application.

Figure 2.11 shows the Action Bar component at the top of the template presented on the 2010 Google I/O session (left) and on a dashboard activity of the official Facebook Android app (right). In case of the Andood application there are only two full screen Activities visible to the user at the moment (many more are to come with the release of the *launch* version). They both have the *ActionBar* element at the top, but slightly modified in respect to the original design pattern. Figure 2.12 shows these two activities: *LoggingActivity* (left) and *CategoryChooseActivity* (right).

Clicking the Dood logo (visible on the left-hand side of the *ActionBar*) does not bring the user back to the main application's activity because there is simply no such activity at the moment. Secondly, instead of the action buttons at the right of the *ActionBar* component there is an activity's name ('what i can do with it now'). Additionally, in *LoggingActivity* there is an image holder with a key which symbolises the action requiring personal (secret) data input. The Andood *launch* application will bring some new features like searching (for businesses or users) or creating new reviews of businesses. These are the most common actions planned for the Andood application so they will eventually find their places in the *ActionBar* as non-contextual buttons.

Chapter 3

Andood – business analysis

3.1. Dood project

Dood is a web service founded by LemoNET company which started in September 2009. Within the Polish market it has a unique functionality which combines a search engine of local businesses with a social network of users who write reviews and rate those businesses. The word *business* is used here as a broad name for all publicly accessible places like restaurants, theaters, cinemas, pubs, clubs, financial institutions or even ATMs. As of March 2011 Dood's database contained more than 50.000 records of *businesses* in 23 of the most populated Polish cities. The total number of their permanent residents exceeds 9 million people. However, all commuters and people like students who come to major academic institutions would certainly find the service's functionality very useful making the potential group of users even larger. The total number of active users has been constantly growing over the past two years and within the last two months (February and March 2011) they all have posted over 1.200 unique reviews of *businesses*.

From the very beginning developers working on a Dood project perceived service's accessibility and flexibility as key issues. That is why apart from the main web application Dood offers a variety of other products built as separate modules but still organised to work on the same data set. In order to make it possible a special API has been designed and consequently implemented. The first version of the API (released in June 2009) is used by Java Scripts and two mobile applications. The first one is called doodMobile (released in early 2010) – is designed for J2ME (Java, MIDP 2.0) phones. The second – *iDood* – supports iPhones and it is now available on Apple's App Store. In August 2010 the author of this thesis and Łukasz Kidziński from LemoNET company together agreed to reorganise the core structure of the API to make it more flexible and better support features like caching on the client side. This joint work resulted in releasing the second version of the API (2.0) and implementing a part of it which is now used by the Andood application. Documentation of the most up-to-date version of the API (but only the parts used by the Andood application) can be found on the project's web page [GCW]. Some of the most important aspects of the API are discussed in chapter 5.

3.2. Business model

The business model of the Andood application is an integral part of Dood service's business model. Business owners or their authorised agents are offered a list of additional features of the Dood web site or other components of the system (for instance mobile applications). They

can choose to activate some or all of them by purchasing one of the predefined subscription packages. Subscriptions are paid once a month or once a year according to the business owner's preferences. The list of additional features available in these subscriptions contains, among others, the following:

- **freedom of selecting a *front* review** – this review is strongly associated with the business and always appears first when there is no place for other reviews. This is also a review which is sent by the server to the Andood application together with *additional businesses* objects, so there is a very high probability that this would be the first one which users encounter. The *front* review is selected automatically by the system if no choice is provided by the business owner (no subscription package has been purchased).
- **sponsored links** – these links are contextual so they appear only when the system detects that the user is interested in a certain type of businesses. Opposite to standard "paid links" (like Google's AdWords system), the target page of those links is a sub-page within a Dood system (the business's profile to be exact). Moreover, they have a rich format which apart form a standard name and rating contains a picture, front review, address etc.
- **sponsored links with promotions** – these links are standard sponsored links which additionally come with the promotion title and description (the title cannot be longer than 140 characters). Sponsored links with promotions are standard contents used by the Dood system to supply the Andood widget with *Promotion* objects. They can be purchased in packages as part of the subscription or separately via an SMS service.
- **category leader** – a selected business can become one of the category leaders and by this it appears on a respective category page when the user navigates through the category's list. Businesses which currently possess the *category leader* status are more likely to appear on the Andood widget when users select filtering by category.

The whole list of additional and paid features can be found on the Dood web site. This section focuses on these aspects of the business model which are related to the Andood application.

From the users' perspective the Andood application will always be free to download from the Android Market (in the predictable future). It is company's top priority to build a huge number of users who regularly utilise the Andood application to receive relevant information about current offers from their whereabouts. The application will not be distributed in any other form than by a direct installation from the Android Market mainly because of the security issues. However, LemoNET will start a marketing campaign that would link people with the application on the Android Market. It is worth mentioning that some new and interesting forms of sharing direct links are available, with the 2D barcodes being the most innovative. People can simply take a photo of the special 2D barcode (square shape) and be automatically redirected to the application's page on the Android Market. Barcodes can be distributed in a printed form (as leaflets or newspaper advertisements) or via the Internet.

Having in mind that Andood offers a unique functionality on the Polish market the author of this paper together with other business partners from the LemoNET company have decided to launch the application with no extra cost for the business owners for at least the first six months. This does not apply in the SMS use case mentioned in section 2.1.1 as the system relies on external services which are not free. Consequently, no additional payment will be charged for using this feature of the Dood service (no increase in prices of subscription packages). The number of available *Promotions* and their specifications is presented on the web site. For LemoNET the most important goal is to attract as many potential users as

possible; business owners who regularly supply new promotions (advertisements) and users who simply use the Andood application and receive this content. As for the SMS system a small price of 1 PLN (ca. $0.35, plus tax) has been set for a single message sent to the system. Moreover, in order to sustain the incoming traffic and protect the system from unwanted messages (SPAM) there is a limit of one new *Promotion* per day per single business.

3.3. Market analysis

Andood is definitely a unique application on the Polish market and currently in Poland there are no signs of any other, similar projects in progress. The only serious competition would be the well known foreign company like Facebook or Google launching an app with corresponding functionality. Fortunately, Facebook has already started a project called *Facebook Places* [FCB] which does not exactly match the functionality of Andood and focuses more on geo-localising friends in respect to *places*. More importantly, Dood's database already contains a large set of businesses and reviews from many major Polish cities so there is no threat of loosing users because of poor application's content.

Google products (like maps, navigation) so far have been only used to search and navigate for different places. Even though the database itself is relatively large, the majority of businesses do not contain a lot of reviews and all of them are taken from external providers. Moreover, *Google Places* has already been added to the standard Android apps pre-installed on some of the devices, and yet, it has not made a spectacular success on the Polish market.

The most important advantage of the Andood application is that it collects the well-known functionality of Facebook (people) and Google (search for places) and combines it in a form that is convenient for the average Polish user. All reviews and businesses' information are in Polish and from the beginning everything has been prepared and implemented by young Polish people who understand the unique needs of potential users of the app.

An interesting trend on the Polish market is the growing popularity of web services like Groupon, CityDeal or MyDeal which offer (in many major Polish cities and also worldwide) a variety of discounted coupons for different services like restaurants, hair dressers, theatres etc. By definition these coupons are not valid instantly after purchase so sometimes it takes even couple of days before a customer can get what he or she has ordered. On the other hand, coupons can be redeemed during a long period of time (even several months), yet it turns out to be one of their disadvantages. In most cases people would want to get what the have purchased as soon as possible, or else they would try to redeem the coupon on the last possible moment in order not to loose the money they have invested. The result is that the *advertisers* would probably be overloaded at the beginning and at the end of promotion.

In this perspective Andood is somehow one step further in terms of business models. First, there is no big pressure on companies who sell such coupons to release mobile applications that would enable users to buy them directly from their phones or tablets as these coupons cannot be redeemed instantly. Hence it not so likely that these kind of applications would appear shorty. Second, Andood presents much more targeted advertisements (which can be regarded also as *coupons*) in terms of geo-localising customers and filtering offers according to their preferences. Andood is basically trying to provide users with information that they can use the moment they open the application, information that would lead them to businesses in their nearest neighbourhood. This approach is much more suitable for mobile platforms than the original concept of services like Groupon or CityDeal. Moreover, the growing popularity of these web sites is a clear sign that users are very much interested in buying discounted goods, not necessarily limited to things that they can purchase in a local supermarket.

Chapter 4

Authentication and Security

One of the main arguments to support Android versions from 2.1 onwards only apart from the market trends described in chapter 1, is the very rich set of new features introduced in 2.x platforms. These features are obviously optional (to withhold backward compatibility), but if properly used, they can leverage developers efforts in implementing new applications. New releases of Android gathered the most common actions for third party applications and pre-implemented some of the mechanism in a form of separate modules that developers can integrate with their own code. One of such examples in the *Account&Sync* mechanism which is extensively used in Andood application.

4.1. The Account Manager in Android 2.x

The core of the *Account&Sync* mechanism is the *AccountManager* instance which holds information about users' credentials for multiple accounts. This is a centralised system which requires applications to explicitly request access permissions for this service in their manifest files. Users can see these permissions and approve of them during the installation process which guarantees security of the whole mechanism.

The second important feature of the *AccountManager* is that it can store users' credentials persistently. It means that in the majority of cases users will not be asked to log in to the same service twice. It definitely follows the idea of personalised smartphones where users expect to access different services (like email, Facebook, Twitter) instantly, without being bothered about providing passwords every time they wake their phones up. The novelty here is that thanks to the centralised mechanism the *AccountManager* can present its active accounts (also from third party application) in the Settings Screen, where users expect such information to appear in the first place. Moreover, one account can be used by many third party applications, so for instance one can easily write an application which uses the Google account necessary to set up the device. This however does not mean that they can retrieve the user's credentials (like password) in plain text. The authentication process runs in a separate service and third party applications only retrieve something called *Authentication Tokens*, which they can use to communicate with external services (like Facebook, Twitter etc.). In case of failure in requesting the token, Android automatically launches the *Authenticator* service which handles the process of authentication. This service however is provided by the developers who established that particular account type, so it is only their code which is used from the moment the system decides to validate the account. It guarantees the security of stored users' credentials and enables developers to let the system manage the whole process of authentication for them, regardless of whether they use a third party application's accounts

or their own.

The *AccountManager* is also integrated with the synchronisation mechanism which was the key argument in designing the architecture of the Andood application. The fundamentals of the Sync mechanism in terms of synchronising RESTful data objects are described in sections 5.1 and 5.4, so this section will only cover these aspects of the process which are related to the security issues. The whole idea of implementing a separate, non-application-dependent module for authentication was probably taken from a simple observation that many Android applications use the scenario of authenticated access to the external resources. Consequently authentication process is only the first part in this general flow. As the second part, synchronisation can also be externalised from the main application to some extent. The *SyncManager* service (also centralised) handles the sync requests from many applications and triggers the appropriate *SyncAdapter* and its *onPerformSync* method. Not surprisingly, this method gets an instance of the Account as a parameter and can both request a validated token or invalidate the existing one in case of any failure. The content of the *Authentication Token* is not specified by the *Account&Sync* mechanism so it can be any string representing randomly selected characters, the session key used to communicate with the third party server, or even the user's password (which would be a very stupid solution). The last example shows that even with the ready-to-use mechanism that provides a secure system of authentication there are still many ways of "breaking" it if the *Authenticator* module had not been written properly.

4.2. The system of authentication in Andood

The *Authentication* module (figure 4.1) in the Andood application consists of three main classes:

- *AuthenticationManager* – a helper class which handles an anonymous account,
- *AuthenticationService* – a service which handles an authentication process and instantiates an *Authenticator*,
- *Authenticator* – it extends *AbstractAccountAuthenticator* – the most important methods are implemented here: *addAccount*, *getAuthToken*, *confirmCredentials*.

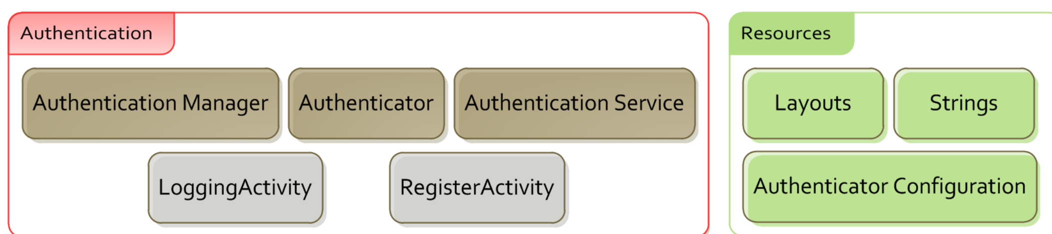

Figure 4.1: Andood's Authentication Module

The *Authenticator* can be registered in the Android system using the *AnstroidManifest* XML file where developers specify a service which should be launched when a user requests a new account of a certain *AccountType*. There is another file called *AuthenticatorConfiguration* where developers should provide the label, icon and the *AccountType* constants in order to populate the Preference Screen (Accounts&Sync section) with new account's type information. The instance of *AuthenticationService* (launched by the system) delegates it's work to

the *Authenticator* which extends the *AnstractAccountAuthenticator* class. The *Authenticator* provides implementation of the most important methods like adding a new account or validating the existing one, and most commonly it starts an Activity to handle this process. In the case of the Andood application the *LoggingActivity* is responsible for logging and validating existing Dood accounts. The *RegisterActivity* has already been implemented but registration requests are not yet supported by the Dood server (that is why the button "register" is disabled, as shown on figure 4.2). Direct registration via the Andood application is on schedule for the next release. The remaining resource files specify layouts of Activities or externalise (multiple language support) strings used in the process of authentication.

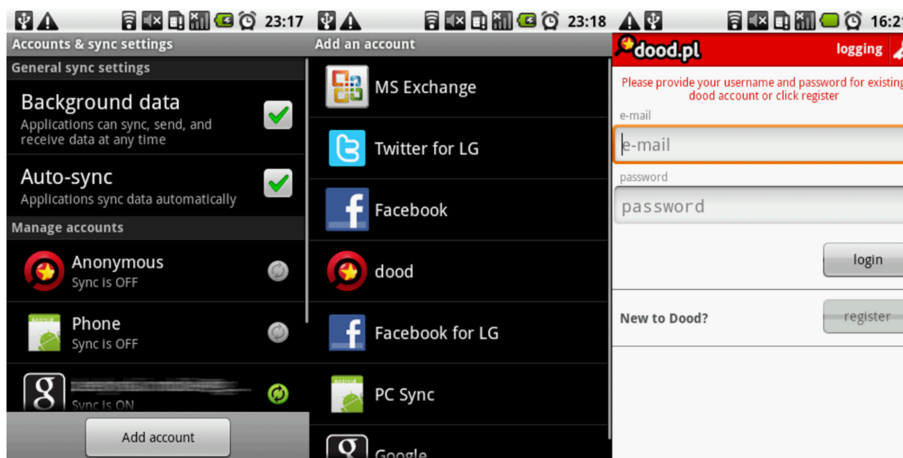

Figure 4.2: The Andood authentication module in the Android system settings.

The reason why the Andood application contains the Authentication module is that some data exchanged with the server can be personalised. It can be either users' credentials (which have to be protected) or simply data which does not represent any value to anybody else. The Dood web service offers a personalised search results based on many user-specific variables like: user's search settings, history of reviews or covariances of ratings among friends and/or other users. However, the idea was not to obligate new users to create new accounts if they want to use the Andood application. Similarly, one does not need to register in order to use the Dood.pl web search engine. By not registering, users agree on receiving slightly worse (meaning not personalised) results. On the other hand, the dood.pl web service as well as the Andood application are both ready for use right after installation, or loading the page.

In terms of the *Account&Sync* mechanism available on the Android platform problem was that synchronisation via *SyncAdapters* (see section 5.4) is not possible without providing a not empty (not null) account. This is quite reasonable as the system does not want to launch background network operation without any visible sign that can be observed by the user. All accounts (and their Sync actions) can be quickly deleted in the *Account&Sync* preference screen. The problem was solved by introducing the anonymous account which gets created automatically whenever the application sends a request for a network operation and no other account has been established by the user. Figure 4.2 shows the anonymous account in the *Account&Sync* preferences. Users can replace the anonymous account by adding a new one (their own), but they cannot create more than one Dood account on a single device (it would not make any sense). Moreover, as soon as they erase their personal account, the Andood application will reestablish the anonymous account during the next Sync operation.

The anonymous account had been created because the API supports unauthorised sessions (to serve data to the users who are not logged in). Anonymous accounts still need to validate

themselves on the Dood web service in order to receive a *session key*. The key is a security mechanism which protects the Dood web service from spy bots. There is a day limit for requests per session key as well as a limit of requests starting the anonymous session (which create a new session key) from a single device. This is how, without any notification, users who install the Andood application communicate with the server only via authorised channels.

4.3. Personal information protection

Security in Android applications is certainly not limited to the safety of establishing authorised sessions when transferring data to and from the external servers. Since Android devices are mainly smartphones which users carry around with them everywhere, they can store a lot of personal information. Sometimes it is not even a matter of politeness to ask users whether they are willing to share this information, but a matter of obeying or breaking a regional law. The most critical example is the user's location described in the following section. However, there are other examples of personal information stored on the Android devices that could leak outside. These are contact lists, dial history, browser history etc. Each time a developer deals with a system managing such information, he or she should always inform the user about possible actions especially if this data could be sent outside of the device.

4.3.1. Location aware applications

A separate, special case among personal data protection policies is related to *location aware* applications. On the one hand these kind of applications are prised for their contextual, targeted data. Users can retrieve content only related to things which are in their nearest neighbourhood or exchange their current location with their friends to find each other more easily. On the other hand, in some countries revealing such information (this includes sending it to the external server) is not legal without a direct user's authorisation. Developers should definitely be aware of that and ask users for permission before sending such information outside. A single pop-up dialog box with the information and "yes" or "no" buttons displayed only once would be quite enough to satisfy law regulations and do not bother users needlessly.

The Andood application takes advantage of geo-location to provide targeted data about current promotions from the user's neighbourhood. Instead of asking users once for their permission, Andood offers a preference option to enable or disable sending location to the server. The default value is set to "true" (OK to send user's location), but the preference screen is the first one that is displayed after choosing the Andood widget from the list, so users can choose to disable it if they want to. The Andood widget still operates normally if users decide to switch off this option, but it simply uses a default city so it can serve a slightly worse content.

Chapter 5

The Data Exchange Model

The Andood application is entirely dependent on the external data provider. Without a source of up-to-date content with every minute it would become less and less useful to the user as it would only be able to present a data backed up in a local memory. It would also be pointless to pre-install a database of businesses or promotions into the device's memory because this data set simply changes too often (not mentioning a significant amount of space that it would take). Even though data transfer is not always free (still, in most cases only WiFi can, but does not have to be free) smartphone users are somehow prepared for that expense. From this perspective one can say that Android has anticipated the market growth as well as reductions in a cost of 2G/3G network data transfer. The whole system is somehow designed to be able to connect to the Internet whenever there is a need. Naturally, there is a huge difference between a 2G/3G network connection and a WiFi connection. And even though they are both handled exactly the same way (Android hides it behind the abstraction layer) developers should be aware of the type of the connection they are establishing and try to minimise the amount of transferred data accordingly.

The decision of implementing the data exchange model is always fundamental from the point of view of application's architecture. Not only does it require a transportation layer for the objects, but also a system of synchronisation between local copies and external providers. Fortunately Android 2.x platforms provide new features which are very useful when developing applications performing some kind of synchronisation. These new features are obviously optional (to withhold backward compatibility with 1.x versions), but they have a major impact on the application's architecture design. In fact, developers should make their decision about using these techniques as soon as possible, because they completely redefine the way the application deals with synchronisation.

5.1. Implementation of REST clients in Android

The REST (or Representational State Transfer) architecture is a commonly used style for distributed hypermedia systems [WRS]. It defines certain constraints such as statelessness, client-server architecture, support for caching etc. Most importantly, in the case of RESTful web services which accept requests via the HTTP protocol it is a widely spread standard adopted by numerous companies.

Android itself does not provide any dedicated libraries to support implementations of RESTful APIs, nevertheless this subject (from the point of view of architecture design patterns) has been broadly presented by Virgil Dobjanschi during the Google I/0 (May 2010) session called *Developing Android REST client applications*. The following section cites the

most important parts of this presentation to collate it with the solutions implemented in Andood application.

Despite not having a dedicated library, Android provides a number of pre-implemented solutions (including some new features of Android 2.x platforms) that can be extensively used during the implementation (provided that developers know about them). There are basically three architectural patterns advised to Android developers who want to implement a RESTful API as part of their applications. In all cases the main problem is how to design an action flow between components of the system from the presentation layer (*Activities*) down to the network and local memory operations.

There is nothing like the best way of implementing a RESTful API client on Android platform because each solution can, and should be modified according to the unique requirements of the protocol and – generally – application. On the other hand, there is one pattern which is definitely not advised (still some developers tend to use it). In this approach RESTful methods (they launch a long-running operations, like network operations) are executed directly from the UI thread (*Activities*), or from the inner class working thread which stores data only in the memory storage. The first option would eventually end up with the application causing an ANR (Application Not Responding error). Network operations on the UI thread would certainly slow down the application and make it less responsive (please refer to chapter 6). The second option is also bad because, even though applying a working thread would prevent the application from causing ANRs, the data would not be stored persistently. It means that whenever the activity is launched, exactly the same data could be downloaded many times in a row wasting not only CPU, but – most importantly – bandwidth. Also anytime the activity stops the effect of the background operation would be lost (it can happen quite often without the user's acknowledgement). So despite the fact that the application works normally and the RESTful methods are executed safely, it is definitely not a good example to follow.

Figure 5.1 shows a user icon on the left of the *Activity* component (it appears also in the following subsections) to indicate that this is the only part of the system which has a graphical representation (can be visible). Android has to allocate a lot of the processor's time to redraw all views attached to the foreground *Activities* and – according to the section 1.2.3 – it honours the process of these components with highest priority. Executing any long running operations (even access to a local database) on the UI thread is therefore not advised. In all presented design patterns the key issue is to get off of the main (UI) thread as soon as possible in order to provide a seamless experience to the user.

5.1.1. Solution I – *Services*

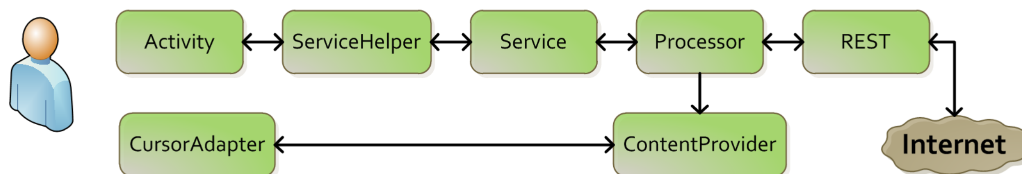

Figure 5.1: Solution I – *Services*

This solution presented on figure 5.1 is based entirely only on the code from Android 1.x platforms so it does not use any mechanisms introduced in newer versions. From the bottom of the system's architecture the *REST* component deals with creating, sending and retrieving objects via the HTTP protocol. It specifies data types and implements a connection as well as a parser. Data can be sent in many available formats (also binaries) like JSON (Android

has a fast library that supports parsing JSON objects) or XML, but definitely should be GZIPed to reduce amount of transferred bytes.

The second component (*Processor*) is responsible for synchronising the local memory with the external source (the one that the *REST* component is using). By adding some flags to each transferable object developers can have a full description of the actual state of each particular object with regards to the synchronisation. The *Processor* stores data using a *ContentProvider* component which most commonly is backed up by a local database. It could also be a more complex solution with the external memory (SD card).

The next component is *Service* which runs in context of the application. This *Service* starts long running (*Processor*) operations in the background and is independent from the life cycle of the *Activity* so all requests can be handled even if the user decides to leave the *Activity* or some new event has caught his/her attention (for instance incoming mail). The service can use an Intent API to receive intents from the upper layer (UI), unpack them and delegate them to the *Processor* component. After the task is completed it notifies the upper layer by calling a proper service via a callback method. In this approach *Service* is a stateless component so it does not need to store any of its data persistently. More importantly, as soon as the *Service* is done with its job, developers should make sure to shut it down. If they forget to do that, the system will not be able to decrease the priority of this process (as the service is still running), so it is less likely to reclaim memory even though it is not used anymore.

Implementing a middle layer as a singleton class which exposes a very simple API is a quite common pattern in developing multi-layer applications. Hence the role of the *ServiceHelper* class (another component of this solution) is only to transmit requests and data between *Activities* and *Service*. However, when developing a RESTful API it is also important to reduce the number of unnecessary requests whenever it is possible. Such situations can happen when users repeatedly click on the refresh button for instance. To solve this problem before the *ServiceHelper* dispatches a new request (as an Intent), it can ask the running *Service* if any corresponding operation is still pending and refrain from sending duplicates.

The last component of this architecture is *Activity* which simply calls a *ServiceHelper* whenever the user "wants" to perform any REST operation (by clicking the button or scrolling down the list etc.). It registers and unregisters (according to its life cycle) certain listeners that are called by the *ServiceHelper* when the REST operation is completed.

5.1.2. Solution II – *ContentProvider* API

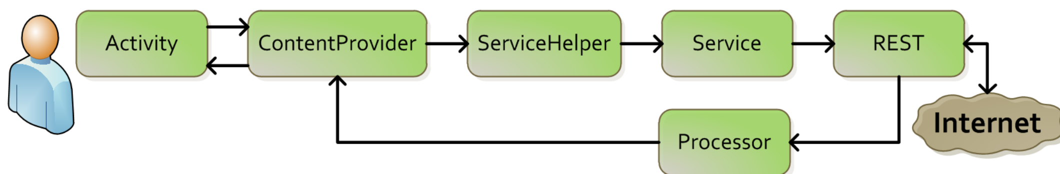

Figure 5.2: Solution II – *ContentProvider* API

This solution (presented on figure 5.2) does not introduce any new components with respect to the previous one. It simply makes a full use of the *ContentProvider* and its exact correspondence with the standard REST operations (in pairs: Query from Content Provider – REST GET method, Insert – POST, Delete – DELETE, Update – PUT). In this solution the *Activity* calls (via asynchronous tasks) the appropriate *ContentProvider* which executes its task (like inserting a new row, or deleting it) keeping the data in a transitional state. However, before it finishes, it calls a *ServiceHelper* to request a new REST operation. From this moment

everything works exactly the same way as in the first solution in the forward path. The return path can be shortened by calling *Processor* directly after the REST method finishes. At this point it makes no sense to pass the message back through the *Service* and *ServiceHelper*, but instead the *Processor* can directly access a *ContentProvider* to change the data or simply update its current state. As soon as the *ContentProvider* detects a modification in its data set it calls all of the *ContentObservers* (listeners) with the notification message. The *Activity* or any other components which rely on the data retrieved by cursors from *ContentProviders* can call *requery* method and refresh the data presented to the user.

As this solution shifts the main responsibility from the *ServiceHelper* and *Service* to the *ContentProvider*, it is also its job to deal with network operation failures. When a REST method fails to execute (because of timeouts or unreachable server for instance) a *ContentProvider* has to keep that in memory and be able to retry sending the request (hopefully in the exponential intervals, not to retry every two minutes). It can be done by setting up an *Alarm* and querying *ContentProviders* data for any rows in a transitional state.

5.1.3. Solution III – *ContentProvider* API and *SyncAdapter*

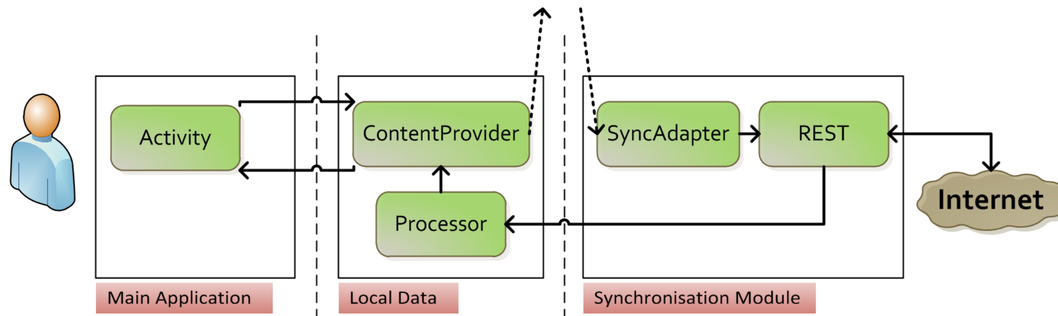

Figure 5.3: Solution IIIA – *SyncAdapter*

The third solution uses the *SyncAdapter* mechanism introduced in Android 2.0. Even though it has been about a year and a half from the moment this platform has been released, many developers are still not well acquainted with this concept and only a fraction of them have adopted it in their applications. Some more details about *SyncAdapters* can be found in section 5.2.

The *SyncAdapter* replaces the *Service* and the *ServiceHelper* components from the previous solution. An *Acitivity* calls an appropriate *ContentProvider* which executes the requests instantly, possibly leaving some data rows in a transitional state. Before it finishes a *requestSync* method is called in a *ContentResolver* adding a new request into the *SyncManagers'* queue. When *SyncManager* decides to execute this requests it launches the *SyncAdapter's* $onPerformSync$ method. Usually implementations of *SyncAdapter* would query a *ContentProvider* for all rows that need to be synchronised, create a new request and execute the *REST* component passing it as parameter. After the network operation is completed, they start a proper synchronisation, having both response from the remote server and access to the local *ContentProvider*. *Activities* are notified about changes by the *ContentObservers* listeners.

Two possible implementation of this solution have been presented on figures 5.3 and 5.4. The only difference is that a *REST* component on the latter one does not have to know about the remaining parts of the system's architecture. This could be useful for instance when this component is in fact a separate library (like in the Andood application).

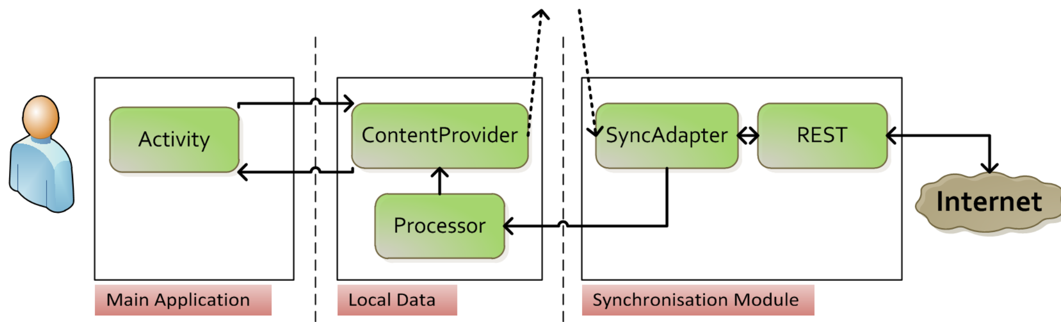

Figure 5.4: Solution IIIB – *SyncAdapter*

5.2. SyncAdapters in Android 2.x platforms

A *SyncAdapter* together with a *SyncManager* is basically a centralised system for synchronisation of local and remote content. Developers can attach their implementations of the *SyncAdapter* (extensions of *AbstractThreadedSyncAdapter*) with the appropriate "sync action" (*onPerformSync* method) to the instance of the *SyncManager*. System deals with executing requests in a queue by means of a separate Service (and thread). It also implements a mechanism of retrying the requests which may fail to execute because of many reasons (IOExceptions, parsing problems etc.).

Undoubtedly the biggest advantage of the *SyncAdapter* and *SyncManager* mechanism is that it gathers all of the requests from all applications that use it across the platform (this includes the Android apps implemented by Google like Gmail). Indeed, the more developers use it, the more powerful this mechanism becomes. However, there is one obvious disadvantage: because the *SyncManager* gathers all requests from many applications it may happen that our requests are not executed instantly. The system tries to balance the network operations and avoid situations when for instance four different application have HTTP connections opened, they all do parsing, they all try to access Content Providers simultaneously. This could have a dramatic impact on battery life not to mention that these applications could block themselves in access to similar resources causing even bigger delays. By using a *SyncAdapter* mechanism developers can simply call a non-blocking *requestSync* method and keep doing other things while the system takes care of the whole process of executing long-running operations. More importantly, the Sync operations can compose a separate module within a new application.

Similarly to the authentication mechanism described in chapter 4, synchronisation via *SyncAdapters* allows developers to write their applications without being bothered so much with the system-related issues. In terms of the authentication the main application component was simply requesting an *AuthToken* and in case it could not be provided, it was a role of the separate Authentication module to handle the authentication process (launch certain Activities etc.). *SyncAdapters* work exactly the same way. It is their unique responsibility to communicate with the remote data source (usually by means of a *REST* component) but also deal with all potential failures (exceptions). The main application would not be suspended (by default) in case the *SyncAdapter* crashes, neither would it be notified (by default) about such an event (this feature has been introduced in Android 3.0).

5.3. Dood API

The second version of the Dood API (ver. 2.0) has been designed and documented by Łukasz Kidziński and the author of this thesis. The fundamental principles behind this work were to guarantee a free access to the Dood database via API for all third party applications and to make the exchange format more flexible compared to the first version. The parts which have already been implemented are presented on a web page hosted by Google (Google Code – http://code.google.com/p/dood-api/). API specifies data types, format of requests, error codes as well as a standard response structure. Additionally, the rules of conduct are available for developers who wish to use the API as part of their applications. This section focuses on the API's aspects related to the Data Exchange Model which are necessary in order to understand data synchronisation mechanisms implemented in the Andood application (see section 5.4).

5.3.1. Data model

At this stage the server side of the implementation of API (ver 2.0) supports four different data types along with headers and session control objects. There are many more transferable objects types across different Dood services and the majority of them are planned for this version of the API, but simply they are not publicly accessible as of April 2011. The Andood application in its current form uses only the available object types, but the following release of the *launch* application will certainly introduce many more (like *Review*, *User*, *Event*, *Lists*, etc.). The table below presents supported objects with their attributes:

Object	Attributes
Business	*name, longitude, latitude, address, city, phone, rating, www, gallery, photo, total_reviews, total_promotions, reviews, categories, open_hours, review_body*
Promotion	*business, title, description, expires*
Category	*name, parent_id, photo*
Location	*latitude, longitude, activity, city, default_city_used*

Moreover, every object is encapsulated in a *Header* which adds four attributes (model, id, version, deleted). The only exception is the *Category* object which does not have all of *Header's* attributes.

The majority of the attributes' names are quite self explanatory, but there are some exceptions:

Attribute	description
review_body	contains a business's front review body
activity	is a float number of a range 0.0 – 1.0 which indicates the density of the businesses and promotions within the area for which the *Location* object has been created
default_city_used	a boolean flag indicating whether a default city has been used by the server to create a *Location* object. This can happen either when a *use_default_location* flag is set, or when the given geographical location is too distant from any record in the database.
deleted	indicates that the requested objects have been deleted from the remote data source and should be removed from all local copies.

5.3.2. Session control

Because of the security issues described in chapter 4, a Dood server requires an authorised *session_key* for all requests that might migrate some of the database content (regardless of the direction). Each time a client application starts a session (see section 5.3.3) an *APISession* object is returned:

Object	Attributes
APISession	*session_key, id, photo_server, category_version*

Along with the newly created instance of the *session_key* client applications are informed about the current version of the category tree. *Category* objects are not so prone to change hence it is easier to handle them at once (as one object). When a client application detects that the local copy is out of date and that the user wants to perform an action where categories are necessary, it downloads the whole tree with one request. Another attribute is a *photo_server* URL which is used to build resource requests. This solution gives the Dood server the ability to balance the network traffic between multiple machines (see Content Delivery Network – CDN [WCD])

5.3.3. Object requests

The complete list of supported requests can be found on the API's web site. The Dood API follows standard REST conventions of accessing remote object where each request consists of a server URL followed by the model name and action that the client wants to perform. Therefore any particular resource can be reached by the URL which concatenates four elements: server URL, model name, "get" action and resource ids passed as GET parameters. So for instance:

```
http://dood.pl/api/business/get?ids=123&session_key=xxxxxxxxxx
```

(with a proper session key) should return a single business object packed in a standard response format (please refer to Data Exchange Structure section on the API's web page for more information about a standard response format).

Instead of quoting all the supported requests from the API, this section focuses on the top three ones which are the most extensively used by the Andood application and only describes a "success path". API tries to honour standard HTTP response codes (including error codes), but sometimes an additional message or code is passed with the response (for more information please refer to [GCW]). The first request starts an anonymous session so it is always used by the authentication module (no direct user's action can launch it):

session/start	
parameters	*none*
response *(success)*	single instance of *APISession* object

The second most common request is the login request:

session/login	
parameters	*username (email), password, system*
response *(success)*	single instance of *APISession* object

The *system* parameter is in fact for statistical purposes, and it is set automatically by the application (user's cannot modify that field). The third request is called the "widget request" because it is used to refresh the widget's content:

business/get	
parameters	*get_promotions, set_location, where, categories, only_open, mode*
response	single instance of a *Location* object and up to 20 *Businesses* and *Promotions* objects

Not all parameters from this request are quoted here, and not all of the quoted ones are obligatory. The first two differentiate this request from a standard *business/get* request which would fetch only *Business* objects. The *where* field can be both a name of the place (street name, number etc) and the exact geographical location (in a *latitude,longitude* format). *Categories* and *only_open* parameters are optional and allow users to filter search results. The *mode* flag indicates the requested frequency of refreshing the application's data and has an indirect impact on the *activity* parameter in a *Location* object which is retrieved from the server.

5.3.4. Local cache support

This part of the API is not yet supported by the server as it was not a key issue to implement it in the Andood application. Nevertheless, the main – *launch* – application will certainly make an extensive use of it. The need of improving local caching mechanisms was the fundamental requirement of the new API. The goal was to reduce the amount of data being transported by the network, because the majority of operators charge users on a per-byte basis. GZIP was naturally a great help, but the API was heading towards more effective solutions, where compression was only a final stage.

The first idea was to split transferable objects into different detail levels. The explanation for that is quite simple. Very often users simply browse through a list of objects (say businesses). They do not enter a detailed view (*Activity*) for every particular list element, where all of the information is required (oppositely to a list view itself). By splitting the objects into detail levels (most commonly only two different levels) the client application can reduce the amount of transferred data. On the other hand, there is an important drawback of this solution. Namely, whenever users enter a details view for any parcelled object they have to wait until the remaining part of this object is downloaded. It turns out that reducing network data transfer sometimes has to be compromised by the need of creating a positive, seamless user experience. Provided that establishing an HTTP connection can take up to 2-8 seconds, the idea of applications waiting on every screen for new data to be downloaded seems like a nightmare for users. Also the number of synchronise requests needed by a single view (screen) is regarded to be a bottleneck. It is totally acceptable to download some photos (like avatars) after the view is presented to the user and update it afterwards. But having two, three or more of synchronise downloads (especially from one server) for a single screen is something that needs to be reconsidered.

Another idea was to allow client applications to send a message (*declaration*) to the server that they have a certain object in their local cache memory. This of course had to be supported by the object versioning system because the data served by Dood servers can change within time. However, to avoid creating a separate "request" this new information can be pasted into every other request.

The Dood API implements a combination of these ideas. Still, the most effective solutions in terms of reducing network traffic are GZIPing all of the content and paging data (search results for instance). The only object which has been divided into two levels of details is a *Business* object (*BusinessShort* being its shorter version) because searching for businesses (thus presenting a list of results) is the most common action for Dood services. All client applications will be able to choose (by using a special request parameter) whether they want to get a full or reduced version of the *Business* object (full is default). However, it is the versioning and "declaration messages" system that the Dood API should be more beneficial from. Every request (except for session control requests) dispatched to the server can hold a number of headers of the objects that the client application is expecting to receive in response. When the server receives such a message it can either return an updated version of this object in response, or simply ignore it which would ultimately mean that the client has the most up-to-date version. Two basic rules are crucial in this solution:

1. Whatever comes from the server is always valid, and most up-to-date. This data should always replace the one that client applications store in local memories. There is no method to request for an old version of any object.
2. If the response for a request containing a "declaration" message does NOT have the object from this "declaration", it means that the client application holds the most up-to-date version of this object.

The positive conclusion from these rules is the fact that when using "declarations", client applications expect NOT to receive new objects in response. There are many situations where this mechanism could be used. A good example would be a *Business* details view where users have the option to see reviews of this particular business. The moment the client application sends a request for this *Business* object can be used to declare that it has some of the relevant reviews already stored in local memory. If all goes well this would save at least one request being sent to the server for the cost of only several additional bytes attached to another request. Most importantly, it would produce a seamless experience to the user when transitioning to the new view (*ReviewActivity*), no long-running progress bar would be needed.

5.4. Data synchronisation – model

The Data Synchonisation Model of Andood is based on the third solution presented by Google (section 5.1.3) with couple of improvements and other modifications related to the specific requirements and characteristics of this particular application. Indeed, in the case of Andood the *Account&Sync* mechanism introduced in platforms 2.x was the most important argument in the decision of supporting only Android versions from 2.1 onwards.

Regarding changes applied to the original Google's model: first of all, the *REST* component has been replaced by the JCL library. The entire library is platform-independent hence it should not call any methods from any specific environments (especially any Android-specific classes). The only two modules which can access a *Session* object from the JCL library directly are the authentication module and the synchronisation module. Both of them are somehow independent from the main application components (*Activities*, *Widget* etc.) and can be triggered directly only by the system.

Secondly, between the presentation layer and the synchronisation layer there is a Local Memory module which is a little bit more autonomous than in the solution Google provides. It contains not only a *Content Provider* itself, but also a couple of other classes that handle

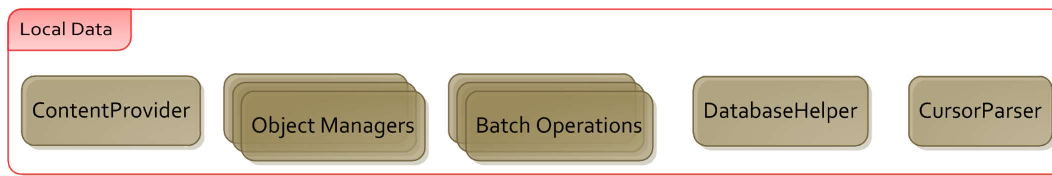

Figure 5.5: Local Data Module in Andood

data management related to the local memory (figure 5.5). *Batch operations* are there for all *Content Provider* operations that can be executed in chunks. This is especially useful in synchronisation when the application retrieves a number of objects in one response and has to update a local memory with all of them. *DatabaseHelper* is just a helper class where a local dood SQLite database with table names and columns is defined. *CursorParser* is another helper class which parses the rows from a database into objects (types are defined by the JCL library). *Object Managers* manage the synchronisation process. They query a *ContentProvider* and synchronise any incoming objects with the existing rows. So, for instance, they decide whether a new row has to be created or an old one should be updated.

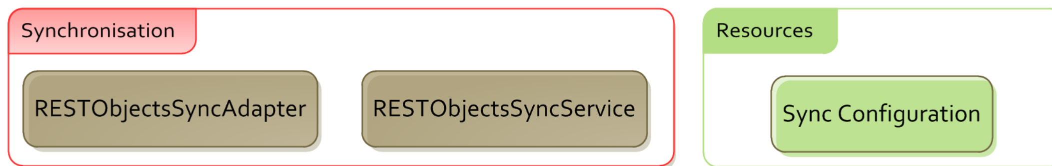

Figure 5.6: Andood Sync Module

The Andood's synchronisation module (figure 5.6) consists of the *SyncService* class and implementation of *AbstractThreaderSyncAdapter*. Both classes are accompanied by the XML configuration file which registers a new Sync mechanism within the system (some changes in the Manifest file are also necessary). *SyncService* is called by the system when the *SyncManager* decides to perform a synchronisation. The system of synchronisation can also be used to refresh the application's content regularly (like synchronising contact lists or "statuses" of friends). Periodic requests can be triggered by the system itself (this should be set up in the configuration file) or using an *Alarm* mechanism (please refer to chapter 6).

The *SyncAdapter* (called *RESTObjectsSyncAdapter*) is responsible for calling REST methods (JCL library) and passing the result to the appropriate *Object Managers* from the Local Data module. At this stage it handles three different types (modes) of synchronisation:

- *categories* – ALL categories are synchronised with the remote data source at once (the *SyncAdapter* does not know ids of those objects).
- *selected objects* – only selected objects are synchronised with the remote data source (the *SyncAdapter* has to know ids and the types of those objects).
- *widget update* – widget content is synchronised (the *SyncAdapter* knows only about the search filters, but does not know the ids of the objects obviously).

5.5. Data synchronisation – action flow

The synchronisation model in the Andood application is quite complicated as it incorporates nearly all of its modules. To avoid getting into too much detail and less probable (special)

cases this section focuses on the single, most common action performed by this application (refreshing the widget content) and presents the whole path of synchronisation. Naturally, with the release of the following *launch* application this action will not be that frequent (compared to the others), nevertheless the mechanism itself is not expected to change significantly.

Figure 5.7 shows the first situation when the *Widget* tries to update its content using only local memory. Such a situation can happen when the user clicks on the *Next* button or a new content has been downloaded by using a synchronisation system and the widget is simply trying to refresh itself.

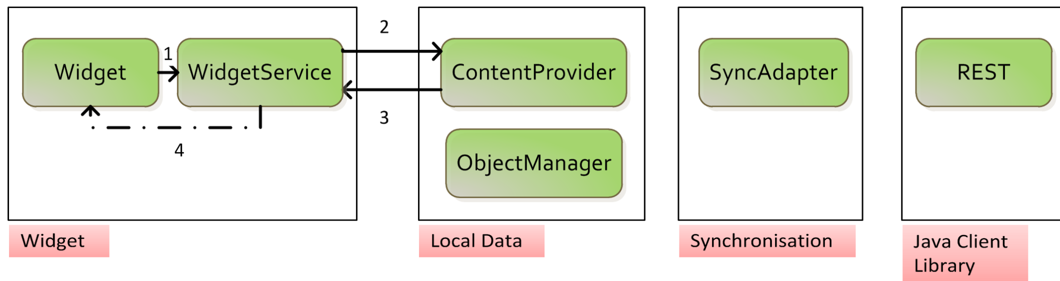

Figure 5.7: Synchronisation in Andood – local data

From the moment the widget is attached to the Home Screen it becomes a part of the Home Application's activity's view. The *Widget* (class) extends a standard *AppWidgetProvider* class which extends a *BroadcastReceiver* class. It updates the visible widget's view using an instance of a *RemoteViews*, but because it is a *BroadcastReceiver* it should be as responsive as possible. It is also vulnerable to the widget's lifecycle (deleting, adding new instances etc.) so any long running operations should not be placed inside of it. Instead, the Widget delegates all of the long running operations to the *WidgetService* instance which implements an Intent API mechanism.

From the left-hand side of figure 5.7 the Widget receives an intent and classifies that it is a local memory update request. It calls **(1)** a *WidgetService* using new intent to delegate this job. Thanks to the Intent API mechanism the *WidgetService* handles its lifecycle automatically. Not only does it start its work in a separate, working thread, but it shuts itself down after the job is done. All requests are also guaranteed to be executed synchronously. Since this is a local memory operation *WidgetService* queries **(2)** a *ContentProvider* for new data, receives a response **(3)** and parses it. When new data is ready to be presented to the user, the *WidgetService* updates **(4)** the widget's view using *RemoteViews*. Again, it is not literally speaking accessing a *Widget* class.

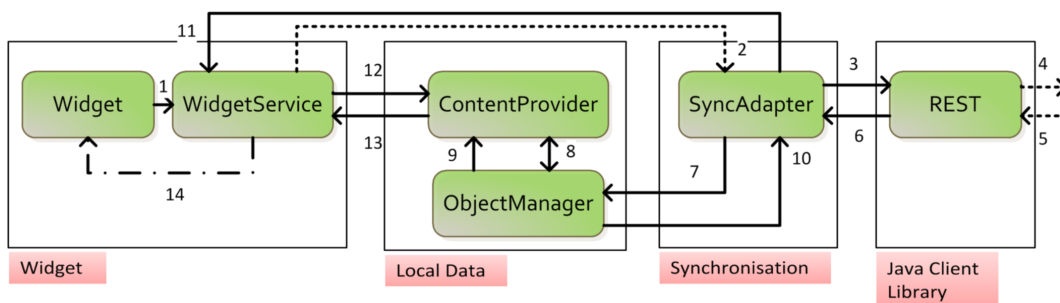

Figure 5.8: Synchronisation in Andood – remote data

The second, more interesting example of synchronisation has been presented on figure

5.8. From the left-hand side: the *Widget* receives an intent and classifies it as a conditional or unconditional remote data update request. The latter one happens when the user intentionally clicks the "Refresh now" button, the first one occurs when the *Widget* has been woken up (by the alarm) and there is a chance that it might need a remote update. Because the *Widget* needs to be responsive it is not its job to decide whether it needs an update or not. Instead it delegates everything to the *WidgetService* instance by sending **(1)** an appropriate intent. The *WidgetService* receives an intent and in case it is a conditional update request it launches a *WidgetHeartbeatControl* to verify if it is a proper time to begin a remote synchronisation. The request is discarded if the *WidgetHeartbeatControl* decides not to trigger a remote update. The whole operation in this path is very fast, and does not require any power-consuming action like sending a data through the network so it can be performed quite frequently (please refer to chapter 6).

If either the *WidgetService* receives an unconditional update request intent, or the *WidgetHeartbeatControl* decides to perform a remote update, the following actions take place. The *WidgetService* updates the widget's view to indicate that it is in the process of updating data, and calls **(2)** the *ContentResolver* adding a new sync operation to the *SyncManager* instance. When the *SyncManager* decides to execute this request it wakes up the SyncService (RESTObjectsSyncService) and this service launches the SyncAdapter's (RESTObjectsSyncService) *onPerformSync* method. The *SyncAdapter* has the *widget update* mode activated (a parameter passed through the sync intent) so it creates a request and calls **(3)** the REST component (Session object within the JCL library) to execute **(4)** a long running network operation (a so-called "widget request"). After the response is prepared by the server (a new instance of *Location* object is created) it is sent back to the JCL **(5)** parsed and sent back **(6)** to the *SyncAdapter*.

After receiving a parsed results *SyncAdapter* updates the *Location* object with the information about *Businesses* and *Promotions* which came in the same response (this data is necessary to retain the widget state) and sends **(7)** all three groups of objects to the appropriate *Object Managers* from the Local Data module. Managers query **(8)** the *ContentProvider* for any existing copies of objects which came from the remote data source and prepare *BatchOperations* which afterwards are executed **(9)** in chunks. When the sync operation is completed the *SyncAdapter* is informed about the result **(10)** so now it can send **(11)** a new intent to the *WidgetService* instance to inform about success. From this moment everything works exactly as if it was a standard local update situation (in fact it is). The *WidgetService* calls **(12)** the *ContentProvider* for a new data and retrieves it **(13)**. Afterwards the widget's view is updated **(14)** using a *RemoteViews* mechanism.

There is a different way of notifying about changes in *ContentProviders* – *ContentObserver* listeners. Such listeners can be set up to catch data-change events for any particular Content URI (with a boolean flag indicating whether changes in sub URIs should also be considered). This solution, however, has one fundamental drawback. Namely, the listeners are called whenever there is a change in a specified object(s), regardless of whether anybody is interested in this particular change or not. The Andood application used to have a *ContentObserver* set on the *Location* table (so any change would trigger the widget's local data update) but this solution was replaced by the one presented on figure 5.8 because it turned out to be more effective also in respect to the future use cases of the *launch* application.

Chapter 6

Performance and power consumption issues

6.1. Performance guideline

If there was only one thing for which people appreciated Google's products, it would definitely be their speed and responsiveness. There are many estimations about what would have happened if the Google's search engine had presented its results only few hundreds of milliseconds longer than usual. These calculations are not always trivial (or trustworthy) because at this point it is simply difficult to imagine the Internet without Google Search. Notwithstanding, reliable estimates have been presented by experts showing that even 100 milliseconds of delay in page load time in the case of websites like Amazon or eBay can lead to millions-dollar losses of those services.

From the very beginning Android (as another Google branded product) has been designed to provide its users a seamless experience and it has quite strict rules regarding responsiveness of third party applications. However, performance requirements had to be compromised somehow by the flexibility of the system. In other words: developers can, and are advised to write fast and responsive applications but they can also do quite the opposite unless they are aware of general design principles and many optimisation tricks (other than those imposed by the system).

Figure 6.1 shows the two screens of situations that can be easily described as a "worst case scenarios" for Android apps. On the left screen the cause was clearly an uncouth exception (like null pointer exception, illegal argument exception etc.) which typically means just a small bug in a code that needs to be fixed. The right screen is much more alarming, it is the ANR (Application Not Responding) error. Android imposes a strict rule that each Activity must respond to any input event in less than 5 second (10 seconds for *BroadcastReceivers*). Otherwise the ANR error is thrown and users are presented a dialog box where they have an option to kill the process (and application) or let it work for another 5-second period of time. In any case this is a clear signal to the users that something is wrong with the application. The vast majority of such applications will not be given a positive reviews on the Android Market so – to say gently – their potential success is very uncertain. ANRs are typically caused by executing long running operations (network, I/O) on the UI thread so they can only be fixed by slightly modifying the architecture of certain components (adding a separate thread for instance).

Brad Fitzpatric from Google gave a speech on the Google I/O conference in May 2010 about *Writing zippy Android apps*. Among many interesting observations he presented a list

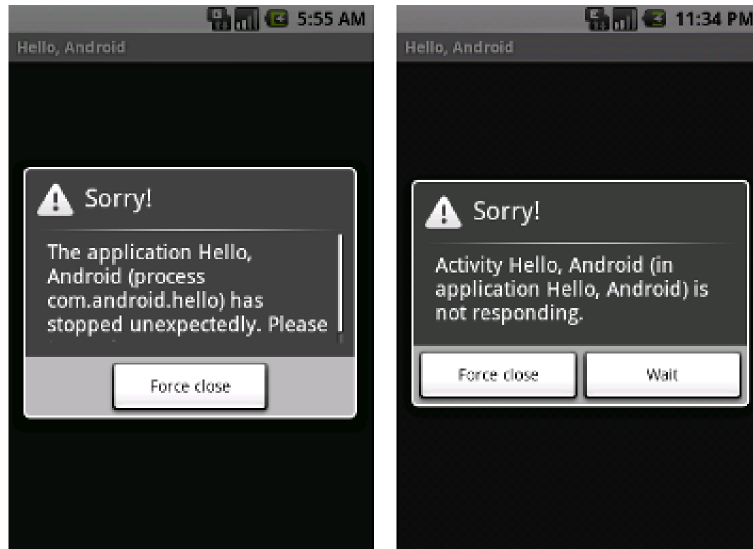

Figure 6.1: Android error dialogs, *source: Android Developers* [AND]

of typical operations with their approximated durations measured on a Nexus One phone. It starts with 0.04 ms needed for writing a byte on a standard process pipe and then moves to the memory operations: 5-25ms for reading an uncached byte from flash memory and 5-200+ms for uncached flash write. The human perception is approximately 100-200ms and the TCP setup and fetching small amount of data over 3G would take about 1-6+ seconds.

There are some practical conclusions from this presentation. First, executing a network operation within the UI thread will most likely cause the ANR and users will find the ANR sooner or later if there is even a slim chance that they can produce it. Some unoptimised SQL transactions can also be too long to fit in a 5 seconds limit. Second, even flash memory operations can cause applications to be 'slow' and 'janky' especially when the flash drive has very little amount of free space available (one of the characteristics of the YAFFS 2 file system which comes with Android). Summing up, to avoid ANRs developers should use the main (UI) thread only to create/update views and delegate long running operations to separate threads as even accessing local memory can be noticeable.

The remaining part of this section will cover only two selected issues of performance and responsiveness improvements implemented in the Andood application. The first one is related to managing large, scrollable lists and the other focuses on increasing the application's responsiveness by delegating expensive computations to separate threads.

6.1.1. Optimising large lists

When speaking about performance of mobile applications one of the biggest challenges is scrolling through the large set of data. It turns out that this action is one of the most common things users do regularly across many different applications. Naturally, games and manipulating huge graphic files are also very demanding, nevertheless scrolling is the one action that developers do not expect to cause any problems in the first place.

The background here is that mobile devices are usually equipped with relatively slow CPUs and a limited amount of internal memory. Only high-end smartphones can offer their users benefits of dual core processors or dedicated graphic chipsets (like Nvidia Tegra). Moreover, people are used to standard PCs where everything works almost instantly and they

expect smartphones to behave the same way.

Google developers have been working hard to find solutions that would enable users to scroll through a large data set seamlessly. However, using these techniques is not obligatory, so it is all a matter of whether developers know about them and use them or not. It seems that this issue was so important in providing a great user experience that Google has decided to hold an hour-long talk during the 2010 Google I/O only about this particular problem. The main concepts of this presentation (lead by Romain Guy and Adam Powell) have been outlined below to collate them with the solutions implemented in the Andood application.

```
1 public View getView(int position, View convertView, ViewGroup parent) {
2   View item = mInflater.inflate(R.layout.listelement, null);
3   ((TextView) item.findViewById(R.id.listelement_text))
4       .setText(SOMEDATA[position]);
5   ((ImageView) item.findViewById(R.id.listelement_icon))
6       .setImageBitmap(mAppIcon);
7   return item;
8 }
```

Listing 6.1: Dummy solution

The first – dummy – solution has been presented on listing 6.1. The method *getView()* is executed when the *Adapter* is asked for a populated instance of a *View* class in order to show it on a list. It happens for example when the user scrolls down the list and new list elements appear on a screen. Line **2** inflates the main list element view from the XML layout file and lines **3-6** populate components of this view (sub views in hierarchy) with the appropriate data. Prepared instance of the list element view is returned in line **7**.

```
1 public View getView(int position, View convertView, ViewGroup parent) {
2   if (convertView == null) {
3     convertView = mInflater.inflate(R.layout.listelement, parent, false);
4   }
5   ((TextView) convertView.findViewById(R.id.listelement_text))
6       .setText(SOMEDATA[position]);
7   ((ImageView) convertView.findViewById(R.id.listelement_icon))
8       .setImageBitmap(mAppIcon);
9   return convertView;
10 }
```

Listing 6.2: The better solution

The second listing (6.2) shows the first and most effective improvement compared to the previous code snippet. Namely, instead of creating a new instance of a *View* class for each element that user can see when scrolling down the list, Android implements a recycle bin and automatically provides developers with initialised instances of a View class (of the appropriate type). So now a new instance is being created only when the *convertView* parameter is null. Provided that the screen resolution and list element size allow to present about 10 list elements at once only 12-15 elements are instantiated (if they are all of the same type). This is a huge leverage because for instance when presenting a 1000-object list the system has to create only 1% of the objects from the previous solution and – more importantly – no new objects are instantiated during the process of scrolling the list.

```
1 public View getView(int position, View convertView, ViewGroup parent) {
2   ViewHolder holder;
3   if (convertView == null) {
4     convertView = mInflater.inflate(R.layout.listelement, parent, false);
5     holder = new ViewHolder();
6     holder.text = (TextView) convertView.findViewById(R.id.listelement_text);
7     holder.icon = (ImageView) convertView.findViewById(R.id.listelement_icon);
```

```
 8       convertView.setTag(holder);
 9     } else {
10       holder = (ViewHolder) convertView.getTag();
11     }
12     holder.text.setText(SOMEDATA[position]);
13     holder.icon.setImageBitmap(mAppIcon);
14
15     return convertView;
16 }
```

Listing 6.3: The even better solution

The third solution presented on listing 6.3 tries to additionally minimise the amount of *expensive* operations. One of such operations is the *findViewById()* method which scans the view hierarchy in search for a child view with the given id. Even though this operation cannot be completely avoided, there is no sense in repeating this process each time a *convertView* is populated. A static *ViewHolder* instance can be attached to the view (by tagging) to hold references to all subviews that have to be populated. Whenever there is a not null recycled *convertView* it will now certainly contain the *ViewHolder* instance.

In this release of the Andood application there is only one Activity which constains a list of elements – *CategoryChooseActivity*. Even though the number of presented categories is limited, the whole component has been optimised according to the third solution advised by Google (the one which uses *ViewHolder* class). Moreover, to improve rendering time of elements while scrolling through the list, the *cacheColorHint* variable has been modified (list has white background, default is black).

The following *launch* application release will certainly contain many more lists within different *Activities* (*BusinessListActivity*, *UserListActivity*, *ReviewListActivity* just to name three). Because these elements are most likely to be relatively complex (many different 'attached' actions) they will be implemented as custom components. This solution allows developers to have full control over the process of inflating the view as well as binding and unbinding data which populates it. To give a real-life example: each of the elements presented on the three Activities mentioned above will have an icon that should be displayed as part of the view. Bitmap decoding is a relatively long operation and in case users scroll down the list it could be quite 'blocking' (causing the application to be 'janky'). One way to deal with that problem would be to create a cache mechanism based on soft references (standard Java *SoftReference*) for all bitmaps. New bitmaps are downloaded (from local or external source) only when the scroll action is stopped, as there is no sense in fetching bitmaps for views that are no longer visible (however, commenced download actions should not be aborted). Bitmaps are then taken directly from the cache when users scroll up and down the list and new data is binded (in this case it should contain a key of the bitmap in cache map).

However, this solution has some significant drawbacks. Namely, Java *SoftReferences* are very prone to any OS signals of low memory. They are the first objects to be released whenever there is a low memory situation (it happens quite often especially on mobile devices). To avoid these 'unexpected' data losses Andood's implementation will be based on the *LRU* cache mechanism introduced in the Compatibility Package released by Google in March 2011. The compatibility package is simply a set of standard classes used in the latest, tablet version of Android (3.0) that developers can use in previous versions (down to 1.6). An instance of the *LRU* cache will be given a small part of the memory which is available to the application and will implement a standard Least Recently Used mechanism to manage bitmap objects. Naturally it will implement the *onLowMemory* mechanism but this method will be called directly by the application, much less frequently and only in the most critical situations.

6.1.2. Improving responsiveness

In order to keep the application's responsiveness under control one has to be constantly aware of the *Activities* lifecycle. After all *Activities*, as a single Android components which have a graphical representation, are the only entry points for any user's action. Therefore all standard methods that are bound to the lifecycle have to be implemented very carefully not to overload the system during transitions between *Activities*.

In general users are prepared to wait a little bit for a fresh data to show up, especially if they know that it needs to be downloaded from the Internet (standard desktop browsers work similarly). However, blocking applications until such operation is completed is unacceptable. Typically when a new *Activity* is about to activate, the overriden *onCreate* method would initialize its components and delegate a long running operation (loading data) to a separate thread (by using the *AsyncTask* mechanism for instance). While the task is running a *progress bar dialog* would be presented to the user to indicate a background operation. *Dialog* is later dismissed by the task callback as soon as it finishes its work. It is definitely a bad practise to block the screen on a progress bar dialog. Users must have a chance to dismiss it and move back if they want to, but of course it does not necessarily mean aborting the whole operation. It is also very important to save the state of *Activities* whenever they are about to be stopped (*onStop* method). There is no point in fetching the same data from local or remote source twice in a very short period of time. Also all of the user's input has to be stored in a Bundle when the activity is about to be stopped because it can be restored soon after.

Thanks to the design patterns developers can (and are advised to) do even more. To avoid a full screen progress dialog box which indeed can be quite annoying on long, remote operations, developers can make use of the *ActionBar* pattern (please refer to section 2.3). A small, indeterminate progress bar (round animation) can be added to the top panel and activate itself whenever the Activity has a background working thread running. This could leave the whole Activity screen unblocked thus more 'user friendly'. Naturally, some data would not be presented as it has to be downloaded from the local or remote source, but keeping the screen lit (as opposed to keeping it dimmed when the dialog is shown) would definitely increase the user's perception of the application's responsiveness.

The Andood application has only one activity where this solution could be applicable – *CategoryChooseActivity*. Indeed the information about a running background task is visible only on the *ActionBar* panel and additionally there is a standard sync icon on the Android's system bar at the top of the screen during the time a *SyncManager* performs sync actions. The second, full screen activity (*LoggingActivity*) is a single exception. In case of fragile actions where security is crucial, it is better to leave a focused, overlaying *progress dialog* on the screen to ensure that the user fully understands what is happening in the background. The *LoggingActivity* is one of such examples. Users need to know exactly when their accounts are being authenticated with the remote server.

The current release of the Andood application is focused on the user's interactions with a widget. The Widget's layout – as can be seen on a desktop – is represented by the *Widget* class which itself is an instance of a *BroadcastReceiver* thus it does not have a lifecycle similar to the Activity. The *Widget* class only responds to certain intents sent from the Home Application where the graphical representation of the widget can be placed. However, because those delegated intents run in the main (UI) thread they need to be handled as fast as possible to ensure the widget's responsiveness. It all comes down to updating a widget's view immediately and delegating all of the requested long running operations to separate threads. The Andood application uses a special service (*WidgetService*) to handle all of such actions regardless whether they request for a local or remote content update. The widget

itself contains an indeterminate progress bar which turns visible as soon as any intent is sent to execute by the *WidgetService*. Moreover, thanks to the *RemoteViews* mechanism to some extent widgets are updated in 'transactions'. Whenever the *WidgetService* fetches new data it prepares and populates a new widget's view (along with all of the actions binded to different view's elements) and 'publishes' it on the Home Application with just one command.

6.2. Reducing power consumption

One thing is to provide users with perfect experiences when they are using different apps, it is another to develop them being aware of battery life limitations. This is especially important now, when due to the technology constraints producers are unable to deliver battery units that would be at the same time small, light and capacious. When users have to charge their mobile devices every single day they start to care about applications being 'gluttonous' or not. Android is mainly installed on a high-end units equipped with many power consuming hardware components like WiFi, Bluetooth or simply 2G/3G adapters. Additionally, on average Android users spend more time browsing the Web, listening to music or watching videos. It is not surprising that preserving battery life is therefore important for both third party application developers (users can watch statistics of the most power consuming applications installed) and developers of Android itself.

There are many techniques implemented in Android that allow developers to reduce the negative impact on a battery life of their applications. This however has to be balanced with keeping applications functional. Not writing apps at all naturally solves the problem, but this is not quite a satisfying solution tough. In general all 'golden rules' of preserving battery life can be cut down into two main advices:

- release resources as soon as possible,
- do not perform any unnecessary operations.

It is worth mentioning that Google developers have contributed significantly in preparing relevant information about how to manage the application's power consumption. Just to name two talks: *A beginner's guide to Android* by Reto Meier and *Writing zippy Andriod apps* by Brad Fitzpatrick both from the 2010 Google I/O. The following section focuses on techniques already implemented in the Andood application.

6.2.1. Reducing power consumption in Andood

Andood refreshes its content using the *SyncAdapter* mechanism introduced in Android 2.x platforms. This alone has a positive effect on preserving battery life as it is not the application that triggers network operation, but a centralised, unified system. The *SyncManager* is managed entirely by the operating system hence it is possible for Android to schedule the incoming requests and avoid opening multiple HTTP clients at once which would eventually cause a lack of responsiveness due to the network bandwidth (especially when there is no WiFi connection). The *SyncManager* implements a queue of requests from all binded applications and controls the exact time when they are executed. It also manages a system of synchronisation retrials in case of any failures during that process.

However, notifying the *SyncManager* about the update request is not good enough. It is also important to control the frequency of such requests. On the one hand updates have to be relatively frequent to keep the data up-to-date, on the other hand waking device every quarter and requesting a network operation is not good either.

To solve this problem Andood uses a mechanism of system *Alarms* to wake the device up every 15 minutes. The *WidgetService* is called first and the *WidgetHeartbeatControl* soon after. The latter one makes the decision whether to update a widget content with a remote source based on many variables:

- *current location* – distance between the current location and the currently displayed *Location's* coordinates. This is taken under consideration only when the current and trustworthy device's location can be retrieved from the *LocationManager*,
- *activity* – one of the *Locations* attributes,
- *elapsed time* – the time since the last remote content update,
- *mode* – one of the user's preferences.

If the *WidgetHeartbeatControl* decides not to refresh the widget's content the *WidgetService* stops itself and no network operation is scheduled in the *SyncManager*. It means that the Andood Widget's behaviour adapts to the user's actions. The more mobile he or she is, the more frequent the updates are. On the other hand when users do not change their positions significantly and stay in relatively low urbanised areas (with less *Businesses* in Dood's database), the widget will not update itself so frequently.

Not only does the Andood Widget control the sync request scheduled in *SyncManager*, but it also lets the system manage the exact wake up time. The widget's content updates can be scheduled in two possible paths. The first one is a direct user's action which is available in the preferences screen – 'Force widget refresh'. Selecting this option would automatically (with no delay) add a new request to the *SyncManager's* queue (it does NOT mean an instant execution though). The second path is indirect, the *Alarm* wakes up the *WidgetService* and the *WidgetHeartbeatControl* class decides to refresh the widget's content. Since this operation is independent from the user's actual notice (it runs in the background) it does not have to be executed precisely every 15 minutes. By setting an *intexact* type of the *Alarm* Andood lets the system decide when exactly to wake the device up. However, the maximum interval between two wakeups cannot be longer than two times the interval passed as the *Alarm's* parameter (15 minutes in case of Andood). The system can choose the best available time respecting the other tasks' needs and schedule the wake up execution accordingly. So even though the Andood Widget 'wakes up' relatively frequently, in most cases it does not schedule a network operation and, most importantly, it does not discriminate other applications nor is it gluttony in accessing resources. The widget lets the system find the most appropriate moment to wake up so that other applications can benefit from this time as well.

Moreover, because Andood uses a standard, centralised *Sync* mechanism, users can disable the indirect refresh actions. They can switch the background synchronisation on and off using the system button available in the main control panel (next to the wifi, bluetooth, gps and screen brightness controls). Not all Android users are aware of what that switcher is actually responsible for, but placing it next to the controls of sensors well known for their significant battery consumption suggests that this could potentially use some of the device's resources. In fact this switcher simply indicates that the user agrees or disagrees to establish a network connection in the background. The *SyncManager* uses this flag to handle (limit) periodic updates, nevertheless it launches synchronisation whenever the user intentionally requests that (direct path of refreshing content in Andood).

Chapter 7

Application release and conclusions

7.1. Application release

The Andood application will be released on the Android Market under the name **dood**. The same argument which says that choosing a separate name containing prefix 'an' was useful in managing and documenting the development process of the Andood application, works exactly opposite in terms of market campaigns. LemoNET does not want to create any confusion among users, so all its products related to the Dood platform should bare the same name (with iDood being a reasonable exception in the iPhone world).

According to the business model presented in section 3.2 from the very beginning the Andood application will be free for all users (*free forever* strategy). The first version to be released will have a 1.0 version number. Each minor change will increase the second number (after the first dot), each major change will increase the first number (before the dot). The *launch* application as a first (already planned) major change will have a 2.0 version number.

The Andood application will be released with an Apache 2.0 license [LIC].

7.2. Future work

The following milestones and features have been planned for the Andood *launch* application:

Andood 2.0 – components and actions, milestones	
1	Integrated searching for businesses, events, users. Search filters.
2	Browsing lists of businesses, reviews, users, events.
3	Adding new reviews to businesses.
4	Publishing a photo review of a business.
5	Full screen business, review, user, event details activities.
6	Cache mechanism for bitmaps based on *LRU* cache.

Andood 2.0 – features	
1	*Where am I* – automatically finds a business based on a the user's current location
2	*Let's meet here* – sets a meeting in one of the businesses (like a restaurant) and invites selected friends. Message is sent via SMS, email or posted on a Facebook wall if the account has been synchronised.
3	*Target campaign* – personalised sponsored business recommendations based on search history, current user's location and history of reviews
4	*Take me there* – navigates user to the selected business (based on Google Map application)

7.3. Conclusions

The final result of six months of designing and implementing is the Andood application. Andood is the first Android application offered by *dood.pl* which provides a system of low-cost and instant advertisements of local businesses. Figure 7.1 presents two screenshots of a typical situation from the user's perspective. The left picture shows the Andood widget placed on a desktop (Android Home Application) displaying information about a promotion in one of the restaurants in Wrocław (the fourth biggest city in Poland). This promotion could be downloaded either by setting a default city preference to Wrocław or simply by revealing the device's location provided that the user is currently nearby. Only the most up-to-date and relevant promotions are retrieved from Dood servers. Details of promotions are presented in a special dialog box (right picture on figure 7.1) along with information about the address, average rank and photo (if available). Users can get to the details dialog by simply touching the promotion on the widget. The dialog offers quick links to the three most common actions:

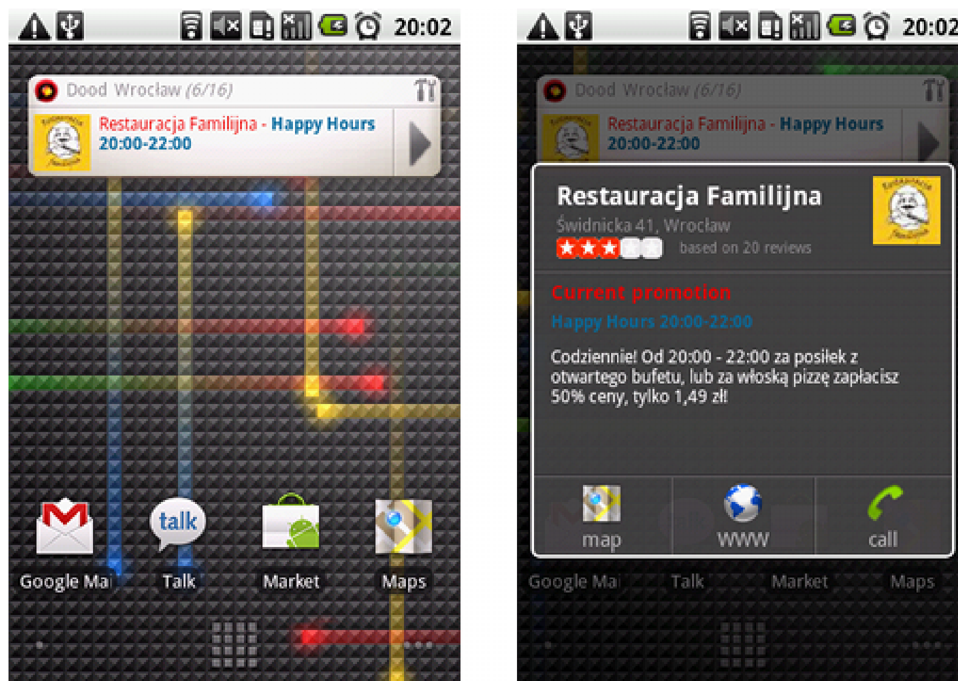

Figure 7.1: Andood widget – business promotion *(on LG 540 swift, normal density screen, 160dpi)*

- *map* – launches Google Maps with a marker showing the exact position of the business,
- *call* – dials the number of the selected business (if available),
- *www* – launches a default browser and opens the business's profile on a *dood.pl* web page.

The Andood application fulfilled nearly all of the planned software requirements. Yet, there is still a huge space for possible improvements even in this limited (compared to the main *launch* application) scope of functionality. Undoubtedly an important achievement was redesigning the API as now not only Andood application can benefit from its new features. Improved flexibility (objects of different types can be placed in one response) and better support for caching on a the client's side (details levels, 'declarations') will certainly reduce

the amount of data which is transferred both to and from the Dood servers. These mechanisms are so powerful and of general purpose that even the current form of the Andood application does not use all of their potential. Detailed sequential graphs of data synchronisation with remote servers which use different aspects of these mechanisms still need to be designed and implemented. It did not make much sense in a relatively small application (as in the current form of Andood) because the greatest improvement could be observed when transitioning between different screens (*Activities*). Bigger, *launch* application with a lot more Activities and possible actions should therefore be much more beneficial from this new, improved API.

Thanks to the Android system the Andood application is very portable. It can be launched on practically any device running Android OS. On the one hand not supporting devices with Android versions lower than 2.x could reduce the number of potential users especially in Poland. On the other hand, statistics show that the process of replacing the old versions with the new ones is very fast. Loosing less than 7% (present data globally) of the market due to this architectural decision would definitely be acceptable and very soon this share will be the reality in Poland. Not mentioning that there are plenty of new solutions and mechanisms introduced in platform 2.x which the Andood application takes a huge advantage of. Clearly, this decision has been justified accurately. Moreover, Andood application supports many screen densities and orientations so not only can it be launched on many devices but it also has different graphic layouts, customised for a variety of screen resolutions and dimensions.

As for the speed and responsiveness Andood has clearly lived up to the expectations. Naturally, 3G or WiFi latencies and bandwidth are things that applications have little or no influence on, nevertheless during the long time of testing different stable versions of Andood there have been no reports of the application causing ANR errors. Notwithstanding, there are still many ways in which the application could become faster. *LRU* caches for bitmaps or better utilisation of the API capabilities would certainly lead to an overall better performance. In terms of responsiveness there is not much more to be done. The Andood implementation follows the guideline described in best practices presented by Google. The widget itself delegates its job to the background *Service* almost instantly so it is fair to say that Andood has managed to avoid some of the common architectural mistakes which would eventually lead to poor application responsiveness.

Andood is also very intuitive because it implements some of the design patterns advised by Google developers. These components, or interface behaviours are constantly gaining popularity among Android developers so many of the most well known applications have already incorporated them. Thanks to this unification nobody who has ever been using Android applications like Facebook, Twitter, Evernote or Google Docs should have any problems when using Andood.

Android is not an easy system for writing complex applications if one wants to make a full use of its functionality. There are many pre-implemented solutions (especially in 2.x and now 3.x versions) like *Account&Sync* mechanism, *LRU* cache, *Alarms*, *Location* and *Sensor Managers* etc. that developers need to understand extensively in order to use them wisely (without a negative effect on the system). So far the Andood application has been developed by only one developer (author of this thesis), with a great help of Łukasz Kidziński in designing the API and implementing a server side of the system and Filip Łysyszyn who contributed to the layout and graphic design. However, the *launch* Andood application (Andood 2.0) is definitely too big of a project to be handled by such a small group of developers. LemoNET will start a marketing campaign to popularise the Andood application and will certainly do all that is necessary to raise funds needed for the implementation of the Andood (*launch*) 2.0 application. However, it is also looking for other companies that would like to take up this project and cooperate with LemoNET in providing Polish users with experiences that the

Andood application can offer. Finding such business partners or venture capital raisers would certainly speed up the implementation of the Andood 2.0 application. It is very important not to miss this unique moment of huge sales growth of Android devices on the Polish market and launch the full scale Andood application as soon as possible.

Bibliography

[ADB] Android Developers Blog,
http://android-developers.blogspot.com/

[ADC] James Steele, Nelson To, *Android Developer's Cookbook*, Pearson Education Inc., 2011

[ADG] Android Developers Group,
http://groups.google.com/group/android-developers

[AND] Android Developers,
http://developer.android.com

[ASO] Stack Overflow, Android tag,
http://stackoverflow.com/questions/tagged/android

[DJO] The Django Book,
http://www.djangobook.com/

[FCB] Facebook Places official web page,
http://www.facebook.com/places/

[GCW] Paweł Bedyński, Łukasz Kidziński, *Google Code Project – Dood API*,
http://code.google.com/p/dood-api

[IO1] Chris Nesladek, German Bauer, Richard Fulcher, Christian Robertson, Jim Palmer, *Google I/O 2010 – Android UI design patterns*,
http://www.google.com/events/io/2010/sessions/android-ui-design-patterns.html

[IO2] Virgil Dobjanschi, *Google I/O 2010 – Developing Android REST client applications*,
http://www.google.com/events/io/2010/sessions/developing-RESTful-android-apps.html

[IO3] Romain Guy, Chet Haase, *Google I/O 2011 – Honeycomb Highlights*,
http://www.google.com/events/io/2011/sessions/honeycomb-highlights.html

[IO4] Xavier Ducrohet, Tor Norbye, *Google I/O 2011 – Android Development Tools*,
http://www.google.com/events/io/2011/sessions/android-development-tools.html

[IO5] Reto Meier, *Google I/O 2011 – Android Protips: Advanced Topics for Expert Android App Develpers*,
http://www.google.com/events/io/2011/sessions/android-protips-advanced-topics-for-expert-android-app-developers.html

[IO6] Vic Gundotra, Hugo Barra, *Google I/O 2011 – Keynote Android: Momentum, Mobile and More at Google I/O*,
http://www.google.com/events/io/2011/sessions/android-momentum-mobile-and-more-at-google-i-o.html

[LIC] Apache 2.0 license,
http://www.apache.org/licenses/LICENSE-2.0.

[PAM] Shawn Van Every, *Pro Android Media*, Apress, 2009

[WCD] Content Delivery Network, wikipedia definition,
http://en.wikipedia.org/wiki/Content_delivery_network

[WRS] Representation State Transfer, Wikipedia definition,
http://en.wikipedia.org/wiki/Representational_State_Transfer

Appendix A

CD Contents

A CD has been attached to this paper. Contents of the CD by folders:

- **doc/** – contains sources of this document (LaTeX)
- **doc/gfx/** – contains graphics used in this document
- **api/** – contains sources of JCL (Java Client Library)
- **andood/** – contains sources of Andood Core Application (Android project)
- **andood/AndroidManifest.xml** – manifest file of Andood application