CA' FOSCARI UNIVERSITY OF VENICE
Department of Environmental Sciences, Computer Science and Statistics
MSc Program in Computer Science

Thesis

Student: Max Perry Perinato - 816507

Development of a Privacy Preserving Liferay Portal document synchronizer for Android

Thesis Supervisor: Prof. Michele Bugliesi

Academic Year 2011-2012

Development of a Privacy Preserving Liferay Portal document synchronizer for Android

by

Max Perry Perinato - 816507

Submitted to the Department of Environmental Sciences, Informatics and Statistics
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

CA' FOSCARI UNIVERSITY OF VENICE

A.Y. 2011/2012

© Ca' Foscari University of Venice 2013. All rights reserved.

Author . . . .

Department of Environmental Sciences, Informatics and Statistics
February 10, 2013

Certified by . . . .

Michele Bugliesi
Full Professor & Head of Department
Thesis Supervisor

Accepted by . . . .

Augusto Celentano
Full Professor
Thesis Examiner

Development of a Privacy Preserving Liferay Portal document synchronizer for Android

by

Max Perry Perinato - 816507

Submitted to the Department of Environmental Sciences, Informatics and Statistics
on February 10, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

In recent years we have seen an overwhelming spread of mobile devices, both for private and corporate use. To push this diffusion were the operating systems iOS by Apple and Android by Google, which currently dominate the market. This year in particular we saw Android reach a share of about 60% of all smartphones sold and about 50% of all tablets. Being these devices well suited for accessing all kinds of information and contents while on-the-go, many enterprises have started developing their own applications to give their employees the ability to access corporate data from anywhere. This however raises the problem of security of sensitive and confidential data, which is considered of great importance in the corporate domain. In this dissertation the problem of protecting data is addressed treating a real world corporate case study: the development of a native Android application (the client) for the synchronization of documents with Liferay Portal (the server), a leading open source web platform for enterprises. This application raises a number of security related issues, including, transferring data from server to client and vice versa, caching of information and off-line access, copying and sharing of data, managing and controlling user permission over each individual information (information rights management), revoking data access to untrusted users, and protecting information from theft or tampering (i.e. rooting) of the mobile device. Some of these problems can be eliminated, whereas many of them represent risks that can only be mitigated. The purpose of this work is to implement a synchronizer able to provide security of confidential data with the highest possible criterion, by harnessing software technologies compatible with the Android platform.

Thesis Supervisor: Michele Bugliesi
Title: Full Professor & Head of Department

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Michele Bugliesi, for giving me the opportunity to work on this interesting project and for providing me confidence, guidance and support throughout the course of this work.

I would like to express my special thanks to Alvise Spanò, Enrico Cavallin, Alessandro Frazza, Wilayat Khan, and, in particular Stefano Calzavara, for their invaluable input and fruitful discussion during the design and implementation of this work. I would also like to acknowledge the SMC company of Treviso for supporting this work.

Also thanks to all my friends for their constant support and for being a lot of fun to be with during these years

Last but not least, I would like to express my deepest thanks to my family, especially my Father, for their endless love and patience. Without their support, I would have never reached this important goal.

Venice, February 2013
Max Perry Perinato

Contents

1 Introduction **1**

1.1 Motivations . . . 2

1.1.1 Securing the Android platform . . . 4

1.2 Problem statement . . . 6

2 The Android platform **7**

2.1 Architecture . . . 8

2.2 The Dalvik Virtual Machine . . . 11

2.2.1 Dex File Format . . . 12

2.2.2 Register-based architecture . . . 13

2.2.3 Zygote . . . 14

2.3 Binder IPC . . . 15

2.3.1 Binder facilities . . . 17

2.3.2 Binder mechanism . . . 18

2.3.3 Transactions and parceling . . . 19

2.4 Security model . . . 20

2.4.1 Application Sandbox . . . 21

2.4.2 Permission mechanism . . . 22

2.4.3 Application signing . . . 23

2.5 Application components . . . 23

2.5.1 Activities . . . 25

2.5.2 Services . . . 26

2.5.3 Content Providers . . . 28

2.5.4 Broadcast Receivers . . . 29

2.6 Inter-component communication using Intents . . . 30

2.6.1 Intent filters . . . 31

2.7 Exporting components . . . 32

2.8 Android Manifest file . . . 33

3 Overview of Liferay Portal **35**

3.1 Portal architecture . . . 36

3.1.1 Users . . . 37

3.1.2 User Groups . . . 37

3.1.3 Roles and permissions . . . 37
3.1.4 Organizations . . . 38
3.1.5 Teams . . . 38
3.1.6 Sites . . . 38
3.2 Portlets . . . 39
3.3 Documents and Media Library portlet . . . 39

4 Client-Server architecture **41**
4.1 Communication layer security . . . 43
4.2 User authentication & Client authorization . . . 44
4.2.1 The OAuth Provider . . . 45

5 File synchronization **49**
5.1 SyncAdapter pattern . . . 51
5.2 Implementation . . . 52
5.2.1 Account creation . . . 54
5.2.2 SyncAdapter implementation . . . 56
5.2.3 UI notification . . . 59
5.2.4 Required permissions . . . 63
5.3 Synchronization algorithm . . . 64
5.4 Logical file representation and persistence . . . 69
5.4.1 The Content Provider . . . 73
5.5 JSON Web Services API . . . 84
5.5.1 Currently used API methods . . . 85
5.5.2 HTTP Client . . . 89
5.6 File download and upload . . . 92
5.6.1 FileDownloader service . . . 93
5.6.2 FileUploader service . . . 98

6 Offline usage and preservation of private data **103**
6.1 A trade-off approach . . . 105
6.2 Marking files as "confidential" . . . 108
6.3 Cache architecture . . . 110
6.4 Encrypted file cache . . . 118
6.4.1 SQLCipher . . . 119
6.4.2 Libsqlfs . . . 120
6.4.3 CacheGuard service . . . 121
6.5 CacheGuard encryption key management . . . 122
6.5.1 Private cache encryption key generation . . . 123
6.5.2 Vulnerabillities . . . 126
6.5.3 Lifecycle and operations . . . 130
6.5.4 Binder interfaces . . . 131
6.6 Viewing and sharing files . . . 131

7 Conclusion and future work **137**

List of Figures

1-1 Anatomy of a Mobile Attack . . . 3
1-2 OWASP Top Ten Mobile Attacks . . . 4

2-1 Android's software stack architecture . . . 8
2-2 .dex file format . . . 13
2-3 The Zygote . . . 15
2-4 Binder IPC facilities . . . 18
2-5 Binder mechanism . . . 19
2-6 Binder interaction . . . 20
2-7 Data parceling . . . 21
2-8 Android application components . . . 24
2-9 Android inter-component communication . . . 31

3-1 Liferay permission model . . . 36

4-1 Client-Server architecture . . . 41
4-2 Liferay Safe . . . 42
4-3 OAuth 2.0 Protocol Flow . . . 46

5-1 Android SyncAdapter pattern . . . 52
5-2 Implementation of the SyncAdapter pattern . . . 53
5-3 Data management and persistence . . . 70
5-4 Data model . . . 72
5-5 Data management classes . . . 74
5-6 Data persistence classes . . . 75
5-7 HTTP client . . . 90

6-1 Marking a file as confidential . . . 109
6-2 Cache architecture . . . 111
6-3 Cache encryption key generation scheme . . . 124
6-4 Android attack chart . . . 127
6-5 Encrypted cache accessibility and protection . . . 129

Listings

5.1 DLSyncService . . . 56
5.2 Manifest service tag . . . 56
5.3 Manifest sync-adapter tag . . . 57
5.4 SyncAdapter manual start . . . 58
5.5 SyncAdapter periodic start . . . 58
5.6 SyncAdapter cancel sync . . . 59
5.7 ContentResolver notifyChange() . . . 59
5.8 Registering the ContentObserver . . . 59
5.9 Custom ContentObserver . . . 60
5.10 Registering an OnSyncBroadcastReceiver . . . 61
5.11 Unregistering an OnSyncBroadcastReceiver . . . 61
5.12 OnSyncBroadcastReceiver . . . 61
5.13 SyncAdapter UI notification local broadcast . . . 63
5.14 SyncAdapter pattern permissions . . . 63
5.15 Synchronization algorithm: main . . . 66
5.16 Synchronization algorithm: GetFolders . . . 66
5.17 Synchronization algorithm: GetFileEntries . . . 67
5.18 Synchronization algorithm: HandleDLSync . . . 67
5.19 Synchronization algorithm: HandleFileContent . . . 68
5.20 Synchronization algorithm: HandleFolder . . . 68
5.21 Synchronization algorithm: PerformRename . . . 68
5.22 Synchronization algorithm: PerformDelete . . . 69
5.23 JSON to Java object conversion . . . 71
5.24 @SerializedName annotation . . . 71
5.25 ProviderMeta . . . 77
5.26 UriMatcher . . . 79
5.27 DataBaseHelper . . . 80
5.28 ContentProvider getType() . . . 81
5.29 ContentProvider update(. . . 82
5.30 DLFileManager updateDLFile() . . . 83
5.31 Manifest provider tag . . . 83
5.32 /get-user-sites JSON response . . . 86
5.33 /get-dl-sync-update JSON response . . . 87

5.34 /get-folders JSON response 87
5.35 /get-file-entries JSON response 88
5.36 PoolingConnectionManager 90
5.37 getDLSyncUpdate() 91
5.38 FileDownloader service intent 93
5.39 FileDownloader service onCreate() 93
5.40 FileDownloader service onHandleIntent() 94
5.41 Private file download 95
5.42 Public file download 96
5.43 HTTP client getFileAsStream() 96
5.44 FileUploader service intent 98
5.45 File upload onActivityResult() 98
5.46 FileUploader onHandleIntent() 99
5.47 HTTP client MultipartEntity subclass 100
5.48 HTTP client addFileEntry 101
6.1 get-tags JSON response 108
6.2 Accessing the external storage 112
6.3 FileObserver service 113
6.4 FileObserver subclass 116
6.5 BootupBroadcastReceiver 117
6.6 Registering the bootup broadcast receiver 117
6.7 FileObserver service start intent 118
6.8 Virtual Encrypted Disk operations 119
6.9 Opening a public file in other applications 132
6.10 Binding to CacheGuard from the PDF viewer application 133

Chapter 1

Introduction

In recent years we assisted to a rapid growth of mobile platforms. Smartphones and tablets are to the early 21st century what the PC was to the late 20th century - a multi-functional tool valued for its productivity and entertainment factor. As a consequence to their rise in popularity and capabilities, mobile devices are becoming a desirable work asset for enterprises who foresee in their adoption an added value to foster mobility and collaboration. However, when used in the enterprise, these devices are likely to process sensitive information. As an example, a common use case for an enterprise is to allow their employees to access and share documents anytime and anywhere, from their own mobile devices. This therefore raises the question of whether this data is secure with current mobile platforms and what actions an enterprise should take to secure sensitive information on mobile devices - whether to protect the enterprise's intellectual property or the privacy of employees and clients.

This thesis describes the development of Lifery Safe, a native application for Android that implements a client for document synchronization with servers running Liferay Portal, the leading open source portal for the enterprise. The purpose of this work is to design an acceptable solution - in terms of security and user experience - for providing preservation of private documents and caching of data for offline use in a mobile environment.

Chapter two gives some background information about the Android platform, with particular interest to its architecture stack, security model and application component framework. The chapter discusses also features specific to Android's architecture stack, such as the Dalvik Virtual Machine and the Binder interprocess communication.

Chapter three provides a brief overview of the Liferay open source enterprise portal, and introduces the document management capabilities integrated in the *"Document and Media Library"* portlet.

Chapter four describes the client-server architecture of the application, with particular attention to the security of the communication channel, including the details about user authentication and secure interaction with protected data.

Chapter five explains the design and implementation of the file synchronizer, which is based on Android's SyncAdapter pattern - an efficient synchronization mechanism perfectly integrated in the Android system.

Chapter six focuses on the dual problem of preserving private documents and providing offline access to these data. A proposed approach to this problem is described, which consists in caching private files in a virtual encrypted disk and entrusting the management of the encryption key to a secure service component. The design of this approach gave rise to several security risks, many of which are related to the Android platform. The chapter discusses how some of these risks can be circumvented, and what assumptions and constraints are necessary for those that can only be mitigated.

1.1 Motivations

The rapid adoption of high end smartphones and tablets, along with the diffusion of mobile applications and cloud based services, is driving a growing trend in the Bring Your Own Device (BYOD) concept among enterprises; as employees become more accustomed to using their own mobile computing devices in their jobs, and accessing corporate private data.

A recent report from Juniper Research [1] has found that the number of employee owned smartphones and tablets used in the enterprise will more than double by 2014, reaching 350 million, compared to almost 150 million this year (about 23% of the total installed base). The report also found that the majority of employees smart devices did not have any form of security software loaded nor were company materials protected. While BYOD is becoming an inevitable trend for the enterprise, it also poses potential security risks. Enterprises need to define their own end-user policies and address the key security issues emerging.

Employees demand access to documents and different types of content anytime, anywhere, from their mobile devices. For companies, however, this introduces significant compliance risks if the proper controls to protect this information are not in place. In particular, confidentiality and liability issues may arise when documents contain private information concerning the company, or personal information about its employees or clients. Organizations of all sizes need a simple, scalable way to distribute, manage and secure documents on smartphones and tablets.

The mobile device security model is erroneously based on the security model of their technological predecessor: the laptop computer [2]. Unlike the laptop, mobile devices are rarely shut down or hibernated. They are always turned on and are almost always connected, making the laptop security model insufficient. This connectivity also raises a new set of security risks with new threats and attack vectors. As illustrated by viaForensics security company (Figure 1-1) a mobile attack can involve the device layer, the network layer, the data center, or a combination of these [3]. In the hands of an attacker, a powered-on device is susceptible to information disclosure via its flash memory (internal/external) and eventually the RAM. Additionally, privilege escalation bugs and public exploits for rooting a device are commonplace.

The primary security issue with BYOD centers around corporate data leakage and misuse. Mobile Device Management (MDM) solutions (e.g., Samsung SAFE) - often referred as *"Secure Containers"* - have been developed to facilitate the adoption of mobile devices into the enterprise and try to mitigate the risks inherent to the platform. However, recent

Figure 1-1: Anatomy of a Mobile Attack

security audits have discovered that such security solutions might still be vulnerable to a number of attacks that plague most third-party applications [2, 4], including, privilege escalation, bad implementation of SSL, usage of vulnerable OS libraries and leakage of data into unprotected parts of the system. Above all, the main weakness of these secure containers stands in the lack of proper implementation of data encryption (e.g., unsafe key generation and storage, weak algorithm, etc.), which unfortunately results to be hindered by the same limitations of the platform's security model. However, in most cases security risks arise from bad design and insecure coding practices. While developing a secure mobile application is not absolutely straightforward, fortunately there are groups of security experts such as the Open Web Application Security Project (OWASP) who are actively collaborating to *"classify mobile security risks and provide developmental controls to reduce their impact or likelihood of exploitation"* [5]. OWASP's annual list of top ten mobile risks (Figure 1-2) can be a starting point for many enterprises that need to assess and mitigate potential risks posed by mobile devices to sensitive information.

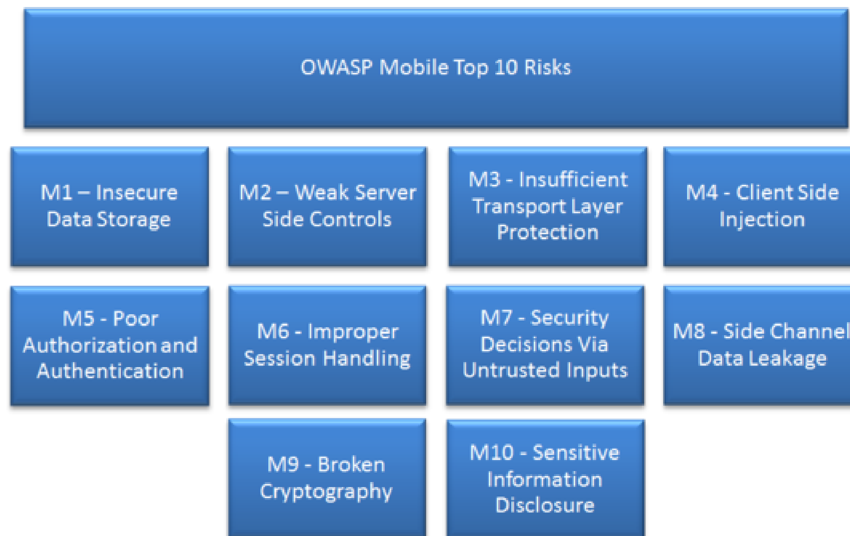


Figure 1-2: OWASP Top Ten Mobile Attacks

1.1.1 Securing the Android platform

Android is the open mobile platform developed by Open Handset Alliance (OHA). The most innovative feature of Android is its openness. This allows Android to be embraced by a large number of smartphone vendors, mobile operators, and third-party developers, driving shipment volumes to an impressive 75% share in the worldwide smartphone market, with over 135 million units shipped in Q3 2012 [6]. However, this popularity comes at a price. In one of its latest reports on mobile security, TrendMicro stated that:

> *"Malware targeting Googles Android platform increased nearly sixfold in the third quarter of 2012. What had been around 30,000 malicious and potentially*

> *dangerous or high-risk Android apps in June increased to almost 175,000 between July and September.”* [7]

While the openness provides various benefits to developers and users, apparently it also increases security concerns. Due to the lack of a control in the application development and distribution processes, it is quite possible for a user to download and install malicious applications. As an attempt to reduce this risk, Google has recently introduced the Bouncer service, which provides automated scanning of the Google Play store for potentially malicious software. However, security experts have promptly demonstrated that the Bouncer can be easily bypassed [8]. Although it's not an easy problem to solve, clearly the consequences to the lack of effective moderation in the application store can include exposure of private information.

The security model of the Android system is based on application-oriented mandatory access control and sandboxing. By default, components within an application are sandboxed by Android, and other applications may access such components only if they have the required permissions to do so. This allows developers and users to restrict the execution of an application to the privileges it has (statically) assigned at installation time. However, enforcing permissions is not sufficient to prevent security violations, since permissions may be misused, intentionally or unintentionally, to introduce insecure data flows [9]. In fact, Davi et al. [10], have shown that Android's sandbox model is conceptually flawed and actually allows privilege escalation attacks. In confirmation of these findings, a later survey provided a taxonomy of attacks to the platform demonstrated by real attacks that guarantee privileged access to the device [11].

The Android operating system also provides a rich inter-application message passing system. This encourages inter-application collaboration and reduces developer burden by facilitating component reuse. Unfortunately, message passing is also an application attack surface. The content of messages can be sniffed, modified, stolen, or replaced, which can lead to breaches of user data and violation of application security policies [12].

A number of security frameworks have been developed with the purpose to enhance Android's security architecture. Saint [13] addresses the current limitations of Android security through install-time permission granting policies and run- time inter-application communication policies. Kirin [14] extracts an application's security policy from its manifest file to detect data flows allowing privilege escalation attacks. SCanDroid [9] statically analyzes data flows through Android applications to help the developer make security-relevant decisions, based on such flows. ComDroid [12] detects application communication vulnerabilities. TaintDroid [15] extends the Android platform to track the flow of privacy sensitive data through third-party applications.

These tools can be useful for developers to certify the security of their Android applications and detect vulnerabilities due to bad design and insecure coding practices. Still, companies supporting BYOD must understand that mobile devices have a very large threat surface which makes full-protection of sensitive information stored in uncontrolled (and possibly compromised) mobile devices an inherently impossible task, even when providing access to enterprise resources through certified secure containers. An ideal approach already

pursued by security and military agencies [16] would be for enterprises to build their own security enhanced version of Android and give company-owned devices to their employees, yet giving up the productivity benefits of the trending BYOD model.

1.2 Problem statement

Considered the several areas of risk to which mobile platforms are exposed, and considered the issues inherent to Android's security model, the problems to be asked are the following:

Can documents organized in an enterprise portal be synchronized with an Android device and still have their privacy preserved? Is it also possible to cache these private data for offline access without losing their control? Can users be effectively revoked and data synchronized on their devices be consistently removed? Finally, what is the impact on the user experience and how can it be minimized?

This thesis will discuss the development of Liferay Safe, a software for the synchronization on Android devices of documents stored in a Liferay Portal. The purpose of this thesis is to provide and explain the implementation of a concrete solution to the problems stated above.

Chapter 2

The Android platform

Android is an operating system for mobile devices developed by Google and the Open Handset Alliance. Android was initially developed by Android, Inc., a small California startup which was later purchased by Google. The initial goal of Android, Inc. was to create an open and extensible operating system for mobile devices, which would give mobile carriers flexibility to create unique devices for their networks. When Google bought Android, Inc. in 2005, the industry assumed Google would launch its own mobile device. However, Google continued development on Android and helped form the Open Handset Alliance in 2007, which is a business alliance comprised of 84 hardware, software and telecom companies.

The first Android mobile handset, the T-Mobile G1, was released in the United States in October 2008, with over 1 million sales by the end of the year. In 2012 more than 1.3 million devices were activated per day [17], and by the end of 2013 cumulative shipments of Android smartphones are expected to exceed 1 billion units[18].

Google's Andy Rubin describes Android as follows:

> *The first truly open and comprehensive platform for mobile devices. It includes an operating system, user-interface and applications — all of the software to run a mobile phone but without the proprietary obstacles that have hindered mobile innovation.* — Where's My Gphone? [19]

Google wanted Android to be open and free; hence, most of the Android code was released under the open-source Apache License, which means that anyone who wants to use Android can do so by downloading the full Android source code from the Android Open Source Project (AOSP).

As the Android platform is open source, manufacturers can tailor Android to their hardware and sub brand the user experience to their own needs, in order to differentiate themselves from other manufacturers who also use the Android platform. And because it is free of charge, they can implement it on their devices without the need to pay expensive licensing fees. This simple development model makes Android very attractive and has thus piqued the interest of many vendors.

Finally, Android offers an open development environment built on an open-source Linux kernel. Compared to many existing proprietary platforms, Android is truly open to continued innovation and new experiences. Hardware access is available to all applications

through a series of API libraries, and application interaction, while carefully controlled, is fully supported. In Android, all applications have equal standing. Third-party and native Android applications are written with the same APIs and are executed on the same run time.

2.1 Architecture

Android's architecture is essentially a software stack made of a Linux Kernel and a collection of C/C++ libraries, which are exposed through an application framework that providers services for, and management of, both the runtime and applications.

Figure 2-1: Android's software stack architecture

The software stack, shown in Figure 2-1, is composed of the following layers and blocks listed from bottom to top:

- **Linux kernel** — The underlying Linux v2.6 kernel - version 3.x from Android 4.0 Ice Cream Sandwich onwards - is responsible of dealing with hardware, process and memory management, security, network and power management, providing an abstraction layer between the hardware and the remainder levels of the stack.

- **Libraries** — Android includes various C/C++ libraries such as libc and SSL, as well as SQLite, WebKit, OpenGL, a media framework for audio/video playback, and a surface manager to provide display management.

- **Android runtime** — The runtime is what really distinguishes Android from a mobile Linux implementation. Designed to be small and efficient for use on mobile devices, the runtime hosts and executes Android applications. Includes the Dalvik Virtual Machine (VM) and the Core Libraries, forming the basis for the Application Framework.

  - **Core Libraries** — Despite most Android application development is written using the Java language, Dalvik is not a Java VM. These libraries provide most of the Java core libraries, as well as Android-specific functionalities.
  - **Dalvik VM** — Dalvik is a register-based Virtual Machine especially designed and optimized to handle multiple running instances on the same device. It relies on the Linux kernel for threading and low-level memory management.

- **Application Framework** — Comprises the classes used to create Android applications. It also agnostically exposes system system services to the Application Layer, including window manager and location manager, notifications, resources, databases, telephony, and sensors.

- **Application Layer** — Both native and third-party applications are built on the application layer from the same API libraries. The Application Layer runs within the Android runtime, using classes and services made available from the Application Framework.

The Android software stack includes also a set of preinstalled applications (Native apps) that form part of the Android Open Source Project (AOSP), including, and not limited to, an e-mail client, an SMS management application, a WebKit-based web browser, a music player and media gallery, a camera and video recorder, a home screen, a calculator, an alarm clock, and nonetheless a calendar and a contact list. Android devices also comprise the following proprietary Google mobile applications: Google Play, Google Maps, Gmail client, Google Talk, and YouTube player.

Although Android devices ship with a core suite of standard applications, what really makes Android a radical change compared to existing mobile development platforms is the user's ability to *completely customize look, feel and function of their devices.* In fact, the data stored and used by many of these native applications - such as contact details - are also available to third-party applications. Similarly, third-party applications can respond to events such as incoming calls. This gives application developers an exciting opportunity to build mobile applications that change the way people use their phones [21].

Android's open-source nature means that carriers and OEMs can customize the user interface and the applications part of the AOSP, but still the underlying platform and SDK are required to remain consistent across OEM and carrier versions, which means that, besides variations of the user interface, applications developed for Android will function in the same way across all compatible devices [21].

The capabilities required for a device to support the software stack are described by the Compatibility Definition Document (CDD) and the Compatibility Test Suite (CTS). While the Software Development Kit (SDK) includes all the tools, plugins and documentation necessary to create applications [21].

Android inherits several features of the original Linux kernel [20] - the udev device manager and the permission model to make some examples - still it does not include the full set of standard Linux utilities; Android does not have a native X Window System by default nor does it support the full set of standard GNU libraries. In fact, Android's Linux kernel, which is based on the ARM architecture (Linux EABI), has been further changed by Google, outside the typical Linux kernel development cycle, to comprise new and unique features specific to Android, including:

- Binder Inter Process Communication (IPC);
- Bionic libc (a derivation of the BSD standard C library, much smaller than GNU libc and optimized for speed);
- ashmem (Android shared memory);
- pmem (Process memory allocator);
- OOM handler improved for low memory (informally called the Viking Killer);
- wakelocks;
- alarm timers (to support Android's Alarm Manager);
- and more...

Android applications are primarily written in Java and compiled into a custom byte-code (DEX). Each application executes in a separate Dalvik virtual machine interpreter instance running as a unique user identity. Therefore, from the the perspective of the underlying Linux system, applications are apparently isolated. All inter-application communication passes through middleware's binder IPC mechanism. Binder provides base functionality for application execution. Applications are comprised of components. Components primarily interact using the Intent messages. While Intent messages can explicitly address a component in an application by name, Intent versatility is more apparent for Intent messages addressed with implicit action strings, for which the middleware automatically resolves how to handle the event, potentially prompting the user. Recipient components assert their desire to receive Intent messages by defining Intent filters specifying one or more action strings [22].

In the next sections, are covered in detail Android's unique Dalvik VM and Binder IPC mechanism. These features are the basis for Android's security model and the component framework used to build applications, which are introduced later in this chapter.

2.2 The Dalvik Virtual Machine

Dalvik is an open-source process virtual machine custom designed for Android to meet specific requirements. It was originally written by Dan Bornstein, who named it after the Icelandic village of Dalvk, where some of his ancestors lived [23].

There are both technical and business reasons behind the choice of Google to abandon both JME and JVM in favor of an alternative deployment target (Dalvik VM) and a limited implementation of the standard Java libraries. Both of these are non-standard Java, and effectively represent forking of the Java platform.

Focusing on the technical reasons, there are important requirements that the Android runtime must meet, these are [24]:

- limited processor speed;
- limited RAM;
- no swap space;
- battery powered;
- diverse set of devices;
- sandboxed application runtime;
- large system library (10 MB).

Hence, given that the Android application runtime must support a diverse set of devices and that applications must be sandboxed for security, performance, and reliability, a virtual machine seems like an obvious choice. However a virtual machine-based runtime doesn't generally fit so well the the limited processor speed and limited RAM that characterizes most mobile devices. Google addressed these conflicting requirements with their own approach for implementing an application runtime environment, which can be summarized in the following key concepts [23, 24, 25]:

- Dalvik has been optimized so that a device can run multiple VMs efficiently;
- Dalvik executes files in the Dalvik Executable (.dex) format, enhanced for minimal memory footprint;
- The VM is register-based, and runs Java compiled classes that have been transformed into the .dex format by the "dx" tool;
- Dalvik VM relies on the Linux kernel for underlying functionality, including security, threading, and low-level process and memory management;
- Uses its own, non-standard class library built on a subset of the Apache Harmony Java implementation.

Regarding application sandboxing, the Android system implements the *principle of least privilege*. Each application is run within its own instance of the VM, which itself is a different Linux user running in its own OS process. Each process is assigned its own unique user ID (the ID is unknown to the process and used only by the system) that can only access a limited set of features within the system and can only access the components that it requires and its own data: the system sets permissions for all the files in an application so that only the user ID assigned to that application can access them [26].

This design creates a very secure environment in which an application cannot access parts of the system for which it is not given permission. However, for these same design decisions, Dalvik is not directly concerned with security, which is rather enforced by the OS at runtime.

The next paragraphs will cover the primary design choices of the Dalvik VM that were influenced by the requirements outlined previously, mainly the .dex file format, the register-based architecture and the Zygote.

2.2.1 Dex File Format

In Android, programs are commonly written in Java and compiled to bytecode. They are then converted with the "dx" tool from Java Virtual Machine-compatible .class files to Dalvik-compatible .dex (Dalvik Executable) files before installation on a device. In standard Java environments, Java source code is compiled into Java bytecode, which is stored within .class files, which are read by the JVM runtime. Whereas a .class file contains only one class, a .dex file contains multiple classes [23, 24, 25].

The compact Dalvik Executable format is designed to be suitable for systems that are constrained in terms of memory and processor speed. Its design is primarily driven by sharing of data. The following diagram contrasts the .class file format used by the JVM with the .dex file format used by the Dalvik VM.

The .dex file format uses *shared* constant pools as its primary mechanism for conserving memory. A constant pool stores all literal constant values used within the class, including string constants as well as name of fields, variables, methods, classes and interfaces. Rather than storing these values throughout the class, in Dalvik they are always referred to by their index in the shared constant pool [24, 25].

As we see from the diagram above, in the case of the .class file, each class has its own private, heterogeneous constant pool, where all types of constants (field, variable, class, etc.) are mixed together. In contrast, the .dex file contains many classes, all of which share the same *type-specific* constants pools. The type-specific pools actually allow further elimination of repetition by making the constants more granular when used across multiple classes.

Duplication of constants across .class files is eliminated in the .dex format; but the use of a shared constant pool has the consequence of significantly increasing the number of references within a .dex file compared to a .class file.

So how much memory is actually being saved with the use of shared constant pools? A study found out that the biggest part of the Java class files is the constant part (61 percent

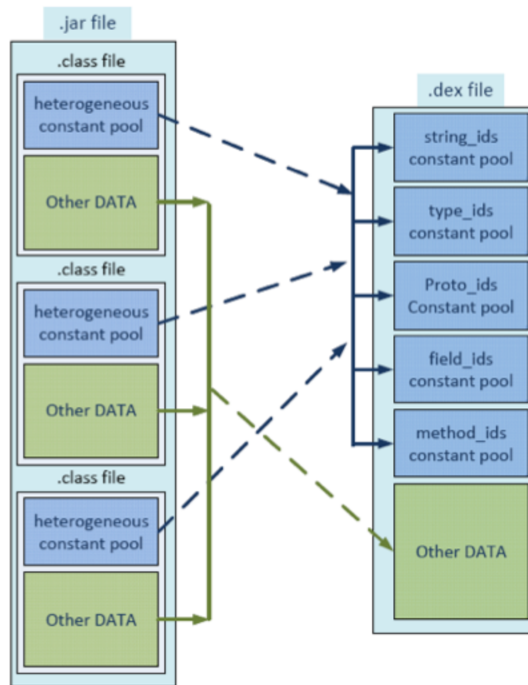


Figure 2-2: .dex file format

of the file) and not the method part that accounts for only 33 percent of the file size. The other parts of the class file share the remaining 5 percent. So it is clear that reducing the constant pool across multiple classes can result in significant memory reduction [24, 25].

In fact, an uncompressed .dex file is typically a few percent smaller in size than a compressed .jar (Java Archive) derived from the same .class files. This allowed to cut the size in half of some of the common system libraries and applications that ship with Android.

Before being executed by Dalvik, .dex files are further optimized specifically for the host environment. This normally happens before the first execution of code from the .dex file, combined with the bytecode verification, or when the application is launched for the first time. The process, called dexopt, loads the .dex and replaces certain instructions with their optimized counterparts. The resulting optimized .dex (.odex) file is then written into the "/data/dalvik-cache" directory [25].

2.2.2 Register-based architecture

Due to simplicity of implementation, many traditional VMs have a stack-based architecture; however, this is coming at a performance cost. In Android, where a variety of devices with constrained processing power and memory should be supported, performance is a priority. For this reason Dalvik is a register-based virtual machine [23, 24, 25].

Studies have shown that this choice seems appropriate, in particular when compared to stack-based architecture [25]:

- Register-based architectures require an average of 47% less executed VM instructions;
- Register code is 25% larger than the corresponding stack code;
- This increased cost of fetching more VM instructions due to larger code size involves only 1.07% extra real machine loads per VM instruction. Which is negligible;
- On average, register-based architectures take 32.3% less time to execute standard benchmarks.

Although register-based code is about 25% larger than stack-based code, the 50% reduction in the code size achieved through shared constant pools in the .dex file compensates the increased code size so there is still a net gain in memory usage as compared to the JVM and the .class file format [25].

Java bytecode is also converted into an alternative instruction set used by the Dalvik VM, providing a higher semantic density per instruction with 30% fewer instructions, 35% fewer code units and 35% more bytes in the instruction stream [25].

In fact, standard Java bytecode executes 8-bit stack instructions, where local variables must be copied to or from the operand stack by separate instructions. Dalvik instead uses its own 16-bit instruction set that works directly on local variables. This lowers Dalvik's instruction count (two instructions are consumed at a time) and raises its interpreter speed; enabling to consume the instruction stream more efficiently by avoiding unnecessary memory access and instruction dispatch [25].

A particular characteristic of Dalvik's bytecode is the ambiguity of primitive types. Dalvik int/float and long/double use the same opcodes. Additionally, Dalvik does not specify a null type, contrary to JVM, instead the zero value is used [25].

2.2.3 Zygote

Since every application runs in its own process within its own virtual machine, not only must the running of multiple VMs be efficient but creation of new VMs must be fast as well.

Android uses the concept of Zygote to address this problem. Zygote is the Dalvik VM master process responsible for starting and managing subsequent Dalvik based components and their associated privileges. It efficiently enables both sharing of code across VM instances and provides fast startup time of new VM instances [24, 25].

Zygote's design is based on two major assumptions; first, that there are a significant number of core library classes and corresponding heap structures that are used across many applications; second, that generally these classes are read-only and never modified by most applications [24, 25].

Zygote is started at boot time by the init process when the kernel is loaded. After the Zygote process is started, it operates as follows (schematized in Figure 2-3):

- First initializes a Dalvik VM which preloads and pre-initializes core library classes;

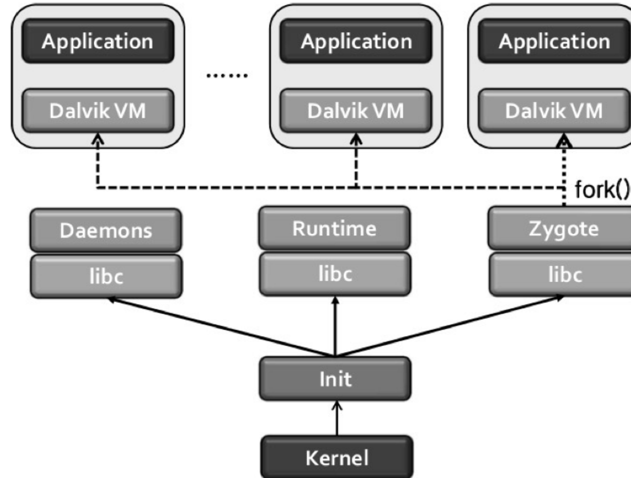


Figure 2-3: The Zygote

- Then is kept in idle state by the system and waits for socket requests;
- Whenever an application execution request occurs, Zygote forks itself and creates a new process with a pre-loaded Dalvik VM.

By spawning new VM processes from the Zygote,the startup time is minimized and slow cold starts are avoided.

Because all Applications are forked from Zygote, they inherit the same Address Space Layout as Zygote. This enables sharing of heavy core library classes that are generally only read by applications. When those classes are written Zygote follows a "copy-on-write" behaviour, so that the memory from the shared Zygote process is copied to the forked child process of the application's VM and writes are committed there. This behaviour prohibits applications from interfering with each other, while still allowing for maximum sharing of memory [24].

Instead, in traditional Java VM design, each instance of the VM will have an entire copy of the core library class files and any associated heap structures. Memory is not shared across instances.

2.3 Binder IPC

When one process exchanges data with another process, it is called *inter-process communication* (IPC). The reasons for an environment to allow this kind of process cooperation are multiple and not limited to information sharing: performance, modularity, and privilege separation are the most relevant of them.

IPC effectively copes with the concept of *process isolation*, according to which, for security reasons, one process must not manipulate the data of another process. In Linux, each process can access only the private virtual memory space that was assigned to; the virtual

memory is then mapped to physical memory by the OS. While ensuring memory security, process isolation hinders communication between processes; however sometimes this is required for modularization, hence IPC is needed. Linux offers a variety of mechanisms for IPC, including, signals, pipes, sockets, message queues, semaphores and shared memory.

There is also a tradeoff when implementing IPC; the need of applications and services to communicate and share data while running in separate processes, vs, the introduction of significant processing overhead and security holes.

Binder is essentially Android's solution to IPC: a lightweight inter-process communication mechanism, and a remote method invocation system. That is, one Android process can call a routine in another process using Binder to identify the method to invoke and pass the arguments for the other process [28].

Originally developed under the name OpenBinder by Be as a feature of BeOS - and later enhanced by Palm - Binder was a centralized construct for encapsulating software interfaces, providing bindings to functions and data from one execution environment to another. The lead architect behind this mechanism was Dianne Hackborn, who is now a key engineer at Android/Google. Binder in Android is a customized re-implementation of OpenBinder with a much simpler user-space; or better a branch of OpenBinder maintained by Android developers, since the original code is no longer being developed [27, 28].

The main technical characteristics of Android's Binder are the following [27, 28]:

- Three levels of IPC abstraction:
  - Intent (highest level);
  - AIDL (Android Interface Definition Language): allows the developer to define the programming interface that both the client and service agree upon in order to communicate with each other using IPC. It generates the tedious code needed to decompose the objects of one process into primitives and handle marshalling to another process;
  - A kernel driver to facilitate inter-process communication: a pool of threads is associated to each service application to process incoming IPC. Binder driver performs mapping of an object between two processes, using an object reference as an address in a process's memory space.
- High performance through shared memory (ashmem);
- Per-process thread pool for processing requests;
- Reference counting and mapping of object references across processes;
- Synchronous calls between processes, with reference counting;
- It is not based on SysV IPC, and is not POSIX-compliant.

2.3.1 Binder facilities

Android's Binder provides more than a simple mechanism for IPC, and is very different from a Linux socket. Binder has an internal status associated to a PID rather than to a file descriptor (fd), so that it can be shared among threads in the same process.

In the Android platform, the Binder is used for nearly everything that happens across processes in the core platform. As a matter of fact, most of the system services are implemented on top of it: package manager, telephony manager, app widgets, audio services, search manager, location manager, notification manager, accessibility manager, connectivity manager, wifi manager, input method manager, clipboard, status bar, window manager, sensor service, alarm manager, content service, activity manager, power manager, surface compositor.

The main facilities - schematized in Figure 2-4 - are summarized below from a description submitted by Dianne Hackborn to the Linux Kernel Mailing List (LKML.org) [29].

- **Permission validation**: The fundamentals of Android's security are a combination of uid-based permissions and binder capabilities. Some capabilities are direct (when the Binder object is given as an interface that the client can call on), some are indirect (when the Binder object is given as a token that the client can compare against other tokens previously received to validate who it is). For permissions, every incoming Binder transaction has associated with it the uid of the initiator, which is used in numerous places where only specific uids should be allowed to access specific features.

- **Binder as a token**: Each Binder is uniquely identifiable, meaning that it can act as a secure access token which can be shared across multiple processes. Binder objects have a single identity as they travels across processes, no matter how many times they do so or where they go. Thus it's always possible to check the origin's identity of a given token, without any way for clients to spoof it.

- **fd passing**: Binder objects can be used to pass a fd (file descriptor) of a shared memory through different processes. This facility enables the surface compositor to allocate an area in a shared heap it manages, when a new surface is requested. The window manager, makes this request on behalf of a client application, and then passes that Binder object over to the client process for it to retrieve the fd and draw directly into the associated surface memory.

- **Transparent call semantics**: Separate components, may be switched between running in the same process or different processes with no change to their code.

- **One-way and two-way calls**: Two way calls are dispatched directly from a thread pool (rather than serializing all calls through one thread) for better multi-threading. They are used extensively by all of the system service for incoming IPCs. Traditional one-way calls are used for communicating back with applications.

- **Link to death mechanism**: Allows a process to get a callback when another process hosting a Binder object is terminated. Many of the system services need to clean up

state they have associated with a client process. For example, if an application process goes away, all of its windows should be removed. The driver provides this facility by telling a process about the death of any objects it is watching.

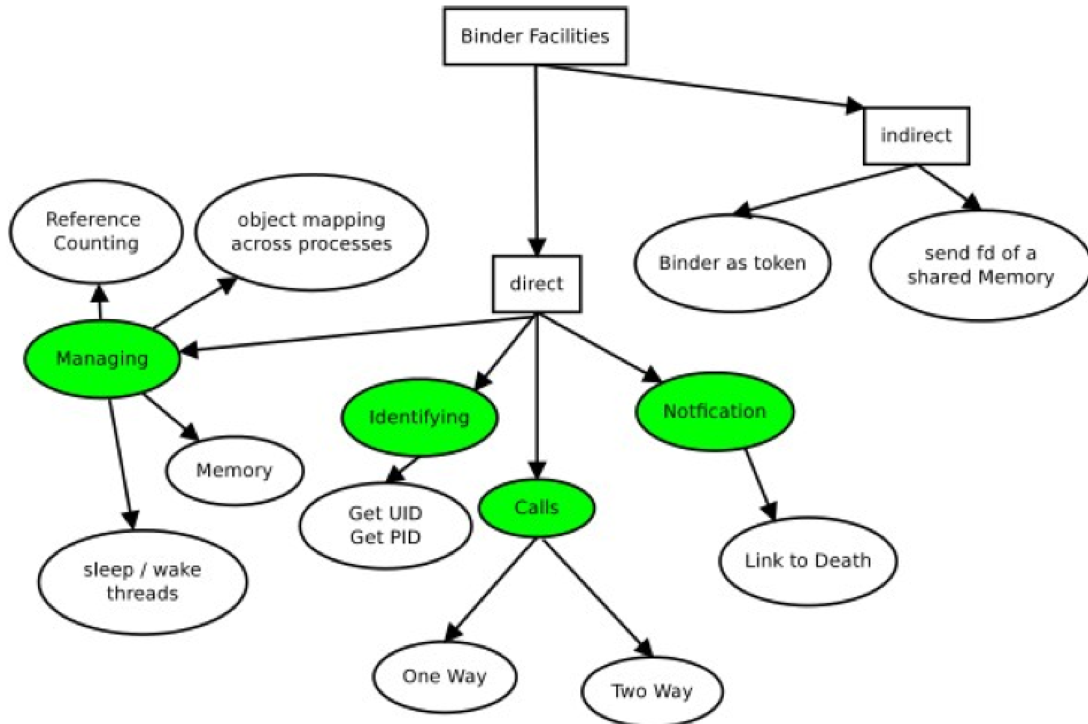


Figure 2-4: Binder IPC facilities

2.3.2 Binder mechanism

The Binder mechanism - simplified in Figure 2-5 - is a client-server model that implements the proxy pattern. On the client side, Binder uses a proxy for communication with the kernel driver. A proxy, in its most general form, is a class functioning as an interface to something else (i.e. a network connection, a file or a large object in memory), which is given to the client to access the service. On the server side, a stub class can be used by the service implementation to extend it to an anonymous class with the implementation of the remote methods; also a pool of multiple Binder threads exists on the server for processing requests [27, 28].

The Binder driver decomposes the method call and all its corresponding data to a level that Linux can understand, transmitting it from the local process and address space to the remote process and address space, and reassembling and re-enacting the call there.

In Figure 2-6 Process A is the client and holds the proxy object that implements the communication with the Binder kernel driver. Process B is the server process, with multiple Binder threads.

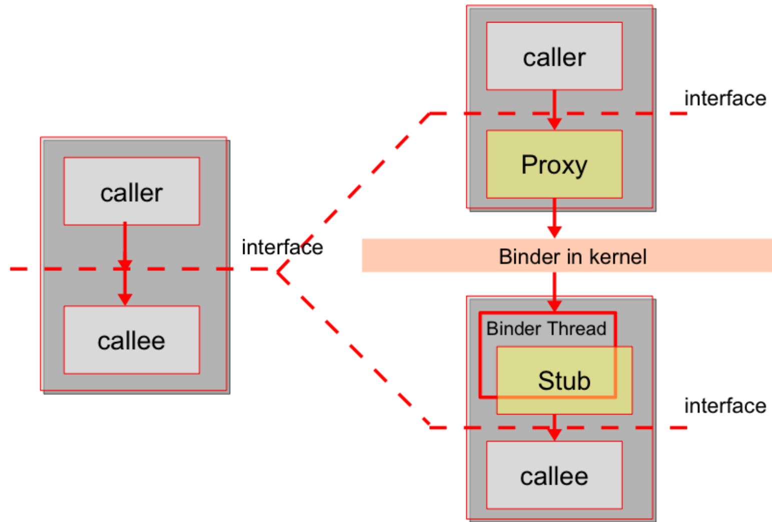


Figure 2-5: Binder mechanism

Service Manager (SM) is the implementation of the *Context Manager*, a special Binder node that serves as name system and makes it possible to assign a name to a Binder interface. Clients do not know the address of a remote Binder a priori, only a Binder interface knows its own address. But if no remote server address is known, no initial communication can happen [28].

Service Manager solves this problem, having the only interface with a fixed Binder address known a priori by all clients. The utilization is simple: first, a Binder submits a name and its Binder token to SM, and then, the client retrieves the Binder address with the service name from SM.

In the diagram below, Process A calls getService() on the server's Binder interface and retrieves an instance of Process B itself, on which the method foo is called. Then, the driver will marshal the message and deliver it to the invoked object; while, the Binder framework will spawn new threads to handle all incoming requests, up to a defined maximum. Finally, when the work is completed, the driver can return to the caller.

2.3.3 Transactions and parceling

If one process sends data to another process, it is called transaction. The data is called transaction data. In an object oriented view, the transaction data is called parcel. A parcel is a container for a message that can be sent through a Binder interface. The procedure of building a parcel is called marshalling (or flattening) an object. The procedure of rebuilding a object from a parcel is called unmarshalling (or unflattening) an object [27, 28].

Parcels are defined by the Parcel class, which is designed as a high-performance IPC transport; however primitive data types can be written/read with the provided functions. A parcel can contain both flattened data that will be unflattened on the other side of the IPC, and references to live Binder objects, which will result in the other side receiving a

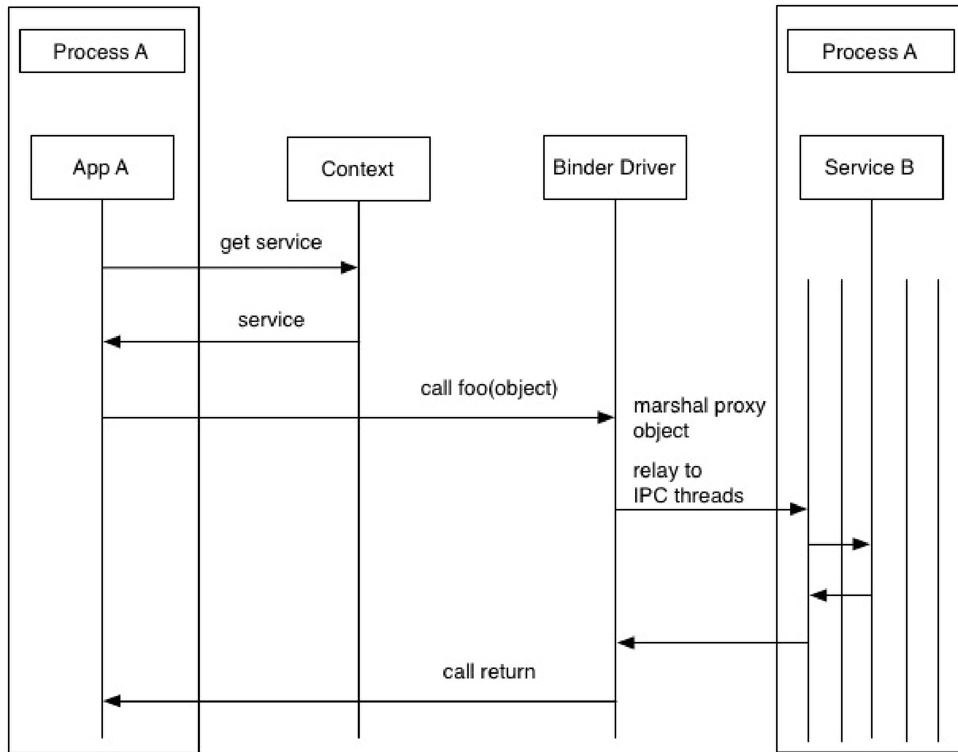


Figure 2-6: Binder interaction

proxy Binder connected with the original Binder in the parcel [27, 28].

Any object, that can be transmitted remotely in a Parcel, must implement the Parcelable interface. The Parcelable protocol provides an extremely efficient protocol for objects to write and read themselves from Parcels. When an object is flattened, both the class type and its data are written to the Parcel, allowing that class to be reconstructed from the appropriate class loader when later reading.

A particular type of Parcelable is represented by the Bundle class. Bundle is a special type-safe container, available for key/value maps of heterogeneous values. This class has many optimizations for improved performance when reading and writing data, and its type-safe API avoids difficult to debug type errors when finally marshalling the data contents into a Parcel.

2.4 Security model

Android's security model provides a set of features based on traditional operating system security controls with the purpose of protecting user data, protecting system resources (including the network), and provide application isolation [26].

Linux is at the heart of the Android system and much of the Android security model is a result of that. The Linux kernel provides Android with several key security features at

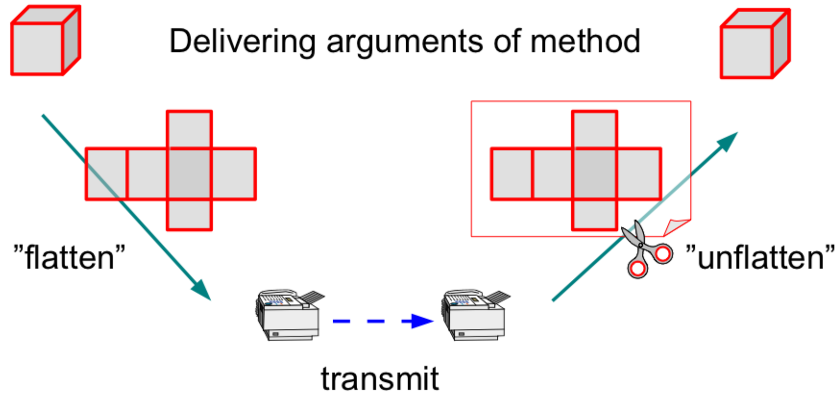


Figure 2-7: Data parceling

the operating system level, including:

- A user-based permissions model;
- Process isolation;
- Secure interprocess communication (IPC);
- The ability to remove unnecessary and potentially insecure parts of the kernel;
- Technologies like ASLR, NX, ProPolice, safe_iop, OpenBSD dlmalloc, OpenBSD calloc, and Linux mmap_min_addr to mitigate risks associated with common memory management errors;

In the next sections are discussed the main security features provided at the application-level by Android's security model, which arise exactly from the underlying Linux kernel security. These are:

- Mandatory application sandbox for all applications;
- Application-defined and user-granted permissions;
- Application signing.

2.4.1 Application Sandbox

The Android platform takes advantage of the Linux user-based protection as a means of identifying and isolating application resources. The Android system assigns a unique user ID (UID) to each Android application and runs it as that user in a separate process [26].

To separate the apps from each other, the discretionary access control (DAC) model of Linux is used, which controls access to files by process ownership. System files are owned by either the system or root user, while other applications have own unique identifiers. Each file is assigned access rules for three sets of subjects: user, group and everyone. In this way,

an application can only access files owned by itself or files of other applications that are explicitly marked as readable/writable/executable for others.

This sets up a kernel-level Application Sandbox. The kernel enforces security between applications and the system at the process level through standard Linux facilities. By default, applications cannot interact with each other and applications have limited access to the operating system. If application A tries to do something malicious like read application B's data or dial the phone without permission (which is a separate application), then the operating system protects against this because application A does not have the appropriate user privileges.

Since the Application Sandbox is in the kernel, this security model extends to native code and to operating system applications. All of the software above the kernel, including operating system libraries, application framework, application runtime, and all applications run within the Application Sandbox.

In some operating systems, memory corruption errors generally lead to completely compromising the security of the device. This is not the case in Android due to all applications and their resources being sandboxed at the OS level. A memory corruption error will only allow arbitrary code execution in the context of that particular application, with the permissions established by the operating system [26].

2.4.2 Permission mechanism

Since the Linux DAC model allows only a rudimentary level of access control, for fine granulated rights Android's middleware offers a permission label system which implements a mandatory access control (MAC) model [26].

A permission label is simply a unique text string that can be defined by both the OS and third party developers. From an OS-centric perspective, applications are statically assigned permission labels indicating the sensitive interfaces and resources accessible at run time; the permission set cannot grow after installation [13].

Security sensitive interfaces are protected by standard Android permissions such as PHONE CALLS, INTERNET, SEND SMS meaning that applications have to possess these permissions to be able to perform phone calls, to access the Internet or to send text messages.

Application developers specify a list of permission labels the application requires in its package manifest. Permissions are approved at installation time based on checks against the signatures of the applications declaring these permissions and on user confirmation [26].

Each definition specifies a "protection level". The protection level can be normal, dangerous, signature, and signatureOrSystem. Upon application installation, the protection level of requested permissions is consulted. The dangerous-level permissions are listed on the screen at the time of application installation, while the normal-level permissions are hidden in a folded menu on the screen [13].

The granting process for the signature-level or signatureOrSystem-level permission requires certificate comparison. The signature-level permission is granted only when the application that requests it and the application that declares it are signed with the same certificate. The signatureOrSystem-level permission is granted in the same manner as the

signature-level permission, and is also granted when the requesting application is signed with the same certificate with which the system images are signed [13].

Applications can also declare custom types of permission labels to restrict access to own interfaces. An application can specify that a caller must have a certain permission by adding a permission requirement to a component's declaration in the manifest or setting a default permission requirement for the whole application. Also, the developer can add permission checks throughout the code [13].

2.4.3 Application signing

Android uses cryptographic signatures to verify the origin of applications and to establish trust relationships among them. Therefore, developers must sign the application code. Applications that attempt to install without being signed will rejected by either Google Play or the package installer on the Android device [26].

On Android, application signing is the first step to placing an application in its Application Sandbox. The signed application certificate defines which UID is associated with which application; different applications run under different UIDs. Application signing ensures that one application cannot access any other application except through well-defined IPC [26].

Applications can be signed by a third-party (OEM, operator, alternative market) or self-signed. Applications do not have to be signed by a central authority. The certificate is enclosed into the application installation package such that the signature made by the developer can be validated at installation time [26].

Applications are also able to declare security permissions at the signature protection level, restricting access only to applications signed with the same key while maintaining distinct UIDs and application sandboxes. A closer relationship with a shared Application Sandbox is allowed via the shared UID feature where two or more applications signed with same developer key can declare a shared UID in their manifest [26].

2.5 Application components

Android applications are written in the Java programming language, and they live in their own security sandbox running within an instance of the Dalvik VM, a register-based virtual machine specifically designed and optimized for Android to provide a seamless run-time environment across a diverse set of performance-limited devices.

Using the Android SDK tools, developers compile all the application's code - along with a mandatory manifest file, plus any additional data and resource files - into a single .apk archive file called Android package, which is then used by Android-powered devices to install the application.

The essential building blocks of an Android application are called *application components*. There are four different types of application components (in Figure 2-8): Activities, Services, Content Providers and Broadcast Receivers. Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed. More

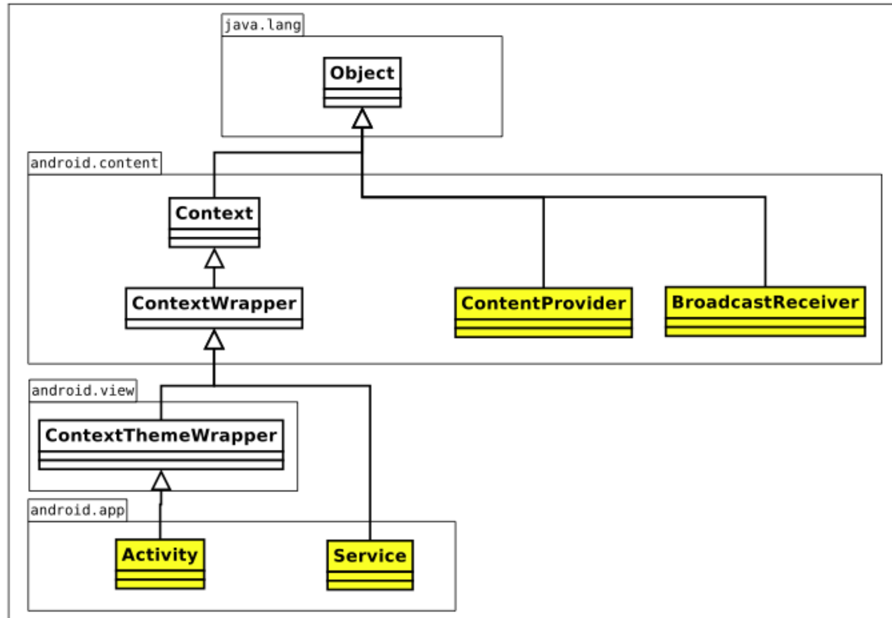


Figure 2-8: Android application components

importantly, each component is a unique building block inside the application, that exists as its own entity and plays a specific role in defining the application's overall functionality and behavior.

Components have the property to be loosely coupled; they are bound only by means of the *application manifest* file that describes each component in an application and how they interact. The main advantage of decoupling the dependencies between application components is that Content Providers, Services and Activities can be shared with other applications, even those of third parties. Therefore, each component is a different point through which the system can enter an application; however, not all components are entry points for the user because some depend on each other. An interesting aspect of this unique design is that, a component, when started by the system, does not live inside the process of the caller; instead, the system starts the process for that application (if it's not already running) and instantiates the classes needed for the component. In such way, Android's application sandboxing is not violated.

In fact, since applications are run in separate processes, with their own user ID and file permissions that restrict access to other applications, actually no other process but the system can directly activate a component in another application. Thus, it is needed a mean to request the system to activate a particular component on behalf of one application; that mean is called an Intent message.

The following sections look more closely at the properties and design of each type of component that model Android applications.

2.5.1 Activities

Activities constitute the presentation layer of an application. An Activity represents a single screen, and is given a window in which to draw a user interface. It is implemented as a subclass of Activity and makes use of Views, Widgets and Fragments to build the user interface and to interact with the user [30].

An Android application can have several Activities that work together to form a cohesive user experience. For example, an email client application might have three Activities: one for listing the inbox, another to compose an email, and another for reading emails. However, every Activity is independent of the others, and as such, can be started by a different application (with due permission). For instance, a camera application could start the "compose mail" Activity in the email application, in order for the user to share a picture.

Usually, but not necessarily, one Activity is declared as the "launcher" - or "main" - Activity, which is presented to the user when newly starting the application.

An Activity can also start another Activity to perform some defined user action (eventually expecting a result). When this occurs, a "last in, first out" stack - called the "back stack" - is used by the system to preserve the previous Activity. The back stack works in two ways: when a new Activity starts, it is pushed onto the stack and gains user focus; when the user presses the Back button, the current activity is popped from the stack (and destroyed) and the previous activity resumes [30].

Activity lifecycle

To provide a sound and slick user experience, developers should take care of managing the lifecycle of Activities in their application, this is done by implementing the callback methods inherited from the Activity class. The lifecycle of an Activity is directly affected by its interaction with other Activities, its task and back stack. There are three major states in which an Activity can exist [30, 31]:

- **Resumed** — When the Activity is running in the foreground of the screen and has user focus.
- **Paused** — If another Activity is in the foreground and has focus, but this one is still visible. This is the case when, another Activity - that is partially transparent or doesn't cover the entire screen - is visible on top of this one. A paused Activity is completely alive and attached to the window manager, but can be killed by the system in low memory conditions.
- **Stopped** — When the Activity is in the background, that is, completely obscured by another Activity and no longer visible to the user. A stopped Activity is also still alive, but is not attached to the window manager. However, it is likely a candidate to be killed by the system when memory is needed elsewhere.

If an Activity is either paused or stopped, the system can release it from memory by asking it to finish (calling its finish() method), in normal conditions, or simply killing

its process, in extreme conditions. When the same Activity is opened again, it must be completely recreated.

2.5.2 Services

Services are components that Android specifically offers to applications for performing long-running operations, or to supply functionality for the other applications to use - for example, handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background. Without having a user interface, Services update data sources and activities by broadcasting Intents, and notify the user via the notification framework in Android. Another application component, such as an activity, can start a Service and let it run, or bind to it in order to interact with it and even perform inter-process communication (IPC); the Service will continue to run in the background even if the user switches to another application [30, 31].

A Service is NOT:

- a *separate process*. Unless otherwise specified, a Service object runs in the same process as the application is part of.
- a *thread*. It is not an alternative to avoid Application Not Responding errors (ANR), and it does not do work off of the main thread by its own.

It is on the responsibility of the developer to create a new thread within the Service to do any CPU intensive work or blocking operations. In such way, the application's main thread can remain dedicated to user interaction within activities, reducing the risk of ANR errors [30].

There are two different facilities that a Service can provide to an application, for which the Service itself can assume as many forms, even at the same time [30]:

- **Started** — When an application component starts a Service by calling startService(), usually to perform a single operation without expecting a result. Once started, a Service can run in the background indefinitely, even if the caller component is destroyed. However, when the operation is done, the Service should stop by itself.
- **Bound** — When an application component binds to it by calling bindService(), usually to expose some of its functionality to other applications. A bound Service offers a client-server interface that allows to establish a long-standing connection to the Service in order to interact with it, send requests, get results, and even do so across processes with IPC. Multiple components can bind to a Service at once; the Service will continue to run until there are no active bindings, after that, the Service is destroyed.

A Service can be both started and have components bound to it. In such a case, the system will keep the Service running as long as either it is started or there are one or more active bindings.

Unless a Service is declared private, in the Android manifest file, and blocking access from other applications, any application component can use it (whenever it is started, bound or both) - even from a separate application - just in the same way any component can use an activity, by starting it with an Intent.

Service lifecycle

A started Service must manage its own lifecycle, which is independent of the component that started it. So a Service must stop itself by calling stopSelf(), or eventually another component can stop it by calling stopService() [30].

Once requested to stop with stopSelf() or stopService(), the system destroys the Service as soon as possible. Otherwise, the Service will continue to run indefinitely in the background after onStartCommand() returns, unless the system must recover memory.

In the event the system kills the Service after onStartCommand() is called, there are three possible modes of operation for restarting the Service, which can be defined by returning one of the following integer values from onStartCommand() [30]:

- START_NOT_STICKY — Tells the system to not recreate the Service, unless there are pending Intents to deliver. This is the safest option to avoid running a Service when not necessary and when unfinished jobs can be safely restarted.
- START_STICKY — Tells the system to recreate the Service and call onStartCommand() with a null Intent, unless there were pending Intents to start the Service. This is suitable for Services that are not executing commands, but running indefinitely and waiting for a job (i.e. media players).
- START_REDELIVER_INTENT — Tells the system to recreate the Service and call onStartCommand(), with the last Intent that was delivered to the Service. Any pending Intents are delivered in turn. This is suitable for Services that are actively performing a job that should be immediately resumed, such as downloading a file.

If a Service handles multiple requests to onStartCommand() concurrently, it shouldn't be stopped when processing to a single start request is completed. To avoid this, the stopSelf(int) method can be called passing the ID of the start request (the startId delivered to onStartCommand()) that should be matched by the stop request.

If a Service is purely bound, then its lifecycle will be managed by the Android system on whether it is bound to any clients. As such, when a Service is unbound from all clients, the Android system destroys it. Additionally, a Service can be both started and bound. A component can bind to a Service that was already started with startService(). In this case, stopService() or stopSelf() does not actually stop the Service until all clients unbind. Vice versa, if all clients call unBindService(), the Service will run until it stops by itself with stopSelf() or another component calls stopService() [30].

The Android system will kill a service only when memory is low and it must recover system resources for the activity that has user focus. In this case, the priority of a process hosting a Service will be the higher of the following possibilities [30]:

- *The Service is currently executing code in its onCreate(), onStartCommand(), or onDestroy() methods.* Then the hosting process will be a foreground process, and the Service can execute the code without being killed.

- *The Service has been started.* Then the hosting process is considered more important than any other currently visible to the user on-screen, but more important than any process not visible. Thus, the Service should not be killed except in extreme low memory conditions. Additionally, if the Service is long-running, the system will lower its position in the list of background tasks over time, thus becoming highly susceptible to killing.

- *There are clients bound to the Service.* Then the hosting process is never less important than the most important client.

- *The (started) Service called startForeground() to enter the foreground state.* Then the system considers it to be something the user is actively aware of and thus not a candidate for killing when low on memory.

If the system kills a Service, it will try to restart it later as soon as resource become available again - depending on the value return from onStartCommand, as discussed before. Finally, other application components running in the same process as a Service will contribute to increase the importance of the overall process beyond just the importance of the Service itself.

2.5.3 Content Providers

Content Providers implement a shareable persistent data storage. Data can be stored in the file system, an SQLite database, on the Web, or any other persistent storage location accessible by an application [30, 31].

Content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. They are the standard interface to share data across application boundaries. Applications can configure their own Content Providers to allow access from other applications, through which can query or even modify the data. However, Content Providers are also useful for reading and writing data that is private to an application and not shared.

Usually, Content Providers are not accessed directly, but through a request to the ContentResolver object in the application's Context, which provides an interface to communicate with the Content Provider as a client. The ContentResolver object handles all direct transactions with the Content Provider, introducing a layer of abstraction for security, so that the component requesting information calls methods on the ContentResolver.

Finally, Android itself includes several native Content Providers that manage data such as audio, video, images, and personal contact information. These providers can be accessed by any application having the required permissions.

2.5.4 Broadcast Receivers

Broadcast Receivers enable applications to listen for system-wide broadcast announcements and handle Intents messages matching a specific criteria; they are useful for creating event-driven applications [30].

Many broadcasts originate from the system - for example, a broadcast announcing that the screen has turned off, the battery is low, or the system completed the boot process. Applications can also initiate broadcasts - for example, to alert other applications that a download was completed and the data is available for use.

Broadcast Receiver components subscribe to specific "action strings" - defined by the system or by the developer - and act as specialized event handlers. By registering the Broadcast Receiver in the Android manifest file, an application can have that receiver's class started by the system and called whenever someone sends the specified broadcast message.

Messages are sent as Intents, and the system handles dispatching them, including enforcing permissions. When an "action strings" is used and no target component name is specified, the Activity Manager uses the IntentFilters applications register to figure out which application to use for a given broadcast [31].

Usually, a Broadcast Receiver is just a "gateway" to other components and is intended to do a minimal amount of work - for example, it might start a service to perform some work based on the event. In fact, Broadcast Receivers don't display a user interface, instead, they may show status bar notifications when an event occurs. Broadcast Receivers may also be used dynamically during the lifecycle of an activity; usually, they are registered in the activity's onStart() method, and unregistered the onStop() method.

There are two major classes of broadcasts that can be sent from within an application [30]:

- **Normal broadcasts** (completely asynchronous) — All receivers are run in an undefined order, often at the same time. While being efficient, they cannot return results or abort their broadcast. In the case they are requiring the creation of a process, only one will be run at a time to avoid overloading the system with new processes - still non-ordered semantics hold.

- **Ordered broadcasts** (delivered to one receiver at a time) — Each receiver is executed in turn and it can either propagate a result to the next receiver, or completely abort the broadcast. A priority order for the receivers can be defined from the Android manifest file; receivers with same priority will be run with an arbitrary order.

Although an Intent object is used for sending and receiving broadcasts, the semantics and the mechanism of the Intent broadcast are completely separate from Intents used to start activities. Broadcast Receivers can't absolutely capture Intents used with startActivity(); as such, it is impossible for a broadcast Intent to find or start an activity. Semantically, a broadcast is a background operation that normally the user is not aware of, and not a foreground operation that modifies the user interaction.

Sticky Broadcasts

A particular type of broadcast messages supported by Android are the "sticky" broadcasts. These messages are usually informational and designed to tell other processes some fact about the system state [31].

Unlike normal broadcasts - that terminate once they are delivered to the intended receiver - a sticky broadcast remains in existence so that it can notify other applications if they need the same information. Existing sticky broadcasts are replaced when the same event occurs again.

Applications need a special privilege, BROADCAST_STICKY, to send or remove a sticky broadcast. Sticky broadcasts must not be used for exchanging sensitive information, since they can't be secured like other broadcasts can.

2.6 Inter-component communication using Intents

Android provides a sophisticated message passing system, in which Intents are used to link applications. An Intent is a message that declares a recipient and optionally includes data; an Intent can be thought of as a self-contained object that specifies a remote procedure to invoke and includes the associated arguments. Applications use Intents for both inter-application communication and intra-application communication. Additionally, the operating system sends Intents to applications as event notifications. Some of these event notifications are system-wide events that can only be sent by the operating system. These messages are called system broadcast Intents [12, 30, 31].

More specifically an Intent represents a blob of serialized data that can be moved between application components to get something done. Intents usually have an action, which is a string like "android.intent.action.VIEW" that identifies some particular goal, and often somedata in the form of a Uri. Intents can have optional attributes like a list of Categories, an explicit type (independent of what the data's type is), a component, bit flags and a set of name value pairs called "Extras" [31].

Intents can be sent between three of the four components (in Figure 2-9): Activities, Services, and Broadcast Receivers. Intents can be used in a number of ways [12, 31]:

- To start an Activity coordinating with other programs like browsing a web page.
  - Using Context's startActivity() method.
- As broadcasts to inform interested programs of changes or events.
  - Using Context's sendBroadcast(), sendStickyBroadcast(), and sendOrderedBroadcast() family of methods.
- As a way to start, stop or communicate with background Services.
  - Using Context's startService(), stopService(), and bindService() methods.
- To access data through ContentProviders, such as the user's contacts.

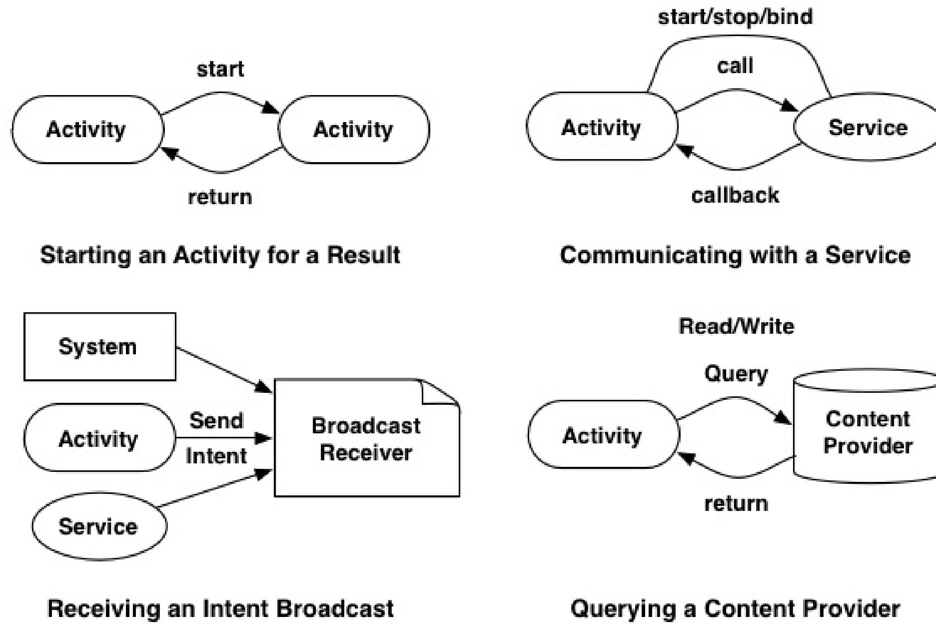


Figure 2-9: Android inter-component communication

  - Using Context's getContentResolver() or Activity's managedQuery().

- As call backs to handle events, like returning results or errors asynchronously with a PendingIntent, that is a description of an Intent and target action to perform with it, that can be handed to other applications so that they can perform the action described on behalf of the caller (with same permission and identity) at a later time.

For a Service or Activity to receive Intents, it must be declared in the manifest. (Broadcast Receivers can be declared in the manifest or at runtime.) Intents can be used for explicit or implicit communication. An explicit Intent specifies that it should be delivered to a particular application specified by the Intent, whereas an implicit Intent requests delivery to any application that supports a desired operation. In other words, an explicit Intent identifies the intended recipient by name, whereas an implicit Intent leaves it up to the Android platform to determine which application(s) should receive the Intent [12, 30, 31].

All of the forms of communication listed previously can be used with either explicit or implicit Intents. By default, a component receives only internal application Intents (and is therefore not externally invocable). Additionally, Intents can be restricted with manifest permissions. This allows to create Activities, BroadcastReceivers, ContentProviders or Services that can only be accessed by applications the user has granted these rights to.

2.6.1 Intent filters

To inform the system which implicit Intents they can handle, Activities, Services, and Broadcast Receivers can have one or more Intent filters. Filters advertise the capabilities

of a component, a set of Intents that the component is willing to receive. They open the component to the possibility of receiving implicit Intents of the advertised type. If a component does not have any Intent filters, it can receive only explicit Intents. A component with filters can receive both explicit and implicit Intents [30, 32].

An Intent filter can constrain incoming Intents by action, data, and category; the operating system will match Intents against these constraints. An *action* specifies a general operation to be performed, the *data* field specifies the type of data to operate on, and the *category* gives additional information about the action to execute. For example, a component that edits images might define an Intent filter that states it can accept any Intent with an EDIT action and data whose MIME type is image/* [12].

To be delivered to the component that owns the filter, an implicit Intent must pass all three tests. If it fails even one of them, the Android system won't deliver it to the component at least not on the basis of that filter. However, since a component can have multiple Intent filters, an Intent that does not pass through one of a component's filters might make it through on another [30].

Multiple applications can register components that handle the same type of Intent. This means that the operating system needs to decide which component should receive the Intent. Broadcast Receivers can specify a priority level (as an attribute of its Intent filter) to indicate to the operating system how well-suited the component is to handle an Intent. When ordered broadcasts are sent, the Intent filter with the highest priority level will receive the Intent first. Ties among Activities are resolved by asking the user to select the preferred application (if the user has not already set a default selection). Competition between Services is decided by *randomly* choosing a Service [12].

Since the Android system must know about the capabilities of a component before it can launch that component, Intent filters are generally not set up in Java code, but in the application's manifest file (AndroidManifest.xml) as ⟨intent-filter⟩ elements. (The one exception would be filters for broadcast receivers that are registered dynamically.) [12, 30]

It is important to note that Intent filters are not a security mechanism. A sender can assign any action, type, or category that it wants to an Intent (with the exception of certain actions that only the system can send), or it can bypass the filter system entirely with an explicit Intent. Conversely, a component can claim to handle any action, type, or category, regardless of whether it is actually well suited for the desired operation.

2.7 Exporting components

At the most fundamental level of access control lies the Android concept of *exporting*. Each component, be it an Activity, a Service, a Content Provider, or a Broadcast Receiver can be public or private. Private components are accessible only by components within the same application. When declared as public, components are reachable by other applications as well, however, full access can be limited by requiring calling applications to have specified permissions [12, 32].

Applications often contain components that another application should never access

- for example, an activity designed to return a user-entered password could be started maliciously. Instead of defining an access permission, a component can be made private by either explicitly setting the *exported* attribute to false in the manifest file or letting Android infer if the component should be private from other attributes in its manifest definition [12, 14, 32].

Private components simplify security specification. By making a component private, the developer doesn't need to worry which permission label to assign it or how another application might acquire that label. Any application can access components that aren't explicitly assigned an access permission, so the addition of private components and inference rules significantly reduces the attack surface for many applications [12].

The default behavior depends on whether the component is likely to be used externally. In the case of Activities, Services, and Broadcast Receivers, the default depends on how the component is configured with regard to Intents. A component can specify an Intent filter that allows it to receive Intents from other apps to carry out tasks. As such, if a component specifies an Intent filter, it is assumed that the developer wants the component to be accessed by other apps and it is, by default, exported and thus public. If, however, no Intent filter is specified for a component, the only way to send an Intent to it is to fully specify the component's class name. Therefore, it is assumed that the developer does not want to make this component publicly accessible, so the default exported is false and the component is private [32].

However, the developer must be careful when allowing Android to determine if a component is private. Security-aware developers should always explicitly define the exported attribute for components intended to be private [12].

2.8 Android Manifest file

The general deployment format of an Android application is a signed archive. Compiled codes of an application and binary resources are archived in a Zip-compatible format, and signed with the author's certificate. The package also accompanies an XML file, called the Android manifest file. The manifest file includes the following authorization-related information [30]:

- **List of application sub-components**: The XML element, ⟨application⟩, is composed of a set of XML sub-elements for components such as ⟨activity⟩, ⟨service⟩, ⟨provider⟩, and ⟨receiver⟩.

- **List of permission declarations:** An application can declare a permission using ⟨permission⟩ element. The permission is added to a system when the application is installed. Let us call the permissions declared-permissions.

- **List of permissions expected to be granted**: An application lists the permissions needed to accomplish its task, using ⟨uses-permission⟩ element. The permissions are requested at the time of installation, and are listed on the screen. The user either allows the installation or aborts it. Allowing installation means granting all of the

requested permissions as well. We use different terms for the requested permissions at the time of installation and for the permissions after being granted: requested-permissions and use-permissions of the application, respectively.

- **List of permissions used for protection**: The ⟨application⟩ element and the component elements have the android:permission attribute. If a permission name is assigned to the attribute, access to the application or the component requires the permission. If the attribute is set by the application element, all of its sub-components are protected by the permission as well. If a permission is assigned to the attribute in the component element, access to the component requires the permission set by the component, but the application-level permission enforcement setting is ignored. Let us call the permissions used to protect an application or a component, enforce-permissions of the application or the component.

Chapter 3

Overview of Liferay Portal

Liferay Portal is the world's leading open source enterprise portal solution using the latest in Java and Web 2.0 technologies. A portal is generally defined as a software platform for building websites and web applications. Modern portals have added multiple features that make them the best choice for a wide array of web applications. More than a portal, Liferay is a platform for creating effective business applications and solutions. It offers a robust feature set, impressive scalability, time-saving development tools, support for over 30 languages, and a flexible, scalable architecture that is open source developed and enterprise refined.

Liferay Portal ships with broad product and technical capabilities to provide immediate return on investment:

- Content & Document Management;
- Web Publishing and Shared Workspaces;
- Enterprise Collaboration;
- Social Networking and Mashups;
- Enterprise Portals and Identity Management;
- Compatible with all major databases, operating systems, and app servers;
- Hierarchical system of sites and organizations;
- Granular, delegable permissioning system;
- Completely exposed API supporting web services (SOAP), JSON and RMI;
- Highly scalable, supporting more than 5,000 concurrent transactions (33,000 simultaneous users) per server;
- Real-world performance of millions of pageviews and 1.3 million users;
- Robust user management and security features including password policies, user reminder settings, and complete log-in security procedures.

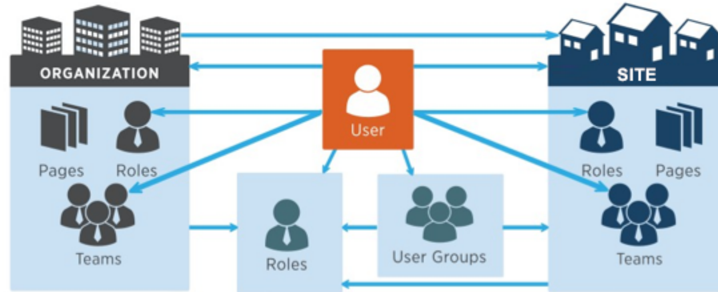


Figure 3-1: Liferay permission model

In this chapter, we will give an overview of the architecture of Liferay Portal and introduce the *"Documents and Media Library"* portlet, which is the basis for our Liferay Safe Android application. The information given in the next section are a summary of the Liferay Portal Users Guide [33] and Administrators Guide [34].

3.1 Portal architecture

Liferay Portal is structured around how users can be grouped together. Some of these groupings follow an administratively organized hierarchy, and other groupings may be done by the users themselves. And other groupings may be done administratively via Roles for other functions that may cut across the portal.

In particular, Liferay uses the following concepts to organize the portal:

- Portals are accessed by Users.
- Users can be collected into User Groups.
- Users can belong to Organizations.
- Organizations can be grouped into hierarchies, such as Home Office → Regional Office → Satellite Office.
- Users, Groups, and Organizations can belong to Sites regarding a common interest.
- Within Organizations and Sites, users can belong to Teams, which are groupings of users for specific functions within a site or organization.

This way of organizing portal concepts may be illustrated as shown in Figure 3-1 below.

In the illustration above, each arrow may be read using the words *"can be a member of"*. So this means that Organizations can be members of Sites, Sites can be members of Roles, Users can be members of anything, and so on. It is important to note that the diagram illustrates only users and their collections. Permissions do not flow through all of these collections: permissions can be assigned to Roles only.

Teams are inside individual organizations and sites, and are only available as they are created within those organizations and sites. Roles that appear inside organizations and sites are roles that are scoped just for organizations and sites. This means that though each organization and site in the portal has this role with its configured permissions, membership in this role is different for each organization and site.

3.1.1 Users

Users represent physical users of the system. These are the user accounts that people use to log into the system. By default, users get their own private sites with public and private pages that they can manage themselves, and this can be turned off or locked down by administrators. The personal space enables users to have their own public blog or their own private calendar, a place to store their documents, and more.

Users can be collected in multiple ways:

- They can be members of organization hierarchies, such as Liferay, Inc. → Security → Internet Security.
- They can be collected into arbitrary user groups.
- They can be members of sites which draw together common interests.

Finally, Users can have roles which define their permissions in the system, and these roles can be scoped by Portal, Organization, or Site.

3.1.2 User Groups

User Groups are simple, arbitrary collections of users. User groups can be members of sites or roles, and are created by portal administrators to group users together who dont necessarily share an obvious hierarchical attribute.

Users can be assigned to multiple user groups. Permissions cannot be assigned to User Groups. User groups can simplify the assignment of several roles to a group of users, and simplify the membership to one or more sites by specifying a group of users.

3.1.3 Roles and permissions

Roles are groupings of users that share a particular function within the portal, according to a particular scope. Roles can be scoped by portal, site or organization. A regular role is a portal-scoped role. Users, User Groups, Sites, or Organizations can be members of a role.

Roles serve as repositories of permissions to be assigned to users who belong to them. Portal permissions cover portal-wide activities that comprise several categories, such as site, organization, location, password policy, etc. Site Content permissions cover the content the installed portlets create. Site Application permissions affect the application as a whole. And the Control Panel permissions affect how the portlet appears to the user in the control panel.

Liferays roles and fine-grained permissions are very powerful and ensure that administrators can customize exactly which areas of the portal theyd like different collections of users to be able to access.

3.1.4 Organizations

Organizations are hierarchical collections of Users. They are handy for defining where a user belongs in a particular hierarchy. Organizations in Liferay are designed to model any group hierarchy, from those of government agencies all the way down to those of small clubs.

Organizations and sub-organizations can be created in a hierarchy to unlimited levels and users can be members of one or many organizations. These organizations can all reside in a single hierarchy or cut across different hierarchies. The rights of an organization administrator apply both to his/her organization and to any child organizations. By default, members of child organizations are members of the parent organizations. Additionally, Organizations can be associated with roles. This allows a portal to be designed to correspond with any existing organization chart and users permissions are granted according to their positions in the chart.

For modeling more complex hierarchical structures, organizations can be combined with teams and scoped roles to assemble the sets of permissions to be granted to particular users.

3.1.5 Teams

Teams are unique within a context of a Site or Organization. Teams are essentially sets of users that can be created within a site. This makes teams different from the Site and Organization Roles because teams appear only in the specific site or organization in which they are created. This is very useful when a team of users is needed for a specific purpose within a site or organization and not for each site or organization in the portal.

Teams can also be essential for some use cases, because they can be created by Site or Organization Administrators. Site and Organization Administrators cannot create roles, so the ability to have teams empowers them to manage permissions at a level they weren't capable of previously.

3.1.6 Sites

Sites are collections of Users who have a common interest. Liferay's default pages are part of a site named for the portal, because everyone - whether they are anonymous or members of the portal - has a common interest in the default, public pages of the portal site.

There are three types of Sites: Open, Restricted and Hidden. An Open Site (the default) allows portal users to join and leave the Site whenever they want to. A Restricted Site requires that users be added to the Site by a site administrator. A Hidden site is just like a Restricted site, with the added concept that it does not show up at all in the Sites portlet or the Control Panel. Users will have to be added to a hidden site by a site administrator.

3.2 Portlets

Liferay Portal is not only a portal platform. It also provides more than 60 applications and tools included in the form of portlets, which are compliant with the JSR 168 Portlet Specification:

> *"Portlets are web components–like servlets–specifically designed to be aggregated in the context of a composite page. Usually, many portlets are invoked to in the single request of a portal page. Each portlet produces a fragment of markup that is combined with the markup of other portlets, all within the portal page markup."* [35]

These portlets are available out of the box, or in the form of third-party plugins created and deployed using the Plugins SDK. Portlets can be developed using any prefered framework, such as Struts, Spring MVC, JSF, or with the Liferay MVCPortlet framework which is simple, lightweight, and easy to understand. The Plugins SDK can be used also to deploy a PHP or Ruby application as a portlet, allowing non-Java developers to take advantage of Liferay's built-in features (such as user management, communities, page building and content management).

3.3 Documents and Media Library portlet

Liferays Documents and Media library is a portlet that provides a mechanism for storing files online using the same type of structure that you use to store files locally. It can be used inside Liferay Portal to store files of any kind; it serves as a virtual shared drive. Users by default, have their own personal sites with public and private pages. They can use their personal sites to host Documents and Media portlets for storing or sharing their own files.

The Documents and Media Library integrates two portlets: the Document Library and the Image Gallery. The Document Library provides document management backed by the Jackrabbit JSR-170 compliant Java content repository. Includes check in / check out, meta data, versioning, and on-the-fly file format conversion. The Image Gallery portlet does not serve as a repository but just displays selected content from the Documents and Media library. It can display image, audio and video files.

Important features included in the Documents and Media library are: customizable document types and metadata sets (in addition to the portals system of tags and categories), automatic document preview generation and support for mounting multiple external repositories.

Customizable document types and metadata sets are an addition to the portals system of tags and categories. When a user assigns a document type to a document, the user is required to fill out the fields defined by the metadata set of the document type. This encourages users not to forget to enter important information about their documents. More importantly, document types and metadata sets can improve document searchability. The values that users enter into the fields determined by their document types metadata set become searchable entities within the portal.

When creating a new document type, one can define "Main Metadata Fields" or "Additional Metadata Fields". Main metadata fields are directly tied to their document type and cannot be made available to other document types. Additional metadata fields, by contrast, can be defined independently and can be used in many different document types. Document types that implement the same additional metadata set can be differentiated by defining different main metadata fields for them.

Chapter 4

Client-Server architecture

The solution is based on a client-server architecture, which is very typical for mobile applications.

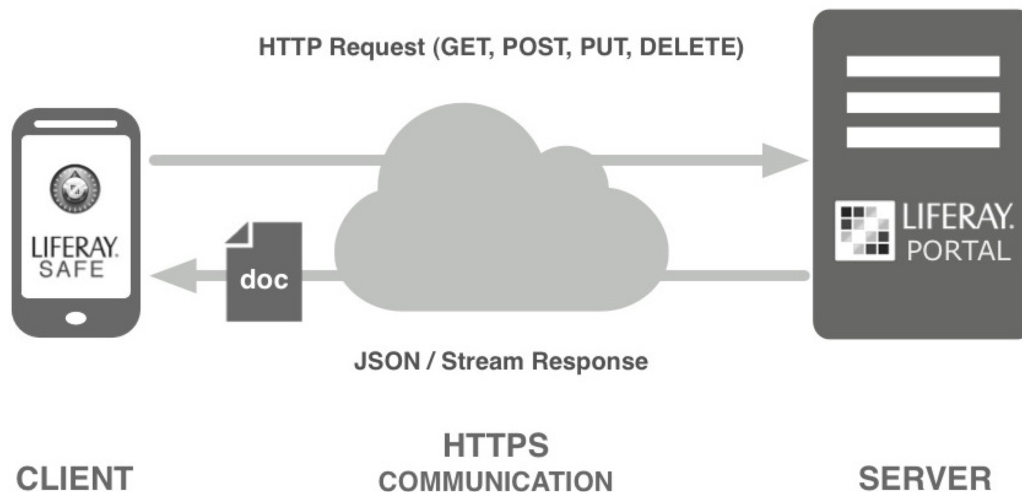
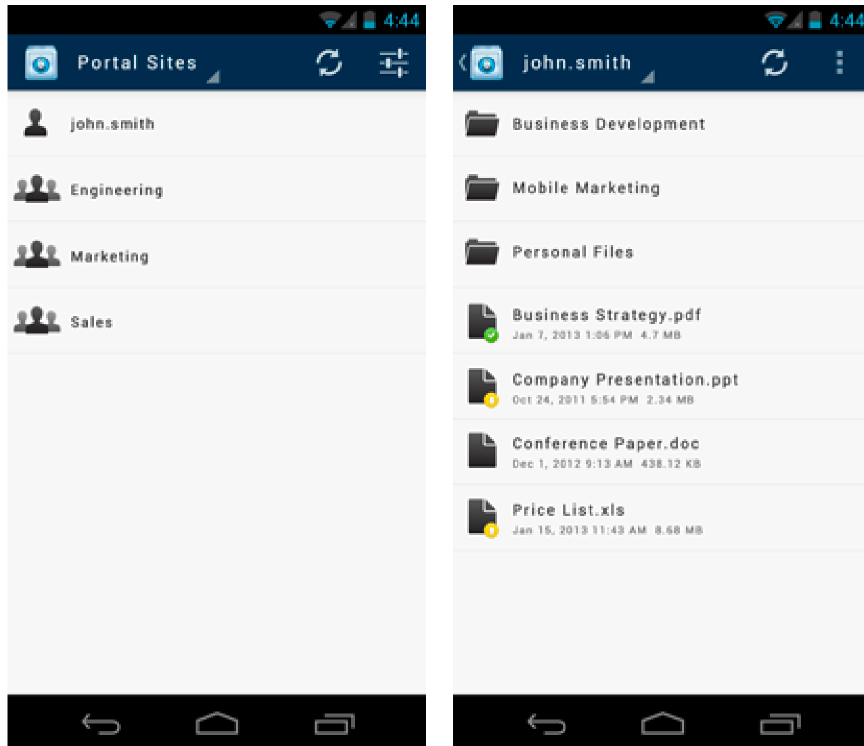


Figure 4-1: Client-Server architecture

The client (in Figure 4-2) is a native application for Android which allows the user to retrieve a synchronized list of all the documents stored in one or more repositories organized in a Liferay portal server. To be clear, a "document" is intended to be any type of file that can be stored in the portal's Documents and Media Library. The user can download a document to his device and access it later, even offline when a connection is not available. The user can open a downloaded document with other applications installed on the device and upload modifications to the server. Additionally, new documents can also be uploaded, as well as new folders can be created. All these operations can be performed only if the user has the necessary permissions, which are managed and enforced by the server. If a document is marked as "confidential" the user is limited to only viewing the document, either in online or offline mode. The client will treat the document with specific security mechanisms in order to preserve its privacy.



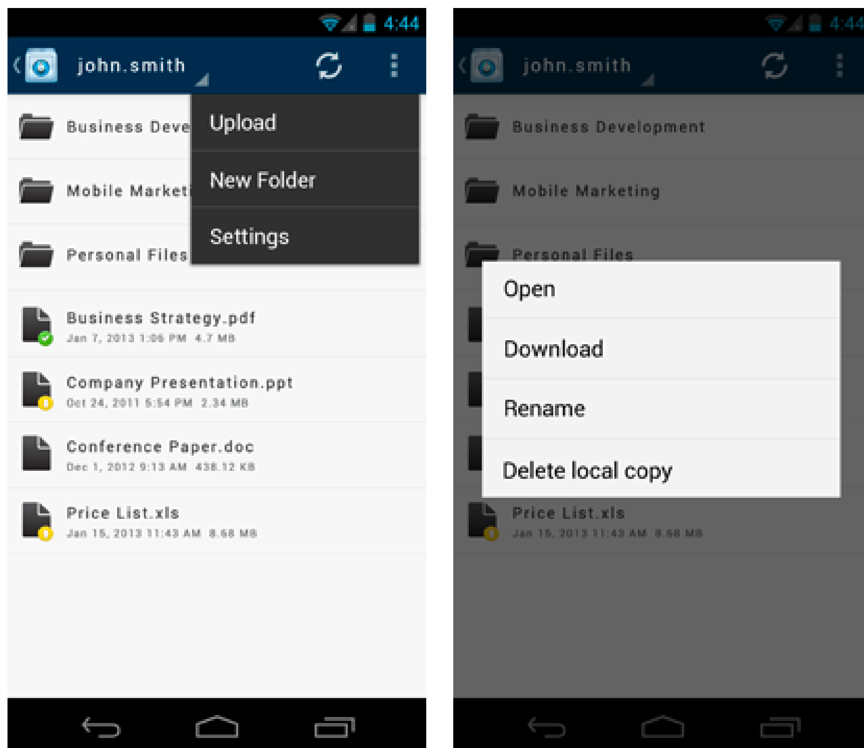


Figure 4-2: Liferay Safe

The server runs an instance of Liferay Portal (version 6.1) which exposes to the client a Web Services API returning document descriptions and synchronization updates in the form of textual JSON responses, as well as the contents of the documents themselves as data streams.

The communication process is initiated by the client invoking one of the portal services methods exposed by the API. The server processes the request, prepares a response and then sends it back to the client. HTTP GET and HTTP POST request methods are used in the communication. GET methods are used when the client requests resources (e.g., document descriptions, sync updates, or document contents). The client does not use GET requests to send updates to the server (so GET requests therefore should not change the server status). Only the POST request method is used for submitting document descriptions and uploads.

Only the secure HTTP protocol (HTTPS) is used for communication, since the security of data in transit relies only on the security of the communication itself. The details of this topic are discussed in the next section.

4.1 Communication layer security

The use of a secure communication channel is important in the context of a mobile client-server application. Users of mobile devices are prone to use relatively insecure networks for Internet connectivity, such as open WiFi networks, which are at high risk for eavesdropping and man-in-the-middle attacks. For this reason all client-server traffic must be transported over an encrypted communication channel, especially if there are confidential data that must be protected.

The protocol used for communication security is the Transport Layer Security (TLS) protocol (version 1.2, RFC 5246). The protocol is composed of two layers: the TLS Record Protocol (at the lowest layer, on top of the TCP transport protocol) and the TLS Handshake Protocol. TLS allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. In particular, the following connection security properties are provided:

- The connection is private. Symmetric cryptography is used for data encryption (e.g., AES). The keys for this symmetric encryption are generated uniquely for each connection and are based on a secret negotiated by the TLS Handshake Protocol.
- The connection is reliable. Message transport includes a message integrity check using a keyed MAC. Secure hash functions (e.g., SHA-1) are used for MAC computations.
- The negotiation of a shared secret is secure. The negotiated secret is unavailable to eavesdroppers, and for any authenticated connection the secret cannot be obtained, even by an attacker who can place himself in the middle of the connection.
- The negotiation is reliable. No attacker can modify the negotiation communication without being detected by the parties to the communication.

One advantage of TLS is that it is application protocol independent. The HTTP protocol is layered on top of the TLS protocol transparently. Conceptually, HTTP is used over TLS precisely as it is used over TCP, as specified in RFC 2818.

The secure transmission of data in transit relies not only on encryption but also on authentication - on both the hiding or concealment of the data itself, and on ensuring that the parties at each end are the parties they say they are. While client authentication is optional, server authentication is mandatory. In order to prevent man-in-the-middle attacks, the client must validate the server's TLS certificate as defined by RFC 6125 and in accordance with its requirements for server identity authentication. Certificates must be fully validated by the client and signed by a trusted root Certification Authority (CA). Lower levels of encryption (such as export-grade 40-bit encryption) must be disabled on the server.

If the client application already knows the server it is connecting to, there is no reason to rely on CA signatures to validate the authenticity of the connection (which is the default behavior), in this case a self-signed certificate can be used. CA certificates are for general purpose network communication where the destinations aren't known ahead of time. Furthermore, because the client knows the server address it will connect to, the client can validate if the self-signed certificate presented by the server is actually the correct certificate, by comparing it with a certificate already bundled with the application. This is called certificate pinning and will prevent against CA compromises such as the well-publicized DigiNotar and Comodo incidents. Certificate pinning and self-signed certificate validation can be implemented in Android using a custom certificate Trust Store [36].

Last but not least is to eliminate all unsecured HTTP traffic both for the API exposed by the server and the client application, and make the client validate that TLS is active on the connection. This protects the connection against a security layer stripping attack. This man-in-the-middle attack transparently hijacks HTTP traffic on a network, monitors for HTTPS requests and then eliminates the TLS leaving the client on an unsecured connection.

4.2 User authentication & Client authorization

Liferay Portal exposes a set of APIs, called JSON Web Services, that external client applications can invoke via HTTP to access different types of resources located in the portal, including the documents stored in the Documents and Media Library. To access any protected resource, a portal user must be authenticated through Basic or Digest HTTP authentication. Therefore, the user's credentials must be shared with the client, causing several problems and limitations:

- The client application is required to store the user's credentials for future use, typically a password in clear-text.
- The client application gains overly broad access to the user's protected resources, without any restriction in the duration or access to a limited subset of resources.

- A server administrator cannot revoke access to an individual client without revoking access to all clients, and must do so by changing the user's password.
- Compromise of any client application results in compromise of the end-user's password and all of the data protected by that password, including access to other services (e.g., online banking) for which the same password was used.

To address these issues, providing more flexibility and control, user authentication is coupled with the OAuth 2.0 authorization framework, an authorization layer that separates the role of the client from that of the user. In particular, as specified in RFC 6749,

> *In OAuth, the client requests access to resources controlled by the resource owner (i.e., the user) and hosted by the resource server (i.e., the portal), and is issued a different set of credentials than those of the resource owner.*
>
> *Instead of using the resource owner's credentials to access protected resources, the client obtains an access token - a string denoting a specific scope, lifetime, and other access attributes. Access tokens are issued to clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server.*

OAuth makes it possible to separate the assignment of API security credentials from the process of authenticating users to a web site or a web portal, eliminating the need to store a password on a mobile device. Unlike the user credentials, an OAuth token is a unique identifier tied to a particular application and device, and can be issued with a restricted scope and limited lifetime. An OAuth token is also hard to guess, so it's not subject to a dictionary attack, and it is hard to share. OAuth allows the server to revoke tokens for an individual user, for an entire app, without requiring the user to change their original password and without affecting other devices and apps. This is critical if a mobile device is compromised or if a rogue app is discovered.

4.2.1 The OAuth Provider

The OAuth framework is flexible enough to be integrated with any existing API. In fact, in a joint work with this thesis, Cavallin [37] has developed an OAuth 2.0 provider as a portlet application for Liferay. This OAuth provider exposes a new set of JSON APIs (discussed in section 5.5 wrapping the existent JSON APIs of the Documents and Media Library services. The provider (in Figure 4-3) acts both as an *authorization server* - issuing access tokens and refresh tokens to the client after successfully authenticating the resource owner and obtaining authorization - and a *resource server* - capable of accepting and responding to protected resource requests using access tokens. A resource can be either a JSON object describing a document or a folder, a JSON object representing an update event regarding a document or a folder, or a stream of the content of a document (i.e., the file).

The OAuth 2.0 flow illustrated in Figure 4-3 describes the abstract interaction between the client and the OAuth provider, which is actually based on the "Authorization Code

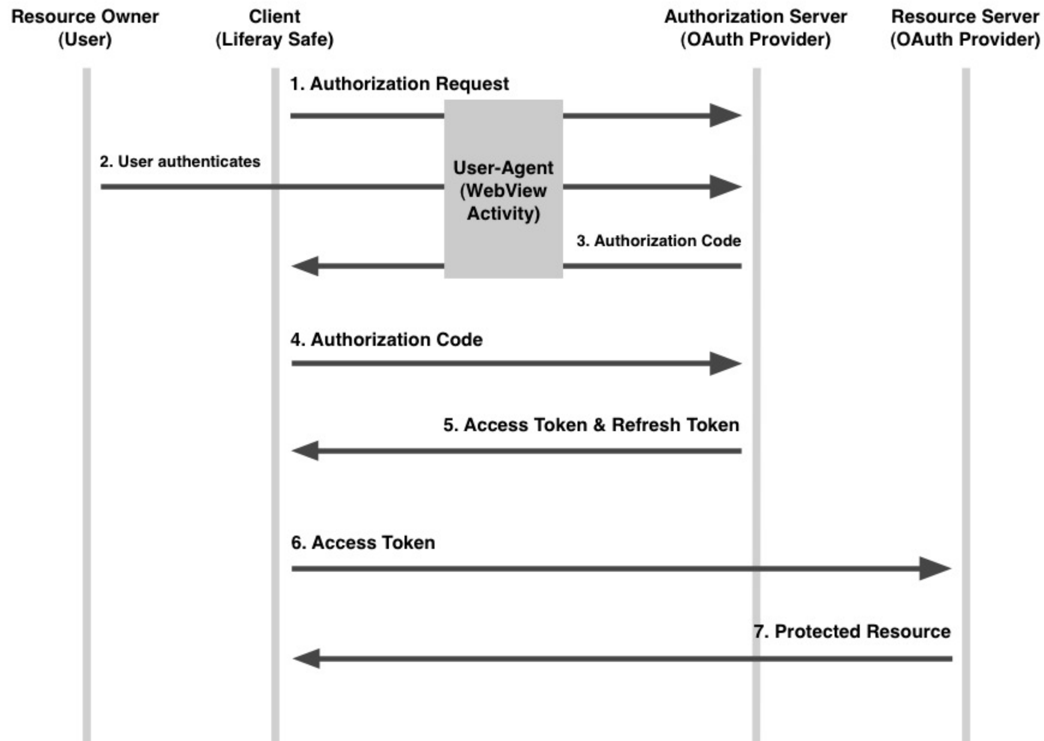


Figure 4-3: OAuth 2.0 Protocol Flow

Grant" flow specified in RFC 6749. Assuming the resource owner (i.e., the user) grants the client's access request, the flow includes the following steps:

1. The client requests authorization from the resource owner. The authorization request is made indirectly to the resource owner using the authorization server as an intermediary. The client therefore directs the resource owner's user-agent to the authorization server.

2. The authorization server authenticates the resource owner (via the user-agent) and obtains authorization.

3. The authorization server redirects the user-agent back to the client with an authorization code, which is a credential representing the resource owner's authorization.

4. The client requests both an access token and a refresh token to the authorization server by including the authorization code received in the previous step.

5. The authorization server verifies the client's identity and validates the authorization code, and if valid, issues both an access token and a refresh token.

6. The client requests a protected resource from the resource server (e.g., invoking the JSON API) and authenticates by presenting the access token.

7. The resource server validates the access token, and if valid, serves the requests.

The user-agent participating in the authorization flow is a WebView embedded in an activity component of the client. The user-agent is necessary for interacting with the resource owner and for receiving incoming requests (via redirection) from the authorization server. The client application obtains the response from the authorization server by communicating with the embedded user-agent through an instance of WebViewClient, which enables monitoring state changes emitted during resource load, and accessing the user-agent's cookie storage. Because the resource owner only authenticates with the authorization server through the user-agent, the resource owner's credentials are never shared with the client.

Included with the access token, the authorization server issues also a refresh token. Refresh tokens are credentials used to obtain a new access token when the current access token becomes invalid or expires. Because a refresh token represents the authorization granted to the client by the resource owner, the flow for refreshing an expired access token does not require the resource owner either to authenticate with the authorization server, or to grant the refresh request. The utility of the refresh token will be discussed in section 6.5.

Finally, the OAuth 2.0 framework explicitly specifies that any access token or refresh token must be kept confidential in transit and in storage. While the confidentiality of these tokens is preserved by TLS during transit (as also specified in RFC 6749), safely storing the access token in a mobile device is not a straightforward operation. In section 6.5 are explained the importance of the access token in the client application and the way this is safely retained until its expiration.

In this section was given only the big picture of the functioning of OAuth in this specific client-server architecture, for a more rigorous and complete discussion please refer to Cavallin [37].

Chapter 5

File synchronization

One of the capabilities that most prominently characterizes the paradigm of mobile computing is the possibility of working with a variety of content (such as documents and media) while being “on the go”; that is, accessing shared and private files anytime and everywhere directly from a mobile device. Indeed, this is a very common scenario in the ecosystem of mobile applications, since many of these applications wouldn't be much interesting without being connected in some way to content repositories or data services on the cloud.

In the design and implementation of the Liferay Safe client, we face directly the problem of file synchronization; more specifically, files are added to a Liferay Portal server application, through the *“Documents and Media Library”* portlet, and synchronized with client applications installed on a diverse set of mobile devices running Android. The synchronization is *two-way*, since updated files are copied in both directions - and across many clients, as the same user can sync multiple devices of his own - with the purpose of keeping the two locations identical to each other (i.e., synced).

However, dealing with any kind of data synchronization in a mobile context is way different than dealing with it in a traditional (desktop) computing environment. In fact, keeping local and remote files synced requires to pose attention to many aspects that are exclusive to the world of mobile computing. The most relevant of these derive from considering some characteristics about the networks, the hardware, and the operating system of a mobile device, including, but not limited to, the following:

- Prolonged or intermittent, network connection unavailability;
- Cost of cellular data connection;
- Limited hardware resources - processor, memory and storage - and network bandwidth;
- Multi-tasking: resources and data connection are shared among multiple concurrently running applications;
- Limited battery power time;
- Implications on software design imposed by the underlying software platform (i.e., Android);

- Lack of a desktop environment: the user interface for file operations must be built from scratch as part of the application's user interface;

- Increased data security risks (wrt. traditional desktop environments).

These aspects, all together, imply that a mobile client for file synchronization should track and persist all file modifications even when there is no connectivity, and automatically sync in the background these changes, with the server counterpart on the cloud, as soon as the connectivity returns - while consistently detecting and managing possible conflicts. Actually, due to the limited storage space, not all files can be automatically downloaded and synchronized, instead only the one's demanded by the user should be. Therefore, the client has to manage a size-adjustable cache for user's favorite and files uploaded from local - to be kept in sync. Additionally, the user should still be able to view a list of all files in the synced repository and browse any folders. Thus, the client has to keep in sync the logical description of every file (i.e., UUID, title, extension, UUID of parent folder, etc.) and locally persist this entry along with a value indicating the synchronization state of the corresponding file's content. When all file entries are retrieved, the full repository structure can be represented and automatically be kept in sync with its evolution over time - without needing to download the content of all files. Finally, the client sync process should be optimized to have the least resource footprint possible, in order to reduce battery consumption and efficiently coexist alongside other applications, with which has to coordinate to responsibly use the available connection bandwidth and perform the needed sync operations.

In short, a mobile client for file synchronization should include at least the following features:

- Automatic synchronization of file entries and cached file contents;

- Detection and tracking of file operations (edit, rename, move, delete);

- Conflict detection;

- Selective download of remote files;

- Import and upload of local files;

- New folder creation;

- Encryption for security of private files, at both transport and storage levels.

In this chapter, we will explore a particular synchronization pattern leveraged by Android, called the SyncAdapter, while also illustrating the criterium of synchronization and its algorithmic implementation. Besides, we will define how the entry of a file is represented across various states and locally persisted, and finally, discuss how downloads and uploads are performed. Later in the next chapter, we will unravel the cache architecture and propose an approach for security of private files.

5.1 SyncAdapter pattern

The Android platform provides, through its component framework, several patterns for implementing a client to synchronize data with HTTP Web Services. In this section, we focus on the most interesting of these patterns, the one using both the ContentProvider API and a SyncAdapter, simply called the SyncAdapter pattern. This pattern was first introduced by Google engineer Virgil Dobjanschi in a session during Google IO 2010 [38]. Of the patterns presented by him, the SyncAdapter was the most interesting due to its level of integration with the Android system.

Through the SyncAdapter, an application can register with the system to perform any synchronization work with a remote server, delegating to the system itself all the burden of managing eventual sync errors and consequent reattempts. In fact, the difference from a pattern not using the SyncAdapter is that with the latter, the available resources and connection bandwidth are efficiently shared among all applications registered with the system through this mechanism. That is, the system will automatically listen for sync requests, enqueue them, and subsequently perform them as soon as connectivity is available. This ensures that any synchronization would not interfere with other running syncs or applications using data connection, while minimizing the network usage. If a synchronization could not be started or successfully completed due to an error, the system will automatically arrange a new attempt - an exponential backoff algorithm will set a priority in the work queue for the reattempt, based on the gathered error information. Moreover, with the ContentProvider API, synchronized data can be neatly persisted and conveniently updated to the User Interface by means of the ContentResolver with small effort. Therefore, the developer can relief from all this dull work, yet keep the code cleaner and be assured that her application is complying with the limitations of the underlying mobile platforms which we discussed before.

Despite Engr. Dobjanschi during his talk hasn't displayed any demo application nor has provided any example code - actually a sample SyncAdapter for the Contacts Provider was later released along the SDK - he illustrated the pattern as clearly as needed to be correctly implemented and customized by any experienced developer. The pattern is schematized in Figure 5-1 below.

In this pattern, the SyncAdapter is the central component responsible for performing the synchronization operations. It relies on the ContentProvider to retrieve all the items that need to be synced (1.), this is the preliminary step of every sync task necessary to send to the server any local pending updates and make the synchronization two-way. After all items are retrieved, the sync algorithm can be started from the SyncAdapter's onPerformSync() method (2.), which was originally called by the system or from another component by means of the ContentResolver. During this phase all communication with the server's Web Services API is performed by the HTTP client via HTTP request methods (3.). Local updates are pushed to the server (POST/PUT/DELETE) and remote modifications are retrieved (GET). All retrieved data must be unmarshaled, compared with existing data to detect eventual conflicts, and finally persisted (4. and 5.). A Processor component should take these responsibilities and interact with the ContentProvider's API (insert, update,

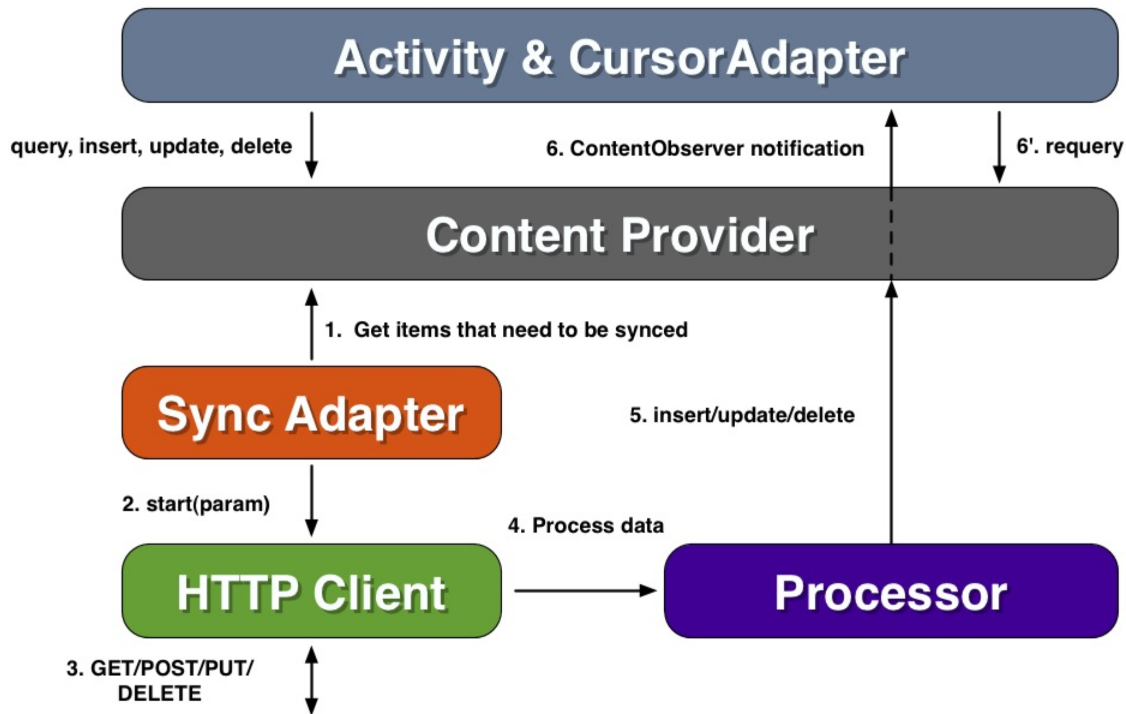


Figure 5-1: Android SyncAdapter pattern

delete). Any Activity can rely on a CursorAdapter to update its views with new synced data. A ContentObserver object will receive callbacks from the ContentProvider whenever its content is changed, and notify the CursorAdapter (6.) which can then requery the dataset (6'.).

In pills, the SyncAdapter pattern allows to decouple the synchronization logic from activities and execute background operations which are not time critical, while minimizing the network usage. Additionally, by relying on the ContentProvider, data can be persisted early and often in a convenient way, thus keeping the application functionality sound wrt. its lifecycle - an application (or one of its components) can be terminated by the user or the system anytime.

5.2 Implementation

Implementing the SyncAdapter pattern might still get a little bit challenging despite its simplicity, as there are not many examples and documentation explaining how the concept works. Fortunately, there are a few articles and the SampleSyncAdapter included in the SDK providing some examples useful to set off [39, 40, 41, 42]. So let's start illustrating the implementation steps and details of the SyncAdapter for Liferay Safe, which is schematized in Figure 5-2 below.

Simply put, the SyncAdapter is just a service that is started by the Sync Manager, which

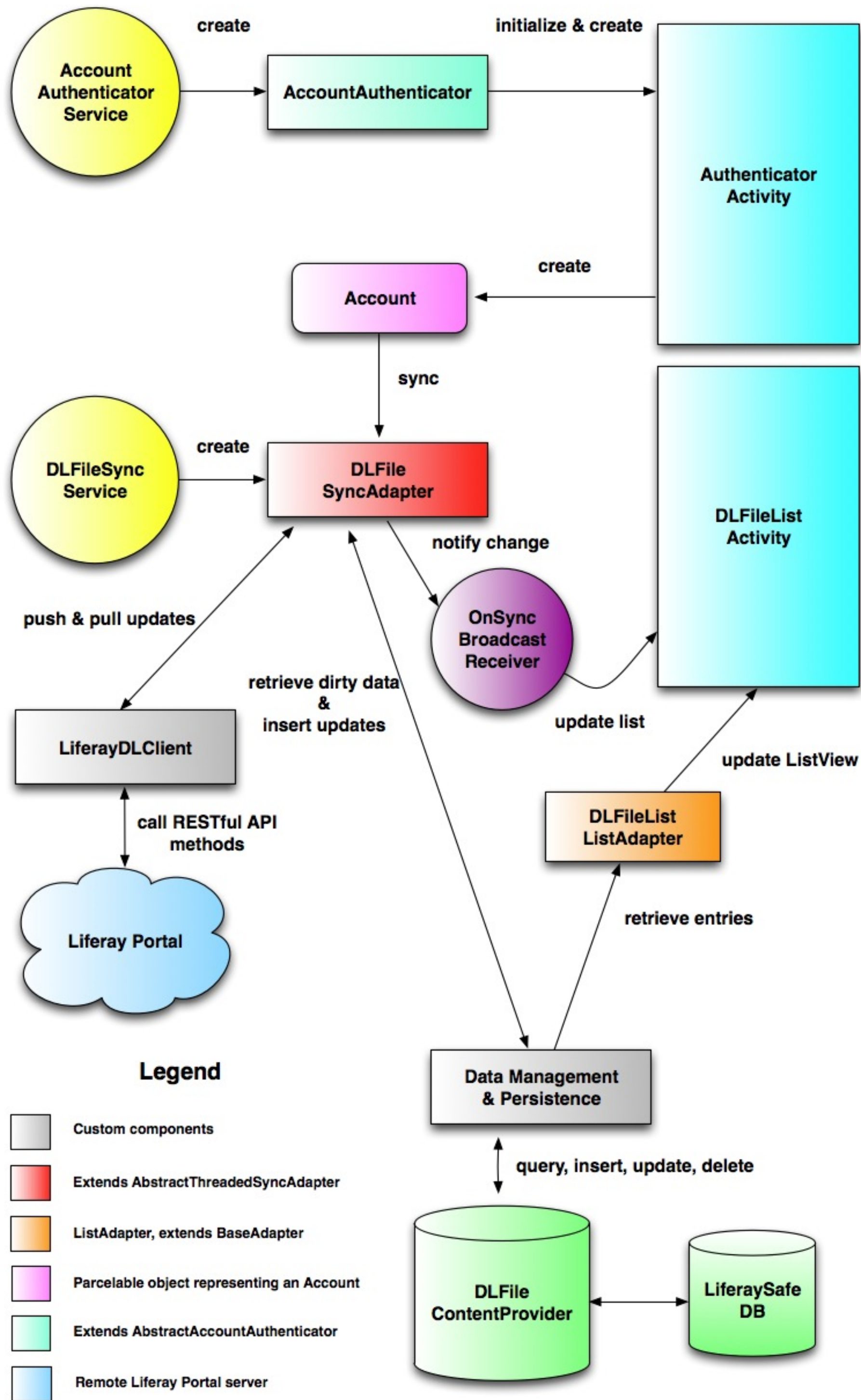


Figure 5-2: Implementation of the SyncAdapter pattern

in turn maintains a queue of SyncAdapters and is responsible for choosing when to perform the sync, and for scheduling resyncs according to the errors reported by the SyncAdapter.

The basic idea is that remote data is mirrored in a local database. Activities can access this data through the ContentProvider and display it to the user - for example, with a ListView and a ListAdapter - allowing him to perform various operations. The SyncAdapter will be in charge of making the remote and local data match; it will push the local changes to the server and fetch the new data.

There are several components involved in the implementation of this pattern. In general, one first needs to write its own implementation of the components, then has to glue all the components together using the AndroidManifest file, and finally make the SyncAdapter interact with the application. Hereafter are described all the steps involved in the implementation, including a description of all the necessary components.

5.2.1 Account creation

In order to use a SyncAdapter an Account is needed. On Android an Account can be exactly any kind of account a user owns to access some online service. It is essentially a pair of strings; one is the username identifying the user on the remote server, and the other is the type distinguishing it from the others available on the system. Different types of online user accounts are organized and stored in a centralized system registry, the component responsible for providing access to this registry is the AccountManager.

Applications interact with the AccountManager to add or retrieve particular types of accounts they are interested in. Since different online services have different ways of handling accounts and authentication, the AccountManager uses pluggable authenticator modules - extending AbstractAccountAuthenticator - for different account types. Authenticators handle the actual details of validating account credentials and storing account information.

To interact with the user to prompt for credentials, present options, or ask the user to add an account, an activity extending AccountAuthenticatorActivity is needed. This class provides the basic implementation necessary to handle requests for and pass responses to an authenticator. A user can also add, modify or remove her accounts from the "Accounts & sync" section of the "Settings" application.

When an account has to be created, an Intent is sent to the system specifying the type of account requested. The system will look up for the authenticator associated exactly with that type (exposed through the AndroidManifest file) and start a service from which the authenticator itself can be instantiated and binded. Next the authenticator's addAccount() method can be called and subsequently the AccountAuthenticatorActivity will be started with an Intent, passing a response and other account related values. The activity will be responsible to collect the user's credentials and perform authentication with server. If the credentials are correct, the last step is to call the AccountManager's addAccountExplicitly() method and effectively add the account by passing in the new Account object and the password. Other user data can be associated to account and stored in the AccountManager with a key by calling its setUserData() method. This is useful for example to store the server's host URL. Finally the activity has to call setAccountAuthenticatorResult() passing

the response back to the authenticator with the name and type of the account that was added, or an error code and message if an error occurred.

Security Remark

Account data protection is based on the Linux user id (UID) of the process making the request. Each account is associated with an authenticator (that has a UID), and the process calling getPassword() (or several other methods) must have the same UID as the authenticator. This prevents extraneous applications to steal the user's credentials from the AccountManager.

Still, it is NOT safe to store in the AccountManager user's passwords and other sensitive data. In fact, all data is stored "as is" (and so eventually in plain text) in a SQLite database located in /data/system. Thus, after gaining root access on the device (which is often extremely simple), a malicious user can easily access these system files via Android's adb and retrieve all stored account info [43].

For the record, in August 2010 was reported an issue on AOSP stating: "The password for email accounts is stored into the SQLite DB which in turn stores it on the phone's file system in plain text." [44]. The issue was closed one year later proposing full-disk encryption as the "most appropriate solution", starting with Honeycomb. More recent comments in fact still confirm the database is stored in plain text.

An alternative and safer approach would be to store user's sensitive information through the Android KeyStore API. This provides access to a credential storage encrypted using an AES 128 bit master key, which in turn is derived from the device unlock password or PIN. This means that secrets cannot be extracted even after gaining root access, unless the password is known. However, the master encryption key is not tied to the device, so it's possible to copy the encrypted key files and perform a brute force attack on more powerful machine(s) [45].

Many servers support some notion of an authentication token, which can be used to authenticate a request to the server without sending the user's actual password. AccountManager can generate auth tokens for applications, so the application doesn't need to handle passwords directly. In Liferay Safe this functionality is not used. As discussed earlier authentication tokens are handled independently by the application and in a different way wrt the AccountManager, in order to mitigate related security risks. For the same security reasons, the user's password is never stored in the device (a dummy string will be passed when calling addAccountExplicitly() on the AccountManager); therefore, the user will be prompted for credentials whenever the authentication token has expired.

The Account is an important piece of the SyncAdapter pattern, because it ties the SyncAdapter and the ContentProvider together. In fact, *"AccountManager, SyncAdapter and ContentProvider go together. You cannot use an AccountManager without a SyncAdapter. You cannot use a SyncAdapter without an AccountManager. You cannot have a SyncAdapter without a ContentProvider"*[46].

5.2.2 SyncAdapter implementation

In order to write a SyncAdapter one must extend the AbstractThreadedSyncAdapter class, provide implementations for the abstract onPerformSync() method and write a service that returns the result of getSyncAdapterBinder() in the service's onBind() when invoked with an intent with action android.content.SyncAdapter.

When a requestSync() is received (from the SyncManager), a thread will be started to run the operation and onPerformSync() will be invoked on that thread.

- If a sync operation is already in progress, then an error will be returned to the new request and the existing request will be allowed to continue.
- If a cancelSync() is received that matches an existing sync operation, then the thread that is running that sync operation will be interrupted.

All the synchronization logic resides in the body of the onPerformSync() method, including communication with the server and interaction with the ContentProvider for data persistence. There is no one standard way for sync logic, as it depends strongly on the server's API.

Instead, setting up a service that filters SyncAdapter intents is relatively straightforward.

Listing 5.1: DLSyncService

```
1  public class DLFileSyncService extends Service {
2          private static final Object sSyncAdapterLock = new Object();
3          private static SyncAdapter sSyncAdapter = null;
4          @Override
5          public void onCreate() {
6          synchronized (sSyncAdapterLock) {
7                  if (sSyncAdapter == null) {
8                  sSyncAdapter = new DLFileSyncAdapter(getApplicationContext(), true);
9                  }
10         }
11         }
12         @Override
13         public IBinder onBind(Intent intent) {
14                 return sSyncAdapter.getSyncAdapterBinder();
15         }
16 }
```

The service definition given above must be registered in the AndroidManifest file and specify the following intent filter and metadata tags:

Listing 5.2: Manifest service tag

```
<service
        android:name=".syncadapter.DLFileSyncService"
        android:exported="true" >
<intent-filter>
            <action android:name="android.content.SyncAdapter" />
</intent-filter>

<meta-data
             android:name="android.content.SyncAdapter"
             android:resource="@xml/syncadapter" />
</service>
```

Where the android:resource attribute points to the following resource in the res/xml folder

Listing 5.3: Manifest sync-adapter tag

```
<sync-adapter xmlns:android="http://schemas.android.com/apk/res/android"
    android:contentAuthority="com.liferay.safe"
    android:accountType="liferaysafe"
    android:userVisible="false"
    android:supportsUploading="true"
    android:allowParallelSyncs="false"
 />
```

The android:contentAuthority and the android:accountType property are particularly important because they tie the SyncAdapter respectively with the ContentProvider and the account type. In fact, the value of the first property should match the android:authorities property value of the ⟨provider /⟩ resource declared in the AndroidManifest file, while the value of the second should be exactly the one declared in the ⟨account-authenticator /⟩ resource located in res/xml. As a matter of fact, the instances of the Account and the ContentProvider are passed as parameters to the onPerfomSync().

As for the remaining properties of the ⟨sync-adapter /⟩ resource,

- android:userVisible="false" specifies that the SyncAdapter should not show up in the "Accounts & sync settings" screen;

- android:supportsUploading="true" enables an upload-only sync to be requested whenever the associated authority's ContentProvider does a notifyChange() with syncToNetwork set to true;

- android:allowParallelSyncs="false" indicates that the SyncAdapter cannot handle syncs for multiple accounts at the same time.

Also note that the service's android:exported is set to true, this means that other applications can invoke the service and interact with it. Although it might be interpreted as a security risk, this is necessary in order to receive sync requests with the intent filter.

An ulterior parameter of onPerformSync() which is of particular interest, is the SyncResult. This object is used to communicate the results of a sync operation to the SyncManager. Based on the values in the SyncResult, the SyncManager can determine - by means of an exponential backoff algorithm - whether to reschedule the sync in the future or not. More specifically, via the SyncResult, the SyncManager evaluates also the SyncStats object, which records various statistics about the result of a sync operation, including, and not limited to:

- authentication exceptions occurring when authenticating the Account specified in the sync request (numAuthExceptions);
- IO exceptions related to network connectivity or timeout while waiting for a network response (numIoExceptions);
- exceptions while processing the data received from the server, usually due to malformed or corrupted data (numParseExceptions).

Using the SyncResult object in a clever manner will allow the SyncManager to be more effective.

A SyncAdapter can be started manually:

Listing 5.4: SyncAdapter manual start

```
1 Bundle bundle = new Bundle();
2 bundle.putBoolean(ContentResolver.SYNC_EXTRAS_MANUAL, true);
3
4 ContentResolver.requestSync(account, "com.liferay.safe", bundle);
```

Or alternatively, it can be scheduled to start periodically, for example every hour:

Listing 5.5: SyncAdapter periodic start

```
1 Bundle params = new Bundle();
2 params.putBoolean(ContentResolver.SYNC_EXTRAS_EXPEDITED, false);
3 params.putBoolean(ContentResolver.SYNC_EXTRAS_DO_NOT_RETRY, false);
4 params.putBoolean(ContentResolver.SYNC_EXTRAS_MANUAL, false);
5
```

```
6 ContentResolver.addPeriodicSync(account, "com.liferay.safe", params, 3600);
7 ContentResolver.setSyncAutomatically(account, "com.liferay.safe", true);
```

Finally, any active or pending syncs that match an account and an authority (or any of them) can be canceled:

Listing 5.6: SyncAdapter cancel sync

```
1 ContentResolver.cancelSync(account, "com.liferay.safe");
```

5.2.3 UI notification

If a synchronization is performed when the related application is running, the user should be notified of the affected changes and the UI should be updated. There are several ways to carry out this task, the most convenient of them involves using a CursorAdapter and ContentObserver notifications.

Whenever data is inserted, updated or deleted, clients should be notified about these changes in the underlying datastore of the ContentProvider. This is accomplished by calling notifyChange() on the ContentResolver, specifying the URI of the content that was changed.

Listing 5.7: ContentResolver notifyChange()

```
1 getContext().getContentResolver().notifyChange(uri, null);
```

To receive these notifications, Android provides the ContentObserver class following the object-observer pattern, which can be implemented as a subclass, and registered to the ContentResolver to listen for changes [47].

Whenever a change occurs, the onChange() method is called, therefore the implementation has to override this method and perform there all the logic necessary to update the UI. To avoid executing this code on the UI thread, it's recommended to create the handler for the ContentObserver on a separate thread, for example using an AsyncTask.

Then, registering a ContentObserver is as simple as calling the ContentResolver's registerContentObserver() method:

Listing 5.8: Registering the ContentObserver

```
1 getContentResolver().registerContentObserver(SOME_URI, true, yourObserver);
```

Where the second parameter specifies if changes to URIs beginning with SOME_URI (descendents) will also cause notifications to be sent. Content URIs can be directory-based or id-based. The first one is more appropriate when needing to update a list of data, while the second should be used for updating a detail screen. However, the choice of the URI depends on how the ContentProvider was implemented.

A ContentObserver can be used almost by any component, but it turns out very useful with a CursorAdapter. A CursorAdapter is an adapter that exposes data from a Cursor (returned from a ContentProvider) to a ListView. When using a CursorAdapter, an activity can register a ContentObserver to receive notifications and call the adapter's changeCursor() to update a ListView whenever the underlying data returned by the Cursor has changed.

Listing 5.9: Custom ContentObserver

```
1  class MyObserver extends ContentObserver {
2      public MyObserver(Handler handler) {
3          super(handler);
4      }
5      @Override
6      public void onChange(boolean selfChange, Uri uri) {
7          Cursor myCursor = managedQuery(uri, projection, where, whereArgs, sortBy);
8          myAdapter.changeCursor(myCursor);
9      }
10 }
```

Note that the original pattern suggested calling requery() on the Cursor, and hence the ContentObserver could have been registered directly on the Cursor itself calling register-ContentObserver(). However, requery() was deprecated suggesting infact to *"Just request a new cursor, so you can do this asynchronously and update your list view once the new cursor comes back."*

This mechanism also provides an effective model to show to the user the progress of a synchronization. In fact, by storing a column in each record in the ContentProvider indicating the sync state of that record, it's possible to have a per-row granularity for information, allowing, for example, to have an independent spinner for each item in the ListView, telling the user exactly what data is being synced.

Unfortunately there are two main downsides when using content observers, one is that for directory-based URIs it's not possible to get a list of id's that have changed, which can be a problem for big datasets if only a few records have changed, second, it's often difficult to reduce the number of notifications of bulk operations.

There is one more issue about UI notification which is more closely related to the SyncAdapter, namely that, at the state of the art, there is no sound way to get notified when the overall sync process has started or ended. This is a major problem especially when the SyncAdapter is started periodically.

Scrolling through the ContentResolver's API we find the addStatusChangeListener (int mask, SyncStatusObserver callback) method. By calling this method a component can request notifications when different aspects of the SyncManager change, such as when a sync is active, pending or the sync settings have been changed. The status change will cause the onStatusChanged() method of the SyncStatusObserver object to be invoked. An example can be found here [48] and here [49].

However, this utility is pretty useless for a SyncAdapter, since addStatusChangeListener() notifies the callback object when the sync status of ANY SyncAdapter changes, not a particular sync identified by an account + authority combination [50, 51]. This would cause too many refresh apart from the fact that it would not be possible know what is being synced nor distinguish the start of the sync event from the end.

An alternative, but dirty, approach would be to spawn a thread which continuously checks the result of isSyncActive() and isSyncPending() from the ContentResolver. These methods return true if there is currently a sync operation for the given account or authority actively being processed, or in the pending list.

In Liferay Safe a different approach was adopted, which consists in using a BroadcastReceiver, and a ListAdapter (instead of a CursorAdapter) to populate the ListView. In the activity managing the ListView, we register the broadcast receiver when onResume() is called:

Listing 5.10: Registering an OnSyncBroadcastReceiver

```
IntentFilter syncIntentFilter = new IntentFilter(DLFileSyncService.SYNC_MESSAGE);
mSyncBroadcastReceiver = new OnSyncBroadcastReceiver();
LocalBroadcastManager.getInstance(this).registerReceiver(mSyncBroadcastReceiver,
syncIntentFilter);
```

And unregister it in onPause() to avoid memory leaks:

Listing 5.11: Unregistering an OnSyncBroadcastReceiver

```
if (mSyncBroadcastReceiver != null) {
LocalBroadcastManager.getInstance(this).
unregisterReceiver(mSyncBroadcastReceiver);
        mSyncBroadcastReceiver = null;
}
```

The BroadcastReceiver is defined as a private inner class of the activity:

Listing 5.12: OnSyncBroadcastReceiver

```
private class OnSyncBroadcastReceiver extends BroadcastReceiver {

@Override public void onReceive(Context context, Intent intent) {
        boolean inProgress =
                        intent.getBooleanExtra( DLFileSyncService.IN_PROGRESS, false);

        String accountName = intent .getStringExtra(DLFileSyncService.ACCOUNT_NAME);

        if (accountName.equals(AccountUtils.getCurrentLiferayAccount(context).name)) {

                String synchFolderRemotePath =
                        intent.getStringExtra(DLFileSyncService.SYNC_FOLDER_REMOTE_PATH);

                boolean fillBlankRoot = false;
                if (mCurrentFolder == null) {
                        mCurrentFolder = getDLFileManager().getDLFile(PathHelper.PATH_ROOT);
                        fillBlankRoot = (mCurrentFolder != null);
                }
                if ((synchFolderRemotePath != null
                        && mCurrentFolder != null
                        && (PathHelper.fixPath(mCurrentFolder.getFilePath())
                                                        .equals(synchFolderRemotePath)))
                    || fillBlankRoot ) {
                        if (!fillBlankRoot)
                                mCurrentFolder = getDLFileManager().getDLFile(synchFolderRemotePath);

                        DLFileListFragment fileListFragment =
                                        (DLFileListFragment) getSupportFragmentManager()
                                                .findFragmentById(R.id.fileList);
                        if (fileListFragment != null) {
                                        fileListFragment.listFolder(mCurrentFolder);
                        }
                }
                setSupportProgressBarIndeterminateVisibility(inProgress);
        }
 }

}
```

When a broadcast intent is received, the receiver checks the progress status and the account name, sent from the SyncAdapter in the intent's extras, and sets a progress indicator in the activity. When the synchronization is notified as completed, the receiver will invoke the update of the ListView.

The SyncAdapter sends a broadcast intent both at the beginning and at the end of the

sync operation. Additionally, the SyncAdapter can also notify the activity when a folder has been updated, by setting its path as an extra in the intent. This allows to refresh the view displaying the contents of that folder, when visible to the user, avoiding to wait the full sync event to complete.

Listing 5.13: SyncAdapter UI notification local broadcast

```
1  private void notifyUI(boolean inProgress, String dirRemotePath) {
2          Intent i = new Intent(getContext(), DLFileListActivity.class);
3          i.setAction(DLFileSyncService.SYNC_MESSAGE);
4          i.putExtra(DLFileSyncService.IN_PROGRESS, inProgress);
5          i.putExtra(DLFileSyncService.ACCOUNT_NAME, getAccount().name);
6          if (dirRemotePath != null) {
7                  i.putExtra(DLFileSyncService.SYNC_FOLDER_REMOTE_PATH, dirRemotePath);
8          }
9          LocalBroadcastManager.getInstance(getContext()).sendBroadcast(i);
10 }
```

By using the LocalBroadcastManger to publish and subscribe for broadcasts, we avoid all the security issues related to global broadcasts, such as leaking private data or receiving the same broadcasts from other applications [52]. Besides, since local broadcast intents never go outside of the current process, the communication is also more efficient [53].

Finally, instead of using Sticky Broadcasts (which are insecure [54]), we can still detect a periodic sync that was started when the application was paused or closed, by calling the ContentResolver's isSyncActive() and eventually update the UI. The application will still register for sync broadcast and be notified by the SyncAdapter when the sync operation has finished.

5.2.4 Required permissions

The final step for implementing the SyncAdapter pattern is to declare in the AndroidManifest file a few permissions to allow several aspects of the application to function appropriately: ability to read/write accounts, ability to interact with the network, and ability to read/write sync settings.

Here is a snippet from Liferay Safe's AndroidManifest.xml:

Listing 5.14: SyncAdapter pattern permissions

```
1 <uses-permission android:name="android.permission.INTERNET" />
2 <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
3
4 <uses-permission android:name="android.permission.GET_ACCOUNTS" />
```

```
5  <uses-permission android:name="android.permission.USE_CREDENTIALS" />
6  <uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />
7  <uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />
8
9  <uses-permission android:name="android.permission.READ_SYNC_STATS" />
10 <uses-permission android:name="android.permission.READ_SYNC_SETTINGS" />
11 <uses-permission android:name="android.permission.WRITE_SYNC_SETTINGS" />
```

In the next sections we will focus on the synchronization logic and we'll cover the remaining building blocks of the SyncAdapter pattern: the Data Management & Persistence layer providing access to the ContentProvider, and the HTTP client executing the methods of the Web Services API (LiferayDLClient).

5.3 Synchronization algorithm

A preliminary step necessary for designing the logic of a synchronization client is to analyze the sync policy enforced by the server. For sync policy are intended all the rules that define the behaviour of the client when performing two-way synchronization of operations carried on syncable items.

In Liferay Safe there are two main classes of syncable items, user sites (UserSite) and file entries (FileEntry). A user site represents a site available in Liferay Portal that is identified by a "groupId". File entries are records of the "Document and Media Library" (DML) that describe file contents stored in a user site's repository or directories defined in the DML. A file entry is identified by a "fileEntryId" and is tied exactly to a user site with a "repositoryId" that corresponds to the user site's "groupId", whereas it is tied to a file content by means of the file's "uuid". A file content indicates precisely a file stored in the server's Content Repository; we may call it simply a file, a document, a media or a content.

A change in a file entry can give rise to three types of synchronization events: "add", "update" or "delete". Synchronization events subsequent to a given timestamp are pulled together from the "Document and Media Library" as a collection of document library sync entries (DLSyncUpdate). A document library sync entry (DLSync) is tied to a user site by a "repositoryId" and to a file entry by a "fileId".

The responsibilities of a client when carrying out these synchronization events depend on the role of the server. Two main issues for a file synchronizer are conflict resolution and file consistency on one hand, and controlling user permissions on the other. Fortunately the responsibilities of Liferay Safe are simplified for the fact that Liferay Portal provides a centralized control of permissions on user operations and a centralized versioned repository.

This practically means that Liferay Safe in most cases can delegate to the server the burden of managing synchronization conflicts. In fact, the DML keeps track of every version of a file entry and its file content; therefore all clients, when synchronized, will always have the latest version of that entry. This means that if a client misses some updates and uploads a new modification before syncing, the server will accept the modification as the

latest version, without creating any conflict. Other clients can still retrieve the previous versions and the user can take care by himself of merging the modifications. Similarly, if a new file is uploaded and has the same full path of an existing one, it will be treated as a new version of that existing file without raising a conflict.

Being Liferay Portal a collaborative platform where more than one person can be working on a file at the same time, "editing conflicts" may occur very often. File versioning allows to keep contents consistent across many clients and more importantly it ensures that newer modifications are never lost by overwrites - while eliminating the clutter of file duplications due to unresolvable conflicts.

In Liferay, contents of the DML can be accessed also via WebDAV, which can be used by applications to write directly on remote files as if they were local. Often, applications use file locking to protect files that are in use. But, having a document open in any application that lock its files can increase the likelihood of conflicting edits taking place. Any changes made in an application that has a file locked will create inevitably a conflict. In this case, the client uploading changes (not holding the lock) will have to abort the upload and create a local duplicate of the file to be sorted out by the user.

For security implications, instead, it's not possible to completely rely on a centralized user permission control. The reason is straightforward: the client needs to know if user's permissions have changed after a file content has already been downloaded. However, since Android devices have a very limited internal storage, most of the times application data has to be stored in the external SD storage, but there, all contents are world-readable because the file system lacks of user permissions. Thus, at this point it is impossible for the client to effectively enforce user's permissions defined on the server, not least it would be worthless.

We postpone to the next section the discussion of a trade-off approach to this problem, when we'll introduce the concept of private file content. For now, let's define a private file content as a file content whose file entry is marked as confidential.

To respond correctly to synchronization events the following set of rules is defined, relying on the server's policy, which apply to all file entries.

- Local updates and uploads must be pushed to the server immediately (if possible) or always before pulling remote updates;

- Contents should be downloaded ("cached") only upon user request; only the updates of contents whose entries are marked as "keep in sync" must be downloaded automatically;

- Local entries must track and persist the synchronization state of their associated cacheable content;

- Remote deletions must be applied immediately, whatever is the file entry's sync state;

- Local deletions are voluntarily not supported in Liferay Safe. The user is only allowed to remove a file content from the local cache storage.

Based on the synchronization policy defined above, the sync mechanism can now be explained in algorithmic form, which should be implemented in the SyncAdapter's onPerformSync() method.

Listing 5.15: Synchronization algorithm: main

```
1  Notify the UI that the sync event has started
2  Request "get-user-sites", update local user site entries and notify the UI
3  For each UserSite site
4         If site.lastAccessDate = 0
5                GetFolders(site.groupId, 0)
6                GetFileEntries(site.groupId, 0)
7                 Assign 1 to site.lastAccessDate
8                Jump to 23.
9         Get local entries marked as "pending upload"
10        For each "pending upload" entry
11               If entry.fileId = -1
12                      Request "add-file-entry"
13               Else
14                      Request "update-file-entry"
15        Request "get-dl-sync-update" and store result in DLSyncUpdate remoteUpdates
16        For DLSync remoteEntry in remoteUpdates and remoteEntry.type = "folder"
17               HandleDLSync(remoteEntry, site.groupId)
18        For DLSync remoteEntry in remoteUpdates and remoteEntry.type = "file"
19               HandleDLSync(remoteEntry, site.groupId)
20        Assign remoteUpdates.lastAccessDate to site.lastAccessDate
21        Notify the UI that site is synced
22 Get local entries marked as "pending download" and start the Downloader service
23 Notify the UI that the sync event has completed
```

Listing 5.16: Synchronization algorithm: GetFolders

```
1  GetFolders(repositoryId, parentFolderId)
2
3  Request "get-folders" and store result in List<Folder> folders
4  For each Folder folder in folders where folder.folderId > 0
5         If local entry for folder does not exist
6                Create local entry
7                Create folder in cache storage
```

```
8         Update local entry and mark it "downloaded"
9         GetFolders(repositoryId, folder.folderId)
10        GetFileEntries(repositoryId, folder.folderId)
11        Notify UI that folder is synced
```

Listing 5.17: Synchronization algorithm: GetFileEntries

```
1  GetFileEntries(repositoryId, folderId)
2
3  Request "get-file-entries" and store result in List<FileEntry> fileEntries
4  For each FileEntry fileEntry in fileEntries
5         If local entry for fileEntry does not exist
6                Create local entry
7         If fileEntry.version > localFileEntry.version and
8                                    localFileEntry.keepInSync = true
9                Update local entry and mark it "pending download"
10        Else
11               Mark local entry "downloaded"
```

Listing 5.18: Synchronization algorithm: HandleDLSync

```
1  HandleDLSync(remoteEntry, repositoryId)
2
3  Get local entry DLFile localEntry from remoteEntry.fileId and repositoryId
4  If remoteEntry.event = "delete"
5         PerformDelete(localEntry)
6         Return
7  If localEntry does not exist
8         If remoteEntry.type = "folder"
9                HandleFolder(localEntry, remoteEntry)
10        Else
11               HandleFileContent(localEntry, remoteEntry)
12        Return
13 If localEntry.filePath != remoteEntry.filePath
14        PerformRename(localEntry, remoteEntry)
15 If remoteEntry.version $\rangle$ localEntry.version
```

```
16        HandleFileContent(localEntry, remoteEntry)
17 If localEntry.directory = true
18        HandleFolder(localEntry, remoteEntry)
```

Listing 5.19: Synchronization algorithm: HandleFileContent

```
1 HandleFileContent(localEntry, remoteEntry)
2
3 If localEntry does not exist
4        Create local entry
5 Update localEntry with remoteEntry data
6 If localEntry.keepInSync = true
7        Mark localEntry "pending download"
```

Listing 5.20: Synchronization algorithm: HandleFolder

```
1 HandleFolder(localEntry, remoteEntry)
2
3 If localEntry does not exist
4        Create local entry with type folder
5        Create folder in cache storage
6 Update localEntry with remoteEntry data
7 Mark localEntry "downloaded"
```

Listing 5.21: Synchronization algorithm: PerformRename

```
1 PerformRename(localEntry, remoteEntry)
2
3 Rename cached content
4 If localEntry.parentId != remoteEntry.parentId
5        Move cached content
6 Update localEntry with remoteEntry data
7 Recursively update children's paths
```

Listing 5.22: Synchronization algorithm: PerformDelete

```
1 PerformDelete(localEntry)
2
3 If localEntry.directory = true
4        Recursively delete children's entries and cached contents
5 Delete localEntry and respective cached content
```

In the next sections we will define the all the object models involved in this algorithm and explain how the HTTP requests to the server's Web Services are carried out by the LiferayDLClient component.

5.4 Logical file representation and persistence

The second most important aspect in the SyncAdapter pattern, after the synchronization algorithm, is the management and persistence of synchronized data. The diagram in Figure 5-3, depicts the overall process of retrieving data from the server and storing it on the client device for local use.

As we have seen earlier, there are two main types of logical data to keep in sync (apart from the contents themselves), which are user sites and file entries. These data are retrieved from the Liferay Portal server through its Web Services API as four different types of responses in JSON format, depending on the API method invoked. JSON is open standard alternative to XML, designed for language-independent, lightweight data interchange. It is derived from the JavaScript scripting language for representing objects as simple data structures and associative arrays.

In order for the Android client to manage and persist these data (in a ContentProvider), every JSON message must be parsed and converted to a Java object of the corresponding type. The responses differ for the fact that their JSON representations have different field names, although they can still be mapped to a single object type. In fact, we have the following method to object mappings:

- "get-user-sites" returns an array of user sites. A user site is converted to a UserSite object;
- "get-dl-sync-update" is converted to a DLSyncUpdate, which contains an array of sync entries, in turn converted to corresponding DLSync objects;
- "get-file-entries" returns an array of JSON structures that are converted to FileEntry objects;
- "get-folders" is mapped to Folder objects.

The JSON to Java object conversion is carried out by the "google-gson" library [55].

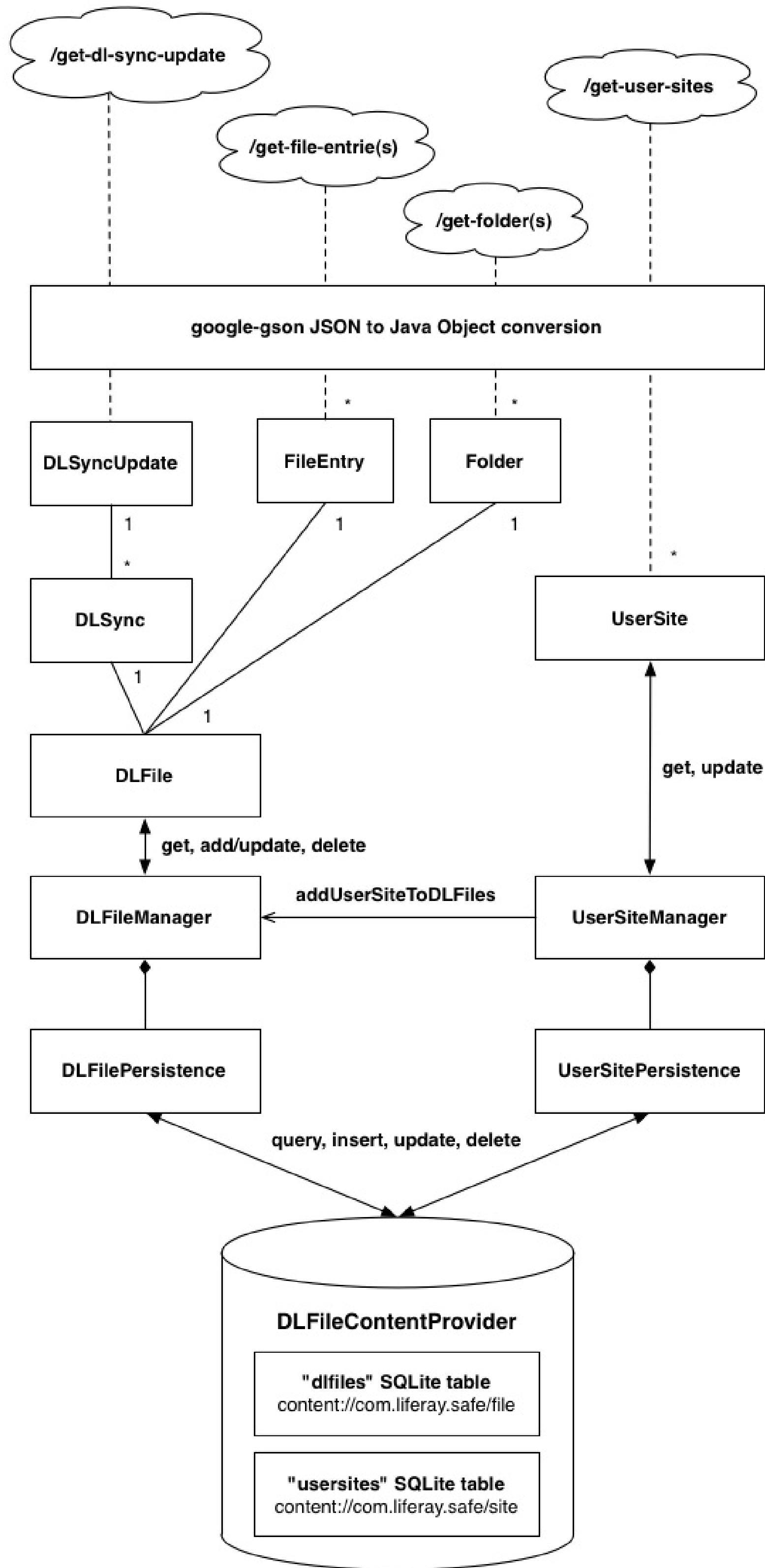


Figure 5-3: Data management and persistence

The following snippet of code shows how a “get-user-sites” response entity is converted to a list of UserSite objects:

Listing 5.23: JSON to Java object conversion

```
1 InputStream instream = entity.getContent();
2 Gson gson = new Gson();
3 Reader reader = new InputStreamReader(instream);
4 Type collectionType = new TypeToken<List<UserSite> > () \{\}.getType();
5 List<UserSite> userSites = gson.fromJson(reader, collectionType);
6 reader.close();
7 instream.close();
```

The use of the library requires no additional step other than defining the class of the Java object itself, as long as the field names correspond exactly to the ones of the JSON object. When they are not, it’s sufficient to annotate the Java fields with the name of the corresponding JSON fields, for example:

Listing 5.24: @SerializedName annotation

```
1 @SerializedName("DLSyncs")
2 private List<DLSync> dlSyncs;
```

This maps “DLSyncs” from JSON to the dlSyncs field of the DLSyncUpdate Java object.

Since FileEntry, Folder and DLSync objects in the end all refer to a file entry, in Liferay Safe all these data are simplified to two main object types: a file entry is represented by a DLFile, while a user site still by a UserSite. The class diagram below illustrates the data model package which includes all the objects just discussed.

Hence a DLFile is the local representation of a file entry stored in the “Documents and Media Library” of a Liferay Portal server. A DLFile has also a fileState property that reflects the synchronization state of the file content it is associated to. The state is a value from an enumeration (DLFileState) and describes either a transition from a “pending” download/upload to a “completed” download/upload or an “error” occurred during an upload. By default the state of a DLFile is NONE, and this means that the corresponding file content is not available yet in the local cache storage.

DLFile and UserSite objects are persisted in a ContentProvider (DLFileContentProvider), which manages a SQLite database holding two tables: the “dlfiles” table for DLFile entries and the “usersites” table for UserSite entries.

All the application logic related to these two types of information, including the access to the ContentProvider, is implemented in a manager-persistence pattern.

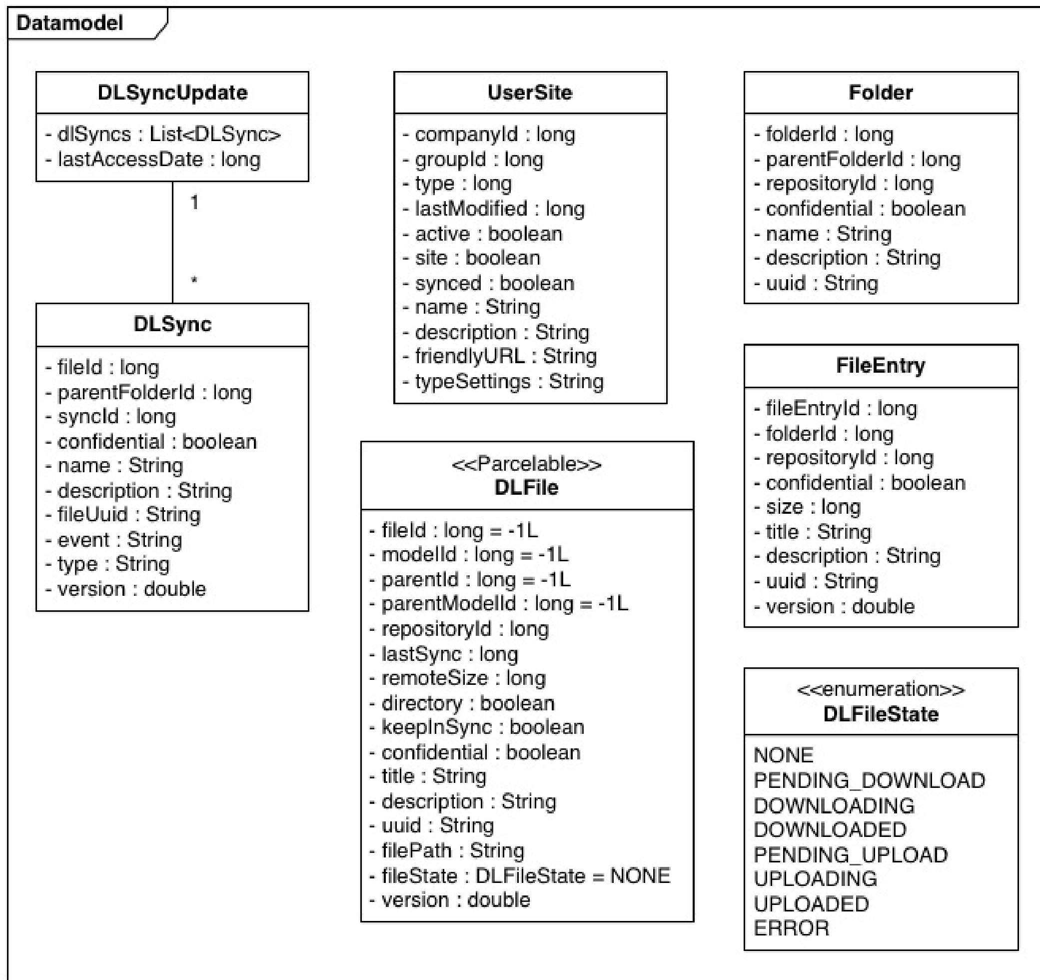


Figure 5-4: Data model

The manager object has the responsibility of performing all the operations defined on the type of model object it is managing. The manager also guarantees that any modification made is immediately persisted on the ContentProvider. For this purpose, it's tightly coupled to a persistence object which has the sole responsibility of calling the ContentProvider's API to persist the modifications. In fact, the manager (or any other application component) never talks directly to the ContentProvider, only the persistence object is allowed to.

DLFileManager and DLFilePersistence manage and persist DLFile objects, while UserSiteManager and UserSitePersistence manage and persist UserSite objects. Their operations are listed in the class diagrams reported below in the next pages.

Finally, since the first screen of the application has to list the sites available at the host Liferay Portal, which work as different content repositories, for convenience, a UserSite is also persisted as a DLFile. The UserSiteManager calls the DLFileManager's addUserSiteToDLFiles() method passing the UserSite to be converted to a DLFile and stored in the corresponding ContentProvider's table.

5.4.1 The Content Provider

The DLFileContentProvider is a custom implementation of the abstract ContentProvider class providing an abstraction from the underlying SQLite database where UserSite and DLFile models are stored in separate SQLite tables.

Content providers support the four basic operations, normally called CRUD-operations. CRUD is the acronym for create, read, update and delete. DLFileContentProvider extends ContentProvider and implements the interface:

- onCreate() which is called to initialize the provider;
- query(Uri, String[], String, String[], String) which returns data to the caller;
- insert(Uri, ContentValues) which inserts new data into the content provider;
- update(Uri, ContentValues, String, String[]) which updates existing data in the content provider;
- delete(Uri, String, String[]) which deletes data from the content provider;
- getType(Uri) which returns the MIME type of data in the content provider.

The persistence classes (DLFilePersistence and UserSitePersistence) do not always use the Content Provider directly (for example, when the interaction started from an Activity), rather they use the Content Resolver.

The Content Resolver is a single, global instance in an application that provides access to content providers. It accepts requests from clients, and resolves these requests by directing the them to the content provider with the given authority. To do this, the Content Resolver stores a mapping from authorities to Content Providers. The ContentResolver class includes the CRUD (create, read, update, delete) methods corresponding to the abstract methods (insert, delete, query, update) in the ContentProvider class.

Managers

DLFileManager

- mDLFilePersistence : DLFilePersistence

+ DLFileManager(account : Account, contentProvider : ContentProviderClient)
+ DLFileManager(account : Account, contentResolver : ContentResolver)
+ getDLFile(repositoryId : long, fileId : long) : DLFile
+ getDLFile(path : String)
+ getDLFileFromModelId(modelId : long) : DLFile
+ getChildrenDLFilesByParentModelId(modelId : long) : List<DLFile>
+ getDLFilesByState(state : DLFileState) : List<DLFile>
+ createDLFileFromPath(name : String, filePath : String, parentFilePath : String, folder : boolean) : DLFile
+ createDLFileFromFolder(dlFolder : Folder, repositoryId : long) : DLFile
+ deleteDLFile(repositoryId : long , fileId : long) : DLFile
+ deleteDLFile(filePath : String) : DLFile
+ addUserSiteToDLFiles(userSite : UserSite) : void
+ renameUserSiteAndDLFiles(repositoryId : long, oldPath : String, path : String) : void
+ containsDLFile(repositoryId : long, fileId : long) : boolean
+ containsDLFile(path : String) : boolean
+ updateParentId(modelId : long) : boolean
+ updateParentId(filePath : String) : boolean
+ updateRepositoryId(dlFile : DLFile) : DLFile
+ handleLocalRenameMove(dlFile : DLFile, newParentPath : String, newName : String) : DLFile
+ handleRemoteRenameMove(repositoryId : long, newParentId : long, newName : long, dlFile : DLFile) : DLFile
+ handleConflict(dlFile : DLFile) : void
+ markDownloaded(dlFile : DLFile) : void
+ markDownloading(dlFile : DLFile) : void
+ markPendingDownload(dlFile : DLFile) : void
+ markUploaded(dlFile : DLFile) : void
+ markUploading(dlFile : DLFile) : void
+ markPendingUpload(dlFile : DLFile) : void
+ markNone(dlFile : DLFile) : void
+ markError(dlFile : DLFile, errorCode : SyncErrorCode) : void
+ markError(path : String, errorCode : SyncErrorCode) : void
- getRootDLFile() : DLFile
- updateDLFile(dlFile : DLFile, state : DLFileState) : void
- updateDLFile(dlFile : DLFile, state : DLFileState, time : long) : void
- findRepositoryId(filePath : String) : long
- performRename(parentDLFile : DLFile, fileName : String, dlFile : DLFile) : String
- updateChildPaths(parentDLFile : DLFile, newPath : String, oldPath : String) : void

UserSiteManager

- mUserSitePersistence : UserSitePersistence
- mDLFileManager : DLFileManager

+ UserSiteManager(account : Account, contentProvider : ContentProviderClient)
+ UserSiteManager(account : Account, contentResolver : ContentResolver)
+ UserSiteManager(account : Account, contentProvider : ContentProviderClient, dlFileManager : DLFileManager)
+ getUserSite(groupId : long) : UserSite
+ getUserSItes() : List<UserSite>
+ setUserSites(userSites : List<UserSite>) : void
+ updateUserSite(userSite : UserSite) : void
+ updateUserSiteLastSynced(repositoryId : long, lastAccessDate : long) : void

Figure 5-5: Data management classes

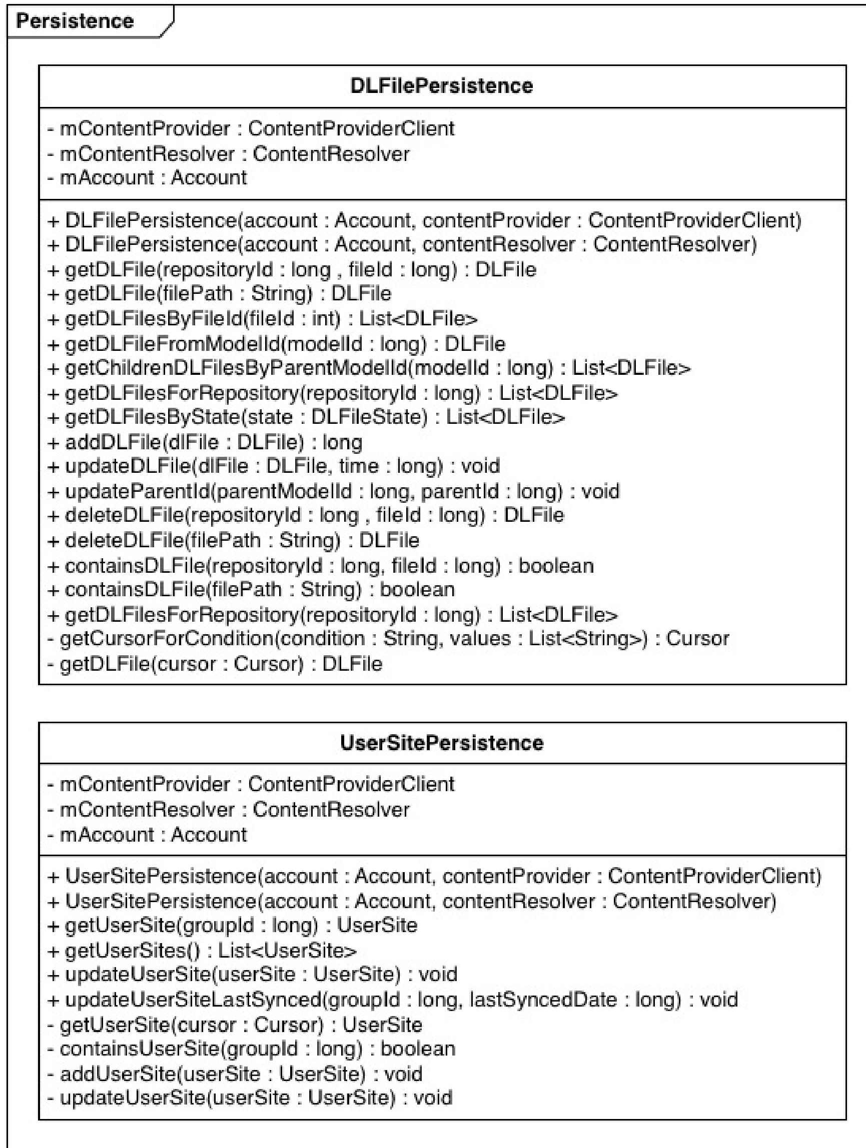


Figure 5-6: Data persistence classes

The Content Resolver does not know the implementation of the Content Providers it is interacting with (nor does it need to know); each method is passed an URI that specifies the Content Provider to interact with.

Data sets of a Content Provider are known by their URI. Each URI uniquely identifies a data set (a single table), or even a specific record in the data set. It's necessary to specify a URI whenever data needs to be accessed from a Content Provider.

URIs for Content Providers have a standardized format that is structured in four parts: `content://authority/optionalPath/optionalId`

- The first part is the **scheme**, that for Content Providers is always "content".
- The next part is the **authority** for the Content Provider. Authorities have to be unique for every content provider. Thus the naming conventions should follow the Java package name rules.
- The third part, the **optional path**, is used to distinguish the kinds of data in a Content Provider. This way a content provider can support different types of data that should be related. If the URI ends with this part it is called a directory-based URI. They are used to access multiple elements of the same type.
- The last element is the **optional id**, a numeric value used to access a single record. If the underlying store is a SQLite database, this corresponds to the mandatory numeric _ID field that uniquely identifies a record within a table. URIs that include this part are called id-based URIs.

Besides defining the content URI patterns, Content Providers based on structured data have to define also their **content types**. Content types are MIME types in Android's vendor-specific MIME format, which consists of three parts:

- Type part: **vnd**
- Subtype part:
  - If the URI pattern is for a single row: **android.cursor.item/**
  - If the URI pattern is for more than one row: **android.cursor.dir/**
- Provider-specific part: **vnd.⟨ name⟩ .⟨ type⟩**
  - The ⟨ name⟩ value should be globally unique, and the ⟨ type⟩ value should be unique to the corresponding URI pattern. A good choice for ⟨ name⟩ is the company's name or some part of the application's Android package name. A good choice for the ⟨ type⟩ is a string that identifies the table associated with the URI.

Implementing a Content Provider working on a SQLite database involves the following steps:

1. Create a class that extends ContentProvider;
2. Define the authority, URIs and database attributes;
3. Create constants for table name and columns;
4. Create a class that extends SQLiteOpenHelper;
5. Implement the getType() method;
6. Implement the CRUD methods;
7. Add the Content Provider to the AndroidManifest.xml.

Based on the defined data model, in the next pages we will expand on all these steps. We assume that the DLFileContentProvider class that extends ContentProvider was already created, therefore we focus on the remaining steps.

Define Content Provider attributes and table constants

Since there isn't a common standard for dealing with these attributes and constants, the definition of a ProviderMeta class was opted as a container for the authority, database attributes, and two inner classes defining URIs, table name and table column names for the two subtypes of our provider: DLFile and UserSite. The class is reported below:

Listing 5.25: ProviderMeta

```
1  public class ProviderMeta {
2
3      public static final String AUTHORITY = "com.liferay.safe";
4      public static final String DB_FILE = "liferaysafe.db";
5      public static final String DB_NAME = "liferaysafe";
6      public static final int DB_VERSION = 1;
7
8          public static final Uri CONTENT_URI = Uri.parse("content://"
9              + AUTHORITY + "/");
10
11     static public class DLFileTableMeta implements BaseColumns {
12         public static final String FILE_TABLE = "dlfiles";
13
14         public static final Uri CONTENT_URI_FILE = Uri.parse("content://"
15                 + AUTHORITY + "/file");
16         public static final Uri CONTENT_URI_FOLDER = Uri.parse("content://"
17                 + AUTHORITY + "/folder");
18
19         public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.liferay.safe.file";
20         public static final String CONTENT_TYPE_ITEM = "vnd.android.cursor.item/vnd.liferay.safe.file";
21
```

```
22            public static final String FILE_DESCRIPTION = "description";
23            public static final String FILE_IS_DIRECTORY = "is_directory";
24            public static final String FILE_ID = "file_id";
25            public static final String FILE_PATH = "path";
26            public static final String FILE_STATE = "state";
27            public static final String FILE_LAST_SYNC = "last_sync";
28                    public static final String FILE_KEEP_IN_SYNC = "keep_in_sync";
29                    public static final String FILE_CONFIDENTIAL = "confidential";
30            public static final String FILE_PARENT_ID = "parent_id";
31            public static final String FILE_PARENT_MODEL_ID = "model_parent_id";
32            public static final String FILE_SIZE = "size";
33            public static final String FILE_REPOSITORY_ID = "repository_id";
34            public static final String FILE_TITLE = "title";
35            public static final String FILE_UUID = "uuid";
36            public static final String FILE_VERSION = "version";
37            public static final String FILE_ACCOUNT_OWNER = "account";
38
39            public static final String DEFAULT_SORT_ORDER = FILE_TITLE
40                    + " collate nocase asc";
41        }
42
43        public static class UserSiteTableMeta implements BaseColumns {
44            public static final String SITE_TABLE = "usersites";
45
46            public static final Uri CONTENT_URI_SITE = Uri.parse("content://"
47                    + AUTHORITY + "/site");
48
49            public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.liferay.safe.site";
50            public static final String CONTENT_TYPE_ITEM = "vnd.android.cursor.item/vnd.liferay.safe.site";
51
52            public static final String SITE_GROUP_ID = "group_id";
53            public static final String SITE_ACTIVE = "active";
54            public static final String SITE_COMPANY_ID = "company_id";
55            public static final String SITE_DESCRIPTION = "description";
56            public static final String SITE_FRIENDLY_URL = "friendly_url";
57            public static final String SITE_NAME = "name";
58            public static final String SITE_SITE = "site";
59            public static final String SITE_SOCIAL_OFFICE_SITE = "social_office_site";
60            public static final String SITE_SYNCED = "synced";
61            public static final String SITE_TYPE = "type";
62            public static final String SITE_TYPE_SETTINGS = "type_settings";
63            public static final String SITE_LAST_MODIFIED = "last_modified";
64            public static final String SITE_ACCOUNT_OWNER = "account";
65
66            public static final String DEFAULT_SORT_ORDER = SITE_NAME
67                    + " collate nocase asc";
68        }
69    }
```

Essentially two different content types and three different content URIs were defined:

- For DLFile data:
  - vnd.android.cursor.item/vnd.liferay.safe.file
  - vnd.android.cursor.dir/vnd.liferay.safe.file
  - content://com.liferay.safe/file
  - content://com.liferay.safe/folder
- For UserSite data:
  - vnd.android.cursor.item/vnd.liferay.safe.site
  - vnd.android.cursor.dir/vnd.liferay.safe.site
  - content://com.liferay.safe/site

To deal with these multiple URIs Android provides the helper class UriMatcher, which eases the parsing of URIs. In the DLFileProvider class the UriMatcher is initialized by adding a set of paths with corresponding int values. The UriMatcher is very important when implementing the CRUD methods, because it allows to detect the type of data is being queried, inserted, updated or deleted, and consequently set the correct table name, where clause and arguments of the corresponding SQL operation.

The following code snippet shows how the UriMatcher is defined:

Listing 5.26: UriMatcher

```
1  private static final int FILE = 1;
2  private static final int FILE_WITH_ID = 2;
3  private static final int FOLDER = 3;
4  private static final int SITE = 4;
5
6  private static final UriMatcher mUriMatcher;
7  static {
8          mUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
9          mUriMatcher.addURI(ProviderMeta.AUTHORITY, "file/", FILE);
10         mUriMatcher.addURI(ProviderMeta.AUTHORITY, "file/#", FILE_WITH_ID);
11         mUriMatcher.addURI(ProviderMeta.AUTHORITY, "folder/#", FOLDER);
12         mUriMatcher.addURI(ProviderMeta.AUTHORITY, "site/", SITE);
13 }
```

Whenever it is asked if a URI matches, the UriMatcher returns the corresponding int-value to indicate which one matches. It is common practise to use constants for these int-values in order to use them with switch statements inside the methods of the Content Provider.

Create a class that extends SQLiteOpenHelper

The SQLiteOpenHelper is an helper class used to manage database creation and version management. A subclass can be created by implementing onCreate, onUpgrade and optionally onOpen, and this class takes care of opening the database if it exists, creating it if it does not, and upgrading it as necessary.

Threfore, the DataBaseHelper subclass was created as an inner class of the DLFileContentProvider class, and the onCreate() method was implemented to create the two database tables using the constants defined in the previous step. Below is the source code of the helper class.

Listing 5.27: DataBaseHelper

```
class DataBaseHelper extends SQLiteOpenHelper {

        public DataBaseHelper(Context context) {
                super(context, ProviderMeta.DB_NAME, null, ProviderMeta.DB_VERSION);
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
                db.execSQL("CREATE TABLE " + DLFileTableMeta.FILE_TABLE + "("
                                + DLFileTableMeta._ID + " INTEGER PRIMARY KEY, "
                                + DLFileTableMeta.FILE_ACCOUNT_OWNER + " TEXT, "

                                + DLFileTableMeta.FILE_DESCRIPTION + " TEXT, "
                                + DLFileTableMeta.FILE_IS_DIRECTORY + " INTEGER, "
                                + DLFileTableMeta.FILE_ID + " INTEGER, "
                                + DLFileTableMeta.FILE_PATH + " TEXT, "
                                + DLFileTableMeta.FILE_STATE + " INTEGER, "
                                + DLFileTableMeta.FILE_LAST_SYNC + " INTEGER, "
                + DLFileTableMeta.FILE_KEEP_IN_SYNC + " INTEGER, "
                + DLFileTableMeta.FILE_CONFIDENTIAL + " INTEGER, "
                                + DLFileTableMeta.FILE_PARENT_ID + " INTEGER, "
                                + DLFileTableMeta.FILE_PARENT_MODEL_ID + " INTEGER, "
                                + DLFileTableMeta.FILE_SIZE + " INTEGER, "
                                + DLFileTableMeta.FILE_REPOSITORY_ID + " INTEGER, "
                                + DLFileTableMeta.FILE_TITLE + " TEXT, "
                                + DLFileTableMeta.FILE_UUID + " TEXT, "
                                + DLFileTableMeta.FILE_VERSION + " REAL );");

                db.execSQL("CREATE TABLE " + UserSiteTableMeta.SITE_TABLE + "("
                                + UserSiteTableMeta._ID + " INTEGER PRIMARY KEY, "
                                + UserSiteTableMeta.SITE_ACCOUNT_OWNER + " TEXT, "

                                + UserSiteTableMeta.SITE_GROUP_ID + " INTEGER, "
```

```
34                                  + UserSiteTableMeta.SITE_ACTIVE + " INTEGER, "
35                                  + UserSiteTableMeta.SITE_COMPANY_ID + " INTEGER, "
36                                  + UserSiteTableMeta.SITE_DESCRIPTION + " TEXT, "
37                                  + UserSiteTableMeta.SITE_FRIENDLY_URL + " TEXT, "
38                                  + UserSiteTableMeta.SITE_NAME + " TEXT, "
39                                  + UserSiteTableMeta.SITE_SITE + " INTEGER, "
40                                  + UserSiteTableMeta.SITE_SOCIAL_OFFICE_SITE + " INTEGER, "
41                                  + UserSiteTableMeta.SITE_SYNCED + " INTEGER, "
42                                  + UserSiteTableMeta.SITE_TYPE + " INTEGER, "
43                                  + UserSiteTableMeta.SITE_TYPE_SETTINGS + " TEXT, "
44                                  + UserSiteTableMeta.SITE_LAST_MODIFIED + " INTEGER );");
45          }
46
47          @Override
48          public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
49                  // Nothing to do here, no previous version exists.
50          }
51
52  }
```

Note that there is no native boolean data type for SQLite, so it's necessary to use integer values instead.

A reference to the DataBaseHelper is initialized in the provider's onCreate() method, and is used in the CRUD methods to get an instance of the database, in read-only or writeable mode.

Implement the getType() and CRUD methods

Every content provider must return the content type for its supported URIs, this is done by implementing the getType() method as follows:

Listing 5.28: ContentProvider getType()

```
1   public String getType(Uri uri) {
2           switch (mUriMatcher.match(uri)) {
3           case FILE: case FOLDER:
4                   return DLFileTableMeta.CONTENT_TYPE;
5           case FILE_WITH_ID:
6                   return DLFileTableMeta.CONTENT_TYPE_ITEM;
7           case SITE:
8                   return UserSiteTableMeta.CONTENT_TYPE;
9           default:
10                  throw new IllegalArgumentException("Unknown Uri id."
11                                  + uri.toString());
12          }
```

```
13 }
```

Note that content URIs for files and folders are associated to the same content type, since they are both modeled as a DLFile.

To give an example of implementation of a CRUD method we take the code of the update operation, which we report below.

Listing 5.29: ContentProvider update(

```
1  @Override
2  public int update(Uri uri, ContentValues values, String selection,
3                              String[] selectionArgs) {
4
5          int updateCount = 0;
6          String table;
7          switch (mUriMatcher.match(uri)) {
8          case FILE:
9                  table = DLFileTableMeta.FILE_TABLE;
10                 break;
11         case SITE:
12                 table = UserSiteTableMeta.SITE_TABLE;
13                 break;
14         default:
15                 throw new IllegalArgumentException("Unknown uri id: " + uri);
16         }
17
18         updateCount = mDbHelper.getWritableDatabase().update(
19                             table, values, selection, selectionArgs);
20         if (updateCount > 0) {
21                 getContext().getContentResolver().notifyChange(uri, null);
22         }
23         return updateCount;
24 }
```

In order to notify other application components when there has been a change in the underlying datastore of a content provider it's necessary to notify the Content Resolver with notifyChange(), passing the URI of the modified record. This notification should be triggered as well when deleting or inserting data.

The CRUD methods are called directly on an instance of the Content Provider (usually when the call originated is the SyncAdapter) or through the ContentResolver. The persistence (DLFilePersistence and UserSitePersistence) will be responsible of performing the call and deal with the necessary paramaters. As an example, below is reported the

updateDLFile() method of the DLFilePersistence class.

Listing 5.30: DLFileManager updateDLFile()

```
public void updateDLFile(DLFile dlFile, long time)
                        throws RemoteException {

    ContentValues cv = new ContentValues();
    cv.put(DLFileTableMeta.FILE_DESCRIPTION, dlFile.getDescription());
    cv.put(DLFileTableMeta.FILE_IS_DIRECTORY, dlFile.isDirectory());
    cv.put(DLFileTableMeta.FILE_ID, dlFile.getFileId());
    cv.put(DLFileTableMeta.FILE_PATH, dlFile.getFilePath());
    cv.put(DLFileTableMeta.FILE_STATE, dlFile.getFileState().ordinal());
    cv.put(DLFileTableMeta.FILE_LAST_SYNC, time);
    cv.put(DLFileTableMeta.FILE_PARENT_ID, dlFile.getParentId());
    cv.put(DLFileTableMeta.FILE_PARENT_MODEL_ID, dlFile.getParentModelId());
    cv.put(DLFileTableMeta.FILE_SIZE, dlFile.getRemoteSize());
    cv.put(DLFileTableMeta.FILE_REPOSITORY_ID, dlFile.getRepositoryId());
    cv.put(DLFileTableMeta.FILE_TITLE, dlFile.getTitle());
    cv.put(DLFileTableMeta.FILE_UUID, dlFile.getUuid());
    cv.put(DLFileTableMeta.FILE_VERSION, dlFile.getVersion());

    if (mContentProvider != null)
        mContentProvider.update(DLFileTableMeta.CONTENT_URI_FILE,
                                cv,
                                DLFileTableMeta._ID + "=? AND " +
                                DLFileTableMeta.FILE_ACCOUNT_OWNER + "=?",
                                new String[] { String.valueOf(dlFile.getModelId()),
                            mAccount.name });
        else
                mContentResolver.update(DLFileTableMeta.CONTENT_URI_FILE,
                                cv,
                                DLFileTableMeta._ID + "=? AND " +
                                DLFileTableMeta.FILE_ACCOUNT_OWNER + "=?",
                                new String[] { String.valueOf(dlFile.getModelId()),
                            mAccount.name });
}
```

Add the Content Provider to the AndroidManifest.xml

The last step is to register the Content Provider within the AndroidManifest.xml file. The next code snippet shows how this is done.

Listing 5.31: Manifest provider tag

```
1 <provider
2         android:name=".provider.DLFileContentProvider"
3         android:authorities="com.liferay.safe"
4         android:enabled="true"
5         android:exported="false"
6         android:label="@string/sync_string_files"
7         android:syncable="true" >
8 </provider>
```

The Content Provider is not exported, which means that only components of the same application, or applications that have the same user ID as the provider will have access to it.

The android:syncable property tells that the provider can be synced using a SyncAdapter. The android:authorities property is important to tie the provider with the SyncAdapter, and in fact must have the same value of android:contentAuthority declared in the ⟨ sync-adapter /⟩ component. Although Content Providers should be used to share data with other applications, there cannot be a SyncAdapter without a ContentProvider and an AccountManager, these three components must be tied together.

Additionally there are a number of benefits to using ContentProvider, such as:

- ContentProvider schedules the database access in a background thread, preventing ANR errors while not requiring you to explicitly handle threading [46].
- ContentProvider ties into ContentResolver's observer: this means it is easy to notify views when content is changed [46].
- It enables to decouple application layers from the underlying data layers, making the application data-source agnostic by abstracting the underlying data source [76].
- It provides mechanisms for defining data security (i.e., by enforcing read/write permissions) and offer a standard interface that connects data in one process with code running in another process [76].

5.5 JSON Web Services API

In Liferay Portal JSON Web Services provide convenient access to portal service methods by exposing them as JSON HTTP API. This makes service methods easily accessible using HTTP requests from any JSON-speaking client [56].

JSON (JavaScript Object Notation) is a lightweight data-interchange text format. JSON is both easy for humans to read and write, and easy for machines to parse and generate. Although it is based on a subset of the JavaScript Programming Language, JSON is completely language independent [57].

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

Each service method registered in the JSON Web Services, is bound to one HTTP method type. All service methods having names starting with get, is or has are assumed to be read-only methods and are therefore mapped as GET HTTP methods, by default. All other service methods are mapped as POST HTTP methods. By default, Liferay Portal works in "non-strict HTTP method" mode; this means that it does not check HTTP methods when invoking a service call, and services may be invoked using any HTTP method.

Furthermore, each service method has the responsibility to determine for itself whether it can be executed by unauthenticated/unauthorized users and whether a user has adequate permission for the chosen action. Most of portal's read-only methods are open to public access.

JSON Web Services can be invoked simply by passing parameters as request parameters directly in the URL. Parameter values are sent as strings using the HTTP protocol. Before a matching Java service method is invoked, each parameter value is converted from a string to its target Java type. Except for methods returning a java.io.InputStream, any returned objects are serialized to a JSON string and returned to the caller.

In the following paragraphs, are listed all the service methods used by Liferay Safe, and is introduced the HTTP client component mentioned in the SyncAdapter pattern, which is responsible for communicating with the server. It will be also provided the sample code of an API invocation.

5.5.1 Currently used API methods

In Liferay Portal a list of all available JSON Web Services can be retrieved from `http://localhost:8080/api/jsonws`. The resulting page lists all registered and exposed service methods of the portal.

As already discussed at the beginning of this chapter, for security constraints it's required that every invocation from the Liferay Safe client to the server's JSON API must be verified and authorized by a portlet implementing an OAuth 2.0 provider & resource server.

In fact, a method exposed by the Document and Media Library services, for example, will never be invoked directly by Liferay Safe, but instead, the client will invoke the exact same method exposed by the Provider Portlet services, passing also the required OAuth parameters. If the client is authorized, the call will be unmarshaled and forwarded to the corresponding Java method of the Local Services API. The response will be returned to the client again from the Provider Portlet, which must first convert the result of the local service call to JSON.

Hence, the JSON HTTP methods of the Provider are actually wrappers of several methods exposed by other different services. These methods are:

- /group/get-user-sites
- /dlsync/get-dl-sync-update
- /dlapp/get-folders
- /dlapp/get-file-entries
- /dlapp/add-folder
- /dlapp/update-folder
- /dlapp/add-file-entry
- /dlapp/update-file-entry
- /dlfileentry/get-file-as-stream

Where group, dlsync, dlapp and dlfileentry are the names of different JSON Web Services. In the case of Liferay Safe, which invokes only the Provider's services, these names are substituted with dlprovider.

In the next pages we report an example of JSON response for each of the first four methods listed; we will focus on the remaining ones when discussing the downloader and uploader services.

/get-user-sites

Returns the guest or current user's layout set group, organization groups, inherited organization groups, and site groups.

Listing 5.32: /get-user-sites JSON response


```
1  [
2      {
3          active: true,
4          classNameId: 10,
5          classPK: 19,
6          companyId: 1,
7          creatorUserId: 5,
8          description: "",
9          friendlyURL: "/guest",
10         groupId: 19,
11         liveGroupId: 0,
12         name: "Guest",
13         parentGroupId: 0,
14         site: true,
15         type: 1,
16         typeSettings: ""
```

```
17          }
18  ]
```

/get-dl-sync-update?companyId=1&repositoryId=19&lastAccessDate=1355499996163

Returns all sync updates occurred in the repository after the last sync.

Listing 5.33: /get-dl-sync-update JSON response


```
1   {
2           DLSyncs: [
3                   {
4                           companyId: 1,
5                           confidential: true,
6                           createDate: 1342279216432,
7                           event: "add",
8                           fileId: 10683,
9                           fileUuid: "c1708549−9672−4f53−aa09−932215d03c34",
10                          modifiedDate: 1342279216432,
11                          name: "research_report.pdf",
12                          parentFolderId: 0,
13                          repositoryId: 19,
14                          syncId: 10684,
15                          type: "file",
16                          version: "−1"
17                  }
18          ],
19          lastAccessDate: 1359763011283
20  }
```


/get-folders?repositoryId=1&parentFolderId=0

Returns all immediate subfolders of the parent folder.

Listing 5.34: /get-folders JSON response


```
1   [
2           {
3                   companyId: 1,
4                   confidential: false,
5                   createDate: 1342792580838,
```

```
6                   defaultFileEntryTypeId: 0,
7                   description: "",
8                   folderId: 12816,
9                   groupId: 19,
10                  lastPostDate: 1359252171687,
11                  modifiedDate: 1342792580838,
12                  mountPoint: false,
13                  name: "Folder",
14                  overrideFileEntryTypes: false,
15                  parentFolderId: 0,
16                  repositoryId: 19,
17                  userId: 2,
18                  userName: "",
19                  uuid: "bc22d971-f4b4-4119-bf46-1138e5db8936"
20          }
21  ]
```

/get-file-entries?repositoryId=19&folderId=0

Returns all the file entries in the folder.

Listing 5.35: /get-file-entries JSON response

```
1   [
2           {
3                   companyId: 1,
4                   confidential: false,
5                   createDate: 1342279217320,
6                   custom1ImageId: 0,
7                   custom2ImageId: 0,
8                   description: "",
9                   extension: "png",
10                  extraSettings: "",
11                  fileEntryId: 10813,
12                  fileEntryTypeId: 0,
13                  folderId: 10683,
14                  groupId: 19,
15                  largeImageId: 0,
16                  mimeType: "image/png",
17                  modifiedDate: 1342279217320,
18                  name: "12",
19                  readCount: 25,
20                  repositoryId: 19,
21                  size: 19163,
22                  smallImageId: 0,
```

```
23                  title: "social_network.png",
24                  userId: 5,
25                  userName: "",
26                  uuid: "8e978614-5d2e-4105-a4a4-32a36e621dfe",
27                  version: "1.0",
28                  versionUserId: 5,
29                  versionUserName: ""
30          }
31  ]
```


An additional field in the JSON objects returned by the Provider Portlet is confidential. This field specifies if the file is marked as confidential in the Documents and Media Library. If it is, Liferay Safe will have to take all necessary measures to preserve the privacy of that file. We discuss how this is done in the remainder of this chapter.

5.5.2 HTTP Client

The component responsible for sending requests to the server, converting JSON responses and processing file streams is LiferayDLClient.

LiferayDLClient is based on Apache HttpClient 4.2.1, a library used also by Web browsers which simplifies the handling of HTTP requests.

In the class diagram below we see that LiferayDLClient encapsulates an instance of DefaultHttpClient. This is the default implementation of the HttpClient class, through which data is sent and received as a HttpUriRequest, such as HttpGet and HttpPost. The response of the HttpClient instead is returned as an InputStream.

To handle HTTP connections, the HttpClient uses an implementation of the ClientConnectionManager interface. The purpose of an HTTP connection manager is to serve as a factory for new HTTP connections, manage persistent connections and synchronize access to persistent connections making sure that only one thread of execution can have access to a connection at a time.

In Liferay Safe a PoolingClientConnectionManager [60] is used to manage a pool of client connections and to service connection requests from multiple execution threads. Connections are pooled on a per route basis. A request for a route for which the manager already has a persistent connection available in the pool will be serviced by leasing a connection from the pool rather than creating a brand new connection. This allows to share a single connection manager among multiple instances of LiferayDLClient and manage multiple connections more efficiently.

The LiferayDLClientFactory is a factory class providing static methods that create different instances of the LiferayDLClient which use a single shared PoolingClientConnectionManager. There are different factory methods depending on whether the HTTP client is used to test the connection to a server or to invoke an API method either passing an access token or not.

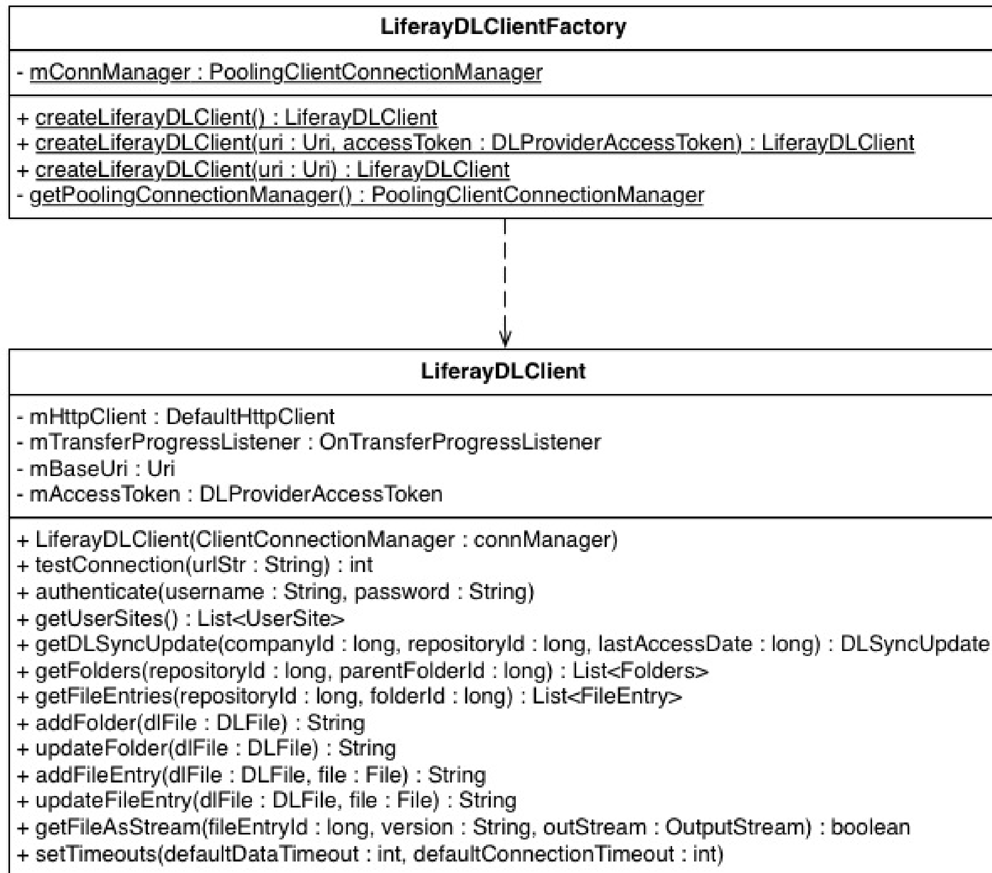


Figure 5-7: HTTP client

The most important method in the factory (reported below) is getPoolingConnection-Manager(), which is used to retrieve the instance of the connection manager necessary to initialize the HttpClient.

Listing 5.36: PoolingConnectionManager

```
1  private static PoolingClientConnectionManager getPoolingConnectionManager() {
2          if (mConnManager == null) {
3
4                          ConnPerRoute connPerRoute = new ConnPerRouteBean(10);
5                  ConnManagerParams.setMaxConnectionsPerRoute(params, connPerRoute);
6                  ConnManagerParams.setMaxTotalConnections(params, 20);
7
8                  SchemeRegistry schemeRegistry = new SchemeRegistry();
9                  schemeRegistry.register(new Scheme("https", SSLSocketFactory.getSocketFactory(), 443));
10
11                         mConnManager = new PoolingClientConnectionManager(schemeRegistry);
```

```
12          }
13      return mConnManager;
14  }
```

Note from the code that the connection manager is initialized with a single scheme to support only the HTTPS protocol. Additionally the scheme is supplied with the instance of SSLSocketFactory (pre-loaded by Android) which is used to validate the SSL certificate against a list of CA's (certification authorities) included in the system trust store.

To explain how the LiferayDLClient works, we consider the following code used to invoke the getDLSyncUpdate method of the JSON Web Services.

Listing 5.37: getDLSyncUpdate()

```
1   public DLSyncUpdate getDLSyncUpdate(long companyId, long repositoryId,
2                       long lastSyncMarker) throws ParseException, IOException,
3                       AuthenticationException {
4
5           DLSyncUpdate dlSyncUpdate = null;
6
7           String url = String.format(URL_GET_DL_SYNC_UPDATE, companyId,
8                               repositoryId, lastSyncMarker);
9
10          // Prepare a request object
11          HttpGet httpget = new HttpGet(mUri.toString() + url);
12
13          // Set the OAuth token in the header
14          httpget.setRequestHeader(Authorization, Bearer  + mAccessToken.getValue());
15
16          // Execute the request
17          HttpResponse response = mHttpClient.execute(httpget);
18
19          // Examine the response status
20          int status = response.getStatusLine().getStatusCode();
21          boolean result = status == HttpStatus.SC_OK;
22
23          if (result) {
24                  // Get hold of the response entity
25                  HttpEntity entity = response.getEntity();
26
27                  if (entity != null) {
28                          InputStream instream = entity.getContent();
29
30                          try {
31                                  Reader reader = new InputStreamReader(instream);
32
```

```
33                                  Gson gson = new Gson();
34
35                                  // Parse the JSON data
36                                  dlSyncUpdate = gson.fromJson(reader, DLSyncUpdate.class);
37
38                          } catch (RuntimeException ex) {
39                                  httpget.abort();
40                                  throw ex;
41
42                          } finally {
43                                  // Ensure that the entity content is fully consumed and
44                                  // the content stream, if exists, is closed.
45                                  EntityUtils.consume(entity);
46
47                                  // Release the connection back to the connection manager
48                                  httpget.releaseConnection();
49                          }
50                  }
51          } else if (status == HttpStatus.SC_UNAUTHORIZED) {
52                  throw new AuthenticationException();
53
54          } else {
55                  throw new IOException();
56
57          }
58          return dlSyncUpdate;
59  }
```

Two important steps when invoking an HTTP method are, first to provide the access token value in the request header, and second to make sure the connection at the end is always released. The flow is analogous for the remaining GET methods; an example of POST execution will be given in the next section after introducing the downloader and uploader services.

5.6 File download and upload

File downloads and uploads are performed by two distinct services of the IntentService class. IntentService is a subclass of Service that uses a worker thread to handle all start requests, one at a time. In particular IntentService does the following:

- Creates a default worker thread that executes all intents delivered to onStartCommand() separate from the application's main thread.
- Creates a work queue that passes one intent at a time to the onHandleIntent() implementation.

- Stops the service after all start requests have been handled, so there's no need to call stopSelf().
- Provides default implementation of onBind() that returns null.
- Provides a default implementation of onStartCommand() that sends the intent to the work queue and then to the onHandleIntent() implementation.

Thus, the only method to implement is onHandleIntent(), which receives the intent for each start request so that work can be done in the background. This is convenient for us because we don't want to handle multiple requests simultaneously.

File downloads are initiated on demand of the user, or whenever a new sync update is received, if the file's keepInSync value is true. The file downloader service must also take care of managing private files differently than public ones, this is done based on the value of the file's confidential property. Similarly, file uploads are started by the user, or whenever a local modification to a file is detected. However, only public files can be uploaded.

In the next paragraphs we look in more detail to the implementation of these services.

5.6.1 FileDownloader service

The FileDownloader service is started with an explicit intent containing an instance of Account, necessary to retrieve the server's base URI, and the DLFile instance of the file being requested for download. Both objects implement the Parcelable interface in order to be bundled with the intent, which is reported below.

Listing 5.38: FileDownloader service intent

```
1 Intent intent = new Intent(this.getContext(), FileDownloader.class);
2 intent.putExtra(FileDownloader.EXTRA_ACCOUNT, account);
3 intent.putExtra(FileDownloader.EXTRA_DLFILE, dlFile);
4 getContext().startService(intent);
```

If the service is being started for the first time, then the service's onCreate() callback is called. At this the server creates a NotificationManager to display notifications about the download progress on the notification bar, and a LiferayDLClient to communicate with the server. Additionally, the service binds to the CacheGuard service (discussed later in the next chapter) responsible for managing the access token and for storing private files. The onCreate() method is reported in the next code listing.

Listing 5.39: FileDownloader service onCreate()

```
1 @Override
2 public void onCreate() {
```

```
3   super.onCreate();
4
5           // Init notification manager
6           mNotificationMngr = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
7
8           // Bind to the CacheGuard service
9           bindService(new Intent(CacheGuardService.class,
10                          mConnection, Context.BIND_AUTO_CREATE);
11
12          // Create the HTTP client
13          mLiferayDLClient = LiferayDLClientFactory.createLiferayDLClient();
14          mLiferayDLClient.setTransferProgressListener(this);
15  }
```

Next the intent is passed to the onHandleIntent() method, where all the work should be performed. Here the service extracts both the Account and the DLFile bundled in the intent, and sets the LiferayDLClient with the current access token returned from the bound CacheGuard service. Next two different methods are called for performing the download, depending on whether the file is marked as confidential or not.

Listing 5.40: FileDownloader service onHandleIntent()

```
1   @Override
2   public void onHandleIntent(Intent i) {
3           if (!intent.hasExtra(EXTRA_ACCOUNT) || !intent.hasExtra(EXTRA_DL_FILE)) {
4                   return;
5           }
6
7           mAccount = intent.getParcelableExtra(EXTRA_ACCOUNT);
8
9           // Set the current base URI and access token
10          String baseUri = AccountManager.get(context).getUserData(mAccount,
11          AccountAuthenticator.KEY_HOST_BASE_URL);
12          mLiferayDLClient.setBaseUri(baseUri);
13          mLiferayDLClient.setAccessToken(mCacheGuard.getAccessToken());
14
15          DLFile dlFile = intent.getParcelableExtra(EXTRA_DLFILE);
16
17          boolean result = false;
18
19          // Notify download start
20
21          if (dlFile.getConfidential())
22                  result = processPrivateFileDownload(dlFile);
23          else
```

```
24                  result = processPublicFileDownload(dlFile);
25
26          // Broadcast result to registered receivers
27          sendFinalBroadcast(result);
28  }
```

If the file is a private file, then processPrivateFileDownload() is called. The method calls the LiferayDLClient's getFileAsStream() method passing an OutputStream where the downloaded stream should be written to. The OutputStream is managed by the CacheGuard service, which will take care of writing to an encrypted storage, and mark the file as downloaded in the persistence if the operation was completed with success.

The outcome of the download is then notified to the user. If any error occurred, the state of the cache is reverted and the file is marked with a specific error code.

Listing 5.41: Private file download

```
1   private boolean processPrivateFileDownload(DLFile dlFile) {
2           boolean result = false;
3
4           try {
5                   result = mLiferayDLClient.getFileAsStream(dlFile.getFileId(),mDLFile.getVersion,
6                                   mCacheGuard.initFileCaching(dlFile).getOutputStream());
7
8           } catch (Exception e) {
9                   // Set notification error message
10          }
11
12          if(result) {
13                  mCacheGuard.finalizeFileCaching(dlFile);
14                  // Notify download success
15          }
16          else {
17                  mCacheGuard.abortFileCaching(dlFile);
18                  // Notify download error
19          }
20
21          return result;
22  }
```

For downloads of public files the procedure is similar, yet the OutputStream is returned from the CacheManager and the result of the download is persisted through the DLFileManager.

Listing 5.42: Public file download

```
1  private boolean processPublicFileDownload(DLFile dlFile) {
2          boolean result = false;
3
4          try {
5                  result = mLiferayDLClient.getFileAsStream(dlFile.getFileId(),dlFile.getVersion,
6                          new FileOutputStream(getCacheManager.openLocalFile(dlFile)));
7
8          } catch (Exception e) {
9                  // Notify download error (I/O or storage space)
10         }
11
12         if (result) {
13                 getDLFileManager().markDownloaded(dLFile);
14                 // Notify download success
15         }
16         else {
17                 getDLFileManager().markError(dlFile, SyncErrorCode.DOWNLOAD_UNKNOWN);
18                 getCacheManager.deleteLocalFile(dlFile);
19                 // Notify download success
20         }
21
22         return result;
23 }
```

The getFileAsStream() method of the HTTP client writes the server's response stream exactly in the OutputStream supplied as an argument. During this operation the client calls the transferProgress() callback defined in the OnTransferProgressListener interface, passing the number of bytes written to the registered listener implementing the interface, which in this case is the FileDownloader service. This allows the service to update a progress bar while the file is being download.

Listing 5.43: HTTP client getFileAsStream()

```
1  public boolean getFileAsStream(long fileEntryId, String version, OutputStream outStream)
2                  throws IOException, AuthenticationException {
3          boolean result = false;
4
5          String url = String.format(URL_GET_FILE_AS_STREAM, fileEntryId, version);
6
7          HttpGet httpget = new HttpGet(mUri.toString() + url);
8          httpget.setRequestHeader(Authorization, Bearer  + mAccessToken.getValue());
9
10         HttpResponse response = mHttpClient.execute(httpget);
```

```
11
12          int status = response.getStatusLine().getStatusCode();
13
14          if (status == HttpStatus.SC_OK) {
15                  HttpEntity entity = response.getEntity();
16
17                  if (entity != null) {
18                          InputStream instream = entity.getContent();
19
20                          try {
21                                  byte[] b = new byte[1024];
22                              int len = 0;
23                              while ((len = instream.read(b)) != -1) {
24                                          if (mTransferProgressListener != null)
25                                                  mTransferProgressListener.transferProgress(len);
26                                  outstream.write(b, 0, len);
27                              }
28                                  result = true;
29
30                          } catch (RuntimeException ex) {
31                                  httpget.abort();
32                                  throw ex;
33
34                          } finally {
35                                  EntityUtils.consume(entity);
36                                  outstream.close();
37
38                                  httpget.releaseConnection();
39                          }
40                  }
41
42          } else if (status == HttpStatus.SC_UNAUTHORIZED) {
43                  throw new AuthenticationException();
44
45          } else {
46                  throw new IOException();
47
48          }
49
50          return result;
51  }
```

Finally, besides notifying the user about the result, when a download is completed the service sends a local broadcast to all other components registered with a BroadcastReceiver.

5.6.2 FileUploader service

Uploading a new file to the remote repository involves first selecting such file either directly from the device's local storage or from other installed applications. Additionally, a user might select a single file or multiple files at once to upload.

For simplicity we will deal with the upload of a single file imported from other applications. Instead of implementing a custom local file browser, it's assumed that one of the many file explorers available for free from the Google Play store is already installed on the device. Anyhow, our implementation can be entirely reused to support other more sophisticated requirements.

The first step for uploading a file is to display to the user a list of available applications where a file can be selected from. This is done with the intent in the next code snippet:

Listing 5.44: FileUploader service intent

```
1  Intent action = new Intent(Intent.ACTION_GET_CONTENT).
2                  .setType("*/*")
3                  .addCategory(Intent.CATEGORY_OPENABLE);
4                  startActivityForResult(Intent.createChooser(action,
5                                  getString(R.string.upload_chooser_title)),
6                                  ACTION_SELECT_CONTENT_FROM_APPS);
```

This intent will start the activity matching the application selected by the user, expecting as a result the Uri of the file that was selected, returned with an intent to the onActivityResult() callback. If the result is valid a new entry in the Content Provider can be created based on the path of the folder the user decided to upload the file to, and the FileUploader service can be started.

Listing 5.45: File upload onActivityResult()

```
1  public void onActivityResult(int requestCode, int resultCode, Intent data) {
2
3      if (requestCode == ACTION_SELECT_FILE_FROM_APPS && resultCode == RESULT_OK) {
4          Uri fileUri = data.getData();
5
6                  String name = new File(fileUri).getName();
7                  String filePath = mCurrentDir.getFilePath() + FILE_SEPARATOR + name;
8                  DLFile uploadFile = mDLFileManager().createDLFileFromPath(name, filePath,
9                                  mCurrentDir.getFilePath(), false);
10
11                 Intent intent = new Intent(this, FileUploader.class);
12                 intent.putExtra(FileDownloader.EXTRA_ACCOUNT, account);
13                 intent.putExtra(FileDownloader.EXTRA_DLFILE, uploadFile);
```

```
14                  intent.putExtra(FileDownloader.EXTRA_FILE_PATH, fileUri.getPath());
15                  startService(intent);
16      }
17  }
```

Note that when a new file is uploaded the file itself isn't cached in Liferay Safe. For this reason, when the user want's to access the file from Liferay Safe, a new synced version of the file will have to be downloaded to the cache from the server, possibly creating a double copy in the local storage.

The logic implemented by the FileUploader service is simpler than its downloader counterpart, since the upload of private files is not supported. The file upload consists in executing from the HTTP client either the addFileEntry() or updateFileEntry() method depending on whether the file is being added or it was modified.

The server's response, which is returned to the service from the HTTP client, can either be a string representing a server exception or a FileEntry as a JSON object. In the first case the service will handle the exception and set the state of the local file to an error value, accordingly. In the second case the service can proceed to parse the FileEntry from JSON and use it to update the local file entry. Finally, if no errors occurred, the file is marked as uploaded.

Listing 5.46: FileUploader onHandleIntent()

```
1   @Override
2   public void onHandleIntent(Intent i) {
3           if (!intent.hasExtra(EXTRA_ACCOUNT) || !intent.hasExtra(EXTRA_DL_FILE)) {
4                   return;
5           }
6
7           mAccount = intent.getParcelableExtra(EXTRA_ACCOUNT);
8
9           // Set the current base URI and access token
10          String baseUri = AccountManager.get(context).getUserData(mAccount,
11                                  AccountAuthenticator.KEY_HOST_BASE_URL);
12          mLiferayDLClient.setBaseUri(baseUri);
13          mLiferayDLClient.setAccessToken(mCacheGuard.getAccessToken());
14
15          DLFile dlFile = intent.getParcelableExtra(EXTRA_DLFILE);
16          String filePath = intent.getStringExtra(EXTRA_FILE_PATH);
17
18          if (filePath == null)
19                  filePath = CacheManager.getLocalFilePath(dlFile);
20
21          // Notify upload start
22
```

```
23          String response;
24          if (dlFile.getFileId() == −1)
25                  response = mLiferayDLClient.addFileEntry(dlFile, filePath);
26          else
27                  response = mLiferayDLClient.updateFileEntry(dlFile, filePath);
28
29          // Check server errors
30          boolean result = checkServerException(response);
31
32          if (!result) {
33                  // Parse JSON response
34                  Gson gson = new Gson();
35                  FileEntry fileEntry = gson.fromJson(response, FileEntry.class);
36
37                  if (fileEntry.getFileEntryId() > 0) {
38                          DLFileUtil.updateDLFile(dlFile, fileEntry);
39                          getDLFileManager().markUploaded(dlFile);
40                          result = true;
41                  } else {
42                          getDLFileManager.markError(dlFile, SyncErrorCode.UPLOAD_UNKNOWN);
43                          // Notify upload error
44                  }
45          }
46
47          // Broadcast result to registered receivers
48          sendFinalBroadcast(result);
49  }
```

The file is actually uploaded as a FileBody content in a MultipartEntity sent with an HTTP POST method. Since the user has to be notified about the upload progress, the MultipartEntity class was extended in order to override the writeTo() method and post the progress to the FileUploader implementing the OnTransferProgressListener interface. In particular a custom FilterOutputStream was used to get the number of bytes written to the output stream.

Listing 5.47: HTTP client MultipartEntity subclass

```
1   public class DLMultiPartEntity extends MultipartEntity
2   {
3
4           private final OnTransferProgressListener listener;
5
6           @Override
7           public void writeTo(final OutputStream outstream) throws IOException
8           {
```

```
9             super.writeTo(new ProgressOutputStream(outstream, this.listener));
10        }
11
12        public static class ProgressOutputStream extends FilterOutputStream
13        {
14
15            private final OnTransferProgressListener listener;
16            private long transferred;
17
18            public void write(byte[] b, int off, int len) throws IOException
19            {
20                out.write(b, off, len);
21                this.transferred += len;
22                if (this.listener != null)
23                    this.listener.transferProgress(this.transferred);
24            }
25            ...
26        }
27    ...
28    }
```

Finally to complete this section, reported below is the implementation of the addFileEntry() method.

Listing 5.48: HTTP client addFileEntry

```
1   public String addFileEntry(DLFile dlFile, String filePath) {
2
3       HttpPost httpPost = new HttpGet(mUri.toString() + URL_ADD_FILE_ENTRY);
4
5       httpPost.setRequestHeader(Authorization, Bearer  + mAccessToken.getValue());
6
7       DLMultipartEntity multipartEntity = new DLMultipartEntity(mTransferProgressListener);
8
9       StringBody changeLogBody = createStringBody(dlFile.getVersion());
10      multipartEntity.addPart("changeLog", changeLogBody);
11
12      String description = dlFile.getDescription();
13      StringBody descriptionBody;
14      if (description != null) {
15          descriptionBody = createStringBody(description);
16      }
17      else {
18          descriptionBody = createStringBody("");
19      }
```

```
20          multipartEntity.addPart("description", descriptionBody);
21
22          StringBody fileEntryIdBody = createStringBody(dlFile.getFileId());
23          multipartEntity.addPart("fileEntryId", fileEntryIdBody);
24
25          StringBody majorVersionBody = createStringBody("false");
26          multipartEntity.addPart("majorVersion", majorVersionBody);
27
28          StringBody titleBody = createStringBody(dlFile.getTitle());
29          multipartEntity.addPart("sourceFileName", titleBody);
30
31          String mimeType = MimeTypeUtil.getMimeType(dlFile.getFilePath());
32          StringBody mimeTypeBody = createStringBody(mimeType);
33          multipartEntity.addPart("mimeType", mimeTypeBody);
34
35          multipartEntity.addPart("title", titleBody);
36
37          try {
38                  File file = new File(filePath);
39                  FileBody filePart = new FileBody(file, mimeType);
40                  multipartEntity.addPart("file", filePart);
41          } catch (Exception e) {
42                  return null;
43          }
44
45          httpPost.setEntity(multipartEntity);
46
47          HttpResponse response = mHttpClient.execute(httpPost);
48
49          int status = response.getStatusLine().getStatusCode();
50
51          if (status == HttpStatus.SC_OK)
52                  return EntityUtils.toString(response.getEntity());
53
54          return null;
55  }
```

The implementation is analogous for the remaining POST invocations to the server's JSON Web Services API that were mentioned in the previous section.

Chapter 6

Offline usage and preservation of private data

There are two important concepts that make the development of Liferay Safe for Android an interesting case of study, these are offline usage and preservation of private data.

The concept of offline usage (or offline mode) is very common in the context of a client-server architecture. In fact, offline usage is the capability of the client application to deliver its functionalities - or at least, part of them - to the user when a connection to the server cannot be established. Furthermore, an application in offline mode must provide the same security level as if operating in online mode. This directly implies that, under this mode of operation, the application must still be able to identify and authenticate the user, as well as preserving any managed private data.

To better define what we mean for private data and its preservation, we should first recall that Liferay Portal (the server) is "The Leading Open Source Portal for the Enterprise". This tagline is important because it clearly states that the domain of study of this work is exactly the Enterprise domain. This means that the party owning and administering the server is not a group of common consumer users, but a company or organization of any size and complexity.

Liferay Portal allows the development of collaborative websites for teams and an entire enterprise. Liferay users can be grouped into a hierarchy of "organizations" or cross-organizational "user-groups". A Liferay user can also create or join one or more communities and organize all work collaboration within that community.

Liferay Portal also provides a central platform for determining enterprise content policy depending on a user's role, including who can edit and publish Web content, files, communities, and applications.

Therefore, Liferay is also a full workflow-enabled Web Content Management System (WCMS). Including a unified document repository (the "Documents and Media Library") that can be leveraged across an enterprise, within a specific group, or for a single user as a web repository. Liferay users can store documents, video, audio, images, and other media types all in one place, and make them available on the portal websites, or across different client applications through the HTTP APIs, to download them for offline use.

The confidentiality of documents (intended as "data" in general) stored in Liferay Portal can be scaled to various level - unless they are of public domain, of course. A document can be private to the single user, or it can be confidentially disclosed within a specific group or the entire enterprise. However, the concept of privacy for some documents overlaps with the concept of private proprietary information.

> *"Proprietary information is sensitive information that is owned by a company and which gives the company certain competitive advantages. Proprietary information assets are critical to the success of many, perhaps most businesses. [...] Proprietary information, also known as a trade secret, is information a company wishes to keep confidential."* [56]

Whereas private personal information, is currently defined by the European Union as:

> *"any information relating to an identified or identifiable natural person ('data subject'); an identifiable person is one who can be identified, directly or indirectly, in particular by reference to an identification number or to one or more factors specific to his physical, physiological, mental, economic, cultural or social identity;"* Article 2(a) of the Data Protection Directive [57]

Hence, in Liferay we deal with two types of private data. On one hand, data that is associated with, and belongs to a user as a private individual, which was uploaded to her own repository and that may be shared with others for personal or business reason. On the other hand, data that is proprietary information of the enterprise, which can be uploaded to a collaborative repository by a user as an enterprise representative.

For preservation of private data we mean information security, which indeed may be defined as:

> *"Preservation of confidentiality, integrity and availability of information; in addition, other properties such as authenticity, accountability, non-?repudiation and reliability can also be involved."* [63]

The first three primary concepts in information security - also known as CIA triad - are defined in ISO 27001 [63] as follows:

- Confidentiality - *"the property that information is not made available or disclosed to unauthorized individuals, entities or processes"*;
- Integrity - *"the property of safeguarding the accuracy and completeness of assets"*;
- Availability - *"the property of being accessible and usable upon demand by an authorized entity"*.

Private data are those information assets for which all three attributes are important; security of private data focuses on provision of protection mechanisms that preserve those three attributes.

In the development of Liferay Safe - a Liferay Portal document synchronizer for Android - different problems must be faced regarding the preservation of CIA over private data. Some of these issues are inherent in a client-server architecture, other are more particularly related to mobility and to the Android platform. Moreover, the requirement of offline access capability poses an inherent challenge to privacy preservation, because it typically results in losing control other data.

When considering a mobile client application for Android, relevant issues related to privacy preservation arise, including and not limited to: data usage and exfiltration, data theft, inadequate data retention, inadequate data deletion, communication spoofing (security of data in transit), client/server impersonation, dynamic provisioning of trust (user revocation), unauthorized access, lack of a root of trust, device tampering, Android malware, privilege escalation, and notification of breach.

Some of these issues can be eliminated, other represent risks that can only be mitigated. In the following sections we propose a trade-off approach towards the challenging task of simultaneously providing preservation and offline access to private data in a mobile Android application.

6.1 A trade-off approach

Considering the issues related to privacy preservation that we mentioned before, the goal of the presented approach is to build a secure implementation of Liferay Safe for Android, that is, one capable of preserving the privacy of private user and corporate data at multiple levels, and against multiple risk factors.

In synthesis, the approach consists in applying security coding best practices to harness features specific to Android - such as the component framework, the security model and lower kernel capabilities - and to integrate state of the art technologies compatible with the Android platform. In particular, the approach aims to give the following guarantees:

- Security of data in transit: client-server communication is secured with HTTPs. Man In The Middle attacks (MITM) and impersonation are mitigated with the use of TLS certificates. See section 4.1;

- Prevention from unauthorized data access: the OAuth 2.0 protocol provides an authorization mechanism that allows to control, restrict and revoke user access to protected data, while preserving their account credentials. The automatic screen lock functionality protects retained data from unwanted eyes when the device is left unattended. See section 4.2.

- Protection of data at rest: private data cached in the local storage must be protected in the event of its unauthorized access or theft. Encrypted cache storage and secure encryption key management are the core solutions to prevent data visibility and mitigate privacy leakage. See sections 6.4 and 6.5.

- Control over the use of data: as any other asset, private data has no value to the user if it cannot be used at least for the purpose of viewing. Unfortunately, it's not possible to entrust either a third party, or the system itself, to provide such usage functionalities, unless without taking into account all the security risks involved. Though, security of private data can be preserved if all functionalities required by the user are implemented by a trusted party, and if there is a single entity holding the control of private data during all its use in memory. See section 6.6.

- Reduction of tampering exposures: Android-compatible devices have proven to be particularly open to user customization. At the same time, Android's architecture stack is considered an interesting attack surface where tools of exploitation and reverse engineering common to the Linux environment can be put in practice to easily gain root privileges and violate user's private data. There is no final protection against tampering of the device, this risk can only be assumed or mitigated.

The approach is called a trade-off because at the state of the art on Android devices it's not possible to provide a fully functional solution able to protect private data and enable offline usage at the same time; at least without posing any constraints and limitations to the application, or taking initial assumptions.

Furthermore, a key aspect of this approach is that the resulting solution does not absolutely imply forking the original Android Open Source Project (AOSP), or building a custom firmware, to leverage the above security provisions. In fact, an important requirement is that enterprises interested in this solution should not be constrained to provide their employees a specific device tailored for this application, but instead employees should be allowed to use their own Android devices in their jobs and install Liferay Safe to access corporate private data. However, this implies that there is little or no control over how devices running Liferay Safe are used, what other applications are installed, and if the devices are rooted or have a customized firmware. Besides, in the scenario where the owner of the device turns out to be a malicious user, even end-user policies defined by enterprises turn out to be both highly ineffective and very difficult to be enforced, yet they are important to educate trusted employees to make proper use of their device.

Taking all this into account, there is no surprise in the fact that the result of this approach should not be deemed perfect, and consequently we shall now state the initial assumptions, constraints and limitations of the presented design:

- Minimum Operating System: lacking a root of trust, the solution heavily relies on the underlying kernel security to store in memory expiring encryption keys and access tokens. Starting from Android 4.1 (Jelly Bean) there is full support for 32-bit Address Space Layout Randomization (ASLR) and Position-independent executables (PIE) [64], which hinder some types of security attacks (such as Return Oriented Programming) by preventing an attacker from being able to easily predict target addresses [65].

- 24-hours limited offline access: to enforce revocation of a user, offline access is limited to 24-hours, after which the application is locked to the authentication screen and

the private cache is erased. The public cache can still be accessed by mounting the external storage.

- Maximum size of private data: because private data is never written to disk unencrypted, private files can only be accessed in such state from the volatile memory. Therefore, the maximum size of a single private file cannot be larger than the available RAM on the device.

- No editing of private data: to avoid losing control over usage of private data, and prevent its exfiltration, a trusted proprietary solution for editing files is needed. However, this would imply to develop and maintain a full "office suite" solution, which is out of the scope of this project. Moreover, there are no open source projects available of that kind.

- PDF format limitation: for the same reason above, at the moment Liferay Safe supports only private files in PDF format. The functionality is delegated to a separate viewer application based on an open source project. Fortunately, Liferay Portal supports automated conversion of several file formats to PDF.

- Vulnerability to pre-rooted devices: there is no 100% effective way to detect rooting [66]. If a device was already rooted prior to the installation of Liferay Safe, any private data subsequently downloaded could be compromised based on the attacker's skills. Enterprise end-user policies should include responsibilities related to device tampering.

- Enterprise policy enforcement: the enterprise must promptly revoke user access when a user isn't trusted anymore, and in case of policy breach.

- Mandatory screen lock and disabled "USB debugging" setting: the Android Debug Bridge (ADB) is the simplest vector for rooting a device, but it requires the "USB debugging" setting to be enabled [11]. The application enforces the use of a screen lock to protect this settings and requires that it is disabled for the private cache to be available. Whenever the ADB is enabled, the private cache is deleted.

- Vulnerability to remote exploitation: an attacker can still gain root access through malicious applications containing privilege escalation attacks, or by taking advantage of vulnerabilities in system components, such as the browser [11]. However, it's assumed that these techniques would require a reasonable amount of time and a highly experienced attacker to be performed.

- Vulnerability to memory dump: after gaining root access, an attacker can dump the application memory and analyze it to find a valid private cache encryption key or a server access token [67, 68]. It's assumed that memory analysis is a difficult and time consuming task. The risk is mitigated by limiting the validity of sensitive data in memory and reducing its duplication.

- Device whitelisting: some devices expose specific vulnerabilities related to hardware or firmware bugs (such as the latest Samsung Exynos exploit [69]), and additionally, vendors adopt different cycles for releasing security patches and system updates. Maintaining a whitelist of devices can reduce the risk of distributing private data to reportedly compromised devices.
- Impact on battery life: several operations must be executed periodically to ensure the security of private data, resulting in a more rapid battery discharge.

Details regarding the implementation of this approach will be discussed in the remainder of this chapter.

6.2 Marking files as "confidential"

The procedure for marking a file as confidential is directly integrated in Liferay's "Documents and Media Library" (DML) without any necessary modification to the source code. To mark a file confidential, a user has to simply add the "confidential" tag from the document's properties screen, as shown in Figure 6-1.

New and existent documents can be tagged and untagged exclusively from Liferay Portal, and not from the mobile client. The tag information applied to each file entry is evaluated by the Provider Portlet in order to set the boolean value of the confidential field that must be included in the JSON objects returned as a response to the client. Since the original API methods don't include this information, the Provider must retrieve it separately from other services.

Fortunately, the AssetTagService allows us to get the list of all tags associated to any asset stored in the portal, including documents of the DML. This local service is based on Liferay's Asset Framework, which was created exactly to allow portal applications and custom portlets to be able to add tags to any kind of asset in the portal without having to reimplement this same functionality over and over. The term asset defines any type of content regardless of whether it's purely text, an external file, a URL, an image, a document, etc.

If for example we call this method via JSON HTTP, by specifying the asset's class and id, the output would be the following JSON array:

Listing 6.1: get-tags JSON response

```
1 /get-tags?className=com.liferay.portlet.documentlibrary.model.DLFileEntry&classPK=16722
2
3 [
4         {
5                 assetCount:1,
6                 companyId:1,
7                 createDate:1359251455864,
```

Figure 6-1: Marking a file as confidential

```
8                       groupId:19,
9                       modifiedDate:1359251455864,
10                      name:"confidential",
11                      tagId:17218,
12                      userId:12604,
13                      userName:"Max"
14              }
15      ]
```

Hence, the Provider can call this service for any file entry, check if it's marked as "confidential" and finally set the corresponding field in the JSON object returned to the client.

6.3 Cache architecture

The local cache is probably the most important feature in Liferay Safe. The reasons for this are multiple and most of them have already been discussed in the previous sections, however, let's summarize them here all together:

- Reduce network traffic and file access time;
- Provide offline access to files and improve data availability;
- Enable opening files in other applications for viewing or editing.

Basically, the responsibility of the cache is to provide offline access to both public and private files. This distinction has important implications on the architecture of the cache, because all private files must always be encrypted before being stored to disk. Additionally it's required that, *the unencrypted version of a file must never be written disk* under any circumstance or operation; it can only be *retained in volatile memory and for a short amount of time*, after which the memory must be correctly zeroized.

In order to implement these requirements in a correct and sound way, two distinct caches are employed: a secure one for private files and a *world-readable* one for the rest. Although these two are stored and managed differently, they share the following functional characteristics:

- Both caches are stored in Android's external storage;
- A file stored in a cache has the same relative path of the corresponding remote file, with respect to the parent site repository;
- The user is be able to configure the size of the overall storage space allocated to the two caches (public + private). This value is persisted in Android's SharedPreferences;
- All file downloads are automatically cached;
- When the limit is reached, the cache will try to free up space by deleting the oldest synced/modified files, starting from the public cache first;
- The user can remove a file from the local cache individually.
- The architecture of the local cache is illustrated in Figure 6-2 below. In the next paragraphs are explained the functioning of the public cache, while the private encrypted cache will be covered more thoroughly in the next two sections.

The diagram provides us a big picture of the components involved in the implementation of the two caches. Focusing for now on the public cache, the CacheManager is the only component entitled to perform operations that change the state of the public cache, such as creating, deleting, renaming and moving a cached file. Including, setting and maintaining the available storage space.

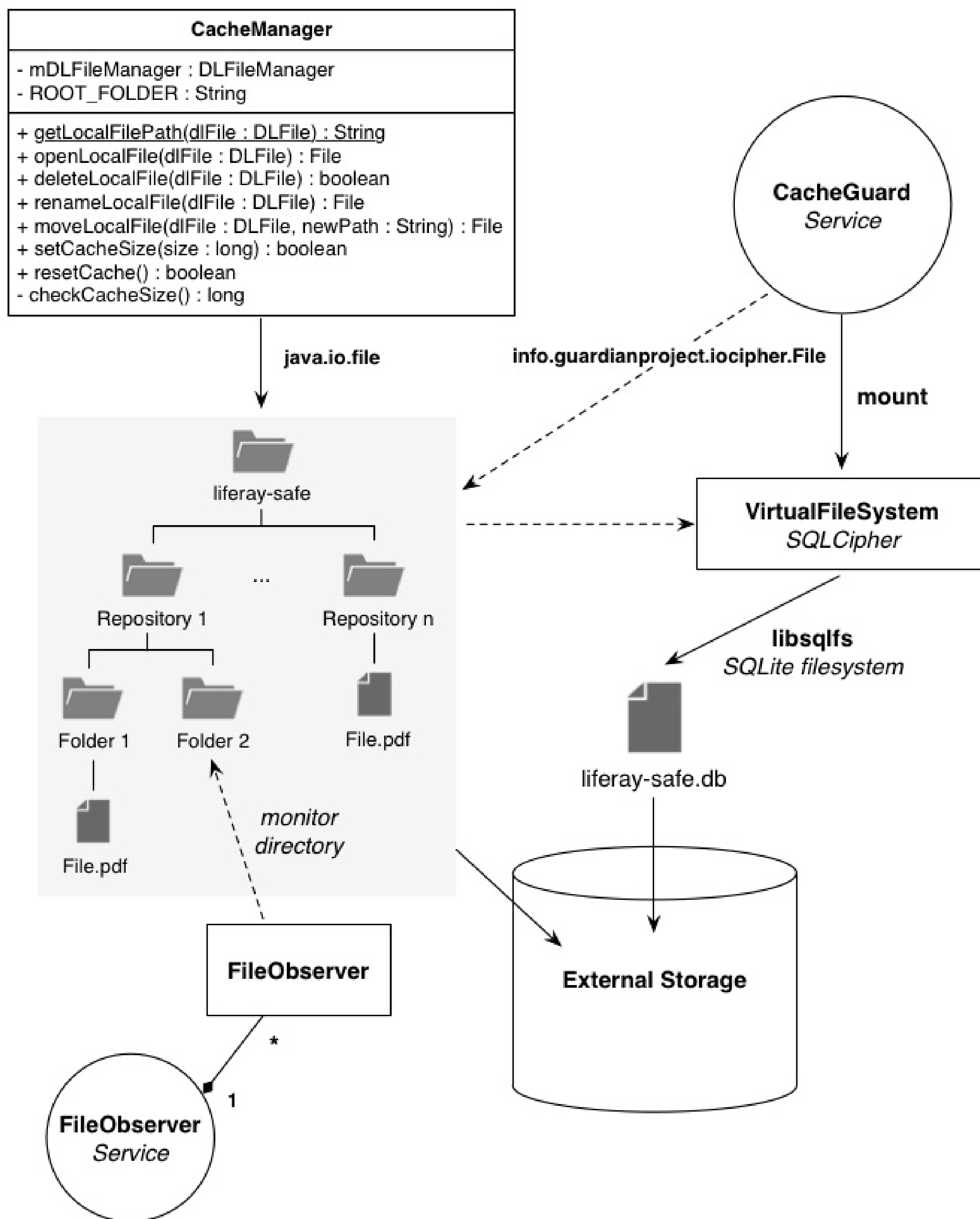


Figure 6-2: Cache architecture

When starting a new download task, the DownloaderService calls the CacheManager's openLocalFile() method, passing the entry of the file going to be downloaded. This causes the CacheManager first to check if the cached file already exists, and then to verify if it's necessary to free up space to complete the download, according to the entry's remoteSize property. Eventually, a list of all synced cached files, ordered by lastSync, can be retrieved from DLFileManager by calling getDLFilesByState(DLFileState) for DOWNLOADED and UPLOADED file states. The combined results can be used to calculate the current occupation size and to select eventual candidates for removal. Finally, a local file is opened (and eventually created) from the entry's filePath property, and returned to the DownloaderService.

Public files are cached to the external storage. This can be thought as a file system supported by every Android-compatible device, that can hold a relatively large amount of data and that is shared across all applications. Traditionally this is an SD card, but it may also be implemented as built-in storage in a device that is distinct from the protected internal storage and can be mounted as a file system on a computer.

Accessing files on external storage is as easy as getting the absolute path to the external storage directory. The following snippet shows how to open a file from the public cache given a filePath:

Listing 6.2: Accessing the external storage

```
1 File sdCardRoot = Environment.getExternalStorageDirectory();
2 File file = new File(sdCardRoot.getAbsolutePath() + "/liferay-safe/" + filePath);
```

The absolute path of any file stored in the public cache can be conveniently retrieved as a String calling CacheManager's getLocalFilePath() static method, by any other component. File operations require no particular coding, since Android uses the standard java.io.File class as an "abstract" representation of a file system entity identified by a pathname. Reading and writing to external storage requires the WRITE_EXTERNAL_STORAGE permission to be declared in the AndroidManifest file.

There are three main issues to take care of when using Android external storage to store files [58]:

- Before doing any work with the external storage, Environment.getExternalStorageState() should always be called before to check whether the media is available. The media might be mounted to a computer, missing, read-only, or in some other state;

- There's no security enforced upon files saved to the external storage. All applications can read and write files placed on the external storage and the user can remove them;

- These files will not be deleted when the application is uninstalled.

Notwithstanding, the use of external storage allows the user to more easily access public files from other applications and to transfer these files quickly on another device or computer.

A very important aspect about the public cache is the detection of modifications to the cached files. Liferay Safe must be able to detect when a cached file has been changed and subsequently upload the new version to the server. Modifications include also renaming and moving a file.

Fortunately Android provides the FileObserver as a convenient way to detect these events. A FileObserver monitors files to fire an event after files are accessed or changed by any process on the device. Each FileObserver instance monitors a single file or directory. If a directory is monitored, events will be triggered for all files and subdirectories inside the monitored one.

The monitoring mechanism is based on inotify, a Linux kernel feature that extends file systems to notice changes and report them to applications (an interesting introduction can be found here: [59]). The main drawback about inotify is that it does not support recursively watching directories, meaning that a separate inotify watch must be created for every subdirectory.

Additionally, to ensure keeping a FileObserver sending events and avoiding the risk of garbage collection, it's necessary to hold a reference to the instance from some other live object.

For the above reasons, the public cache architecture includes a FileObserverService which manages a list of FileObserver's. Each FileObserver is associated to a single file in the cache, but only the files having the keepInSync property set to true in their corresponding entry stored in the ContentProvider are monitored by a FileObserver.

A user can either decide to download a file content for a single visualization or explicitly request the synchronizer to keep the file in sync (both-ways with the server) after the first download. In this case, modifications on the file will be monitored by a FileObserver which will start the UploaderService for any event of change detected.

The FileObserverService also manages a list of broadcast receivers to receive notifications from the DownloaderService when the download of a file has been completed. This is necessary for a file that needs to be synced before a FileObserver starts to monitor its changes.

Below are reported the class definitions for both the FileObserverService and the implementation of the abstract FileObserver.

Listing 6.3: FileObserver service

```
1  public class FileObserverService extends Service {
2
3      public final static String EXTRA_CMD = "EXTRA_CMD";
4      public final static String EXTRA_PATH = "EXTRA_FILE_PATH";
5
6      public final static int INIT_OBSERVED_LIST = 1;
7      public final static int ADD_OBSERVED_FILE = 2;
```

```
8       public final static int RM_OBSERVED_FILE = 3;
9
10      private static List<CacheFileObserver> mObservers;
11      private static List<DownloadCompletedReceiver> mDownloadReceivers;
12      private static Object mReceiverListLock = new Object();
13
14      @Override
15      public int onStartCommand(Intent intent, int flags, int startId) {
16          if (intent == null) {
17              initializeObservedList();
18              return Service.START_STICKY;
19          }
20
21          if (!intent.hasExtra(EXTRA_CMD)) {
22              return Service.START_STICKY;
23          }
24
25          switch (intent.getIntExtra(EXTRA_CMD, -1)) {
26              case INIT_OBSERVED_LIST:
27                  initializeObservedList();
28                  break;
29              case ADD_OBSERVED_FILE:
30                  addObservedFile(intent.getStringExtra(EXTRA_FILE_PATH));
31                  break;
32              case DEL_OBSERVED_FILE:
33                  removeObservedFile(intent.getStringExtra(EXTRA_FILE_PATH));
34                  break;
35          }
36
37          return Service.START_STICKY;
38      }
39
40      private void initializeObservedList() {
41
42          /*
43           * 1. Instantiate mObservers and mDownloadReceivers.
44           * 2. Retrieve file entries having keepInSync = true.
45           * 3. For each entry, get the absolute path and check if file exists.
46           * 4. If yes, create a new CacheFileObserver and set the target file.
47           * 5. Call startWatching() to start the observer.
48           */
49      }
50
51      private void addObservedFile(String path) {
52       /*
53             *  1. Check if an observer for this path already exists.
54             *  2. If not, retrieve the file entry for this path and create
55             *  a CacheFileObserver.
```

```
56              * 3. And the observer to the list of observers.
57              * 4. Register a new DownloadCompleteReceiver for this file.
58              */
59      }
60
61      private void removeObservedFile(String path) {
62              /* 1. Find the observer of this path in the list.
63                  * 2. Call stopWatching() to stop the monitoring.
64          * 3. Remove the observer from the list.
65              */
66      }
67
68      private static void addReceiverToList(DownloadCompletedReceiver r) {
69          synchronized(mReceiverListLock) {
70              mDownloadReceivers.add(r);
71          }
72      }
73
74      private static void removeReceiverFromList(DownloadCompletedReceiver r) {
75          synchronized(mReceiverListLock) {
76              mDownloadReceivers.remove(r);
77          }
78      }
79
80      private class DownloadCompletedReceiver extends BroadcastReceiver {
81          String mPath;
82          CacheFileObserver mObserver;
83
84          public DownloadCompletedReceiver(String path, CacheFileObserver observer) {
85              mPath = path;
86              mObserver = observer;
87              addReceiverToList(this);
88          }
89
90          @Override
91          public void onReceive(Context context, Intent intent) {
92              if (mPath.equals(intent.getStringExtra(FileDownloader.EXTRA_FILE_PATH))) {
93                  context.unregisterReceiver(this);
94                  removeReceiverFromList(this);
95                  mObserver.startWatching();
96              }
97          }
98
99                  @Override
100         public boolean equals(Object o) {
101             if (o instanceof DownloadCompletedReceiver)
102                 return mPath.equals(((DownloadCompletedReceiver)o).mPath);
103             return super.equals(o);
```

```
104             }
105         }
106 }
```

Listing 6.4: FileObserver subclass

```
1  public class CacheFileObserver extends FileObserver {
2
3      public static int CHANGES = CLOSE_WRITE | MOVED_FROM | MODIFY;
4
5      private String mFilePath;
6      private int mMask;
7      Account mAccount;
8      DLFile mDLFile;
9      static Context mContext;
10
11     public CacheFileObserver(String path) {
12         this(path, ALL_EVENTS);
13     }
14
15     public CacheFileObserver(String path, int mask) {
16         super(path, mask);
17         mPath = path;
18         mMask = mask;
19     }
20
21     public void setAccount(Account account) {
22         mAccount = account;
23     }
24
25     public void setDLFile(DLFile dlFile) {
26         mDLFile = dlFile;
27     }
28
29     public void setContext(Context context) {
30         mContext = context;
31     }
32
33     public String getPath() {
34         return mPath;
35     }
36
37     @Override
38     public void onEvent(int event, String path) {
```

```
39          if ((event | mMask) == 0) {
40              return;
41          }
42          Intent i = new Intent(mContext, FileUploader.class);
43          i.putExtra(FileUploader.KEY_ACCOUNT, mAccount);
44          i.putExtra(FileUploader.KEY_DLFILE, mDLFile);
45          i.putExtra(FileUploader.KEY_FILE_PATH, mFilePath);
46          mContext.startService(i);
47      }
48  }
```

Note also that the CacheFileObserver's event mask is specifically set as CLOSE_WRITE — MOVED_FROM — MODIFY to report exactly those event type changes.

In order to correctly watch a file, every FileObserver must be started immediately after the device has booted and should continue to detect any changes as long as the system is running.

To cope with that, a BootupBroadcastReceiver is used to receive from the system the Intent.ACTION_BOOT_COMPLETED broadcast and subsequently start the FileObserverService:

Listing 6.5: BootupBroadcastReceiver

```
1   public class BootupBroadcastReceiver extends BroadcastReceiver {
2
3       private static String TAG = "BootupBroadcastReceiver";
4
5       @Override
6       public void onReceive(Context context, Intent intent) {
7           if (!intent.getAction().equals(Intent.ACTION_BOOT_COMPLETED)) {
8               return;
9           }
10          Intent i = new Intent(context, FileObserverService.class);
11          i.putExtra(FileObserverService.EXTRA_CMD,
12                      FileObserverService.INIT_OBSERVED_LIST);
13          context.startService(i);
14      }
15  }
```

Where the BootupBroadcastReceiver is registered in the AndroidManifest file, along with the required permission declaration:

Listing 6.6: Registering the bootup broadcast receiver

```
1 <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
2 ...
3 <receiver android:name=".receivers.BootupBroadcastReceiver">
4 <intent-filter>
5         <action android:name="android.intent.action.BOOT_COMPLETED"/>
6        </intent-filter>
7 </receiver>
8 <service android:name=".cache.FileObserverService"/>
9 ...
```

Furthermore, we make the service return Service.START_STICKY from onStartCommand() to tell the system to keep the service running in the background. However, the system can still decide to kill the service in certain low memory conditions, and therefore all FileObserver's must be reinitialized when the it is restarted.

Finally, when a user sets a file to be kept in sync, the FileObserverService is started with the following intent, passing the absolute path to the cached file.

Listing 6.7: FileObserver service start intent

```
1 Intent intent = new Intent(getActivity().getApplicationContext(),
2 FileObserverService.class);
3 intent.putExtra(FileObserverService.EXTRA_CMD,
4 (dlFile.getKeepInSyn() ?
5             FileObserverService.ADD_OBSERVED_FILE:
6             FileObserverService.RM_OBSERVED_FILE));
7 intent.putExtra(FileObserverService.EXTRA_FILE_PATH, CacheManager.getLocalFilePath(dlFile));
8 startService(intent);
```

6.4 Encrypted file cache

Liferay Safe's encrypted cache is based on the IOCipher library developed by The Guardian Project group [70], and released under the LGPL 2.1 license. IOCipher brings app-level virtual encrypted disks to Android applications without requiring the device to be rooted. It uses a clone of the standard java.io API for working with files in a secure and transparent way.

To use IOCipher encrypted storage, there are only two things the developer needs to do: manage the password and mount the volume using VirtualFileSystem.mount(); and replace the relevant java.io import statements with info.guardianproject.iocipher. All in all using IOCipher is in fact very simple, as the following example demonstrates:

Listing 6.8: Virtual Encrypted Disk operations

```
// Create the virtual encrypted disk in the app's external storage
java.io.File db = new java.io.File(mContext.getExternalFilesDir(null),
 ''liferay-safe.db'');

VirtualFileSystem vfs = new VirtualFileSystem(db.getAbsolutePath());

// Mount the virtual file system
vfs.mount();

// Get the root of the vfs
info.guardianproject.iocipher.File ROOT = new File(''/'');

// Create a new file
info.guardianproject.iocipher.File f = new File(randomFileName("testFile"));

// Write to the new file
info.guardianproject.iocipher.FileOutputStream out = new FileOutputStream(f);
out.write(123);
out.close();

// Unmount the vfs
vfs.unmount();
```

IOCipher is built on top two important libraries: SQLCipher and libsqlfs.

6.4.1 SQLCipher

SQLCipher is an SQLite extension that provides transparent, secure 256-bit AES encryption of database files. It is an open source project, sponsored and maintained by Zetetic LLC, which has been adopted by many organizations, making it one of the most popular encrypted database platforms for mobile, embedded and desktop applications.

As claimed by The Guardian Project group [71] - who maintains the Android porting of SQLCipher - the data stored in this type of encrypted databases will be less vulnerable to access by malicious apps, protected in case of device loss or theft, and highly resistant to mobile data forensics tools.

SQLCipher's encryption is transparent and on-the-fly. Applications use the standard SQLite API to manipulate tables using SQL, while behind the scenes the library silently manages the security. SQLCipher does not implement its own encryption. Instead it uses OpenSSL libcrypto for all cryptographic functions.

The main security features of SQLCipher are [72]:

- The default algorithm is 256-bit AES in CBC mode.
- Each database page is encrypted and decrypted individually.

- Each page has it's own random initialization vector, generated by OpenSSL's RAND_bytes.
- IVs are regenerated on write so that the same IV is not reused on subsequent writes of the same page data.
- Every page write includes a Message Authentication Code (HMAC_SHA1) of the ciphertext and the initialization vector at the end of the page.
- The MAC is checked when the page is read back from disk.
- When initialized with a passphrase SQLCipher derives the key data using PBKDF2 (OpenSSL's PKCS5_PBKDF2_HMAC_SHA1), with a default number of 4000 iterations.
- Each database is initialized with a unique random salt in the first 16 bytes of the file, used for key derivation.
- The key used to calculate page HMACs is different that the encryption key. It is derived from the encryption key and using PBKDF2 with 2 iterations and a variation of the random database salt.
- When encrypted, the entire database file appears to contain random data.

6.4.2 Libsqlfs

Created as part of PalmSource's ALP mobile platform, the libsqlfs library implements a POSIX style file system on top of an SQLite database. It allows applications to have access to a full read/write file system in a single file, complete with its own file hierarchy and name space, without using SQL statements [73].

Libsqlfs can accommodate small preference values such as a number, and large binary objects such as an video clip. The library provides a generic file system layer that maps a file system onto a SQLite database, and supports a POSIX file system semantics.

IOCipher wraps the SQLite encryption features of SQLCipher on top of the SQLite-based file system implemented in the libsqlfs library, providing a secure transparent app-level virtual encrypted disk, with few limitations:

- files cannot be larger than the available RAM on the device;
- no users, groups, or permissions implemented;
- folder renaming is not supported (by libsqlfs);
- crashes possible under extremely heavy, concurrent load.

Currently The Guardian Project is working side to side with Zetetic to improve the performance and stability of IOCipher. They have recently released some optimizations on the libsqlfs library giving it a huge improvement in read/write concurrency and performance [74].

6.4.3 CacheGuard service

Similarly to the CacheManager seen for the public cache, the component responsible for managing the virtual encrypted disk is the CacheGuard service. Just like the CacheManager, it takes care of opening, storing, deleting, renaming and moving files. But more importantly, it is responsible for preserving the privacy stored in the virtual file system (VFS).

Inside the VFS, private files are stored according to the remote directory structure, with respect to their repository membership. The VFS is abstracted by the libsqlfs library from a single SQLite database file encrypted with SQLCipher. This virtual encrypted disk is stored in the /Android/data/⟨package_name⟩/files path in the external filesystem - retrieved from the Context's getExternalFilesDir() method. The directories returned here are owned by the application, and their contents will be removed when the application is uninstalled.

There is no risk in storing the VFS in the external storage, since the underlying SQLite database once initialized with a key is always in an encrypted state.

Alternatively, Android's internal storage could have been used, where files are private to the application, and other applications (and the user) cannot access them. However, this storage space is very limited in most devices and cannot be used to store large files. Infact, when it comes short, other applications could stop functioning. Therefore, it's better to store the cache in the external storage to benefit of a larger and more versatile space.

The CacheGuard service is the only component communicating with the VFS. Other application components must bind to this service in order to store (when downloading) or retrieve (for viewing) any private file. During these operations a private file is never written to disk. The reasons thereof are twofold.

First of all the NAND Flash technology used in mobile devices and SD cards tends to hold onto data longer than traditional spinning hard drives [75]. This is due to the wear leveling technique used for enhancing the drive's longevity, which works by arranging data so that erasures and re-writes are distributed evenly across the memory. Therefore, to completely erase any written data from a NAND Flash memory, it's necessary to rewrite the full medium, which of course is not possible in real conditions.

Secondly, the external storage is often implemented as a removable storage media (eg. an SD card). If private files are written to this storage, even temporarily, when performing an operation, it is possible that the file is not removed at the end of the operation if a particular event occurred such as, for example, the storage medium was removed, the application or the system stopped responding, or the device was powered off.

Instead, a private file is written to a MemoryFile, both when being downloaded from the server, and when loaded from the VFS. MemoryFile is a wrapper for the Linux ashmem driver. Android Shared Memory (ashmem) is a component of the Android operating system that facilitates memory sharing and conservation [77].

A MemoryFile can be shared with other application components or applications that bind to the CacheGuard service, as described later in section 6.7.6. Ashmem has reference counting so that if many processes use the same area the area will not be removed until all processes has released it, the memory is also virtual and not physically contiguous.

The CacheGuard service is responsible for ensuring that private data is visible for the least amount of time necessary, in particular it must take care of:

- Mounting and unmounting the VFS;
- Zeroing out and closing a MemoryFile;
- Managing crashes of the VFS and errors gracefully;
- Controlling the cache usage and erase least recently used files.

Beyond that, the CacheGuard service is responsible also for managing the key used to encrypt the VFS and the OAuth access token necessary to interact with the server. In the next section we discuss exactly how such service component must be structured to perform these security-critical tasks.

6.5 CacheGuard encryption key management

The CacheGuard service is the most important component of the Liferay Safe application. Essentially CacheGuard is a *foreground service* that other components in the application necessarily have to bind in order to give the user access to the application, and to perform invocations of the remote JSON Web Services API. In particular the service has the following responsibilities:

- Management of the private cache (as seen in the previous section);
- Request of user authentication with the remote server;
- Management and security of the expiring access and request tokens provided by the server after user authentication;
- Generation, management and security of the encryption key for the private cache;
- Application lockout and cache erasure in case of user revocation or authorization expiration;
- Detection of anomalous variations in the system's state or properties.

The necessity of confining such important responsibilities in one single component is due to the lack of a *root of trust*. In fact, the server (or better, the enterprise) can trust neither their employees, nor the mobile devices used by the employees themselves. First because an employee can be revoked from the enterprise portal at any given time (e.g., for termination of the employment relationship), and second because mobile devices are not issued by the company and can easily be compromised. Furthermore, while some Android devices embed a Trusted Platform Module (TPM) called Secure Element, the access to this module is currently not allowed to third-party applications [78, 79]. Therefore the private files cached on a device must be always under the control of the server. Yet, since the user

can operate the client also in offline mode, the client turns out to be the last resort of trust when the server is not reachable. The problem is now that the entire client application must be safe in order to be trusted. However, by making the trusted component small enough, the problem of protecting a generic application is reduced to the problem of protecting a very specific, smaller piece of code. Hence, the security of the private cache depends on the security of the CacheGuard service.

6.5.1 Private cache encryption key generation

In order to understand how the service works, we should first understand how the key used to encrypt the virtual encrypted disk (VED) is generated. There are three constraints that must be observed.

1. Once the VED is initialized with a key, this cannot be changed without initializing the VED again, which means that the key must not be changed unless the VED file is erased.
2. The encryption key cannot be generated from a user password, because the user is not trusted, which means that the key must be generated randomly in the device.
3. The key must be tied to some information known by the server, and must expire after a fixed amount of time making the cache inaccessible by the user.

The resulting approach (depicted in Figure 6-3) is to initialize the VED with a randomly generated master key, then encrypt the master key with another key derived from the access token provided by the server, and finally store the encrypted master key in the external storage along with VED file.

The scheme is based on the 256-bit Advanced Encryption Standard (AES) cipher pro. The master key used to initialize/mount the VED, is generated with a size of 256 bits using a random generator (java.security.SecureRandom) based on the SHA1PRNG algorithm (which is currently the only PRNG algorithm available on Android), and seeded from an internal entropy source, such as /dev/urandom, which should be fairly unpredictable. The 256 bit AES encryption key is derived from the access token using the PBKDF2 key derivation function defined in PKCS#5. The standard is based on two concepts: salting and key stretching. Salting consists in using a random salt to protect the derived key from dictionary attacks, while key stretching consists in iterating the function a large number of times to make the key derivation computationally expensive. Actually, because the access token is already a random and unguessable number, salting is not necessary, however, the algorithm expects a salt and this is the reason it is used anyways. The pseudorandom function applied by PBKDF2 can be any cryptographic hash, cipher or HMAC. However, Android's Java Cryptography Extension (JCE) provider (Bouncy Castle) currently supports only HMAC-SHA1. The seed is generated from SecureRandom with a size of 512 bits, while the iteration count is fixed to 10K rounds. Finally, the AES cipher is initialized with the derived key and a random initialization vector (IV) generated with SecureRandom. Hence the encrypted master key is obtained from the cipher and can be stored in the external storage along with

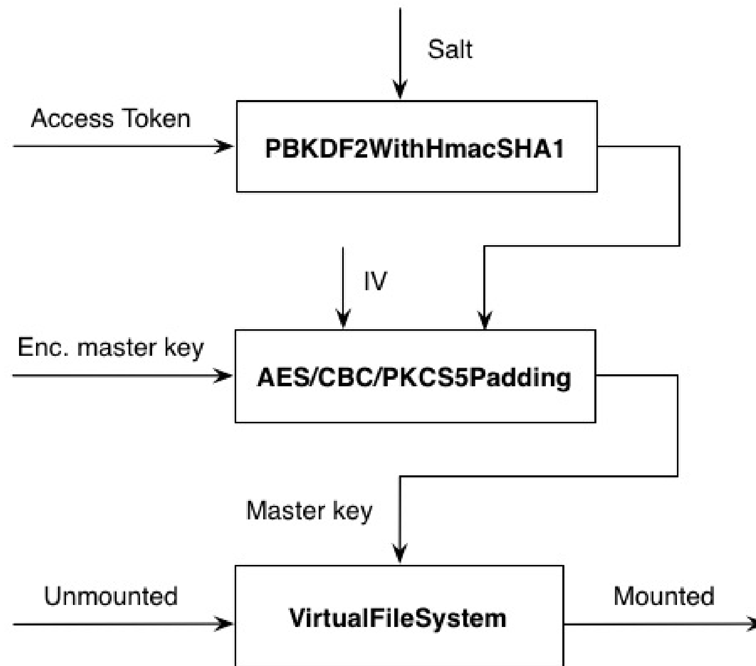


Figure 6-3: Cache encryption key generation scheme

the salt and the IV in a single blob (e.g., a Base64-encoded string) for later decryption. Just to make a comparison, the same cipher and derivation function are used by Android's backup system, with the same key strength, seed size and iteration count [80].

The initialization of the virtual encrypted disk can be summarized as follows:

1. Generate a random master key (256 bits).
2. Generate a salt (512 bits) to use for PBKDF2.
3. Generate a random initialization vector (IV).
4. Process the access token with PBKDF2 (PBKDF2WithHmacSHA1) and obtain the token derived key (256 bits, 10K rounds).
5. Setup the master key cipher (AES/CBC/PKCS5Padding) with the token derived key and the IV.
6. Encrypt the master key with the master key cipher.
7. Save the encrypted master key, the salt and the IV to storage.
8. (Initialize the VFS with the master key)
9. (Destroy master key copy in memory)

Mounting the virtual encrypted disk happens as follows, where steps 2-3 are identical to generation-mode steps 4-5:

1. Read encrypted master key, salt and IV from storage.
2. Process the token with PBDKF2 and obtain the token derived key.
3. Setup the master key cipher with the token derived key and IV.
4. Decrypt the encrypted master key with the master key cipher.
5. Mount the VFS with the master key.
6. (Destroy the master key copy in memory)

As a general rule, the encrypted cache can only be accessed as far as the access token is still valid, or a new one can be requested with a refresh token. When the access token expires, the master key can be recovered as shown above, and it can be re-encrypted using the new access token issued by the server, derived with PBKDF2.

To allow the user to access the encrypted cache when there is no network connectivity, the access token has a fixed duration of 24 hours. If at the moment of the expiration the server is reachable, a new access token can be requested using the refresh token, without needing user authentication. Otherwise, the expired tokens will be completely erased and the current cache won't ever be accessible, and for this reason it can be safely erased along with the encrypted master key. Additionally, to ease the token refresh and avoid losing the cache in vain, CacheGuard can repeatedly attempt to refresh the token already one hour before the expiration (as soon as the connection is available). At the same time, the service can also alert the user about the imminent expiration with a notification. Therefore, the encrypted cache lives as long as a new access token can be retrieved within every 24 hours, and as long as CacheGuard isn't killed making the key unrecoverable. The service can be killed by the system, in extreme low memory conditions, or by the user, intentionally (via system settings or a task manager application), or when the device is powered off or rebooted.

As a consequence of operating offline, CacheGuard must be able to detect autonomously when the expiration time is over. Obviously this cannot be done based on the value returned by System.currentTimeMillis(), because this can be modified by the user. Fortunately, Android provides the Alarm Manager as a convenient way to register an alarm based on the value of SystemClock.elapsedRealtime(), which is the time since the system was booted, including deep sleep. Therefore, the CacheGuard can register an alarm broadcast receiver and have its startService() method called whenever the alarm is triggered, allowing the service to check the token expiration. The interesting thing about the Alarm Manager is that registered alarms can optionally wake up device if they go off while the device is asleep.

In fact, the Alarm Manager holds a CPU wake lock as long as the alarm receiver's onReceive() method is executing. This guarantees that the phone will not sleep until the receiver has finished handling the broadcast. However, it is still possible that the phone will sleep before the service is invoked from the receive. To prevent this, the alarm BroadcastReceiver and CacheGuard must implement a separate wake lock policy to ensure that the CPU continues running until the service becomes available. A solution suggested by Murphy from

the CommonsWare group [81] consists in arranging for a new wake lock overlapping the Alarm Manager's. The new wake lock is acquired in the BroadcastReceiver, during the onReceive() execution, and is hold until the work is completed by the CacheGuard service. The greatest benefit in using the Alarm Manager comes at battery life. Think that only keeping the CPU awake by holding a wake lock indefinitely without performing any work, the battery life of a Samsung Galaxy Nexus was reduced by over 40%.

Besides having an expiration, an access token can also be revoked explicitly by the server. In this case CacheGuard must erase the revoked access token as soon as possible. This can be done in two ways. The first possibility is to contact the server periodically, from CacheGuard, and check the token validity. The second possibility, which is more efficient, is to use the Google Cloud Messaging API and make the server promptly push-notify the client about the revocation. Of course, the token revocation is effective only if there is an Internet connection available.

6.5.2 Vulnerabillities

Clearly at this point, the security of the encrypted cache depends on the security of the access token. Both the access token and refresh token provided by the server when the user authenticates (from the AuthenticatorActivity embedding a WebView), are managed exclusively by CacheGuard, which retains them securely in memory until these have expired or the service is killed.

Obviously an attacker could search for the access token in a (volatile) memory dump - using any memory analysis tools such as Eclipse MAT - and then use this value anytime later to decrypt the cache. However, acquisition of physical memory requires gaining root privileges on the device so that code can be loaded into the OS kernel to read and export a copy of physical memory [67]. Even more easily, with ADB enabled and having root privileges, the attacker can send a SIGUSR1 signal (kill -10) to the process running CacheGuard and cause it to dump its memory in the data/misc directory [82]. Moreover, having root privileges not only means the attacker can directly read and write the application's memory, but can also control flow of execution. The way to do this would be to attach to the app's process using ptrace and use /proc/pid/map to get the memory addresses to access [82].

Vidas et al. [11] devised a taxonomy of attacks to the Android platform that guarantee privileged access to the device. Figure 6-4 shows how an attacker - who is not the owner of the device - could rely on this taxonomy to decide which attack path to pursue, given their own capabilities.

Based on the same taxonomy we can define a mitigation strategy to contrast the attacker performing different preliminary exploitations to execute a memory dump:

- **Complete access to the device**: If the device user has decided to not employ any kind of obstruction, the attacker can perform different types of exploitations to gain root privileges.
  - *Mitigation:* Enforce the use of a password screen lock. Lock down the application until a screen lock isn't enabled, erase any existent cache. Check routinely for

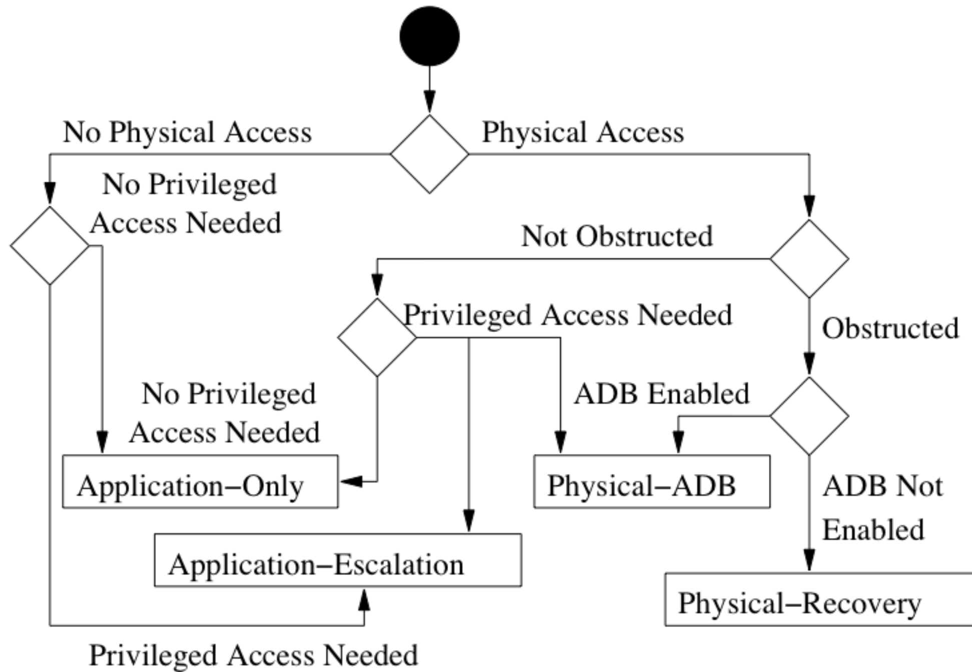


Figure 6-4: Android attack chart

variations.

- **Exploit via Android Debug Bridge**: If the attacker finds a device left unattended, yet obstructed via a password or screen lock, the attacker may be able to exploit the device through the Android Developer Bridge.
  - *Mitigation:* Enforce the disabling of the "USB debugging" setting under the "developer options" section of the system's settings application. Lock down the application until this setting isn't disabled, erase any existent cache. Check routinely for variations.
- **Exploit via Recovery Boot**: If the attacker finds an obstructed Android device left unattended, but is unable to use the ADB service, the attacker may still gain privileged access via recovery boot.
  - *Mitigation:* The exploit requires flashing the recovery partition with a custom ROM, which means the device needs to be rebooted making the cache inaccessible. However, it's not guaranteed that the access token is cleared from memory after the reboot (because the RAM can remember the last stored state [85]), thus it might still be recovered from a full memory dump. But this is assumed to be very unlikely.
- **Malicious applications**: An attacker may rely on convincing the user to install a malicious application. Such an application may present an enticing feature to the

consumer but contain code that executes a privilege escalation attack, or a memory corruption attack - such as a return-oriented programming (ROP) attack or a GOT overwrite attack. A remote attack may not even require the installation of a new application. Android's use of commodity software components, such as the web browser and Linux base can be leveraged for an attack that requires no physical access.

- *Mitigation:* This risk is mitigated by requiring a minimum version of Android corresponding to Jelly Bean (4.1). In fact, in Android Jelly Bean, Address Space Layout Randomization (ASLR) is implemented fully, protecting stack, heap/brk, lib/mmap, linker, and executable from such exploits [65].

Note that the attacker involved in these scenarios is not the owner of the device, however, there could be the case of a revoked user, owner of the device, trying to recover the contents of the cache before its expiration. This risk is mitigated by assuming that such user does not have sufficient skills to perform the memory dump and recover the access in less than 24 hours, after the revocation of his account.

If the application is installed on a device which has been already rooted, there isn't so much that can be done to guarantee the protection of the cache because there are multiple areas where the system might be compromised. There are a number of checks that can be performed to detect whether a device has been rooted, and most of the times these work when the exploit is performed with commonly available tools such as SuperOneClick. Nonetheless, it would be best to have a device examined by a security officer before installing the application.

In Figure 6-5 is summarized how the encrypted cache is protected by CacheGuard service upon the occurrence of the various events discussed so far.

Since the access token is retained in memory by CacheGuard for a long period of time (24 hours), its value needs to be obfuscated so that it is considerably harder (although still not impossible) for an attacker to find it when analyzing a memory dump. The in-memory obfuscation can be achieved using an hardcoded function that performs a combination of operations over the original string value in order to produce a larger and erasable byte array. The operations can include, for example, substituting char bytes with hardcoded integer values, byte splitting, scrambling, etc. Whenever the token needs to be used, the string is de-obfuscated, still with an hardcoded function, and stored in a short-lived temporary char or byte array.

The reason this value isn't stored in a temporary String, is that Strings are immutable and as such cannot be zeroed out once the value is no longer needed [83]. Primitive array types instead can be zeroized (as soon as no longer needed) and have also the benefit of being passed by reference, which avoids creating multiple copies in memory. However, the access token must still be converted to a String when setting the HTTP request header, and the httpclient library might create multiple copies of this value when executing the method. Therefore the plain access token cannot be effectively zeroed out and will stick around in multiple places of the memory until the garbage collector has passed. Fortunately, the Dalvik VM implements a mark-and-sweep garbage collector [84], which, in contrast to

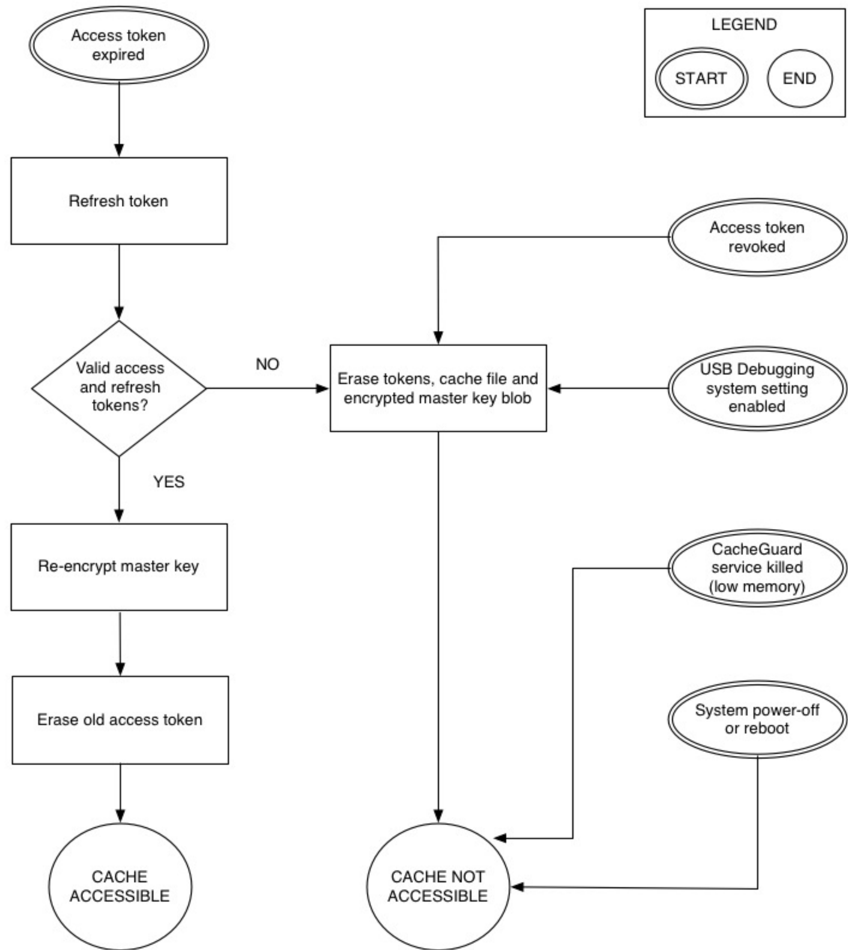


Figure 6-5: Encrypted cache accessibility and protection

generational GCs used by most virtual machines, does not involve copying objects between heap spaces.

In addition to in-memory obfuscation, the access token can be furtherly protected by periodically reordering the obfuscated bytes. This allows to reduce the problem of RAM (in)volatility [85] and to countermeasure a memory dump analysis attack [86].

It's also important to obfuscate all the application code with ProGuard, as suggested by Google, to make the code more difficult to reverse engineer. If ProGuard is not sufficient, critical portions of code must be obfuscated separately using other tools.

6.5.3 Lifecycle and operations

The service is started as a "foreground service" as soon as the application is launched the first time, and then, every time the systems has booted by means of a BOOT_COMPLETED broadcast receiver. A foreground service is a service that's considered to be something the user is actively aware of and thus not a candidate for the system to kill when low on memory. This allows the CacheGuard service to be an "everlasting service" without needing to acquire a power-expensive wake lock.

At the first start there is no valid access token available, therefore the service sits in idle until the user has completed the authentication process. After obtaining the necessary access and refresh tokens, CacheGuard performs a number of periodic operations in the following order of time (based on an estimate of the time required to carry out the attacks seen in the previous section):

- **Every 5 minutes** (only when device is in use): Check if the "USB debugging" setting is enabled.

- **Every hour**: If the server is reachable, verify the access token. Otherwise check if the token has expired.

- **1 hour before token expiration**: Attempt to acquire a new access token using the refresh token. If it succeeds, re-encrypt the cache master key and reset the Alarm Manager. Otherwise retry until expiration.

- **When token has expired**: Perform last refresh attempt. If it fails, erase the tokens and the encrypted cache.

Furthermore, when CacheGuard starts, it makes an attempt to identify if the system is compromised. In this regard, besides checking if USB debug mode is enabled, a number of system tests are performed to check if malicious applications are installed and to find clues about whether the device is rooted or not. The RootTools library [87] provides a collection of root-checks used by many common applications. Although it is not possible to detect if the devices is rooted in a perfectly accurate way, most of the time these checks are sufficient to defeat the malicious intent (or unawareness) of an unexperienced user.

6.5.4 Binder interfaces

CacheGuard is not an exported service, so components of other applications can't either invoke the service or interact with it. Instead, only components of the same application or applications with the same user ID can start the service or bind to it. In fact, the service provides two different binder interfaces. A local binder interface exposes the methods to retrieve the access token needed by the HTTP client to set the Authorization header when invoking the remote API. Other methods include operations related to the encrypted cache, such as the storage of a private document downloaded by the DowloaderService, as well as the delete/rename/move operations needed to keep the encrypted cache in sync with the remote server. Additionally, the local interface exposes a method to allow other components to check whether the user needs to authenticated, and eventually start the AuthenticatorActivity.

The service can be bound through the local interface by several components at the same time:

- Activities bind to check if the user has to be authenticated;
- Downloader service binds to download private files to the encrypted cache;
- SyncAdapter binds to get the access token needed to initialize the LiferayDLClient.

These components must unbind the service as soon as it's no longer needed. Additionally, these components must take care of completely erasing any sensitive information provided by CacheGuard, which must never be stored as immutable data types.

Finally, CacheGuard provides a remote interface written in AIDL, which provides only a method for reading private files. This interface is used only by trusted applications signed with the same developer key and running under the same user ID. Additionally, for security and performance reasons the service cannot be bound through this interface more than once at the same time.

6.6 Viewing and sharing files

Liferay Safe allows the user to perform different operations on cached files depending on whether a file is marked as "confidential". In particular:

- Public files - can be shared with any other application (eg. an email client), both from within Liferay Safe with an intent, or directly from the desired application by browsing the external storage. Any application can access a public file either for reading or writing. The user can also take screenshots and copy parts of text. The content of the public cache can also be accessed from another computer that mounts the external storage.
- Private files - cannot be shared with other third party or system applications and cannot be accessed for writing. Only trusted applications can read from private files.

Additionally, both the screen capture and the clipboard are disabled, and the content of the private cache cannot be mounted from other computers.

To open a public file in another application from within Liferay Safe it's sufficient to create an intent with specified the Intent.ACTION_VIEW action, the file's path Uri and MIME type, like in the following example:

Listing 6.9: Opening a public file in other applications

```
String filePath = CacheManager.getLocalFilePath(dlFile);
String fileType = null;
String extension = MimeTypeMap.getFileExtensionFromUrl(filePath);
if (extension != null) {
        MimeTypeMap mime = MimeTypeMap.getSingleton();
        fileType = mime.getMimeTypeFromExtension(extension);
}


Intent i = new Intent(Intent.ACTION_VIEW);

i.setDataAndType(Uri.parse("file://" + filePath), fileType);
i.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION |
Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
startActivity(i);
```

The standard activity picker will be displayed by the system listing all installed applications whose intent filters match the intent's properties, allowing the user to pick the desired application before proceeding. The selected activity will be started and pushed on top of the activities backstack, so that when the user presses the back button the Liferay Safe screen that had left is restored.

Alternatively the user can access the entire public cache with any file browser application available free on the Google Play store.

Dealing with private files is a lot more different and difficult. We assumed initially that Liferay Safe must provide a secure way to allow the user to view these files at least in PDF format. Still, building a PDF viewer for Android is not straightforward, since there are no system components providing this functionality.

Fortunately, there are a number of open source applications available that could be modified and embedded in Liferay Safe, such as DroidReader (`http://code.google.com/p/droidreader/`) and EBookDroid (`http://code.google.com/p/ebookdroid/`). Many of these are based on MuPDF library (`http://www.mupdf.com/`), which provides accurate and fast PDF rendering.

In spite this, being the rendering library licensed under GNU GPL v3, any derived work must be licensed, as a whole, under the same license (subsection 5c).

Since this could not be compatible with Liferay Safe's licensing, instead the open source code could be modified in a separate application, called Liferay Safe Viewer, and be released independently under the GPL v3 license.

The open source code of the viewer will have to be modified in order to communicate with Liferay Safe to read files from the private cache, and to ensure that any functionality that could compromise the privacy of the shared data is completely blocked or removed - such as opening the file in other applications, capturing screenshots or copying text.

Additionally the viewer application must guarantee that any opened file (or parts of it) is never written to disk, and that any allocated volatile memory is correctly zeroized when the file is closed - leaving no traces of the unencrypted file around.

To make this possible, we rely on Android's Binder Inter-process Communication. An intent from Liferay Safe will start the viewer's activity passing as an extra the id of the private file to be opened. The viewer will then bind to the CacheGuard service using an AIDL interface for remote communication (since the two applications run on different processes), and request the service to open the file. CacheGuard will instantiate a file stream in the shared memory as a MemoryFile and return it directly to the viewer, which can then execute the rendering.

No data is written to disk during all these steps. The interesting thing about using the shared memory (ashmem) is that the allocated space is never counted in the VM budget, therefore as much space can be allocated as the amount available in memory without incurring in OutOfMemory errors.

To allow the viewer to bind CacheGuard without needing to declare an IntentFilter for the service in the AndroidManifest file and making it exported, it is necessary to run both applications under the same Linux User ID. For this purpose we can use the sharedUserId attribute in the AndroidManifest.xml's manifest tag of each package to have them assigned the same user ID.

Only two applications signed with the same signature (and requesting the same sharedUserId) will be given the same user ID. The two packages are then treated as being the same application, with the file permissions and user ID.

The certificate with which an application is digitally signed, and whose private key is held by the application's developer, is used by the Android system as a means of identifying the author of an application and establishing trust relationships between applications. In this way, it's not necessary to define a specific permission for binding the CacheGuard from the viewer - to avoid impersonation - because the identity of the application is already ensured by the application signing.

The viewer can then bind to the service using the AIDL interface as in the following example:

Listing 6.10: Binding to CacheGuard from the PDF viewer application

```
1 interface CacheGuard {
2         void openCachedFile(long fileId);
```

```
3  }
4
5  public class PDFViewer extends Activity implements OnFileReadyListener{
6          private CacheGuard service = null;
7
8          private ServiceConnection svcConn=new ServiceConnection() {
9          public void onServiceConnected(ComponentName className,
10                                         IBinder binder) {
11                 service=CacheGuard.Stub.asInterface(binder);
12         }
13
14         public void onServiceDisconnected(ComponentName className) {
15                 service=null;
16         }
17         };
18
19         @Override
20         public void onCreate(Bundle icicle) {
21                 super.onCreate(icicle);
22
23                 bindService(new Intent(CacheGuard.getName(),
24                         svcConn, Context.BIND_AUTO_CREATE);
25
26                 Intent i = getIntent();
27                 service.openCachedFile(i.getIntExtra(FILE_ID, -1);
28         }
29
30         @Override
31         public void onFileReady(MemoryFile result) {
32                 InputStream input;
33                 if (result != null) {
34                         input = result.getInputStream();
35                 }
36                 // Render the file
37         }
38
39         @Override
40         public void onDestroy() {
41                 super.onDestroy();
42
43                 unbindService(svcConn);
44         }
45 }
```

A benefit from running the PDF rendering in another process is that this operation can be often resource intensive and result in a crash. If the viewer was running in the same

process of Liferay Safe, and consequently in the same process of CacheGuard, a crash of the viewer would cause a restart of the service and access to private cache would be lost.

Finally, Liferay Safe Viewer should not have a launcher activity specified in the AndroidManifest.xml, since this application is not meant to be opened directly from the user but it can only be started from Liferay Safe.

Chapter 7

Conclusion and future work

Companies around the world are exposed to increasing security risks due to a variety of consumer-grade technology brought into the enterprise by the employees and inevitably used for work-related activities. As mobility and workplace flexibility transform the way today's organizations operate, consumer mobile devices have raised in popularity within the enterprise creating a new trend called "consumerization of IT", which includes BYOD [88].

More and more organizations are considering bring-your-own-device (BYOD) initiatives [1], which encourage employees to bring their personal devices to work in order to increase mobility and productivity, while also reducing the costs of device ownership and management [89].

Bolstering this trend is the increasing popularity of Android devices, which has rapidly pushed the mobile operating system to the leading position in the global smartphone market [6]. Over one million of new Android smartphones and tablets are activated every day [17]. As an open platform with roots in open-source software, Android has become a key operating system in the pursuit for increasing mobility and productivity. Features and benefits that are targeted at both consumers and enterprises make Android an attractive choice for both individuals and organizations.

However, for organizations to realize the full value of the BYOD model, employees must be able to access and sync all their enterprise files and data on-demand and collaborate securely, from any device or location. When employees bring their own devices to work and use them to share files and data inside and outside the office, one major challenge must be addressed: *protecting and securing the privacy of sensitive data at all times while allowing unrestricted access to public data.*

Personally owned and managed mobile devices represent a threat to enterprise networks, including the potential for security gaps that expose confidential business information or sensitive data — not to mention possible damage caused by *malicious insiders*. Thus, information security becomes highly dependent on situational information, such as security of the device, its location, and the user, without ignoring also the network and the applications being used.

Organizations need solutions that can quickly and easily make existing applications accessible to tablets and smartphones, while still maintaining control of the ever evolving

array of mobile devices and consumer apps that proliferate with BYOD. Under these requirements, access to sensitive data from mobile devices can be allowed through "security containers", that enterprises can trust, providing mitigation of business risks surrounding the confidentiality, integrity and availability of resources. These solutions can include a full range of encryption, device control, and data loss prevention capabilities - such as password enforcement, remote lock and remote wipe - that make it easier to keep employee's private data and enterprise intellectual property safe - whenever accessed from mobile devices.

This work is, in fact, an attempt to build a "security container" for Android devices, capable of safely synchronizing documents with a Liferay Portal server, allowing caching of data for secure offline usage, and ensuring a proper user experience.

The main challenge faced in the design and implementation of this application (called Liferay Safe) was about providing offline access to private documents and ensuring at the same time their security. In fact, although Android was designed with a focus on security, its security model is inherently open to a number of attack vectors that expose the device to privilege escalation exploitations [10, 11], hindering the protection of data from theft and misuse. Therefore, it was inevitable to put assumptions and constraints that simplify the attacker model and restrict the usage of private data. Thereby, the resulting application is capable of protecting private data from malicious outsiders, and consequently from device theft or loss. However, when the attacker is the owner of the device, the application must rely on the timely application of enterprise policies that involve the provisioning and deprovisioning of employee devices and accounts. Hence, if the account is revoked, only an experienced employee, with enough time and required skills, can attempt to extract the private data from the application during the offline access lease.

Ultimately, the key problem of this challenge is the "lack of a root of trust", since the enterprise can trust neither its employees nor their own devices. Using a Trusted Platform Module (TPM) establishes a ground of truth on which device security could be built, providing secure management and storage of encryption keys, as well as mitigating the effects of certain attack vectors. Currently a number of Android devices already support a TPM (called the Secure Element), however access is not allowed to third party applications. Additionally, although latest versions of Android (e.g., ICS and JB) have made considerable improvements on the security front, with respect to other mobile platforms (i.e., iOS) the disparity is still wide. Important security features, including authenticated downloads, authenticated Android Debug Bridge, Secure Element access [11], and possibly device rooting detection, would provide a more solid ground for enterprises to build their applications and keep up with the BYOD trend. Hopefully these concerns will be addressed in Key Lime Pie, the next version of Android to be announced at Google I/O 2013.

Concluding with the future works, it would be interesting to submit the application code to a static information flow analysis tool to test the existence of privacy leaks. Moreover, the implementation could be optimized to obtain better responsiveness, lower battery usage and higher security, with less assumptions and constraints and more functionalities. Eventually, the application could be extended to support more use cases, becoming a holistic solution to bring secure mobility to Liferay Portal.

Bibliography and URLs

[1] *Security Issues to Escalate as 350m Employees to Use Personal Mobile Devices at work by 2014*, available at http://www.juniperresearch.com/viewpressrelease.php?pr=330

[2] Marc Blanchou, *Auditing Enterprise Class Applications and Secure Containers on Android*, iSEC Partners, 2012.

[3] *Anatomy of a Mobile Attack*, available at https://viaforensics.com/resources/reports/best-practices-ios-android-secure-mobile-development/mobile-security-primer/

[4] *Security in a BYOD Era*, available at https://viaforensics.com/mobile-security/security-byod-era-webinar-questions-answers.html

[5] *OWASP Mobile Security Project*, available at https://www.owasp.org/index.php/OWASP_Mobile_Security_Project

[6] *Android Marks Fourth Anniversary Since Launch with 75.0% Market Share in Third Quarter*, available at http://www.idc.com/getdoc.jsp?containerId=prUS23771812#.UQlAn-jfYQU

[7] *TrendLabs 3Q 2012 Security Roundup*, available at www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-3q-2012-security-roundup-android-under-siege-popularity-comes-at-a-price.pdf

[8] Jon Oberheide and Charlie Miller, *Dissecting the Android Bouncer*, SummerCon 2012.

[9] Adam P. Fuchs, Avik Chaudhuri, Jeffrey S. Foster, *SCanDroid: Automated Security Certification of Android Applications*, University of Maryland, College Park, 2009

[10] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Marcel Winandy, *Privilege Escalation Attacks on Android*, Ruhr-University Bochum, Germany, 2010

[11] Timothy Vidas, Daniel Votipka, Nicolas Christin, *All Your Droid Are Belong To Us: A Survey of Current Android Attacks*, CyLab, Carnegie Mellon University, 2011

[12] Erika Chin, Adrienne Porter Felt, Kate Greenwood, David Wagner, *Analyzing Inter-Application Communication in Android*, University of California, Berkeley, CA, USA, 2011

[13] Machigar Ongtang, Stephen McLaughlin, William Enck and Patrick McDaniel, *Semantically Rich Application-Centric Security in Android*, Pennsylvania State University, 2009

[14] William Enck, Machigar Ongtang, Patrick McDaniel, *Understanding Android Security*, Pennsylvania State University, 2009

[15] William Enck, Peter Gilbert, Byung-Gon-Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, Anmol N. Sheth, *TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones*, Proceedings of the 9th USENIX conference on Operating systems design and implementation, USENIX Association, 2010

[16] *NSA Releases a Security-enhanced Version of Android*, available at http://www.pcworld.com/article/248275/nsa_releases_a_securityenhanced_version_of_android.html

[17] https://plus.google.com/u/0/110023707389740934545/posts/R5YdRRyeTHM

[18] http://www.isuppli.com/Mobile-and-Wireless-Communications/News/pages/Cumulative-Android-Smartphone-Shipments-Will-Exceed-1-Billion-in-2013.aspx

[19] http://googleblog.blogspot.com/2007/11/ wheres-my-gphone.html

[20] *Android Kernel Features*, available at http://elinux.org/Android_Kernel_Features

[21] Reto Meier, *Professional Android 4 Application Development*, John Wiley & Sons, 2012

[22] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri, *A Study of Android Application Security*, The Pennsylvania State University, 2011

[23] Dan Bornstein, *Dalvik VM Internals*, Google I/O 2008, available at https://sites.google.com/site/io/dalvik-vm-internals/

[24] David Ehringer, *The Dalvik Virtual Machine Architecture*, 2010

[25] Jim Huang, *Understanding the Dalvik Virtual Machine*, 0xlab, 2012, available at http://0xlab.org/~jserv/tmp/dalvik.pdf

[26] *Android Security Overview*, available at http://source.android.com/tech/security/

[27] Jim Huang, *Android IPC Mechanism*, 0xlab, 2012, available at http://www.slideshare.net/jserv/android-ipc-mechanism

[28] Thorsten Schreiber, *Android Binder*, Ruhr University Bochum, 2011

[29] https://lkml.org/lkml/2009/6/25/3

[30] *App Components*, available at http://developer.android.com/guide/components/index.html

[31] Jesse Burns, *Developing Secure Mobile Applications for Android*, iSEC Partners, 2008

[32] Jeff Six, *Application Security for the Android Platform*, O'Reilly, 2012

[33] *Liferay Portal 6.1 - User Guide*, available at http://www.liferay.com/it/documentation/liferay-portal/6.1/user-guide

[34] Richard L. Sezov, Jr. and Stephen Kostas, *Liferay Portal Administrator's Guide*, Liferay, Inc. 2010

[35] *JSR 168: Portlet Specification*, available at http://www.jcp.org/en/jsr/detail?id=168

[36] Nikolay Elenkov, *Using a Custom Certificate Trust Store on Android*, available at http://nelenkov.blogspot.it/2011/12/using-custom-certificate-trust-store-on.html

[37] Enrico Cavallin, *Analisi del protocollo OAuth 2.0 e sua implementazione con Spring Security Framework*, DAIS, Universita' Ca' Foscari di Venezia, 2013

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

[39] *Writing an Android Sync Provider*, available at http://www.c99.org/2010/01/23/writing-an-android-sync-provider-part-1/

[40] *Connecting The Dots with Android SyncAdapter*, available at http://ericmiles.wordpress.com/2010/09/22/connecting-the-dots-with-android-syncadapter/

[41] *The revenge of the SyncAdapter: Synchronizing data in Android*, available at http://naked-code.blogspot.it/2011/05/revenge-of-syncadapter-synchronizing.html

[42] *Concept of SyncAdapter*, available at https://sites.google.com/site/andsamples/concept-of-syncadapter-androidcontentabstractthreadedsyncadapter

[43] *Security Issue Exposed by Android AccountManager*, available at http://security-n-tech.blogspot.it/2011/01/security-issue-exposed-by-android.html

[44] *Issue 10809: Password is stored on disk in plain text*, available at http://code.google.com/p/android/issues/detail?id=10809

[45] *Storing application secrets in Android's credential storage*, available at http://nelenkov.blogspot.it/2012/05/storing-application-secrets-in-androids.html

[46] *What should I use Android AccountManager for?*, available at http://stackoverflow.com/questions/2720315/what-should-i-use-android-accountmanager-for/8614699#8614699

[47] *Use Androids ContentObserver in Your Code to Listen to Data Changes*, available at http://www.grokkingandroid.com/use-contentobserver-to-listen-to-changes/

[48] *Obtaining Sync Status Information using SyncStatusObserver or by other means?*, available at http://stackoverflow.com/questions/7214142/obtaining-sync-status-information-using-syncstatusobserver-or-by-other-means

[49] *How to know when sync is finished?*, available at http://stackoverflow.com/questions/6622316/how-to-know-when-sync-is-finished

[50] *SyncAdapter questions: monitoring sync status; getting sync settings; sync icon*, available at https://groups.google.com/forum/?fromgroups=#!topic/android-developers/uT93EUsKXSA

[51] *How does one listen for progress from Android SyncAdapter?*, available at http://stackoverflow.com/questions/5268536/how-does-one-listen-for-progress-from-android-syncadapter

[52] http://developer.android.com/reference/android/content/BroadcastReceiver.html

[53] http://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html

[54] *Android sticky broadcast perils*, available at http://porcupineprogrammer.blogspot.it/2012/09/android-sticky-broadcast-perils.html

[55] http://code.google.com/p/google-gson/

[56] *Liferay Portal 6.1 - JSON Web Services*, available at http://www.liferay.com/documentation/liferay-portal/6.1/development/-/ai/json-web-services

[57] *Introducing JSON*, available at http://www.json.org

[58] *Using the External Storage*, available at http://developer.android.com/guide/topics/data/data-storage.html#filesExternal

[59] *Intro to inotify*, available at http://www.linuxjournal.com/article/8478

[60] Oleg Kalnichevski, *HttpClient Tutorial*, available at http://hc.apache.org/httpcomponents-client-ga/tutorial/html/connmgmt.html#d5e627

[61] *Proprietary Information Law & Legal Definition*, available at http://definitions.uslegal.com/p/proprietary-information/

[62] *Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995*, available at http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML

[63] *ISO (2005) 27001: Information Security Management Specification With Guidance for Use.*

[64] *Android Security Overview - Memory Management Security Enhancements*, available at http://source.android.com/tech/security/index.html#memory-management-security-enhancements

[65] Jon Oberheide, *Exploit Mitigations in Android Jelly Bean 4.1*, available at https://blog.duosecurity.com/2012/07/exploit-mitigations-in-android-jelly-bean-4-1/

[66] *How can you detect if the device is rooted in the app?*, available at http://stackoverflow.com/questions/3576989/how-can-you-detect-if-the-device-is-rooted-in-the-app

[67] J. Sylve, A. Case, L. Marziale, G. G. Richard, *Acquisition and analysis of volatile memory from Android devices*, University of New Orleans, 2012.

[68] A. M. de L. Simao, F. C. Sicoli, L. P. de Melo, R. T. de Sousa Junior, *Acquisition of digital evidence in Android smartphones*, University of Brasilia, 2011.

[69] *Root exploit on Exynos*, available at http://forum.xda-developers.com/showthread.php?t=2048511

[70] *IOCipher: Virtual Encrypted Disks*, available at https://guardianproject.info/code/iocipher/

[71] *SQLCipher: Encrypted Database*, available at https://guardianproject.info/code/sqlcipher/

[72] *SQLCipher - Design*, available at http://sqlcipher.net/design/

[73] http://www.nongnu.org/libsqlfs/

[74] *Report on IOCipher beta dev sprint*, available at https://guardianproject.info/2013/01/31/report-on-iocipher-beta-dev-sprint/

[75] *Security in a BYOD Era*, available at https://viaforensics.com/mobile-security/security-byod-era-webinar-questions-answers.html

[76] *Content Providers and Content Resolvers*, available at http://www.androiddesignpatterns.com/2012/06/content-resolvers-and-content-providers.html

[77] *An Introduction to Android Shared Memory*, available at http://notjustburritos.tumblr.com/post/21442138796/an-introduction-to-android-shared-memory

[78] Nikolay Elenkov, *Accessing the embedded secure element in Android 4.x*, available at http://nelenkov.blogspot.it/2012/08/accessing-embedded-secure-element-in.html

[79] Nikolay Elenkov, *Android secure element execution environment*, available at http://nelenkov.blogspot.it/2012/08/android-secure-element-execution.html

[80] Nikolay Elenkov, *Unpacking Android backups*, available at http://nelenkov.blogspot.it/2012/06/unpacking-android-backups.html

[81] Mark L. Murphy, *The Busy Coder's Guide to Advanced Android Development*, pp. 223-234, version 1.9.2, CommonsWare

[82] Thomas Cannon, *Android Reverse Engineering*, available at http://thomascannon.net/projects/android-reversing/

[83] http://docs.oracle.com/javase/1.5.0/docs/guide/security/jce/JCERefGuide.html#PBEEx

[84] https://android.googlesource.com/platform/dalvik/+/master/vm/alloc/MarkSweep.cpp

[85] Peter Gutman, *Secure Deletion of Data from Magnetic and Solid-State Memory*, University of Auckland, 1996

[86] Hyun Jun Jang, Dae Won Hwang, Eul Gyu Im, *Data Protection in Memory Using Byte Reordering*, Hanyang University, Seoul, Republik of Kora, 2008

[87] http://code.google.com/p/roottools/

[88] *Consumerization Survey Report*, available at http://www.trendmicro.com/cloud-content/us/pdfs/rpt_consumerization-survey-report.pdf

[89] *Key Strategies to Capture and Measure the Value of Consumerization of IT*, available at http://www.trendmicro.com/cloud-content/us/pdfs/business/white-papers/wp_forrester_measure-value-of-consumerization.pdf