

Johannes Anderwald

tagstore: A Mobile Tagging Application with Synchronization

Master's Thesis

Graz University of Technology

Institute for Softwaretechnology
Head: Univ.-Prof.Dipl-Ing.Dr.techn. Wolfgang Slany

Supervisor: Dipl-Ing. Karl Voit

Graz, October 2012

This document was written with T_EX Maker, is set in Palatino, compiled with pdfL^AT_EX2e and Biber.

The L^AT_EX template from Karl Voit is based on KOMA script and can be found online: <https://github.com/novoid/LaTeX-KOMA-template>

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____
Date Signature

Eidesstattliche Erklärung¹

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am _____
Datum Unterschrift

¹Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Zusammenfassung

In den letzten Jahren wurden viele Tagging Systeme entwickelt. Diese Systeme erlauben es, Schlüsselwörter einer Ressource zuzuordnen und diese anhand der Schlüsselwörter wiederzufinden. An der Technischen Universität Graz wurde am Institut für Softwaretechnologie eine Forschungssoftware namens tagstore entwickelt, die dieses Konzept nützt um Dateien zu klassifizieren. Im Rahmen dieser Masterarbeit wurde ein Prototyp für die mobile Plattform Android entwickelt. Die Masterarbeit beschreibt den entwickelten Prototypen. Zusätzlich wurde eine Analyse von existierenden Synchronisationssystemen durchgeführt. Ein Schwerpunkt der Analyse lag in den Cloud Computing Systemen. Des Weiteren wurde ein Algorithmus entwickelt, der die Dateien und assoziierten Schlüsselwörter mit der tagstore Software synchronisiert. Abschliessend wurde das implementierte Synchronisationssystem mit anderen tagstore Alternativen verglichen und es wurde gezeigt, dass das Synchronisationssystem für Benutzer einsetzbar ist.

Abstract

In the last few years, many tagging systems were developed. These systems allow keywords to be assigned to a specific resource and reacquire the resource by the keywords used. At the Institute for Software Technology at Graz University of Technology, a research software called tagstore was developed, which uses this concept to associate files with tags. In the context of this master thesis, a prototype for the mobile platform Android was developed. This thesis describes this developed prototype. Further, existing synchronization systems were analyzed. The main focus of this analysis was on cloud computing systems. In addition, an algorithm was implemented, which allows to synchronize files and keywords with the tagstore research software. The final chapter evaluates the implemented synchronization system with other tagstore alternatives and demonstrates that the synchronization system is deployable for users.

Contents

1	Introduction	1
1.1	Introduction to tagstore	3
1.1.1	Operation of tagstore	7
1.1.2	Store Format of tagstore	8
1.1.3	Configuration File Format of tagstore	9
1.2	Aspects for a Mobile Tagging Application	9
1.2.1	Graphical User Interface Aspects	10
1.2.2	Mobile Environment Aspects	10
1.2.3	User Input Aspects	11
1.2.4	Mobile Phone Platforms Aspects	11
1.2.5	Software Portability Aspects	13
1.2.6	Android Platform Aspects	13
1.3	Android tagstore	15
1.3.1	Differences of Android tagstore	15
1.3.2	Navigation in Android tagstore	18
1.3.3	Android Configuration Settings	21
1.3.4	Android Tagging of Files	23
2	Synchronization Systems	26
2.1	Characteristics of Synchronization Systems	26
2.2	Online Synchronization Systems	28
2.2.1	Network File Systems	28
2.2.2	Distributed File Systems	28
2.3	Offline Synchronization Systems	30
2.3.1	Rsync	30
2.3.2	Unison File Synchronizer	32
2.3.3	Sysex File Synchronization Middleware	33

Contents

2.4	Cloud-based Synchronization Systems	34
2.4.1	Cloud Storage System Benefits	35
2.4.2	Cloud Storage Usage Pattern	36
2.4.3	Cloud Storage Security Requirements	37
2.4.4	Cloud Storage Interfaces	38
2.4.5	Cloud Storage Providers	39
3	tagstore Synchronization	44
3.1	Synchronization Platforms	44
3.2	Synchronization Requirements	45
3.2.1	Synchronization Algorithm Requirements	45
3.2.2	Communication Channel Requirements	46
3.2.3	tagstore Synchronization Requirements	47
3.3	Synchronization Algorithm Conflicts	48
3.3.1	File Conflict Classes	48
3.3.2	Meta-data Conflict Classes	49
3.4	tagstore Synchronization Algorithm	50
3.4.1	Synchronization Algorithm	50
3.4.2	Synchronization Store File	53
3.4.3	Synchronization Algorithm Modes	54
3.4.4	Synchronization Back-ends	54
3.4.5	Synchronization Conflict Handling	58
3.4.6	Synchronization Serialization	59
3.4.7	Synchronization Limitations of tagstore	60
4	Synchronization Evaluation	62
4.1	TaggedFrog	62
4.2	Tabbles	64
4.3	Taggtool	65
4.3.1	Taggtool Desktop	65
4.3.2	Taggtool Business Server	67
4.4	Evaluation Summary	68
5	Conclusion	70
	Bibliography	72

List of Tables

1.1	Gartner Research Report	12
3.1	Synchronization platform factors	44
4.1	Evaluation of tagstore	69

List of Figures

Listings

1.1	Folder Hierarchy of tagstore	7
1.2	Sample Log File	8
1.3	Folder Hierarchy of Android tagstore	17
1.4	Sample Android Store Format	25
3.1	tagstore Synchronization Algorithm	51

1 Introduction

In the last few decades, mobile phones received an enormous amount of improvements. Besides the miniaturization of the phones, advancements in the area of network, battery, chip and display technology have helped to establish a new computing platform. The phone is no longer a device, which is used solely to make calls, but is a mobile computing platform. This has led to the coining of the term “smart phone”. The first smart phone, developed in a joint venture by International Business Machines (IBM) and Bell-South, was named Simon Personal Communicator.² Besides the common telephone features at that time, it used a touch screen for text input. Consequently, this enforced a paradigm change concerning the operation of the mobile phone, as the users were accustomed to press physical buttons. Today smart phones have become a powerful computing device. Smart phones are not only equipped with several gigabytes of personal storage and several hundreds megabytes of memory, they are also driven by a fast processor. As a result, common user tasks such as e-mail, browsing the web, or word processing can be deployed with a smart phone. However, smart phones are normally not equipped with a file browser, which helps organizing personal files stored on the mobile phone. Although users can download a decent file manager from an application store for their mobile platform, it does not solve the problem of fragmentation of personal files. Currently, mobile applications store user files in a preset folder. This folder is not only different for each application, but also is unchangeable. As a result, the personal files are fragmented in the file system. In order to solve this problem, a mobile application has been developed. The mobile application helps the users to organize their personal files. The mobile application lets the users associate one or multiple tags with a file. The tags can be chosen arbitrary. The file can be accessed by selecting the associated

²Wikipedia, 2012b

1 Introduction

tags. However, another problem exists when the users want to synchronize the personal files with another computer. In general, synchronization software's main focus is to keep user files updated. However, preserving the meta-data of the user files in a synchronized state is not accomplished. The effect is the associated tags are lost during the synchronization. This problem has been addressed by the development of an algorithm, which deals with the synchronization of files and their associated tags. The synchronization algorithm is implemented in two independent systems, which allow users to synchronize their files and tags seamlessly.

The master thesis is organized in four chapters. In the first chapter the tagstore research software is presented as well as the Android tagstore. In the following chapter synchronization systems are analyzed. In the third chapter the synchronization algorithm is described in full detail. Finally, in the fourth chapter alternative tagstore systems are compared to tagging functionality as well as to synchronization capabilities.

1.1 Introduction to tagstore

In the last few years many collaborative tagging systems have been developed. The first website, which used this concept, was the website Del.icio.us³ in the year 2003. It allowed users to attach tags to bookmarks. This new technique was quickly adopted by many other websites to explicitly describe content. The content was no longer limited to bookmarks, but extended to videos, images, or articles, for example YouTube⁴, Flickr⁵ or Blogger⁶. Today the area of research focuses on the usage of tagging systems for personal data management. In general, most users structure their personal files in a folder hierarchy, where each folder contains files associated with a specific topic. However, a problem arises when a file belongs to different projects. For this problem there exist two intuitive solutions, which are not satisfying. The first solution is that the users copy the file in one location and references it from the other locations. The disadvantage appears, when the file is being renamed or moved to another location. As a result, the link becomes invalid. The other solution is to copy the file to all required locations, but this creates a merge problem when those files get edited simultaneously. In order to deal with these problems, a research software is developed at the Institute for Software Technology (IST) at Graz University of Technology. The research software called tagstore (Voit, 2012) is built on the concept of tagging. Users associate a file with one or several tags. The files can then be navigated by using the tags. The research software builds a folder structure called TagTree (Voit, Andrews, and Slany, 2011) in the file system. This TagTree creates for each permutation of the associated tags a corresponding folder path, where each folder represents a tag. The contents of a folder are links to the associated files. On the Microsoft Windows platform the links are constructed by using so called shortcut files. These shortcut files are recognized by the file extension `lnk`. On other platforms, symbolic links are utilized to achieve that feature. Since tagstore uses folders in a file system to implement the TagTree, it is file browser independent. Hence, it does not depend on the operating system's file browser applica-

³<http://Del.icio.us>

⁴<http://www.youtube.com>

⁵<http://www.flickr.com>

⁶<http://www.blogger.com>

1 Introduction

tion programming interface (API) to construct the same user experience. Thus the compatibility for different file browsers on multiple platforms is ensured.

The development of tagstore started in June 2010. It is developed in the programming language Python. The Python programming language was founded by Guido van Rossum around 1990 (Lutz, 2001). The language supports the object programming model as well as some features from the functional programming model. The syntax of Python is clear and intuitive. In addition, it has a large set of libraries for common tasks, which facilitate software development. Furthermore, applications written in Python are interpreted. Fortunately, there exist an interpreter for the most popular platforms such as Microsoft Windows, Mac OS X or Unix-compatible environments. Since tagstore is an application with a GUI, which runs on many different operating system platforms, it also requires a cross platform GUI framework to deal with architectural differences. This is achieved by using the Qt framework.⁷ The Qt framework is more than a GUI framework. It also supports network sockets, threads, SQL databases and much more. In order to use this framework in Python, it needs a set of Python wrappers. For that purpose the PyQt⁸ library, developed by Riverbank Computing Limited, is used. Currently, tagstore uses PyQt4, which supports Qt 4. As a result, tagstore can be run on many different operating systems, which is one of the major application requirements.

The tagstore application can be divided into two components: the tagging component and the tagstore manager. The tagging component is responsible for the tagging of the files. It also creates TagTrees and proposes tags when a new file is created in the tagstore's storage folder. The latter component copes with the task of creating new tagstores and removing tagstores. In addition, the tagstore manager supports the following features, which can be controlled for each tagstore individually:

- The manager provides four different modes, which influence the construction of the TagTrees. The first mode is the unrestricted mode. In this mode every tag can be used. In the second mode it is the opposite.

⁷<http://www.qt-project.org/> last visited on 4/1/2012

⁸<http://www.riverbankcomputing.co.uk/software/pyqt/intro> last visited on 4/1/2012

1 Introduction

The manager lets the users construct a tag vocabulary. The vocabulary is stored in the file `vocabulary.txt` inside the `.tagstore` folder. Each tag of the vocabulary is separated by a new line. The manager imposes no rules on the composition of the tag vocabulary. Once the vocabulary is created, the system restricts the users to choose tags from the vocabulary. In general, this technique is also referred to as controlled vocabulary. Controlled vocabularies provide a means for organizing information in a consistent way. Furthermore, it can aid information retrieval (ANSI/NISO, 2005, Section 5.1).

The third mode is an extension of the first mode. When this mode is used, two TagTrees are constructed. The vocabulary of both TagTrees is unrestricted. The first TagTree is made up of tags, which describe the contents. The second TagTree consists of tags, which categorize the contents. The fourth mode is a combination of the first and second mode. Analogous to the third mode two TagTrees are constructed. However, one TagTree uses controlled vocabulary whereas the other TagTree can be composed of user-defined tags. Currently, this is the default mode when a new tagstore is created.

- The date function can automatically assign a file a date tag, which facilitates file retrieval. The manager lets the users choose between two date formats. In the first format the date tag includes year and month separated by a hyphen. The second date format also includes the day when the file was originally tagged. The date function is optional and can be turned off in the tagstore manager.
- The expiry function lets the users assign a special crafted tag, which the tagstore system recognizes as a date tag. When enabled, the tagstore system checks if the current system date has moved beyond the assigned date tag. In that case, the file is moved into a folder, where all expired files are stored. In order to distinguish between date tags and expiry tags, the system prefixes expiry tags with custom prefix. The custom prefix can be configured in the tagstore manager. The default prefix is `exp`. Again, this feature is entirely optional.
- The re-tag feature allows the users to re-associate a file. In this process, all former tags are removed and the new chosen tags are associated. The re-tag feature can either be accessed within the tagstore manager in the Re-Tagging tab or by launching `tagstore_retag.py` with the Python interpreter.

1 Introduction

- The tagstore manager lets the users rename a tag. This benefits the usability as the users do not need to re-tag each file separately. After a new tag has been entered, the relevant TagTree is being updated with the new tag name.
- The synchronization function enables users to synchronize different tagstores. It also supports different synchronization modes. In the Chapter 3 this feature is explained in greater detail.

As already stated, the tagstore manager lets the users create, modify, and delete a store. Although tagstore provides an elegant solution to administer unrelated files, there are applications, which require a disjunction. For example private files and work related files. Therefore, it is reasonable to separately store those files. Given that tagstore manager supports multiple tagstores, it is advised to create a separate tagstore.

Before the users are able to deploy the tagstore for the management of their files, it is required to create a first tagstore. This task is accomplished by using the tagstore manager. The manager can be started by launching `tagstore_manager.py` with the python interpreter. The tagstore manager lets the users choose a root folder of the tagstore. After the folder has been selected it creates five sub folders. These folders are called `.tagstore`, `categories`, `descriptions`, `expired_items` and `storage`. Except the first folder, all other folder names are language dependent. Therefore, the tagstore manager creates a different folder structure, when the operating system's locale is set to another language. At the moment tagstore only supports English and German localization. If there is no localization available, tagstore falls back to English.

In Listing 1.1 the folder structure is shown. The `.tagstore` folder contains configuration files, which are specific to a tagstore. The `description` folder holds TagTree of description tags, whereas the `categories` stores the TagTree of the class tags. The `expired_items` folder stores the items which have been moved into it as the expiry function triggered. The `storage` stores the user files of tagstore.

1 Introduction

```
tagstore root folder \  
|-- .tagstore  
|-- categories  
|-- descriptions  
|-- expired_items  
|-- storage
```

Listing 1.1: This listing shows the folder hierarchy, which is build when a new tagstore is created. It constructs five folders, which have different purposes. The *.tagstore* folder contains the configuration files of the tagstore. The *categories* folder stores the TagTree of the class tags, when tagstore operates in the third or fourth mode. The *descriptions* folder saves the TagTree of the description tags. Moreover the *expired_items* holds the files, which were moved, when the assigned expiration date was triggered. The *storage* folder keeps the unordered files of the tagstore.

1.1.1 Operation of tagstore

The tagstore tagging component is started by launching `tagstore.py` with the Python interpreter in the tagstore installation folder. The tagging component then enumerates all available tagstores by analyzing the main tagstore configuration file. The file named `tagstore.cfg` is stored in the folder `tsresources/conf` of the tagstore installation folder. The tagstore configuration file uses the INI file format. The tagstores are listed in the *stores* section. Afterwards, it initializes each tagstore and starts monitoring the storage folder for file changes. A file change is detected when a file is created, renamed or deleted. In the event of a new file, the *tagging dialog* is launched. This dialog requests the users to enter tags for the new file. It is a requirement that at least one tag must be entered. After the tags have been entered, the corresponding TagTree is created. Finally, the file is then accessible from the TagTree with the chosen tags. When a file is deleted, tagstore removes all created shortcut files or symbolic links. At the same time no longer used tags are also removed from the TagTree. Furthermore tagstore supports detecting a file has been renamed. As soon as this event is detected, tagstore updates the relevant shortcut files or symbolic links.

The tagging component displays the tagging dialog when a new file is created. Figure 1.1.1 demonstrates the tagging dialog. The dialog shows

1 Introduction

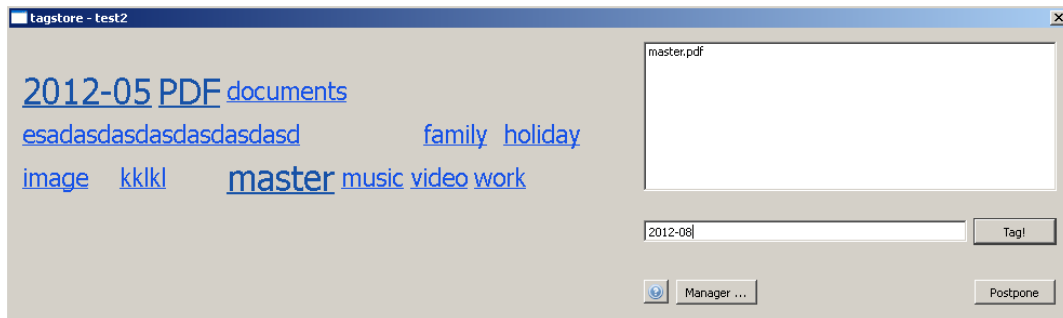


Figure 1.1: A screenshot of the tagging dialog of tagstore. The window bar displays the name of the tagstore, which is *test1*. On the left side tagstore proposes a few tags. The tags were chosen based on the file extension, already used tags, and standard tags of tagstore. Users can select these tags by clicking on them. On the right side is a list box, which is designed for files. These files are called *pending* files, because they have not been tagged yet. There is currently only one file named *master.pdf*. The text field is used to type tags. It is a requirement that tags need to be separated by a colon. Once the users have finished entering tags, the users can initiate building the TagTrees by hitting the *Tag!* button. It is also possible to delay this by using the *Postpone* button.

a set of tags on the left side, which are proposed by tagstore. Users can select these tags by clicking on them. Users can also enter own defined tags in the text field. It is a requirement that each tag is separated by a colon. However, it is not required to immediately enter tags as the users can be busy with more important tasks. The tagstore software stores a list of already tagged files in the file `store.tgs`. When tagstore is started, it enumerates all files in the storage folder. By excluding the file entries from the `store.tgs`, tagstore has a list of files, which have not been tagged yet. If that list is non-empty, tagstore displays the tagging dialog immediately after application start-up. Hence, it is safe to dismiss the tagging dialog.

1.1.2 Store Format of tagstore

The information of a tagstore is saved in a file named `store.tgs` which resides in the `.tagstore` folder. The file format follows the conventions of the INI format. The INI format is a standard for configuration files. The file

1 Introduction

format consists of two sections which may contain several keys and their associated values. A new section is defined with an opening square bracket, the section name, and the closing square bracket. A section ends when the next section is defined or the end of the file is reached. In addition, the section definition must start on a new line. Other keys, values, and sections are not allowed on the same line. In general, a key value pair is delimited by the equal sign. In addition special characters of the key name must be escaped using RFC 2396⁹.

Furthermore, the key value may be escaped using double quotes. The first section is the *settings* section. It contains one supported key *config_format*, which defines the configuration format. The value should be set to "1". The second section called *files* holds the the available files and their associated tags as well as the time stamps.

```
[ settings ]
config_format=1
```

```
[ files ]
example-short-chapter.tex\tags="test , manual , tutorial"
example-short-chapter.tex\timestamp=2012-03-29 15:18:35
example-short-chapter.tex\category=2012-03
```

Listing 1.2: This is a sample log file obtained from a tagstore. The file stores the file names and used tags. In this listing the tags *test*, *manual* and *tutorial* were used with the file *example-short-chapter.tex*. The entry also contains a time stamp, which stores the date when the file was tagged. The *category* key contains the class tags of the entry.

Listing 1.2 demonstrates a file with the name *example-short-chapter.tex*. This file was associated with the tags *test*, *manual*, and *tutorial*. The date stamp key indicates when the file was tagged. If a file is re-tagged, this time stamp gets also updated. Finally, the category key stores the defined class tags of the file.

⁹<http://www.ietf.org/rfc/rfc2396.txt> last visited on 8/29/2012

1.1.3 Configuration File Format of tagstore

Each tagstore stores configuration settings in the file `store.cfg` inside `.tagstore` folder. The configuration format also uses the INI file format. The configuration settings control the tagstore mode¹⁰ and date stamp format. These configuration settings can be manipulated in the tagstore manager. Furthermore, the configuration settings store a key called *android.store*, which determines the type of the tagstore.¹¹ The value can either be “0” or “1”.

1.2 Aspects for a Mobile Tagging Application

An application, which is developed for mobile devices, differs in many aspects from the personal computer platform. In this section the different aspects are presented. The first sub-section illustrates the GUI aspects of mobile phones. Afterwards, the mobile environment aspects are presented. In the third sub-section the user input characteristics are presented. Later, the major mobile phone operating systems are inspected regarding their market share. In the fourth sub-section software portability aspects are shown. Finally, the chosen mobile phone platform is analyzed.

1.2.1 Graphical User Interface Aspects

Mobile devices have – due to their limited physical appearance – a small screen. As a result, an application can not display all user interface elements on the screen at the same time because the elements would overlap. Therefore, elements must be properly aligned to achieve an operable user interface. In general, this is achieved by dividing the user interface into several sections called *screens*. The screens can be navigated by performing a gesture, which the operating system recognizes. Another important aspect is the screen resolution since mobile devices not only differ in respect to

¹⁰The tagstore modes are elaborated in Section 1.1.

¹¹See Section 1.3

1 Introduction

their screen size but also screen resolution. Finally, the screen orientation also needs to be considered.

1.2.2 Mobile Environment Aspects

Mobile applications are influenced by their hosts hardware capabilities. The first important factor is the speed of the central processing unit (CPU). Although modern mobile phones certainly provide the computing power to host a mobile tagging application, the impact for a less developed mobile device is not to be underestimated. Furthermore, CPU intensive applications have a negative influence on the battery consumption. In addition, it can shorten the battery life cycle.

The variable available amount of random access memory (RAM) per mobile phone type can influence the deployment. The mobile phone operating system sometimes needs to kill applications due to memory requirements. Therefore, applications which are deployed on hosts with sufficient amount of RAM, have a higher chance to run uninterrupted.

Mobile phones are dependent on the network accessibility and capability. Since mobile devices do not persist on a fixed location, the device is not assured to have network access. In addition, the network connection can suffer from frequent disconnection or low bandwidth. It may also be the case that the network connection is disabled by phone settings. This scenario typically appears during the usage of foreign global system for mobile communication (GSM) networks. Since roaming fees accumulate, the default behavior is to disable Internet access of GSM.

1.2.3 User Input Aspects

Nowadays mobile phones use a touch screen to receive an input from the users. The touch screen is also responsible for entering a text, which is achieved by providing a screen keyboard. However, a device may also provide a hardware keyboard. In general, phone manufacturers abstain from

1 Introduction

it because it increases the device's weight and complexity. In order to facilitate text input, the mobile phone's operating system provides several methods, which lighten the text entering process. The first technique is the automatic completion approach, which targets the entering of words. As soon as a word is started to be typed, the operating system searches a dictionary. If there is a match, then the match is proposed to the users. Another supporting technology is to add the current word to the dictionary. That way the next time this word is started to be entered, the operating system can propose it. Finally, the integrated correction support of misspelled word assists the user's input process.

1.2.4 Mobile Phone Platforms Aspects

It is required that tagstore should run on the most popular mobile phone operating system. At the time of writing there are three major platforms. The most popular platform is developed by the Open Handset Alliance, which is a group of technology and mobile companies led by Google. The platform is referred to as the Android platform. The second important platform is the iOS manufactured by Apple. The sole product of this platform is the iPhone. Finally for completeness, the Symbian OS developed by the Symbian Foundation must also be mentioned. In February 2011, Nokia announced as the only remaining active developer in the Symbian Foundation, to abandon the development of Symbian OS based devices. Instead Nokia intends to use Microsoft's phone operating system in the future.

Table 1.1 shows the research report from Gartner titled *Forecast: Mobile Communications Devices by Open Operating System, Worldwide, 2008-2015*. The report released in April 2011, forecasts the Android platform to have a market share in the smart phone operating system area of 49.2 per cent. In addition, the analysis estimates Symbian share in the smart phone operating area will be irrelevant. Furthermore, the report highlights two differentiating trends. On one hand Gartner sees a big growth of market share for Microsoft based phones, on the other hand the iOS market growth is declining.¹²

¹²<http://www.gartner.com/it/page.jsp?id=1622614> last visited on 3/3/2012

1 Introduction

OS	2010	2011	2012	2015
Symbian	37.6 %	19.2 %	5.2 %	0.1 %
Android	22.7 %	38.5 %	49.2 %	48.8%
iOS	15.7 %	19.4 %	18.9 %	17.2 %
Microsoft	4.2 %	5.6 %	10.8 %	19.5 %
Other	3.8 %	3.9 %	3.4 %	19.5 %

Table 1.1: The Gartner research report forecasts the market share of smart phone operating systems. It says that the Android platform will nearly have half of worldwide smart phone operating system market share by the end of the year 2012. It also predicts Microsoft's phone operating system a large growth of sold smart phone units.

The Android platform is becoming the leading mobile device platform. This claim is emphasized by Andy Rubin, who is the Senior Vice President of Mobile at Google. In his profile on the social network platform Google+, he states that more 850,000 Android devices are activated per day.¹³ As a result, the Android platform was chosen as the main target for the mobile tagstore.

1.2.5 Software Portability Aspects

In the previous section, the Android platform has been chosen as the main target for the mobile tagstore version. Since the other mobile phone platforms differ in many ways from the Android platform, it is not easy to preserve software portability. However, there are frameworks available, which allows a set of applications to run on multiple mobile phone platforms. In general, the application developers construct a web application, which is bundled with the framework. The framework provides library functions, which can be accessed by using a browser language such as JavaScript. By using the JavaScript language, access to the mobile phone's hardware is provided. The developers only need to bundle the web application with

¹³<https://plus.google.com/u/0/112599748506977857728/posts/Btey7rJBaLF> last visited on 3/3/2012

1 Introduction

the target platform framework version. In short, the frameworks allow developers to build applications, which run on multiple platforms by using Web technologies such as HTML, CSS, and JavaScript. A comparison of cross platform mobile frameworks is listed on the Mashable's website.¹⁴

The Android tagstore is not using these frameworks. The first reason is obviously the speed penalty, which is introduced by adding another software layer. The next disadvantage is the user interface. Web applications do not provide the original look and feel of native Android applications. Finally, the frameworks do not provide an API to observe folders for changes. However, this functionality is essential for a mobile tagstore.

1.2.6 Android Platform Aspects

The Android platform is based on Java technology. Java is a programming language, which was invented by James Gosling at Sun Microsystems. Sun Microsystems was later acquired by Oracle Corporation. Like Python, it supports the object oriented programming model. In addition, the syntax is similar to the programming languages C and C++. Java programs are compiled with the Java compiler into an intermediate code called byte code. This byte code is then interpreted by the Java VM and finally executed. On the Android platform the Dalvik VM is used. However, Dalvik VM is different from the Java VM. In general, the Dalvik VM was designed for embedded systems, which have limited disk space and memory.

The Java VM is a stack-based machine where as the Dalvik VM is using registers. In a stack-based machine the operation code and operands are fetched from the stack. In contrast to a stack-based machine, a register machine fetches the operation code and operands directly from virtual registers. These virtual registers can be mapped to physical processor registers, which in turn can accelerate the execution speed.¹⁵ A study on the analysis of different VM architectures shows that register based VMs require 47 %

¹⁴<http://mashable.com/2010/08/11/cross-platform-mobile-development-tools/> last visited on 5/31/2012

¹⁵<http://www.fhnw.ch/technik/imvs/publikationen/artikel-2009/einblick-in-die-dalvik-virtual-machine> last visited on 5/31/2012

1 Introduction

less instructions than stack-based machines. Although the register code is 25% larger than the corresponding stack code, the increased cost due to fetching more VM code involves only 1.07% extra real machine load per VM instruction. All in all the register based machine requires 32.3% less time to execute standard benchmarks.¹⁶

Before an Android application can be executed by the Dalvik VM, it needs to be converted into an instruction set which is understood by Dalvik. This conversion task is performed by the dex tool, which is included in the Android software development kit (SDK). This tool uses the generated byte code from the Java compiler to produce a so called *dex* file. Typically, multiple class files are converted into one dex file. The advantage of that conversion is that uncompressed dex files are a few percent smaller than compressed Java archives due to the elimination of duplicate strings. Furthermore, byte order and other optimizations can be applied to reduce file size. Afterwards an application package file (APK) is constructed from the the generated dex file together with application resources, manifest, assets, and certificates. The APK file format is based on the JAR file format, which uses the ZIP archive format as a container.¹⁷

1.3 Android tagstore

The Android tagstore is presented in this section. In this section there are four sub-sections. In the first sub-section, the differences between the Android and desktop tagstore are elaborated. Afterwards, the navigation in the Android tagstore is explained. In the third sub-section, the configuration settings are highlighted. Finally, the tagging dialog of Android tagstore is presented.

The original tagstore implementation is written in the Python programming language. In order to differentiate the implementation between Android and Python, the tagstore developed in Python is referred to as the

¹⁶http://static.usenix.org/events/vee05/full_papers/p153-yunhe.pdf last visited on 5/31/2012

¹⁷[http://en.wikipedia.org/wiki/Dalvik_\(software\)](http://en.wikipedia.org/wiki/Dalvik_(software)) last visited on 5/31/2012

1 Introduction

desktop tagstore. In addition, the Android tagstore is sometimes also labeled as the mobile tagstore.

1.3.1 Differences of Android tagstore

The Android tagstore needs to break with a concept, which the desktop tagstore has introduced. The desktop tagstore creates folders in the file systems for the tags and uses *shortcut* files for the files. There is a number of reasons why this concept needs to be adapted. First of all, Android is a Linux based environment. Hence, only the symbolic link mechanism is available. However, the Android platform uses Microsoft's FAT32 file system for the external storage media. Unfortunately this file system does not support symbolic links.¹⁸ As a result, symbolic links can also not be used. Though it is technically feasible to change to a file system with symbolic link support such as EXT2, the amount of work is unacceptable. In addition, the majority of users would be excluded from trying tagstore. Another disadvantage is the requirement of needing a file browser to navigate in the TagTrees. Currently, Android phones are not shipped with a file browser. Although users can install a decent file browser from an Android application store, it is another inconvenience. As a consequence, the Android tagstore restrains from creating folder structure for the TagTrees in the file system. However, it provides the ability to browse the TagTrees in a virtual view, which is accessible within Android tagstore. Currently, the Android tagstore only supports visualization of one TagTree. The mobile tagstore supports two alternating view modes called *tag cloud view* and *icon view*. In the tag cloud view mode, tagstore displays the tag names and file names as text in a freely arranged way. Complemental, the icon view shows an icon above the text. This view resembles the visualization of folder hierarchies in graphical file system browsers. Due to the fact that many users are accustomed to this visualization type, it is the standard view when the mobile tagstore is installed.

The next difference to the desktop tagstore is the way files are stored. In the desktop tagstore, all files are stored in the `myfilenamestorage` folder. Due

¹⁸<http://msdn.microsoft.com/en-us/library/windows/desktop/ee681827%28v=vs.85%29.aspx> last visited on 4/16/2012

1 Introduction

to the requirements of the Android platform, the Android tagstore needs to break with that concept. The reason behind that change is the way Android applications store user files. As already stated, there is no standard where Android applications should store user files. As a consequence, user files are saved in many different folders in the file system. In order to deal with this fact, the Android tagstore provides the ability to observe any folder in the external storage file system for file changes. A file change in this context is the creation, or deletion of a file. When a new file is created, tagstore informs the users to associate the file with tags. Once the file has been tagged, it is accessible with the used tags in the virtual view. On the other hand as soon as a tagged file is removed, all associated tags are removed. However, an associated tag is not removed entirely from tagstore. A tag is only removed if its usage count becomes zero. If that tag is still associated with another file, only its usage count is decremented. When the usage count eventually becomes zero, the tag is also removed.

Since the Android tagstore supports observing multiple folders, the question arises how to deal with files having the same name but residing in different folders. Although it is feasible to support multiple files with the same name, tagstore requires a file to have a unique name. The first reason is that those file names may have the same tags attached. As a result, users need to check each file if it is the desired file. Another disadvantage appears when the tagstore is being synchronized. When the synchronization task is about to synchronize a file with no unique name, it has several candidates available. The synchronization task may choose the first, last, or all of them depending on the synchronization method. Furthermore, one file may have been edited since the last synchronization. The synchronization process has no information whether the modified file should be updated. Consequently supporting this feature creates many problems and increases the complexity of the synchronization. Thus tagstore abstains from this feature. However, the situation may still happen when Android tagstore observes multiple folders. The Android tagstore then offers four possibilities. The tagged file can be renamed or ignored. Ignored means in that context that the file is removed from the tagstore. The new file can also be either renamed or ignored. Ignoring the new file denotes that the tagged file remains in the tagstore.

The Android tagstore supports controlled vocabulary. As already stated, it

1 Introduction

```
root of external storage \  
|-- tagstore \  
|---- .tagstore  
|---- storage
```

Listing 1.3: The folder hierarchy of the Android tagstore is created when the Android tagstore is initialized for the first time. It creates the folder `tagstore` and sub-folders `.tagstore` and `storage`. The first sub-folder stores the configuration files whereas the second sub-folder stores files, which were added during the synchronization.

requires the `vocabulary.txt` to be put into folder `.tagstore`. Furthermore, it must be enabled in the *Configuration Settings* screen. Unfortunately, the Android tagstore does not support the capability to manage a controlled vocabulary. The reason is primarily that typing in a smart phone is an expensive task. Therefore, it is not encouraged by the tagstore application. The recommended process is to edit the file while the mobile phone's storage is mounted as a mass storage device by `USB`.

The Android tagstore does not support the expiry feature of the desktop tagstore. The expiry function brings no benefit as it requires an external file browser. As a result, it is not supported in the Android tagstore.

When Android tagstore is initialized for the first time, it creates a folder named `tagstore` in the root of external storage file system. Inside that folder the folders `.tagstore` and `storage` are created. In addition, the Android tagstore creates the tagstore configuration file and the tagstore store file. The `storage` folder is used for new files, which have been added during a tagstore synchronization. Listing 1.3 shows the created folder hierarchy. In comparison to the desktop tagstore, the folders *descriptions* and *categories* are missing. Since Android tagstore realizes the `TagTree` in a virtual view, there is no need to create the folders in a file system. The Android tagstore uses a database for storing the relations between the files and tags. The primary reason is speed. However, the Android tagstore stores a backup of the database in the tagstore store format.¹⁹ The file is updated every time a change in the Android tagstore occurs.

¹⁹See Section 1.1.2

1.3.2 Navigation in Android tagstore

In the Android tagstore, navigation is performed in a virtual view. In the virtual view, there are two types of items, namely *tags* and *files*. The start view displays all used tags. In general, navigation is performed by clicking on the item. The Android tagstore supports two types of hand gestures. A short click on an item performs the default action for that item. This is also referred to as a *touch* gesture. In the case the item is a tag, the contents of the view is cleared. Later, all tags and files associated with it are collected and displayed. If the item is a file, then tagstore launches the default application registered to handle that file. Furthermore, a *long press* gesture on an item opens a menu. Depending on the item type, a specific menu is opened. For a tag item, there are three operations available such as *rename*, *statistics* and *delete*. The *statistics* menu opens a screen when it is selected. That screen shows the number of files, which have been tagged with that tag and the number of tags which have been used together with that tag. The menu for a file is similar. However, more options are available. Despite the standard operations *rename* and *delete*, it supports the operations *share*, *re-tag* and *open as*. The operation *share* invokes a menu, which lists all registered applications, which are interested in that specific content type. Once an application is selected, the selected file is explicitly made available to that application. Furthermore, the *retag* option lets the users associate new tags to that file. By using this method, a file can be re-associated with new tags. Finally, the option *open as* displays another menu, which asks the users how that file is supposed to be interpreted. Available choices are *text*, *audio*, *video* or *image*. Once the selection has been made, the standard application registered for that content type is started to handle that file. However, there can also be multiple applications registered for that content type. In that case the users need to select the desired application from a list.

The screen size of a phone is spare. Therefore, elements must be carefully chosen not to occupy too much of the available space. The tagstore application abstains from displaying a window title in the navigation screen. Furthermore, tagstore uses the available hardware buttons like *menu* and *back* button to provide additional functionality. If the Android phone does not provide hardware buttons, then the Android system automatically pro-

1 Introduction

vides onscreen buttons. The *menu* button displays a menu, which has three available choices. The first choice shows the configuration screen, which is elaborated in the Section 1.3.3. The second choice opens the synchronization screen, which is explained in Section 3.4. The last option displays a screen, which provides information about tagstore. The *back* button is used to navigate backward in the TagTree. For example the current TagTree is made of the tags *cartoon* and *test*. After the *back* button is pressed, the *test* tag is removed from the TagTree. The view now gets cleared and refilled with the contents of the changed TagTree.

The Android tagstore visualizes the TagTree in a virtual view. When the users navigate in the TagTree, it is essential to know the current contents of the TagTree. The first approach is to provide an area of screen, which displays all tags in a text field. Although this solution solves the problem of knowing the quantity of the TagTree, it is not sufficient for all users needs. In a scenario, the TagTree contains three tags named *cartoon*, *test* and *foo*. In order to display all tags and files entirely related to the tag *foo*, the users need to navigate back to the initial screen and select the *foo* tag. The Android tagstore solves this usability problem by providing so-called *navigation buttons*. Each navigation button represents an item of the TagTree. Once a navigation button is activated, all other items are removed from the TagTree and the view is updated.

Icon View

Figure 1.3.2 demonstrates the start screen of the icon view. This screen shows all tags, which have been associated with one or several files. In this figure two items are displayed. Each item represents a tag, which is shown by the green tag in the image. The items are named *cartoon* and *test*.

Figure 1.3.2 displays another screenshot of the icon view. The TagTree contains the *cartoon* tag. The view was generated by touching the *cartoon* tag. In the upper corner a button with the tag *cartoon* is shown. This button represents a navigation button. In this figure the file *cartoon.jpg* is the tagged file.

1 Introduction

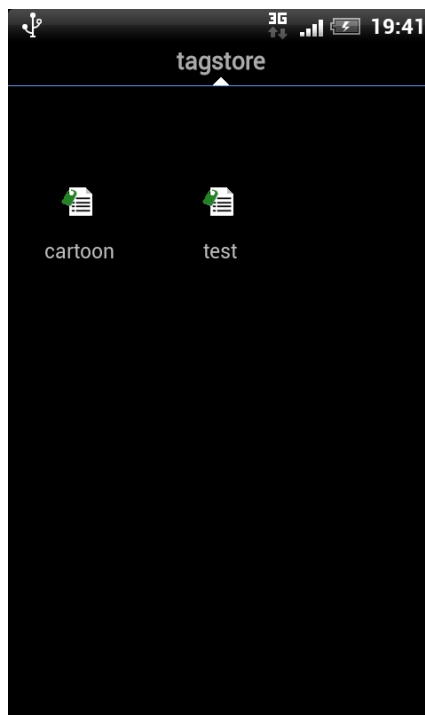


Figure 1.2: This screenshot displays the start screen of the icon view. It displays two items called *cartoon* and *test*. Each item represents a tag, which has been associated with one or several files.

1 Introduction

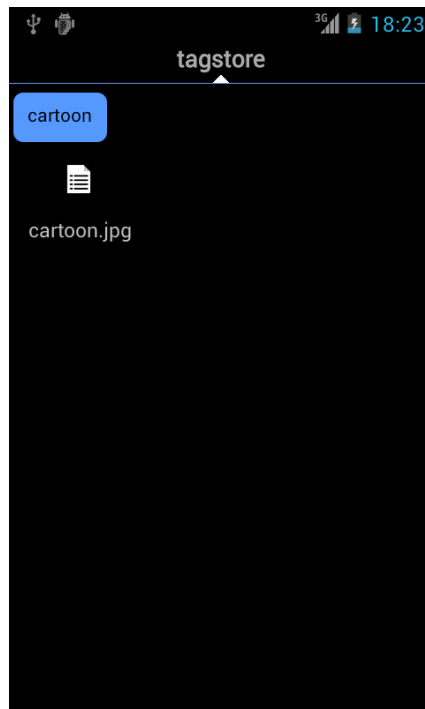


Figure 1.3: This screenshot displays the icon view. The TagTree contains the tag *cartoon*. The contents of the TagTree is displayed by the navigation button. The navigation button is placed above the icon view. There is one file named *cartoon.jpg*, which is associated with the tag *cartoon*.

1 Introduction

Tag Cloud View

Tag clouds were first deployed by Jim Flanagan's Search Referral Zeitgeist, which visualized web site referrals.²⁰ Tag clouds present text in a freely arranged way. In this context text is a tag or a file name.

In comparison to the icon view, the tag cloud view only shows the tag name or the file name. Therefore, it requires special measures to improve the user experience. Rivadeneira et al. describe several features and evaluate their performance corresponding the tag cloud navigation. The tag cloud view of tagstore uses several methods to enhance the user experience.

The first method targets the distinction of the importance of tags. In order to recognize important tags rapidly, tagstore renders the tags differently depending on their importance. The importance is measured on the magnitude, how often a tag has been associated with a file. First, the minimum and maximum tag usage from the entire tag collection is determined. Next the largest font size is assigned to the most important tag. The font size is then decreased proportional to the tag usage for less important tags.

The second method works similarly to the former method. However, the importance of the tag is highlighted by using different colours. The tag with the highest usage is displayed in white. Less used tags are drawn in orange, dark orange and brown. In the Figure 1.3.2 this technique is demonstrated.

Finally, the file names are drawn in the colour blue, which helps the users to distinguish tags and file names. In addition, the font size of the file name is dynamically adjusted to fit the screen size requirements. As a result, files which are composed of a long name are drawn in a smaller font size. This effect is visible in Figure 1.3.2. In addition, the navigation button is visible in the top of the virtual view.

²⁰<http://web.archive.org/web/20020906055006/http://jimfl.tensegrity.net/zeitgeist/> last visited on 9/25/2012

1 Introduction

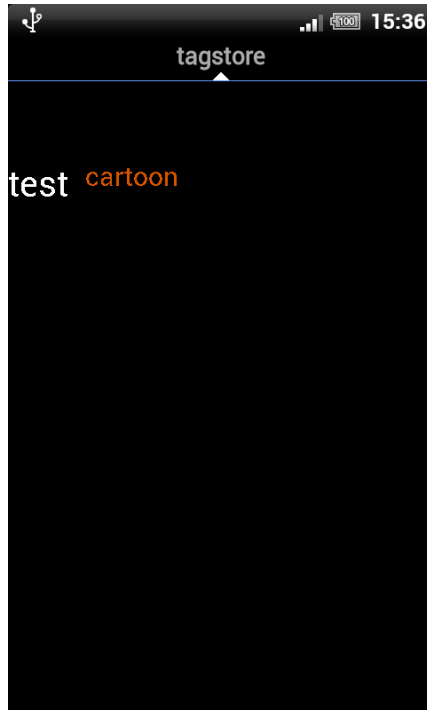


Figure 1.4: This screenshot shows the start screen of tag cloud view. It illustrates two tags called *test* and *cartoon*. The tag *test* is used with more files than the other tag. Therefore, it is drawn in the colour white, which emphasizes higher importance. In addition, the increased font size helps the users to recognize the tag's importance.

1 Introduction

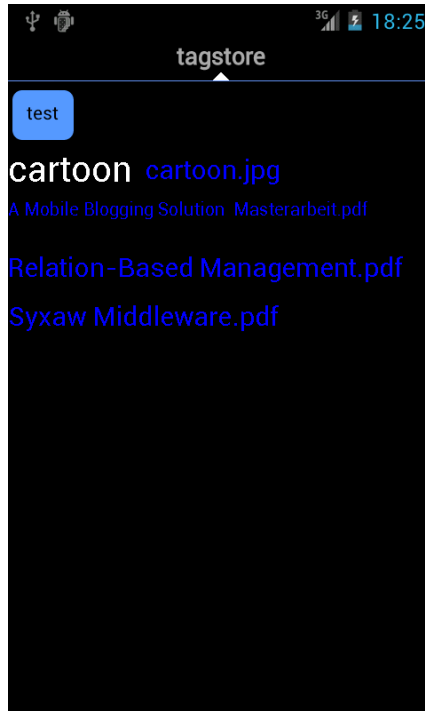


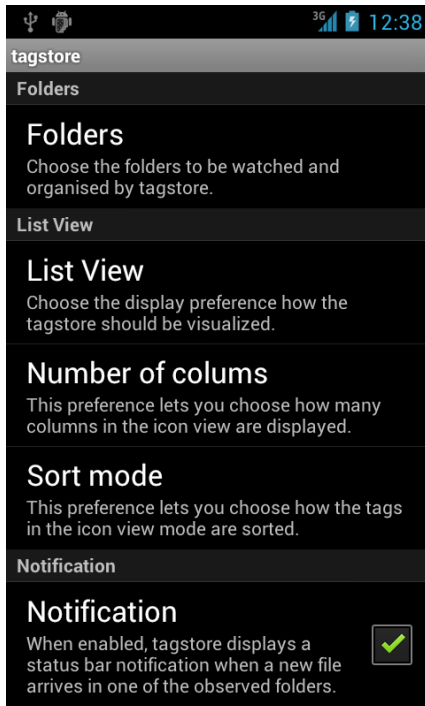
Figure 1.5: This screenshot shows the tag cloud view. The TagTree is made up of the tag *test*. This screen has been accessed by clicking on the *test* tag in the start screen. In general, all file names are painted in the color blue. Due to the excess length of some file names, the font size is reduced. An example is *A Mobile Blogging Solution Masterarbeit.pdf*. In addition, there is a tag visible called *cartoon*. This tag has been added to the file *cartoon.jpg* to illustrate the visualization when multiple tags are associated with a file.

1.3.3 Android Configuration Settings

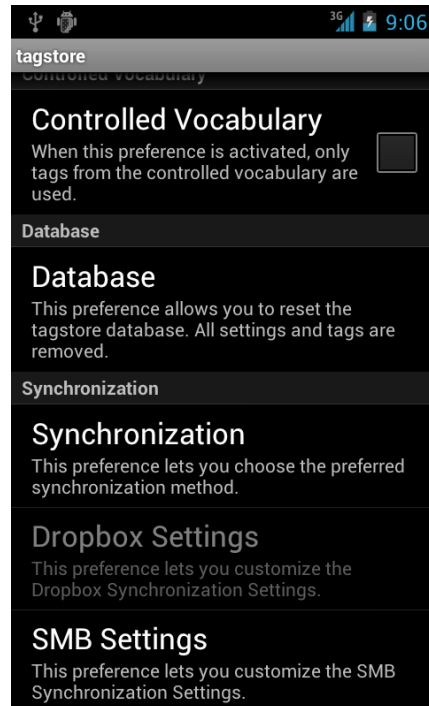
The Android tagstore's configuration screen is accessed by using the menu button and choosing the *Configuration* item. Figure 1.3.3 shows this screen. The Android tagstore supports five configuration options. The options deal with application settings as well as with user interface preferences. The first configuration option is called *Folders*. The setting is used to select folders, which are observed for file changes. A screen is shown, which displays the current monitored folders. Per default the *storage* folder inside the tagstore folder is observed. In addition, the users can insert additional paths by clicking on the *Add Folder* button. On the opposite, folders are removed by hitting on the remove button which is displayed next to the folder path. Due to the spare size of the screen, the remove button is displayed as a pictogram. In general, all observed paths can also be removed again. This also has an effect on the files in the tagstore. In general, all files, which have not yet been tagged, are put into a so called *pending file queue*. When an observed path is removed, all descending files in the pending file queue are also removed. However, files which already have been tagged, are not removed. In that way users have an easy method to stop and resume observing a folder for file changes.

The next configuration option named *List View* controls the user interface appearance. Principally the option lets the users choose, whether the icon view or the tag cloud view should be used for displaying the tagstore. In addition, there are two more sub options available when the icon view is active. The first sub option defines the maximum number of tags or files which are displayed per row. The available range is between one and four. The reason for limiting the number of items is the limited screen space. The other sub option influence the sorting order of the tags in the icon view. There are two modes available, labeled *alphabetic* and *popular*. Whereas the first mode sorts the tags alphabetically, the latter operates on the tag importance. The tag importance is measured by the number of files, the tag is associated with. In other words, the most frequently used tag, is the most important tag. The most important tag is displayed in the view first followed by less significant tags. Nevertheless, it is possible that two tags achieve the same importance. In that situation the tag, which has been created first, is displayed first.

1 Introduction



(a) This screenshot shows the top of the configuration screen. It displays the available configuration items such as *Folders*, *List View* and *Notification*. These items influence the tagstore behaviour in many ways.



(b) This screenshot shows the bottom of the configuration screen when the users navigate downwards. It displays additional configuration items labeled *Controlled Vocabulary*, *Database* and *Synchronization*.

Figure 1.6: Configuration Screen

1 Introduction

The configuration item referred to as *Synchronization* controls the synchronization settings. The Android tagstore supports three synchronization methods. Their functioning is fully explained in Section 3.4.3.

The tagstore application sometimes needs to inform the users that an event happened or an intermediate task was completed. The Android platform provides two independent mechanisms to accomplish this functionality. In the first mechanism labeled *toast*, informal messages can be displayed in the application. In addition, the messages automatically hide after a defined period of time. The drawback is that the application must be currently focused. As a result, this mechanism is only used when the tagstore application is active. The other mechanism circumvents the restriction by using the system's status bar to display information. In that way applications can inform the users of new activity. The tagstore application uses both mechanisms. The *toast* mechanism is used for events, which were triggered actively by the users while interacting with the tagstore. An example is the completion of the synchronization task. The system's status bar mechanism is solely used for informing the users that a new file has arrived in one of the observed folders. This behaviour can be deactivated by unchecking the setting. Once deactivated, tagstore stops informing the users of new file events.

In the previous section, the advantages of the controlled vocabulary have already been elaborated. The Android version of tagstore also supports the controlled vocabulary. Once enabled, the controlled vocabulary affects how tags are used. The setting requires all tags, which are assigned eventually, to originate from the quantity of the controlled vocabulary. This also applies to files, which get re-tagged by the users.

The database configuration option is intended to reset the tagstore to factory settings. As a result the tagstore is set to the initial empty condition and all used tags are removed. In addition, the pending file queue is cleared. In order to prevent unintended resets of the tagstore, the Android tagstore displays a confirmation screen before the action is performed. Needless to say, no user files are touched. However, all stored folders are also removed from the database. Therefore, watched folders need to be re-added after database reset.

1.3.4 Android Tagging of Files

The Android tagstore shows the tagging screen, when a new file is created in one of the observed folders in the file system. The tagging screen is displayed by performing a *swipe* gesture.²¹ Figure 1.3.4 displays a screenshot, which illustrates the tagging screen. In the top of the tagging screen, the full file system path of the new file is shown. The file is named `mynewfile.txt` and is stored in the `storage` folder of Android tagstore. The file can also be opened by clicking on it. Furthermore, additional operations are provided by performing a long press gesture. These operations are elaborated in Section 1.3.2. In the middle of the screen, there is a text field. This text field can be used to type tags. Each tag needs to be separated by a colon. Furthermore, there are buttons displayed above the text field. Each button represents a tag. The tag is inserted into the text field by clicking on it. Afterwards, the button is updated with a new tag, which has not yet been used. If there are no more used tags, then the button is hidden.

The tagging process is invoked as soon as the users have pressed the *Tag me!* button. It is assumed that the contents of the text field consists of one or more tags each separated by a comma. The first task of this process is to split the contents of the text field into a list of tags, where each comma separates a tag. Afterwards, each tag is checked if it is allowed. In the Microsoft Windows family there are certain names reserved for system usage.²² In order to ensure compatibility with these os, those keywords are also restricted. Currently, these reserved keywords are `com1`, `com2`, `com3`, `com4`, `com5`, `com6`, `com7`, `com8`, `com9`, `lpt1`, `lpt2`, `lpt3`, `lpt4`, `lpt5`, `lpt6`, `lpt7`, `lpt8`, `lpt9`, `con`, `nul`, `prn`. Furthermore, tagstore also prevents the usage of following characters in a tag. These are `"\"`, `"/"`, `"?"`, `"<"`, `">"`, `":"`, `"*"`, `"|"` and `""`.

If tagstore detects an invalid tag, the initiated tagging task is stopped and the users are informed by a *toast* message of an error. Otherwise after all tags have been validated successfully, the database is updated with the new

²¹<http://developer.android.com/design/patterns/gestures.html> last visited on 10/8/2012

²²<http://blogs.msdn.com/b/oldnewthing/archive/2003/10/22/55388.aspx> last visited on 4/16/2012

1 Introduction

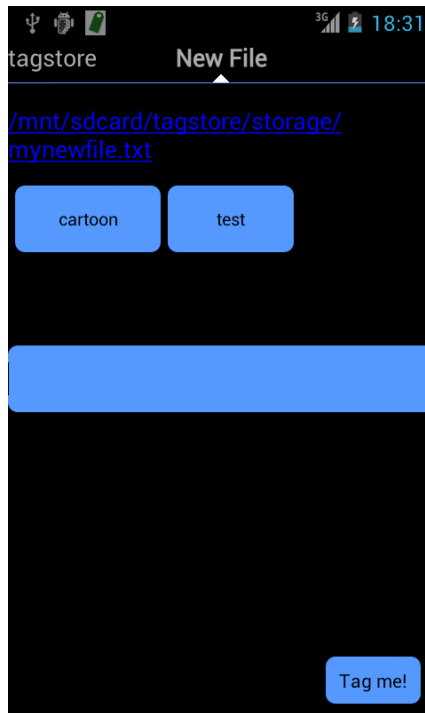


Figure 1.7: This screenshot displays the tagging screen of Android tagstore. It is displayed when tagstore detects a new file in one of the observed folders. In the top of the figure, the full path of the file is displayed. The users can open the file by performing a short click on the item. Other operations are performed by doing a long click – see Section 1.3.2 for more details. There are two tags available for use, namely *cartoon* and *test*. These are inserted into the text field by clicking on their buttons

1 Introduction

```
[ settings ]  
config_format=1
```

```
[ files ]  
tagstore%5Cstorage%5Ccartoon .jpg\tags="test , cartoon , android"  
tagstore%5Cstorage%5Ccartoon .jpg\timestamp=2012-04-16 13:39:46
```

Listing 1.4: The Android store format is very similar to the desktop tagstore store format. The Android store is saved in the `store.tgs`, which is stored in the `.tagstore` folder. However, the Android tagstore supports observing multiple folders in the file systems. Therefore, the Android store format needs to store the relative file system path from the root of the external storage. In addition, the path separator needs to be escaped due to requirements of the desktop tagstore. This Android sample store file contains one file, which is stored in the `storage` folder of the Android tagstore. The file `cartoon.jpg` has been associated with the tags `test`, `cartoon`, and `cartoon`.

file and tags. Later, a new log file entry is created in the file `store.tgs`. Finally, the screen is dismissed and the `TagTree` is refreshed when there are no further pending files. Otherwise the tagging screen is reloaded with the next pending file.

The file format is elaborated in the section 1.1.2. Although the Android tagstore intends to use the same file format, there is a difference in the log format. As already stated the Android tagstore allows the users to observe any folder for file changes. Unfortunately, this feature introduces a problem when the files and tags of a tagstore are examined. In the desktop version of tagstore, the files reside inside the `storage` folder. Therefore, the Android tagstore needs to provide the path to the file in the log file. In Listing 1.4 there is a sample file log entry with file `cartoon.jpg`. The file was associated with the tags `test`, `cartoon`, and `android`. In addition, the relative file system path from the root folder of the external storage is prepended to the file name. Furthermore, the path separator is being escaped. In the Android platform the path separator is represented as a forward slash ("`/`"). Due to a restriction of the Python run-time, the path separator needs to be a back slash ("`\`"). Otherwise the run-time interprets the following character as a special character, which leads to undefined behaviour.

2 Synchronization Systems

In the previous chapter the tagstore research software as well as the Android tagstore were presented. In order to synchronize files between the Android and desktop tagstore, it requires a synchronization system. In this chapter established synchronization systems are analyzed. First, characteristics of synchronization systems are elaborated. Based on these characteristics, existing synchronization systems are grouped. Furthermore, the weaknesses and strengths of the synchronization systems are examined.

2.1 Characteristics of Synchronization Systems

Synchronization systems can be distinguished by several different factors. There are four characteristics, which allow to distinguish synchronization systems (Schütt, 2002, Section 2.1). First of all synchronization systems can be distinguished by their method of communication. Furthermore, synchronization systems can be distinguished by the point in time, when synchronization is performed. The next characteristic is the role of repositories. Finally, synchronization systems can be grouped by their understood file contents.

Synchronization systems have several options, how information exchange can be performed. Information exchange can either be sent to all systems residing in the same network, a group of systems in the same network, or to one system only. Schütt refers to these communication methods as *Broadcasting*, *Multicasting*, and *Unicasting*.

Concerning the point in time when synchronization is performed, there are two types. The types are named *online synchronization* and *offline synchronization*. Schütt defines online synchronization as:

2 Synchronization Systems

Online synchronization means that changes to a file are instantly propagated to all repositories. Therefore it is required that a steady communication channel between repositories consists (Schütt, 2002, Section 2.1.2).

Online synchronization systems need to propagate changes to all repositories. Schütt remarks that this requires on a tight integration with the underlying operating system core because such modification is only permitted with administrator rights. The integration is required in order to serialize file changes and to prevent file corruptions. Furthermore, Schütt mentions that online synchronization systems are typically implemented as a file system. Hence, access from user point of view is simple. Offline synchronization systems are not required to have a steady communication channel. Instead synchronization is triggered by the users or at a scheduled time. Once the synchronization has been completed, the repositories have the same contents.

Synchronization systems can be distinguished by the roles of the participating repositories. In general, there are systems which only support content changes at one repository. That repository is also named as the Master. Repositories, which are only allowed to adopt these changes, are named Slave. However, there are also synchronization systems which are capable of content changes at any repository. These are called Multi-Master synchronization systems.

In synchronization systems an event called conflict may occur. A conflict is an exception during the synchronization process. The exception is caused by situations, which the synchronization system can not handle. A sample conflict is the case, when a file is modified in both repositories. Synchronization systems can deal in many different ways with this event. If the file content is understood, the synchronization system can attempt to merge the file. If not supported, users need to resolve the conflicts manually. Alternatively users can define a policy, which is applied in conflict situations. An example policy is to ignore conflicted files, or to overwrite the file descending from the remote repository. However, such policies are not recommended as it is prone to errors. Hence, users should decide in conflict situations.

2.2 Online Synchronization Systems

Online synchronization systems use a steady network connection to access files. As already stated online synchronization systems are implemented as file systems. Online synchronization systems typically use a server system, which stores the files. The files in the server system construct a repository. Clients only embed the files in a virtual repository. As the files are physically stored at the server system, it does not need file synchronization (Schütt, 2002, Section 2.2.1). In the this section, network file systems and distributed file systems are elaborated.

2.2.1 Network File Systems

Network file systems use a server system to provide network access. Such systems appear like a network hard disk but are accessed in the same manner as local hard disks. Examples for a network file system are the Network File System (NFS) and Server Message Block (SMB). The advantages of network file systems are the simplified access and the elimination of file conflicts. File conflicts are not possible as there is only a single copy of a file, which resides at the server. However, some network file systems are implemented in star network topology. This requires all file accesses to be centralized on the server. Therefore, the server becomes a Single Point of Failure (SPOF). In the event of a server downtime the repository cannot be accessed. Furthermore, such systems do not scale (Schütt, 2002, Section 2.2.1).

2.2.2 Distributed File Systems

Distributed file systems (DFS) are network file systems of a new generation. Levy and Silberschatz (1990) define a distributed file system as:

A DFS is a file system, whose clients, servers, and storage devices are dispersed among the machines of a distributed system.

2 Synchronization Systems

In a conventional network file system, there is one server and many clients. However, in a DFS, data can be stored on several servers. This technique is also referred to as *replication*. Replication can enhance performance and reliability (Bzoch and Safarik, 2011). Another technology of DFS is the so called *transparency*. Clients should access files on a DFS in the same way as local files. Furthermore, clients should be unaware that the accessed files are distributed. Another aspect of DFS is the scalability. In a DFS additional servers can be added at any time, which increases the storage capabilities.

Examples of distributed file systems are the Hadoop Distributed File System (HDFS), Google File System (GFS), Moosefs, Andrew File System (AFS), Coda file system, and Lustre file system. A performance study of HDFS, Moosefs, and Lustre has been conducted by Bai and H. Wu. Andrea Arpaci-Dusseau (2012) analyzed the AFS. The successor of AFS is the Coda file system. It is presented by Satyanarayanan et al. (1990).

The HDFS is part of Apache Hadoop, which is an open-source framework for reliable, scalable, distributed computing.²³ The Apache Hadoop project also contains MapReduce, which is a framework for processing large data sets in-parallel on large clusters in reliable, fault-tolerant manner.²⁴

Microsoft provides a technology to arrange several SMB shares into one distributed file system. The technology named Distributed File System (DFS) lets files, which are distributed across multiple servers, appear to the users as if they reside in one place on the network. In addition, the File Replication Service (FRS) provides replication for files between servers (Microsoft Technet, 2005). Replication has also been brought to the NFS in version 4.²⁵

²³<http://hadoop.apache.org/> last visited on 9/29/2012

²⁴http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html last visited on 9/29/2012

²⁵Shepler et al., 2003

2.3 Offline Synchronization Systems

Offline synchronization systems are used in scenarios where a steady network connection is not feasible due to insufficient network reliability or network infrastructure. In addition, costs accumulated by steady network operation can require an offline synchronization system.

In this chapter a few offline synchronization systems are investigated. The first system explained is Rsync, which was developed by Andrew Tridgell and Paul Mackerras (Wikipedia, 2012a). Afterwards the Unison File Synchronizer, which is built upon Rsync, is analyzed.

2.3.1 Rsync

One of the first popular systems is Rsync. Based on the characteristics in Section 2.1, Rsync is a Master-Slave synchronization system. As a result Rsync does not synchronize repositories but duplicates the state of a source repository to a target repository. In general, this technique is also called mirroring.

Rsync is a system which supports mirroring of files and nested folders. Mirroring of local data as well as remote data is supported. In addition, Rsync can act as a Linux daemon. RSync uses an efficient algorithm to minimize data transfer. Rsync aims to minimize data transfer by implementing a delta-encoding algorithm. Delta-encoding algorithms are used to find the minimum difference between two files.

The Rsync algorithm is performed in five steps (A. Tridgell, 1999). There are two computer systems, named A and B, which are linked by a slow communication link. System A constructs list of files and directories, containing file names, ownership, permissions, size, and modification time. This list is then transmitted to system B, which sorts the list lexicographically by the paths. Afterwards, each file is inspected if it is candidate for an update. In general, files are excluded, when modification time and size are equal. However, Rsync can be configured to calculate a check sum for each file. This check sum is then used as a comparison for the equality of files instead of modification time and size. In principle this approach provides

2 Synchronization Systems

a more robust method for checking equality of files. Unfortunately, it slows down the mirroring process. Finally once a file in system B has been found, which differs from system A, the following steps are applied:

1. The system B divides the file into non-overlapping fixed size blocks of size k . The last block can be shorter. The size k is calculated for each file individually.
2. The system B generates two check-sums for each block. A weak check-sum called rolling check-sum and a stronger check-sum. Since protocol version 30 (release 3.0.0), the hash sum algorithm MD5 has replaced the older and broken MD4.
3. The system B transmits the calculated check-sums to system A.
4. System A also divides the file into non overlapping fixed size blocks. However, it also computes the rolling check-sum at each byte offset. When the rolling check-sum is matched to a block received from system B, the stronger check-sum is used to determine if the blocks are equal.
5. System A sends a list of instructions how to reconstruct the file to system B. The instructions can include references to received blocks or new data blocks. New data blocks are only sent when the check-sum of this block is not found.²⁶

$$a(k, l) = \left(\sum_{i=k}^l X_i \right) \mod M \quad (2.1)$$

$$b(k, l) = \left(\sum_{i=k}^l (l - i + 1) X_i \right) \mod M \quad (2.2)$$

$$s(k, l) = a(k, l) + b(k, l) * 2^{16} \quad (2.3)$$

The rolling checksum is computed by using the result of the first two formulas. Formula 2.1 calculates the sum of the byte values of the current block modulo M , which is defined as 2^{16} . Formula 2.2 calculates the sum of the products of the byte values with the offset $l - i + 1$ of the current

²⁶Andrew Tridgell, 2012

2 Synchronization Systems

block modulo M . Formula 2.3 defines the rolling check sum using the results of the previous formulas.

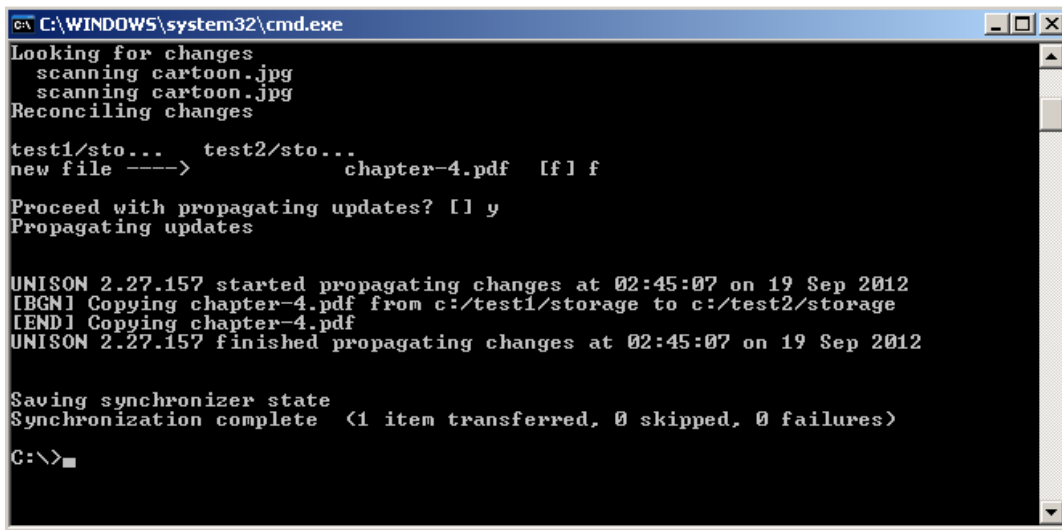
The Rsync algorithm uses rolling check-sum to find similar blocks. As already stated the rolling check-sum is computed for each byte offset. In order to save processing power, a property of the rolling check-sum is exploited. The rolling check-sum allows to re-use the previously computed check-sum at offset i to compute the new check-sum at offset $i + 1$.

Although Rsync can save unnecessary data transfer, there are cases where the transfer of the complete file is more efficient. The first case appears when the target file differs a lot from the source file. In this case Rsync will not find many matching blocks. Furthermore, the efficiency of Rsync algorithm depends on a carefully chosen block size. On one hand reducing the block size increases the data transfer, while the files are analyzed. On the other hand large block sizes reduces data transfer but also increases the chances of not finding a matching block. The performance gain of Rsync compared to full transferred files also depends on the available network bandwidth. As RSync is designed for low speed and high latency networks, the performance gain is reduced when faster networks are used. A study has shown that the deployment of Rsync becomes a disadvantage in comparison to full transferred files when the network operates at 100 Mbits speed (Schütt, 2002, Section 3.2).

2.3.2 Unison File Synchronizer

Unison is an offline synchronization system. The software is supported on many popular operating systems such as Linux, Mac OS X, Solaris, and Microsoft Windows. Unison is written in the OCaml programming language, which extends the Caml language with object oriented capabilities. Unison uses the RSync algorithm for efficient file transfer. However, Unison operates differently than RSync. RSync is used to mirror a repository, whereas Unison is able to detect file conflicts and let users handle them. A conflict happens when a file is modified in both repositories at the same time. RSync lets the users correct conflicts by choosing the appropriate

2 Synchronization Systems



```
C:\WINDOWS\system32\cmd.exe
Looking for changes
scanning cartoon.jpg
scanning cartoon.jpg
Reconciling changes
test1/sto... test2/sto...
new file ----> chapter-4.pdf [f] f
Proceed with propagating updates? [ ] y
Propagating updates

UNISON 2.27.157 started propagating changes at 02:45:07 on 19 Sep 2012
[BGN] Copying chapter-4.pdf from c:/test1/storage to c:/test2/storage
[END] Copying chapter-4.pdf
UNISON 2.27.157 finished propagating changes at 02:45:07 on 19 Sep 2012

Saving synchronizer state
Synchronization complete (1 item transferred, 0 skipped, 0 failures)
C:\>
```

Figure 2.1: This figure shows the output of Unison file synchronizer during a synchronization of two local folders

modified file. Furthermore, Unison performs a two way synchronization. Updates are propagated in both repositories instead of only one.

Unison collects the meta data of files and stores them in an internal Unison file archive format. The file format stores the file name, file path and last modification time. The archive is kept in the `.unison` folder, which is a sub folder of an user's home directory. The file's meta-data is created on the first run of Unison. After each subsequent run any changed meta-data is updated as well as the meta-data of new files is appended.

2.3.3 Syxaw File Synchronization Middleware

Syxaw is a file synchronization middleware, which was developed in the Fuego Core series at the Helsinki Institute for Information Technology (HIIT).²⁷ The Syxaw's name derives from Synchronizer with XML-awareness.

²⁷Lindholm, Kangasharju, and Tarkoma, 2009

2 Synchronization Systems

SyxAw is designed for resource limited devices such as mobile phones and personal digital assistants.

SyxAw supports a delta-encoding algorithm to minimize bandwidth during updates. Alike as Rsync, it can compress data to reduce bandwidth usage. In addition, SyxAw supports separate synchronization of metadata.

SyxAw performs synchronization of repositories by implementing a synchronization server. The synchronization server accepts incoming connections which use the SyxAw client protocol. This protocol uses the HTTP as the transport protocol and uses the standard *get* and *put* HTTP operations for file transfer.

SyxAw implements an XML reconciler. The reconciler uses the algorithm described in (Lindholm, 2004). SyxAw is integrated in Dessy, which is a desktop search and synchronization framework (Lagerspetz, Tarkoma, and Lindholm, 2010).

2.4 Cloud-based Synchronization Systems

Cloud-based synchronization systems are cloud storage systems. Cloud storage is an application of the cloud computing model. The National Institute of Standards and Technology (NIST) defines cloud computing as:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.²⁸

Cloud storage is a system of distributed data centers, which utilize cloud computing technologies like virtualization and offer an interface for storing

²⁸<http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf> last visited on 6/8/2012

2 Synchronization Systems

large sets of data.²⁹ This model is also referred to as Storage as a Service (SAAS). Cloud storage uses the IAAS service model of the cloud computing model to achieve the storage model. Furthermore, cloud storage systems may also use distributed file systems for their storage infrastructure. An example is Amazon. Amazon uses the Apache Hadoop framework in the Amazon Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (S3).

Cloud storage systems use a *cloud master-replica synchronization* approach (Uppoor, Flouris, and Bilas, 2010). These systems provide a master replica. The master replica is stored as a central copy in the cloud storage system. The users synchronize their replica with the master replica. In addition, users can also transfer changes to the master replica.

This section is divided into five sub-sections. The first sub-section elaborates the benefits of cloud storage systems. In the following sub-section, the usage patterns of cloud storage systems are described. In the third sub-section the security and privacy aspects of cloud storage systems are inspected. In the fourth sub-section the cloud storage access methods are explained. Finally, a selection of cloud storage providers is presented.

2.4.1 Cloud Storage System Benefits

The advantage of cloud storage systems is the accessibility of data at any-time and anywhere. There are five key benefits of cloud storage systems (J. Wu et al., 2010). These are ease of management, cost effectiveness, lower impact outages and upgrades, disaster preparedness, and simplified planning.

The first benefit is the ease of management. Software, data, and hardware are maintained in a simpler method in a cloud storage model than in a conventional IT infrastructure because this task is left to the providers of the cloud storage. In addition, network administration tasks such as enlarging storage space, can be performed with a web browser by using the cloud storage web administration interface.

²⁹http://www.sit.fraunhofer.de/content/dam/sit/en/studies/Cloud-Storage-Security_a4.pdf last visited on 6/12/2012

2 Synchronization Systems

The next benefit listed is the cost effectiveness. Since cloud storage enables the elimination of expensive systems and the necessary people involved to maintain it, it helps to reduce costs. These savings often exceed the cloud storage fees. In addition the high availability provided by cloud storage systems is unmatched. Furthermore the economies of scale achieved by cloud storage systems are only matched by very large organizations.

Data redundancy is another advantage of cloud storage systems. In general cloud storage providers achieve a high level of redundancy. As a result cloud storage systems provide a high availability, which eliminates service interruptions during network maintenance measures.

The next advantage displays during the occurrence of a disaster. Cloud storage systems need to have highly developed data redundancy policies and data recovery processes as their business model relies on availability and accessibility of the stored data. Therefore, these systems are highly trained for such an event.

The last benefit is titled simplified planning. Since cloud storage systems provide the feature of easy and instant capacity upgrades, it relieves planning for IT managers. The managers are no longer required to provide a detailed capacity plan. However, additional capacities can be acquired instantly and be released at any given time. This eliminates unnecessary acquisition costs or operational costs.³⁰

2.4.2 Cloud Storage Usage Pattern

Nowadays the usage of cloud storage systems is not reserved for companies for storing mission critical data. Instead it is also used by individuals for a lot of different purposes. In a study four individual use cases were described for choosing a cloud storage system. The use cases are named *backup*, *restore*, *synchronize*, and *share* (Borgmann et al., 2012, Section 2.1).

The *backup* use case performs a duplication of local data in the cloud system. The reason is to have another backup of the data, which is important

³⁰http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5565955&tag=1 last visited on 6/12/2012

2 Synchronization Systems

in the event of a hardware failure. In addition, the chance of theft or vandalism affecting the hardware is a good reason for using cloud storage systems.

The next use case is *restore*. As already stated cloud storage systems achieve a higher accessibility than conventional backup solutions. Moreover, cloud storage providers also provide the option to backup a file multiple times to increase durability. However, this durability improvement is typically connected with expenses. In addition, each changed file of users in the cloud storage system is also archived. Therefore, it is possible to restore user files to a previous version. This feature is efficient in fighting data corruption or other events which require a previous version of a file.

The *synchronization* use case gets important when multiple devices are used to access data. Users want to access the most recent version of a file regardless of the device involved, which can be a desktop, laptop, tablet, or a smart phone. In order to synchronize their files, users need to use a proprietary software of the cloud manufacturer. Typically, the software automatically detects changes and performs the appropriate changes. On the one hand if a local file is out of date, the software updates it to the latest version, on the other hand if the local file is newer than the latest version, the file in the cloud storage system is updated. However, it is possible that a file is edited by multiple persons at the same time. In that event the cloud storage software must notify the users of a possible conflict.

The last use case is *share*. This use case is important when a file needs to be shared with a group of people. Cloud storage providers need to provide facilities which comply with the share intent. Furthermore, the cloud manufacturers need to provide a mechanism to remove access from selected persons or everybody.

2.4.3 Cloud Storage Security Requirements

The security of cloud storage is a highly discussed matter as cloud storage systems are gaining increased popularity. There are five security requirements (Borgmann et al., 2012, Section 4).

2 Synchronization Systems

The first requirement called *registration* appears during the sign-up of a new user. Users are required to provide an email address when registering. If the email is not verified by the cloud provider, it is possible to launch an incrimination attack. The attacker A registers as a new user with the email address of user B. Next the attacker attaches illegal content on cloud storage account of user B and in turn notifies the authorities (Borgmann et al., 2012, Section 4.1).

The next requirement is the *transport security*. When files are transmitted to or received from the cloud storage service provider, it is necessary to apply encryption to prevent eavesdropping attacks. The current industry standard for transport encryption is SSL/TLS (Borgmann et al., 2012, Section 4.2).

The requirement labeled *encryption* has another security impact. Files saved in the cloud storage need to be encrypted. The prime reason is to prevent the cloud storage provider staff to access the files. In addition, the legal situation must be considered. Cloud storage providers based in the United States need to comply with the USA Patriot Act Justice, 2001. This act was signed into law in 2001 by the former president George W. Bush after the terror attacks of September 11th. The act allows authorities to gain access to data, even if the data is physically stored in a data center outside United States territory (Borgmann et al., 2012, Section 4.3).

The fourth requirement is named *sharing*. Files which are shared among a group of users can be accessed by the proprietary software or a web interface. In the latter case the generated URL by the cloud storage provider needs to be obfuscated when no login credentials are required. Otherwise attackers are able to guess the URL, which grants them access. Thus the URL needs not to include information about the users, files, or folder structures. In addition, the cloud storage provider needs to make sure that search engines do not index user files (Borgmann et al., 2012, Section 4.4).

The last requirement is called *deduplication*. Cloud storage providers are required to store millions of user files. Many of these files are just a duplicate of another, for example audio, image, or video. Thus it makes sense to only save one example of these duplicated files. The software of cloud storage providers therefore checks if a certain file is already existing in the storage

2 Synchronization Systems

cloud. If the file is already present, then the file is not uploaded. As a result bandwidth and storage are saved. However, this approach yields to a problem. An attacker can create an account and place the file into a folder, which is monitored by cloud storage software. If the file is not uploaded, then the attacker knows that this file is already present in the cloud. Although the attacker does not know which user has uploaded the file, the attacker knows that at least one user has uploaded the file (Borgmann et al., 2012, Section 4.5).

2.4.4 Cloud Storage Interfaces

Cloud storage providers need to provide a set of interfaces to let the users access their data. Typically, cloud storage providers manufacture a proprietary software for the users. Ideally, the software supports all the use cases, which have been elaborated in Section 2.4.2. In addition, cloud storage providers may also grant access to the files by providing a web interface. This access method is popular when the client software is not supported on the user's platform. In addition, security restriction policies may also prevent the usage of the software client.

Cloud storage providers may also present an API to give third party developers a method to use their services in their products. The advantage is the creation of a biotope of new services around the cloud storage service. A successful example is the cloud storage provider Dropbox. Many additional services have been developed for Dropbox.³¹ In general cloud storage providers expose access by using web services, which use the HTTP as the transfer protocol. The data can be encoded by using the Simple Object Access Protocol (SOAP)³² or by using the REpresentational State Transfer (REST) protocol which utilizes RESTful web services (Fielding, 2000). Nowadays the REST protocol has become more popular than SOAP.³³ The first reason is that RESTful services are easier to develop. In addition, the web service does not

³¹<http://thenextweb.com/apps/2011/04/15/the-top-10-best-dropbox-services-addons-and-hacks/> last visited on 6/14/2012

³²<http://www.w3.org/TR/2003/REC-soap12-part1-20030624/> last visited on 6/14/2012

³³<http://www.oreillynet.com/pub/wlg/3005> last visited on 6/14/2012

2 Synchronization Systems

need to maintain the session state for each connected user as all information is passed in the requests. As a result, multiple servers can be used to handle user requests as each request contains the full session state. Furthermore, it eliminates the requirement of a proprietary client software as any arbitrary web browser can be used to access the web service (Rittinghouse, 2009, Section 7.5.4).

2.4.5 Cloud Storage Providers

In this section a selection of commercial cloud storage providers is presented. The providers are evaluated in regards to online storage capabilities, platform availability, and platform restrictions. In addition the available cloud storage interfaces of each provider are examined.

Dropbox

Dropbox is a cloud storage provider, which was founded in 2007 by Drew Houston and Arash Ferdowsi. It offers three types of storage models. The *basic* model includes 2GB of free storage and a maximum of 16GB extension for referred clients. The *pro* model offers 100GB, 200GB, or 500GB at a monthly / yearly fee. Finally the model labeled *Teams* is intended for companies. It offers 1TB storage with 5 included user accounts and is priced per year. Additional users can be added with an extra charge. Furthermore, Dropbox offers phone support and Active Directory integration for Microsoft Windows Servers.

Dropbox supports file versioning. In the basic model file changes such as modification and deletions are saved for thirty days. Afterwards a file cannot be recovered. However, Dropbox provides unlimited file history for Dropbox pro model customers as well as Dropbox team model customers.

In October 2011 Dropbox reached the 50 million user mark. The client software of Dropbox is developed for many platforms, such as Microsoft Windows, Mac OS X, Linux, Android, iOS and BlackBerry OS. Figure 2.4.5 shows the Android client. Users can also manage their account by using a

2 Synchronization Systems

web browser. The Dropbox client and server back-end are written in Python programming language except for the iOS and Android platform due to memory constraints.

Dropbox uses the Amazon Simple Storage Service (s3) for hosting of the user files. In addition, it uses Amazon's Elastic Cloud Compute (EC2) when transferring files to the cloud (Wang et al., 2012). As Amazon is a host based in the United States, the USA Patriot Act can be used to give authorities access to user files. However, Dropbox recently signed the Safe-Harbor agreement.³⁴ The agreement requires the signer to follow seven principles and must be renewed every twelve months. The compliance with these principles is overseen by the Federal Trade Commission (FTC). In March 2011 the FTC demonstrated by charging Google Inc. that the compliance of the agreement is important.³⁵

Dropbox uses the deduplication technology to save storage and bandwidth. Dropbox uses the SHA-256 hash algorithm to identify duplicate files. In addition the hash algorithm is used to detect file changes. The client software then uploads the new file to the cloud storage service automatically. However, it is inefficient to upload the whole new file when only a few parts have been changed. The Dropbox client overcomes this problem by creating multiple hash sums of a file. Each analyzed file is split into chunks of 4096 KB.³⁶ Next, the hash sums of the chunks are calculated and compared with the last hash sum. Finally, only altered chunks are uploaded. Currently Dropbox is the only provider, which is capable of uploading partial changed files (Axel Kossel, 2012).

As Dropbox is becoming a big player in the cloud storage, it has received attention of security researchers, which revealed several security flaws.³⁷ A flaw which was caused by a faulty deduplication scheme was revealed in April 2011. The open source developer Wladimir van der Laan released Dropship, which enabled Dropbox users to remotely copy other users files

³⁴<https://www.dropbox.com/help/238> last visited on 6/15/2012

³⁵<http://www.ftc.gov/opa/2011/03/google.shtm> last visited on 6/15/2012

³⁶<http://blog.fosketts.net/2011/07/11/dropbox-data-format-deduplication/> last visited on 6/18/2012

³⁷<http://dereknewton.com/2011/04/dropbox-authentication-static-host-ids/> last visited on 6/18/2012

2 Synchronization Systems

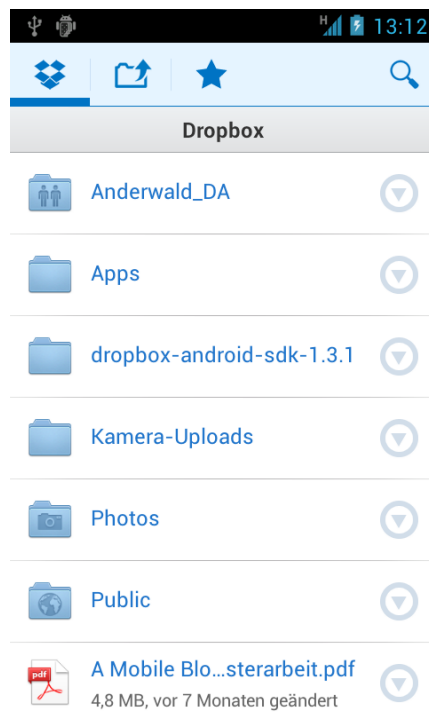


Figure 2.2: This screenshot shows the proprietary Dropbox client running on the Android platform. It displays six folders and one PDF file. The folder `Anderwald_DA` is a shared folder. Dropbox signals this with a pictogram.

2 Synchronization Systems

provided that hash sum is known. The file automatically appeared in the Dropbox folder after Dropship made the Dropbox's server believe a user wanted to upload this file. The tool was recognized by Dropbox as an opportunity to distribute copyrighted material. As a result, Dropbox changed the deduplication scheme making Dropship obsolete. Dropbox now uses single user deduplication (Borgmann et al., 2012, Section 8.4). However, Dropbox has an open flaw regarding the security requirement during registration. Since email addresses are not verified during registration, an attacker can launch an incrimination attack.³⁸

Dropbox allows third party developers to integrate Dropbox into their products. Dropbox provides an official SDK for multiple platforms (iOS, OS X, Android) and programming languages (Python, Ruby, Java). For other platforms, developers can directly use the underlying REST API. However, Dropbox generates a unique key and secret for each application. That key and secret are used during authentication. A key and secret can be obtained by registering the application at the Dropbox developer site³⁹. Dropbox uses the OAUTH protocol for authentication and authorization. In addition, Dropbox provides the opportunity to revoke OAUTH access tokens by using the web interface. This feature is important in the case a smart phone gets lost or stolen. Furthermore, Dropbox provides two different modes for third party applications. In the first mode named *App folder* applications only have read-write access to a specific folder and related sub-folders residing in the Apps folder. In this connection the Dropbox synchronization back-end uses the folder tagstore. In the other mode called *Full Dropbox* applications have full access to all folders residing in Dropbox. As this method enlarges the security risks by malicious applications, Dropbox grants only full access in exceptional cases during approval phase for a wide release.

SkyDrive

SkyDrive is a cloud storage service run by Microsoft. In contrary to Dropbox which uses the Amazon S3 as a storage back-end, SkyDrive is entirely

³⁸See Section 2.4.3

³⁹<https://www.dropbox.com/developers/apps> last visited on 9/29/2012

2 Synchronization Systems

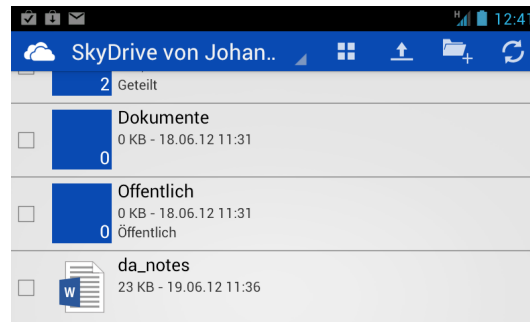


Figure 2.3: Screenshot of SkyDrive client running on the Android platform.

operated by Microsoft. It provides 7GB of free storage and 25GB for long-term users. In addition, it provides 20GB, 50GB, and 100GB storage upgrades for business clients. SkyDrive is available for Microsoft Windows and Mac OS X. In addition, Microsoft released a client for Windows Phone, Android and iOS. Figure 2.4.5 shows the Android SkyDrive client. Furthermore, Microsoft also provides a SDK for developers. In opposition to Dropbox, SkyDrive does not impose any security restrictions for third party applications. As soon as user has been logged in, every application has access to all files and folders. SkyDrive uses the OAUTH protocol like Dropbox.

SkyDrive is part of Windows Live, which is a family of combined services. It provides online services as well as software applications. A remarkable feature of SkyDrive is the ability of editing documents in a web browser. SkyDrive supports Word, Excel, PowerPoint, and OneNote document formats. Moreover, file versioning is supported for these document formats. Unfortunately, file versioning is not supported for other file types. Furthermore, SkyDrive supports audio and video streaming. However, it requires the Silverlight plugin on the Microsoft Windows platform.

As already stated Microsoft Windows has a few restrictions regarding the file names. These restrictions are also applied to SkyDrive. No files containing reserved characters or reserved names can be uploaded to SkyDrive or be created by utilizing the web interface.

2 Synchronization Systems

Google Drive

Google Drive is a new storage service run by Google. The service was launched on April 24 2012. Google uses their distributed data center infrastructure as a storage back-end. In addition, Google uses the distributed file system Google File System (GFS). The file system is elaborated by Ghemawat, Gobioff, and Leung.

In general the service provides 5GB of free storage. For users who require a lot of storage, there are storage extension available starting from 25GB up to 16TB at a monthly fee.⁴⁰ Google Drive also supports the editing documents in a web browser. Besides the Microsoft office document formats, it also supports viewing Adobe Portable Document (PDF), Scalable Vector Graphics (SVG), archive formats (ZIP, RAR), and more.⁴¹

Google Drive is available for multiple platforms. At time of writing the platforms Microsoft Windows, Mac OS X, iOS, and Android are supported. Figure 2.4.5 shows a screen shot of the Android client.

Google Drive does not oppose any file name restrictions. Furthermore all characters are supported. However, this brings up problems in Microsoft Windows as there are reserved file names and characters as already stated in Section 1.3.4. In order to deal with this limitation, the Microsoft Windows Google Drive client converts reserved characters to underscores.

⁴⁰<http://support.google.com/picasa/bin/answer.py?hl=en&answer=39567> last visited on 6/19/2012

⁴¹<http://support.google.com/docs/bin/answer.py?hl=en&answer=1738646> last visited on 6/18/2012

2 Synchronization Systems

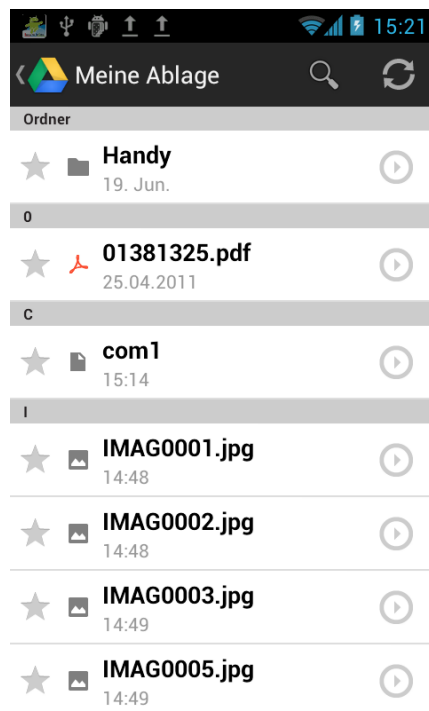


Figure 2.4: Google Drive client running on the Android platform.

3 tagstore Synchronization

In the last chapter file synchronization systems were represented. In order to synchronize files and associated tags, it requires a decent algorithm. The algorithm must also fit the requirements of tagstore. Furthermore, the special needs of the Android platform extends the requirements. In this chapter, the characteristics and needs of the platform are investigated. Afterwards, the requirements of a synchronization algorithm are analyzed. In addition, the synchronization algorithm is inspected concerning the border cases. Finally, the synchronization algorithm is presented in detail.

3.1 Synchronization Platforms

The synchronization algorithm must take the platform requirements of the Android and desktop tagstore into account. Table 3.1 presents a list of factors, which describe the properties of these platforms.

Property	Android	Desktop
Storage capability	32GB SDHC / 2TB SDXC	several TBs
Network connection	slow, variable bandwidth frequent disconnects	fast steady connection
Usage behavior	read / inspect files	read / modify files
Platform	Android(Linux based)	Linux, Windows, Mac OS X

Table 3.1: The table lists the most important factors, which influence the design of the synchronization algorithm.

In contrast to the personal computing platform the Android platform's storage is very limited. Android devices provide storage extension with

3 tagstore Synchronization

memory cards of type SDHC or SDXC cards. Currently the maximum storage size is 128 GByte.⁴² Furthermore, the network connection is not as reliable as the personal computing platform, which mostly connects on broadband Internet connection. In addition, the usage behavior also differs between these platforms. Editing documents is an expensive task due to the small screen size and keyboard. As a result, it is assumed that smart phone users generally only view documents. However, as the trend goes to larger screens, the possibility is kept as a requirement for the future.⁴³

3.2 Synchronization Requirements

In Section 3.1 the special characteristics of the platforms have been presented. Based on these characteristics requirements have been derived. In general there are requirements for the synchronization algorithm as well as the underlying communication channel. Furthermore, the tagstore software also incorporates additional needs for the synchronization system.

3.2.1 Synchronization Algorithm Requirements

In order to design a decent synchronization algorithm, the functional requirements must be considered. The following list presents a selection of requirements, which should be supported by the synchronization algorithm.

- The synchronization algorithm should be designed to enable a quick synchronization of user files. The resulting synchronization tool should operate seamlessly and require as less user input as possible.
- The synchronization algorithm should perform a two way synchronization. Changes should be propagated in both repositories.

⁴²<http://hothardware.com/News/SanDisk-Introduces-Worlds-Largest-SDXC-Memory-Cards/> last visited on 7/9/2012

⁴³<http://www.strategyanalytics.com/default.aspx?mod=reportabstractviewer&a0=7194> last visited on 7/9/2012

3 tagstore Synchronization

- The synchronization algorithm should be able to detect conflicts. Conflicted files should not be modified or deleted, until the users have made the appropriate decision.
- The synchronization algorithm needs to provide a mechanism to ignore files. These files are not included during the synchronization. An example deployment is revealed when android and desktop tagstore are synchronized. The desktop tagstore contains files, which consume too much space or are not needed for the Android tagstore.
- The synchronization algorithm should be able to detect new used tags. These tags should then be automatically assigned during the synchronization.
- The synchronization algorithm should detect deleted files. Since disk space is limited for Android devices, deleted files should not be restored after the synchronization has been completed.

3.2.2 Communication Channel Requirements

The communication channel is responsible for transferring files. Therefore, it is important to have a reliable communication channel. The following list shows the most important requirements of a communication channel.

- The communication channel must establish a communication link within a short period of time. If it fails to establish a link, it can retry immediately.
- The communication channel can be established at any given time and remains active until the synchronization has been completed or the synchronization process is abandoned.
- The communication channel must detect incomplete or corrupted transfers. Furthermore, it must ensure the integrity of the transferred files.
- The communication channel must ensure security. All data transfers need to be performed securely. If the communication channel is established outside a LAN, then the data must be protected by applying transport encryption.

3.2.3 tagstore Synchronization Requirements

The tagstore system is a software, which brings up specialized requirements for a synchronization system. The synchronization system needs to adopt to these requirements. The following list presents these requirements.

- The synchronization system should be able to synchronize two tagstores. The synchronization system should update files during this process. Furthermore, it should also add new tags to the corresponding TagTrees automatically without any user interaction.
- The synchronization system needs to support Android and desktop tagstores. Android tagstores need special treatment as they are stored on mobile phones, which are not always connected to a PC. Since the desktop tagstore system interprets inaccessible tagstores as a serious error condition, Android tagstores need to be handled specially.
- The synchronization system must provide a GUI. The GUI lets the users select the tagstores from a list.
- The synchronization system should be able to synchronize any arbitrary pair of tagstores. It should not make a difference if both tagstores have already been synchronized. Furthermore, all types of tagstore should be sync-able such as Android or desktop tagstore.
- The synchronization system needs a mechanism to signal an in-progress synchronization. This mechanism is needed to detain tagstore from interfering with the synchronization system.
- The synchronization system should provide two different synchronization modes. These modes are called full-sync and tag-sync. In the first mode all files are synchronized. In the other mode only files are synced, which are tagged with a special tag. That tag must be adjustable and changeable at any time. However, changes to this tag during a synchronization process must not have an effect.
- The synchronization system must preserve the file attributes such as creation date, modification date, and access date when a file is being synchronized. The file attributes are necessary for conflict detection.

3.3 Synchronization Algorithm Conflicts

File synchronization algorithms need to be carefully designed to match the requirements raised by their environment. In Section 3.2 these requirements have been elaborated. However, it needs a detailed inspection on the border cases. As already stated a synchronization system has to recognize and deal with exceptional cases, which are also known as conflicts. These conflicts have a big influence on the design of synchronization algorithms. In this section a list of conflicts is described.

3.3.1 File Conflict Classes

The first file conflict class is referred to as the *create* conflict. Such a conflict happens when a file is being added to both repositories before a synchronization is performed. The synchronization system has no way of knowing which file should have preference.

A very common conflict case is the *write* conflict. This conflict case appears when a file is modified in both repositories upon synchronization. In general, synchronization systems need to understand the file contents to resolve this conflict.

The next conflict case is the *delete* conflict. When a file is deleted in one repository, the synchronization system has three options. The first option is to restore the deleted file. The next opportunity is to delete the file also in the other repository as well. The latter option is to ignore the deleted file and do nothing.

The *rename* conflict is a special case for synchronization systems. As the name implies a file is renamed. This brings up problems as the synchronization system detects a deleted file as well as a new file. Synchronization systems are dealing differently with this conflict. Ideally, synchronization systems can detect such events and handle them appropriately. One possible solution is to cache the meta-data of files and compare them with new files. Alternatively the synchronization system can compute the hash sums of the files. However, both approaches are not error resistant. As a result, many synchronization systems do not handle the rename conflict.

3 tagstore Synchronization

The *delete-write* conflict is an extension of the delete conflict. It appears when a file is deleted in one repository and modified in the other repository. Synchronization systems have the same options as in the delete conflict.

3.3.2 Meta-data Conflict Classes

The meta-data of the files need to be preserved during synchronization. Meta-data consist of file modification date, file access date, file creation date and the tags. Regarding the file date a synchronization system has two choices during the update of a file. The synchronization system can either use the time stamp of the replacement file, or it uses the time stamp of the synchronization initiation. However, tags construct new conflict classes.

The associated tags in the tagstore deliver a different set of conflicts. The conflicts resemble file conflict classes elaborated in the previous section. However, their cause is different. Typically, a file is associated with one or more tags. The first conflict arises when a tag is removed in one repository upon synchronization. The synchronization system may now either restore the tag or remove the tag in the other repository. This conflict is referred to as the *delete-tag* conflict.

As already stated a rename conflict is a special conflict for a synchronization system. Likewise, a *rename-tag* conflict needs to be handled carefully. The synchronization system can either re-assign the removed tag or ignore the removed tag. In general, users do not expect the renamed tag to be restored.

Tags are associated with a file. If all tags of a file are detected to have changed during the synchronization, the file content has changed. The synchronization system recognizes this situation as a *semantic-tag* conflict. In such case the users need to decide which file and tags should be kept.

3.4 tagstore Synchronization Algorithm

Synchronization of the Android and desktop tagstores is a challenging task due to platform differences and requirements. In this section the developed synchronization algorithm is explained. The synchronization operating modes, platforms, and back-ends are elaborated. Finally, the limitations of the synchronization algorithm are highlighted.

3.4.1 Synchronization Algorithm

The synchronization algorithm synchronizes two tagstores including their files and associated tags. The file list of the tagstores is retrieved by accessing `store.tgs` in the folder `.tagstore`. Depending on the operating mode, all files or a selection of these files are used during synchronization. The differences and consequences are highlighted in Section 3.4.3.

The synchronization algorithm performs a two-way synchronization. This means all changes of both tagstores are propagated to the other tagstore. The synchronization algorithm performs the following steps:

1. The synchronization system collects a list l_a of files from tagstore A depending on the operating mode.
2. The synchronization system tries to synchronize the file f_n of tagstore A with the other tagstore B .
3. The synchronization algorithm notes any conflicts appeared during the synchronization of file f and continues to synchronize the next file f_{n+1} .
4. After all files have been processed of tagstore A , the files of tagstore B are collected in list l_b and steps 2-4 are repeated.
5. After all files have been processed, conflict processing is triggered. If there are no conflicts then the synchronization is complete. Else-wise the synchronization system prompts the users to resolve the conflicts manually.

In Listing 3.1 the detailed synchronization algorithm is displayed. The synchronization algorithm is performed on a list of files, which is derived from

3 tagstore Synchronization

```
1 for each file  $f_i$  in list  $l$  of  $t_{source}$ 
2
3   if  $\exists t_{target}(f_i)$  then
4     if  $t_{source}(f_i) \equiv t_{target}(f_i)$  then
5       sync new tags
6     else if  $\nexists syncdate(t_{source}(f_i))$  then
7       conflict( $f_i$ )
8     else if not modified( $t_{source}(f_i)$ ) then
9       sync new tags
10    else if not modified( $t_{target}(f_i)$ ) then
11      sync file and new tags
12    else if  $tags(t_{source}(f_i)) \equiv tags(t_{target}(f_i))$  then
13      conflict( $f_i$ )
14    else
15      conflict( $f_i$ )
16
17  else
18    if  $\exists syncdate(t_{target}(f_i))$  then
19      if modified_after_sync( $t_{source}(f_i)$ )
20        conflict( $f_i$ )
21      else
22        skip file
23    else if  $\nexists sync\_folder(t_{target}(f_i))$  then
24      sync new file and tags
25    else if  $t_{source}(f_i) \equiv sync\_folder(t_{target}(f_i))$  then
26      sync all tags
27    else
28      conflict( $f_i$ )
```

Listing 3.1: This listing shows the synchronization algorithm of tagstore.

3 tagstore Synchronization

one of the tagstores. This tagstore is referred to as the *source* tagstore. The tagstore, whose files and tags are updated is labeled as the *target* tagstore.

The algorithm starts to check if the target tagstore also stores a file with the same file name. If the target tagstore does have such a file, there are multiple choices available. If the files are identical, only new tags from the source tagstore are synchronized to the target tagstore. If the files are not equal, the algorithm checks if that file has already been synchronized once. This verification is performed by obtaining the synchronization date from the synchronization store file with the *syncdate* function. The synchronization store file is elaborated in Section 3.4.2. If it fails to fetch a synchronization date, the algorithm detects a *create* conflict. The conflict is explained in Section 3.3.1.

In Line 8 the algorithm tests if a file from source tagstore has been modified. This operation is performed with the *modified* function. Alike if the file has not been modified, only new tags are synchronized. Otherwise the file from the target tagstore is probed for modification (Line 10). If the algorithm detects no modification, then the file of the target tagstore is updated and new tags of the source tagstore are added.

In Line 12 the algorithm checks if the tags of the source tagstore and target tagstore are equal. At this point the algorithm has already established that the files have been modified in both tagstores since the last synchronization. If all associated tags are the same, a *write* conflict is active. Otherwise the algorithm handles it as a *semantic* conflict.

The synchronization algorithm handles *delete* conflicts. This is achieved with the help of the synchronization store file. In Line 18 the algorithm tries to obtain a synchronization date from the synchronization store file. Since the file was already synchronized once, a synchronization date can be recovered. Afterwards, the algorithm tests if the file was modified since the last synchronization date. In case the file was modified, a *delete-write* conflict is raised. Otherwise the algorithm ignores the file and continues with the next file.

The synchronization algorithm needs to handle the special requirements of the Android tagstore. As a reminder, the Android tagstore requires that

3 tagstore Synchronization

unique file names although it supports observing multiple folders. Furthermore, it supports excluding files from the tagstore. Therein lies the problem, if the files are excluded from the Android tagstore and are stored in the default synchronization folder of the Android tagstore.⁴⁴ During synchronization new files are stored in the default synchronization folder. If there is already a file with the same name, the files would be replaced. In order to circumvent this problem, the synchronization algorithm explicitly checks for files with the same name in Line 23 with the function *sync_folder*. If there is no such file, the file and tags are synchronized. On the contrary if the file exists, it is checked if it is equal to the file of the source tagstore. In this case all tags of the file from the source tagstore are added. Finally, if the files remain different, a *write* conflict is raised.

3.4.2 Synchronization Store File

The synchronization algorithm uses a so-called *synchronization store file* for a variety of purposes. The file format uses the INI file format like the tagstore store file.⁴⁵

The synchronization store file is created upon each synchronization. If the file does not yet exist, an empty store file is constructed. If it already exists, it is opened. The synchronization store file is stored in the `.tagstore` folder of the actively synchronized tagstore. In order to support synchronization with more than one tagstore, the file name of the synchronization store file is defined with pattern. The pattern is set as `<target tagstore name>sync.tgs`.

The synchronization store file saves files and tags during synchronization. By using the synchronization store file many conflict types can be detected. From the file conflict class the *create*, *delete* and *delete-write* conflict can be detected. Write conflicts can be identified by saving the file modification date or hash sum of the file. Furthermore, meta-data conflicts can also be detected. However, the synchronization algorithm only identifies and processes *semantic-tag* conflicts.

⁴⁴See Section 1.3

⁴⁵See Section 1.1.2

3 tagstore Synchronization

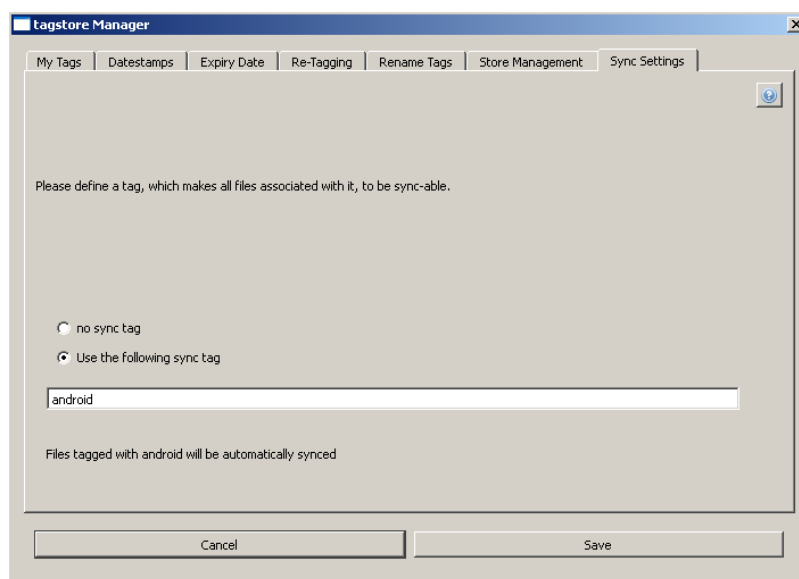


Figure 3.1: This screenshot displays the tagstore manager synchronization setting tab. The special tag is set to *android*. This tag is used in *tag-sync* synchronization mode.

3.4.3 Synchronization Algorithm Modes

The synchronization algorithm supports two modes as already listed in the requirements. In the *full-sync* mode, all files are included in the file list for the synchronization algorithm. This operating mode is intended for synchronizing two desktop tagstores.

The other mode named *tag-sync* is designed for synchronizing a desktop tagstore with an Android tagstore. When this mode is used, the TagTrees are only updated in the desktop tagstore. In addition, this mode only synchronizes files, which have been used with a special tag. This tag can be customized in the tagstore manager in the tab *Sync Settings*. Figure 3.4.3 displays the setting. The special tag is set to "android" by default.

3.4.4 Synchronization Back-ends

The synchronization algorithm is implemented in two independent systems. In the first system it is integrated into the desktop tagstore. In the other system, it is integrated into the Android tagstore application. The Android tagstore supports two synchronization back-ends. The first back-end implements a cloud based synchronization using the Dropbox storage service. The second back-end supports synchronization with an network share using the SMB protocol.

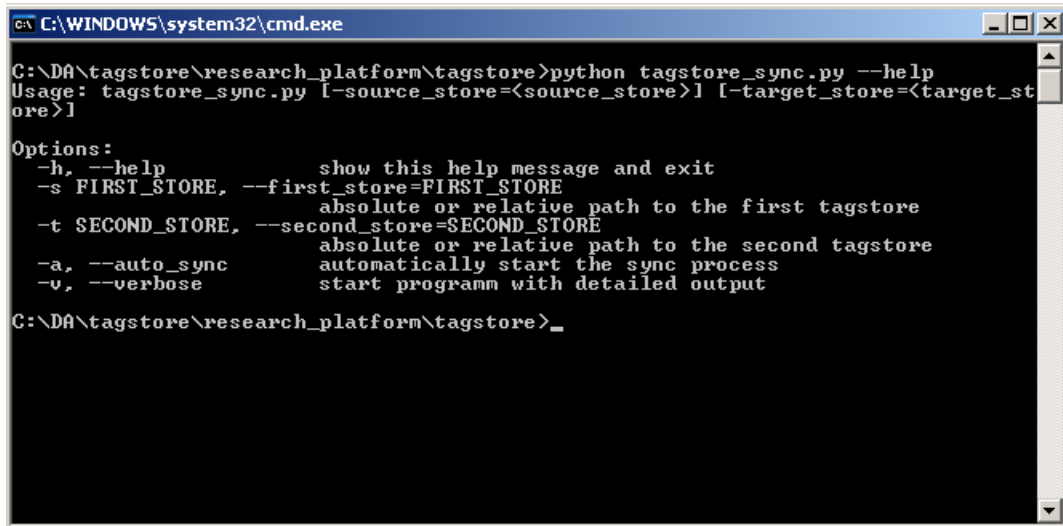
Desktop tagstore Synchronization Back-end

The Desktop tagstore synchronization back-end is integrated in the desktop tagstore research software. It is also written in the Python programming language and can be launched with the Python interpreter by running `tagstore_sync.py`. The synchronizer supports four command line options, which facilitates custom synchronization deployments. Figure 3.4.4 displays the available command line options.

The synchronizer allows the users to select a tagstore for synchronizing from the command line. This can be executed by providing the parameters `-s` and `-t` and appending the absolute or relative path of the tagstore. The command line refers to the parameters as first tagstore and second tagstore. As the synchronization algorithm performs a two-way sync, the first tagstore and second tagstore are synchronized equivalent. Furthermore, the synchronizer supports starting the synchronization automatically. The users can trigger the synchronization by providing the parameter `-a`. However, it is required that the first and second tagstore parameters are provided.

Figure 3.4.4 displays the initial screen of the synchronizer. When the users provided the first or second tagstore parameter, the GUI only displays the desired tagstore in the corresponding list. However, the Android tagstore is a special case. Since Android tagstores are accessible as an USB mass storage device, the corresponding path can change. But it is a requirement that

3 tagstore Synchronization



```
C:\WINDOWS\system32\cmd.exe
C:\DA>tagstore\research_platform>tagstore>python tagstore_sync.py --help
Usage: tagstore_sync.py [-source_store=<source_store>] [-target_store=<target_store>]
Options:
  -h, --help            show this help message and exit
  -s FIRST_STORE, --first_store=FIRST_STORE
                        absolute or relative path to the first tagstore
  -t SECOND_STORE, --second_store=SECOND_STORE
                        absolute or relative path to the second tagstore
  -a, --auto_sync       automatically start the sync process
  -v, --verbose         start programm with detailed output
C:\DA>tagstore\research_platform>tagstore>_
```

Figure 3.2: This screenshot displays the command line of the desktop tagstore synchronization back-end. It is invoked by launching the Python interpreter with the tagstore_sync.py and the parameter *-help*.

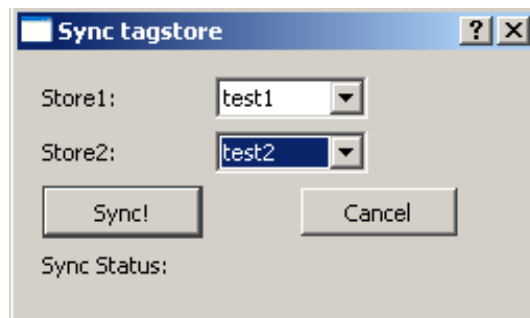


Figure 3.3: This screenshot displays the start-up screen of the synchronizer. It has two lists, which enable the users to select the desired tagstores for syncing.

3 tagstore Synchronization

all tagstores listed in the *stores* section are valid.⁴⁶ As a workaround the Android storage path is stored as a value in the tagstore's main configuration file. The key is named *android_store_path* and must point to the root folder of an Android tagstore. If the path is valid, the synchronizer system shows the tagstore "Android" in the tagstore list.

The desktop tagstore synchronizer detects the appropriate synchronization mode. Before the synchronization is started, the synchronizer checks whether the tagstore is an Android or a desktop tagstore. Each tagstore stores its configuration in the file *store.cfg* in the folder *.tagstore*. The key *android_store* defines the tagstore type. If one of the tagstores has a non zero value for that key, then the tag-sync mode is used. Otherwise the full-sync mode is chosen.

The *tag-sync* mode requires additional processing. The Android tagstore uses a database for speed instead of the *store.tgs*. After a synchronization, the Android's *store.tgs* is updated but the changes are not yet applied in the database. Gladly the Android tagstore supports adding these changes with a synchronization screen. The screen can be accessed with the menu key and selecting the *Sync* button. Figure 3.4.4 shows a sample screen. The figure also demonstrates the last time stamp of the synchronization.

Figure 3.4.4 displays the screen after a successful synchronization. In that figure the tagstores *test1* and *test2* have been synchronized. In addition, the figure shows the completion time stamp of the synchronization.

Dropbox Synchronization Back-end

The Android tagstore implements synchronization with a tagstore, which is hosted inside a Dropbox account. In this way users can synchronize files on the fly without needing to connect their phone to a PC.

The implementation the Dropbox's Android SDK version 1.5.1, which is directly available at the Dropbox's developer site.⁴⁷ The Dropbox synchro-

⁴⁶See Section 1.1.1

⁴⁷<http://www.dropbox.com/developers> last visited on 9/5/2012

3 tagstore Synchronization

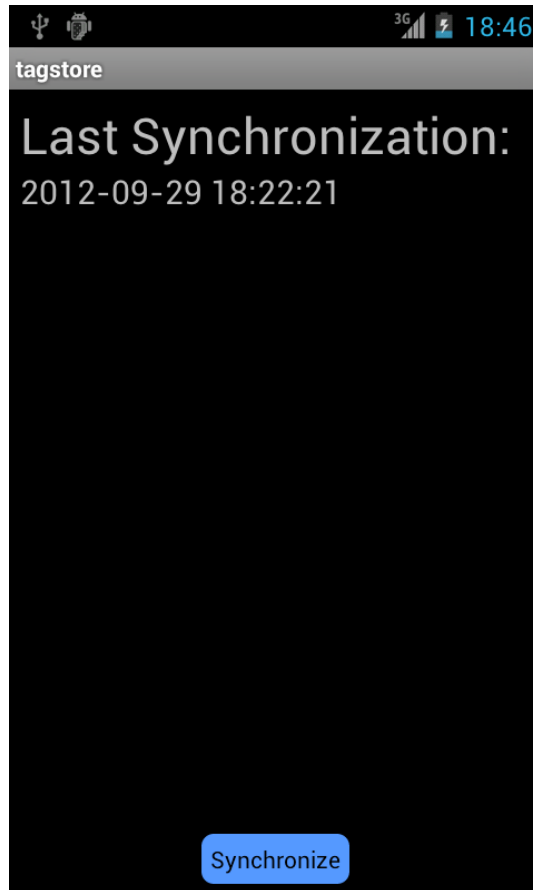


Figure 3.4: This screenshot shows the screen when the Android tagstore has applied the synchronization changes after a *tag-sync* using an usb type sync.

3 tagstore Synchronization

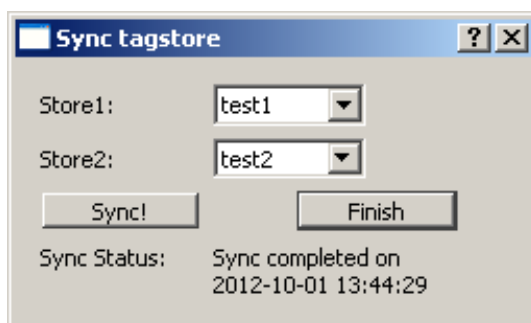


Figure 3.5: This screenshot displays the screen after the tagstores named *test1* and *test2* have been successfully synced. It also shows the time when the synchronization has been completed.

nization settings can be accessed from the *Android Configuration Settings* screen when the synchronization type is set to *Sync via Dropbox*.

The Android Dropbox SDK provides many classes, which facilitate dealing with the Dropbox web service. It provides functionality for uploading or downloading files and querying meta-data of files. In addition, the SDK provides support for authentication and authorization, which is handled with the OAUTH protocol.

Dropbox requires third party applications to be authorized by the users. Otherwise malicious applications can access users Dropbox's files. The authentication is performed either in a web browser accessing the authentication URL, or by using the installed Dropbox application. Figure 3.4.4 shows a screenshot of the Android Dropbox client's authentication screen.

After the users have confirmed access, the synchronization back-end receives an access token. This token is then stored and used for later authentication requests. Figure 3.4.4 shows a screenshot after the users have granted the synchronization back-end access. The *Authenticate* button is disabled as the access was already granted. The *Unlink* button performs the opposite of the *Authenticate* button. It closes the link to the Dropbox service and deletes the access token. As a result, users need to grant access to the synchronization back-end before the Dropbox service can be accessed.

3 tagstore Synchronization

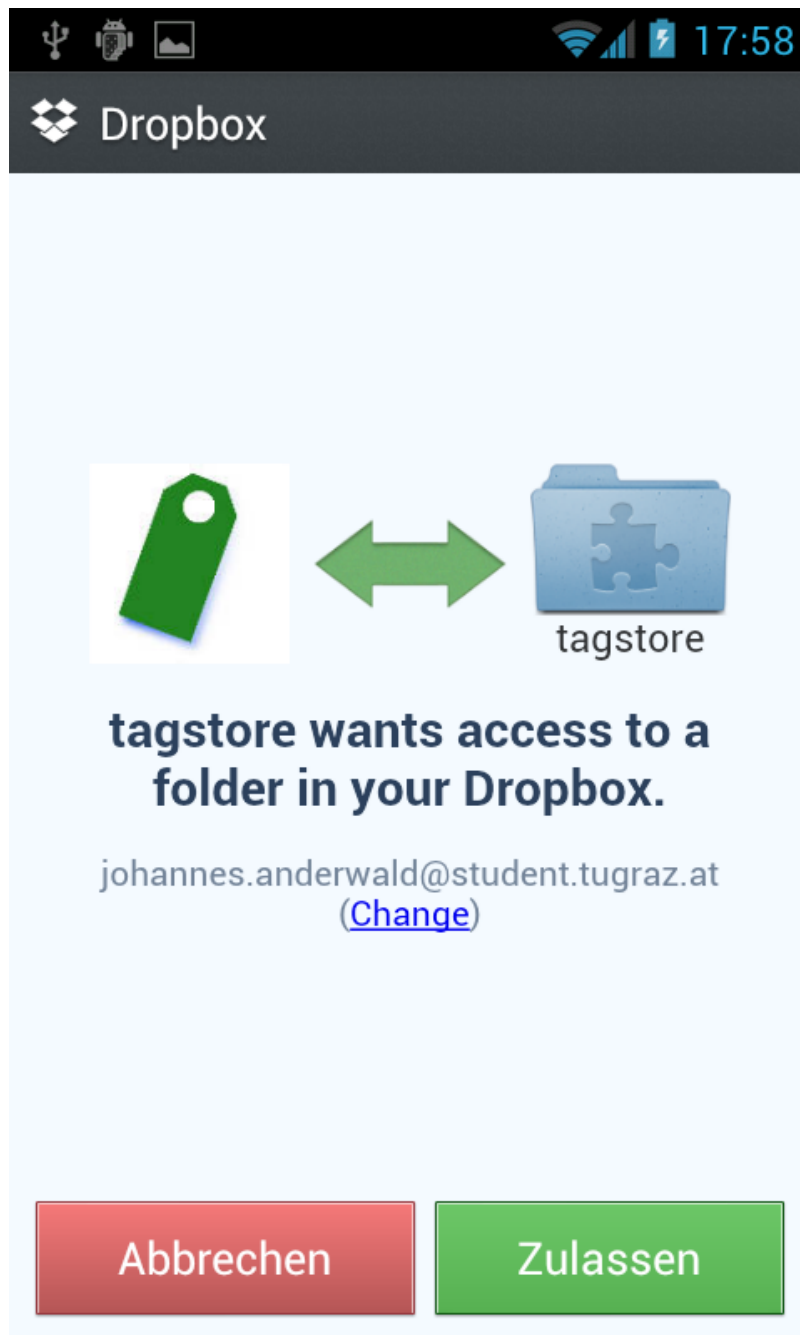


Figure 3.6: This screenshot displays the authentication screen of Dropbox, when a third party application wants to authenticate. If Dropbox is not installed on the Android phone, then a web browser is used to handle the OAUTH authentication.

3 tagstore Synchronization

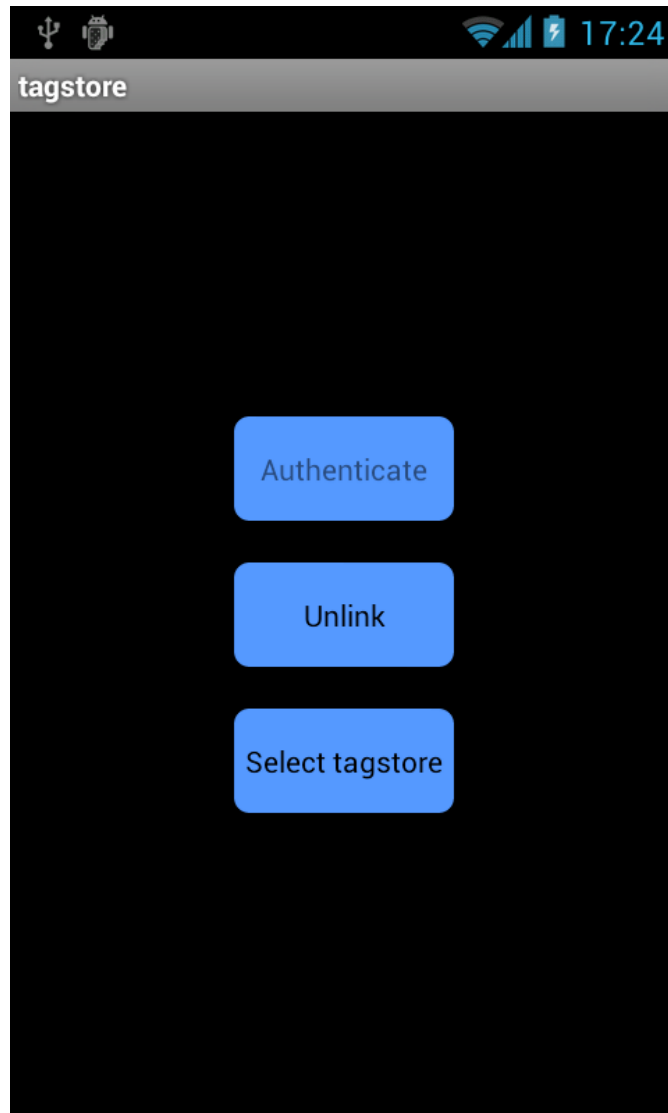


Figure 3.7: This screenshot displays the Dropbox settings screen when Dropbox is used for synchronizing. The screen provides three buttons named *Authenticate*, *Unlink*, and *Select tagstore*. The first button requests users to grant authentication access. Currently the user has already granted access already authenticated. Therefore the button has been disabled. The second button lets the user remove the granted authentication for the synchronization back-end of tagstore. The third button allows the user to select a tagstore. The tagstore must reside in the Dropbox folder Apps/tagstore.

3 tagstore Synchronization

After the users have granted access to Android tagstore, the users need to configure a synchronization tagstore. The Dropbox synchronization supports synchronization with multiple tagstores. However, only one tagstore can be synchronized at the same time. The tagstore can be selected in the *Dropbox Settings* screen by using the *Select tagstore* button. It then starts to search for tagstores because it has no information in which folder a tagstore resides. In the first step the synchronization back-end performs a query to retrieve all folders residing in the `App/tagstore` folder. Each folder is then examined if it has a file `store.tgs` in the sub-folder `.tagstore`. All folders having this file are added to the list of tagstores. This list is presented to the users to select a tagstore for synchronization. Figure 3.4.4 shows this dialog, where one tagstore named *store1* was found.

As soon as the users have granted access to Android tagstore and have configured a synchronization tagstore, they are able to synchronize with the configured tagstore. The synchronization is initiated by using the *menu* key and selecting the *sync* button.

SMB Synchronization Back-end

The Android tagstore supports synchronization with a SMB share. The synchronization back-end uses the JCIFS library.⁴⁸ The library is an open source implementation of the CIFS/SMB network protocol in Java. The back-end uses the JCIFS library in version 1.3.17. The size of the Android tagstore application increases by 178 KB.

The SMB setting dialog can be accessed when the synchronization type is to *Sync via SMB*. The synchronization type can be altered in the synchronization configuration option of the configuration screen. Furthermore, the SMB setting screen provides the same options as the Dropbox setting screen. Therefore, the screen has the same graphical layout as the Dropbox setting screen, which is presented in Figure 3.4.4.

The SMB synchronization back-end requires access credentials in order to access a SMB share. The credentials can be entered in the SMB configuration screen by displaying the SMB settings dialog. Figure 3.4.4 visualizes this

⁴⁸<http://jcifs.samba.org> last visited on 9/10/2012

3 tagstore Synchronization



Figure 3.8: This screenshot displays the tagstore selection screen when users want to set up a synchronization tagstore. The screen enumerates all available tagstores in the Apps/tagstore folder. The enumerated tagstores are then presented to the users.

3 tagstore Synchronization

dialog. The back-end requires the user name, password, server address and share name to be entered. The settings are verified as soon as the connect button is pressed. The back-end then attempts to connect to the *SMB* share. If the connection succeeds, its settings are saved and the screen is dismissed.

As soon as the synchronization back-end has established a connection, a synchronization tagstore needs to be selected. Similar to the Dropbox synchronization back-end, it can be accessed with the *Select tagstore* button.

The algorithm for enumerating the tagstores follows the same design pattern as in the Dropbox synchronization back-end. The algorithm enumerates all files and folders in the configured share path.

After the synchronization tagstore has been selected, users are ready to start to synchronization. The synchronization can be invoked by using the *menu* key and using the *sync* button.

3.4.5 Synchronization Conflict Handling

The synchronization algorithm detects file conflicts and meta-data conflicts. The conflict handling is left entirely to the users. The conflict resolution is handled by displaying a GUI dialog. The dialog lets the users choose if the file of the one tagstore should be replaced by the file of the other tagstore. Figure 3.4.5 shows two sub-figures. The left sub-figure shows the desktop tagstore synchronizer, whereas the right sub-figure displays the Android tagstore synchronizer.

As the synchronization algorithm processes the file list of each tagstore independently, conflicts will be reported each run. As a result, some conflict types will be listed twice. Therefore, the synchronization system checks before the conflict is handled, if the conflict has already been resolved by the users.

3 tagstore Synchronization

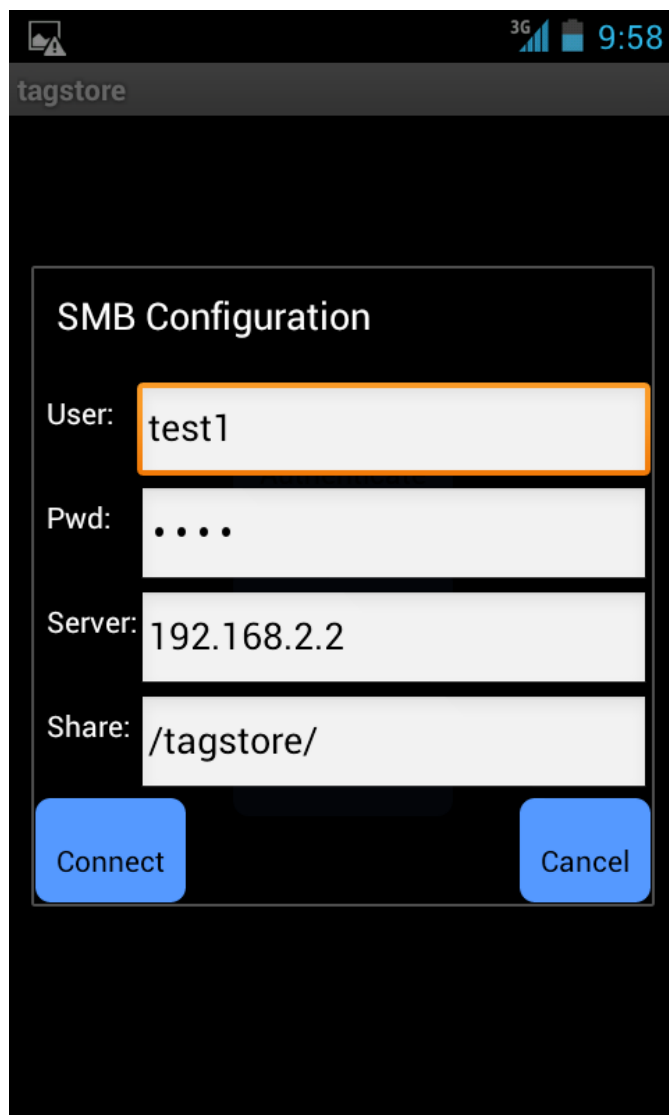
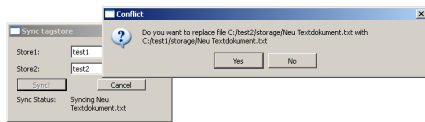
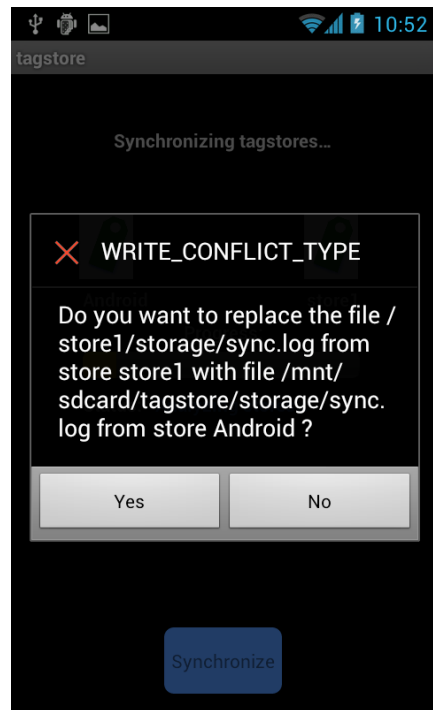


Figure 3.9: This screenshot displays the SMB configuration screen. The screen is displayed when the users have activated the *Sync via SMB* option in the synchronization configuration type and is about to connect to a server. In order to connect to a SMB share, user name, password, server address and the share name are required. If the connection succeeds, then the access credentials are saved and restored when the screen is invoked again.

3 tagstore Synchronization



(a) This screenshot shows the screen of the desktop tagstore synchronizer when a conflict is handled. The synchronizer lets the users choose if the file from one tagstore should replace the file from another tagstore.



(b) This screenshot shows the screen of the Android tagstore when a *write* conflict needs to be resolved. The dialog is presented when a conflict is found during the synchronization. The users can either replace the file or resolve the conflict later.

Figure 3.10: Android / desktop Synchronization Conflict

3.4.6 Synchronization Serialization

The synchronization system needs to ensure that accesses to the `store.tgs` are performed serialized. Otherwise corruptions can occur when the synchronization system and the tagstore application modify it simultaneously. In addition, the synchronization system needs to detect if another synchronization system is already active. Furthermore, the synchronization system also needs to verify that tagstore is not observing file changes in the storage folder. The reason is that tagstore will display the tagging dialog when a new file is stored although the synchronization system has already added these tags.

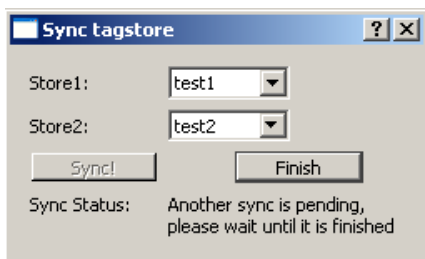
As the tagstore application and synchronization application are not running in the same process, it needs inter-process communication (IPC) to exchange information. A well established solution is to use a so-called *lock file*. The file is stored in known location and has a fixed name. The file contains the process identification number of the synchronization process.

The lock file solves the synchronization problems elaborated above. The synchronization system and the tagstore application are able to detect if a synchronization system is active. The examination is performed by reading the process identification number and testing if the process is still alive. If the process is still running, then the synchronization system does not start the synchronization and the tagstore application ignores file changes. If the process identification number is invalid, the lock file is removed and the synchronization is initiated.

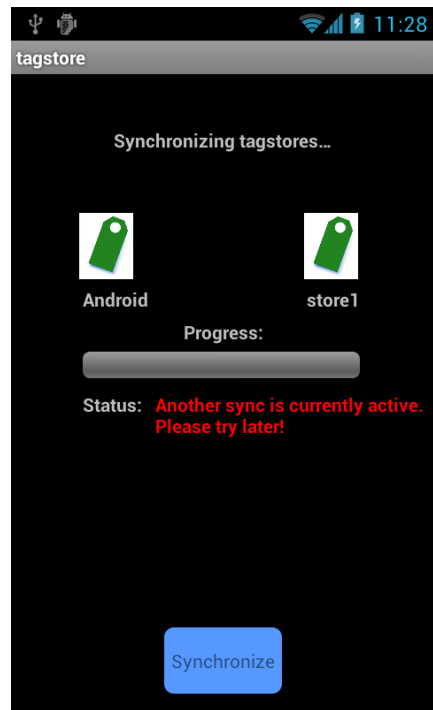
Figure 3.4.6 displays the stand-alone synchronization tool and the Android synchronization back-end. In both figures the corresponding synchronization system has detected that another synchronization system is active and modifying the selected tagstores. Users need to defer the synchronization until the other synchronization system is finished.

The lock file is a solution for synchronizing access to tagstores. However, the model needs to be adopted when synchronization is performed networked. The Android tagstore supports networked synchronization by using a SMB share or the Dropbox storage service. When the synchronization is executed networked, the synchronization system only has access

3 tagstore Synchronization



(a) This screenshot shows the desktop tagstore synchronizer, which detected that another synchronizer is running and modifying the desired tagstores.



(b) This figure shows the android tagstore Synchronization Serialization. It detected that another synchronization system is running and modifying the desired tagstore, which is configured in the *SMB Settings* screen or the *Dropbox Settings* screen.

Figure 3.11: Android / Desktop Tagstore Synchronization Serialization

3 tagstore Synchronization

to files. Therein lies the problem that manifests when the lock file is created. The Android tagstore synchronizer can not provide a process identification number as there is no synchronization process running at the target system. In order to overcome this limitation the Android synchronizer stores the identifier *android* in the lock file. The tagstore application and the stand-alone synchronization tool now have a method to detect networked synchronization processes. Unfortunately, this approach brings a disadvantage concerning a hanging Android synchronization system. The Android synchronization system can possibly at sometime hang due network reliability. As a result, the Android synchronization system and the stand-alone synchronization tool are blocked on retry attempt. Therefore, it is necessary to manually delete the lock file in the storage directory of the corresponding tagstore.

3.4.7 Synchronization Limitations of tagstore

The tagstore synchronization does have a few limitations. First of all it does not support synchronization of folders. Folders are not supported in the Android tagstore as it requires an external file browser.

Another limitation is the rename support. The tagstore synchronization system uses the tagstore store files when performing the synchronization. Since these files are updated when a file is renamed, it contains no information on renamed files. If a file exists in both tagstores and is renamed in one tagstore, it is synchronized with the new name in the other tagstore. As a result, renamed files appear duplicated in the other synchronized tagstore. The same limitation also applies to tags but in a different context.

The Android network-based synchronization implementation has a restrictions concerning the TagTrees. The desktop tagstore uses file system folders and links to implement the TagTree. As already stated on Microsoft Windows family it uses shortcuts whereas on Linux it uses symbolic links. However, the Android synchronization implementation has no knowledge whether the remote tagstore is hosted on Linux, Mac OS X, or Microsoft based. In addition, the Android framework provides no facilities for creating symbolic links or shortcut files. As a result, the TagTree in the remote

3 tagstore Synchronization

tagstore is not updated. As a workaround users can synchronize that tagstore with another tagstore using *full-sync* mode. In this mode the TagTree are rebuild during the synchronization.

4 Synchronization Evaluation

There are many systems, which provide a similar functionality as tagstore. These tagstore alternatives have been investigated (Binder, 2012). However, these systems provide less support for synchronization. In this chapter a selection of tagstore alternatives is examined. The alternatives are investigated concerning their workings, visualization, and synchronization capabilities.

4.1 TaggedFrog

The first alternative presented is TaggedFrog. TaggedFrog is developed by Andrei Marukovich. It is available from the author's web site.⁴⁹ The application is distributed as free-ware and requires the .NET 2.0 framework. Currently, the latest version is 1.1. Furthermore, the application is closed source. However, TaggedFrog provides an API for plugins. At present there are two plugins available with source code in the download section of TaggedFrog. The first plugin called *croak* lets users preview audio files. The other plugin labeled *meta* is a tag extractor. It can extract tags from the file name and from Windows meta-information.

TaggedFrog supports tagging of files and URLs. However, TaggedFrog has a different method of performing the tagging process. Unlike tagstore it does not monitor a defined folder for file changes. Instead users are required to initiate the tagging process. This can either be performed by dragging the file into the application window, or by using a menu in the Windows Explorer.

⁴⁹<http://lunarfrog.com/taggedfrog/> last visited on 9/10/2012

4 Synchronization Evaluation

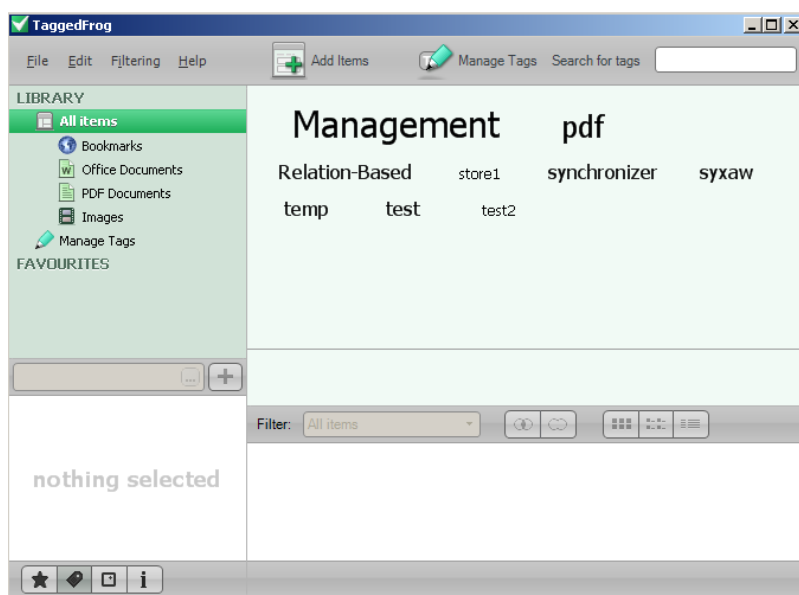


Figure 4.1: This figure shows the TaggedFrog startup screen. On the right side there is a list of tags, which have been used to associate files.

TaggedFrog uses a database for storing files and tags. Furthermore, it presents the tags in a tag cloud view within the application. Figure 4.1 demonstrates the view.

TaggedFrog supports importing and exporting files with their associated tags. TaggedFrog uses the comma-separated values (csv) style file format. Values are delimited by a semicolon. Each file entry begins on a new line with a value named *local*, followed by the complete path of the file. The tags are appended after the path. Each tag is separated by a semicolon. Furthermore, TaggedFrog also exports URLs. Each URL entry starts with a value *url*, followed by the actual URL, and ends with the used tags.

```
local;C:\store1\storage\Syxaw Middleware.pdf;pdf;syxaw;synchronizer  
local;C:\store1\storage\sync.log;sync;log;2012-09
```

This is a sample file, which was exported by TaggedFrog. It contains two entries. The first entry is the file *Syxaw Middleware.pdf*, which is associ-

4 Synchronization Evaluation

ated with the tags *pdf*, *syxaw*, and *synchronizer*. The other entry is the file `sync.log`, which is associated with *sync*, *log*, and the date *2012-09*.

As already mentioned, TaggedFrog supports importing of csv files. TaggedFrog imports files and tags in an additive method. New files and their used tags are added automatically. New tags for existing files are joined with existing tags. Identical to the tagstore synchronization, TaggedFrog does not delete tags, which have been removed in the imported files. As expected deleted entries are not removed during the import.

TaggedFrog does not support synchronization of files. In order to provide a similar functionality as the tagstore synchronizer, it needs an external synchronization back-end. The synchronization back-end can either be an online synchronization system like SMB, or an offline synchronization system. The Unison file synchronizer can be used for that purpose, provided that the file paths are extracted from the csv. However, TaggedFrog imports files with an absolute path. Therefore, the files must be mirrored in the same path in the target system.

The proposed synchronization system has several drawbacks. First, the synchronization system only performs an one-way sync of the meta-data. As a result, users need to perform the import and export process at both TaggedFrog systems. The next disadvantage displays when an offline synchronization system is used. The offline synchronization system requires to understand the csv format. Furthermore, it must be started manually.

4.2 Tables

Tables is a commercial tag based file management software.⁵⁰ According to its online manual, the name stands for “tag-bubble”.⁵¹

Tables supports tagging of files, folders, and URLs. Tables stores the relationship between files, folders, URLs, and associated tags in a database. The tags are displayed in a virtual view of the application.

⁵⁰<http://tables.net/> last visited on 9/11/2012

⁵¹http://http://tables.net/wiki/index.php/Tables_Manual_%28EN%29 last visited on 9/11/2012

4 Synchronization Evaluation

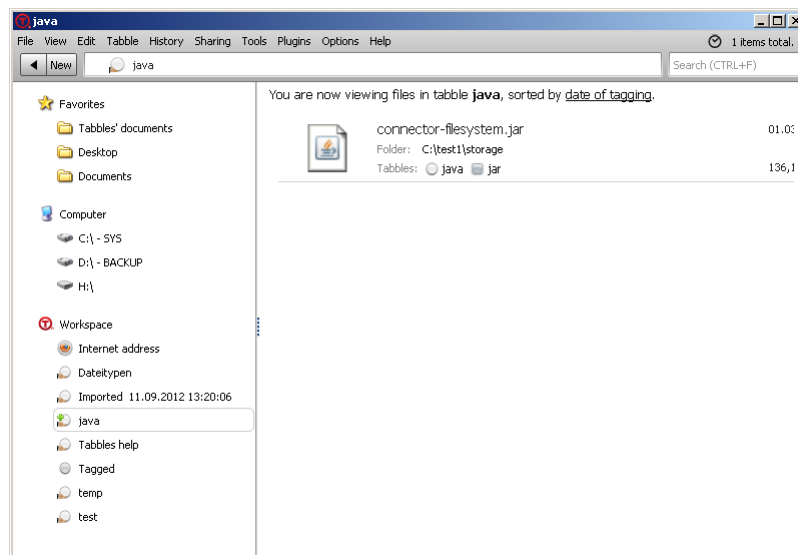


Figure 4.2: This figure shows the table named *java*. It has one file *connector-filesystem.jar*. On the left side, there is a list of tables. The tables labeled *java*, *temp*, *test* have been created by the users. The other tables were set by the Tabbles application.

4 Synchronization Evaluation

A tag is called a *table*. Figure 4.2 displays the table named *java*. That table contains one file `connector-filesystem.jar`. On the left side there are three lists shown. The list named *Favorites* stores the important folders or tables. The next list displays local hard disk drives and network drives, which can be navigated into. The last list referred to as *Workspace* shows predefined tables and tables created by the users.

Tables supports importing and exporting of the database in an XML file, which is embedded in a ZIP file. Furthermore, Tables supports sharing of the database. The database is shared by configuring a target folder and executing the *Synchronize now* button. Tables then exports the current database. After it has completed, it imports databases, which have been shared by other users. It is also possible to share a single table. Thereby the XML file only contains the shared table.

Tables does not support file synchronization. Though this functionality can be performed by an online synchronization system, offline synchronization systems are an additional option, though the XML schema is undocumented.

In comparison to TaggedFrog, Tables only performs an one-way meta-data synchronization as well. Indeed, the synchronization of the meta-data is simplified by using the share feature. In combination with an online synchronization system, it is a solution for synchronizing meta-data in a networked system.

4.3 Taggtool

Taggtool is a commercial file management software.⁵² The stand-alone client is referred to as *Taggtool Desktop* and the server is called *Taggtool Business Server*. Taggtool is available as shareware and can be tried for 30 days. Taggtool supports tagging of files, folders, and bookmarks. Currently, only an installer for the Microsoft platform exists.

⁵²<http://www.taggtool.com/> last visited on 9/13/2012

4 Synchronization Evaluation

The Taggtool client and server are written in Java. Therefore, it requires Java runtime, which is bundled with the application installer. However, the Java runtime is placed in a sub folder inside the Taggtool installation folder. As there is no Java update functionality provided, users manually need to update their Java runtime, which is a requirement due to security flaws.

The following sub-sections describe the Taggtool Desktop application and the Taggtool Business Server. In addition, the different operating modes are explained.

4.3.1 Taggtool Desktop

Taggtool Desktop allows tagging of files from different work flows. Files can be tagged by using a context menu from the Windows Explorer. Furthermore, files can also be tagged by using a built-in file browser of Taggtool Desktop. In addition, users can select hard disk volumes or folders, which are monitored. These files are listed under the *Fresh Files* tab.

Taggtool Desktop does not provide a virtual TagTree for browsing the tag hierarchy. Instead it provides a search interface. Files can be searched by their tags, a modification date range, or content type filter. Figure 4.3.1 displays the search screen of Taggtool.

Taggtool Desktop uses a database for storing files and their associated tags. Taggtool Desktop uses the relational database Apache Derby⁵³ of the Apache Software Foundation. Apache Derby is an open source software and distributed under the Apache License 2.0 version.

Taggtool Desktop supports backing up of tagged files. The files are packed inside a ZIP archive in an user defined folder. In addition, the archive is named with the current date stamp. Since files can be distributed on different drives, Taggtool Desktop stores all files with their full path and drive letter.

Taggtool Desktop does not remove deleted files from the database. Therefore, deleted files must be removed manually by the users. However, Taggtool

⁵³<http://db.apache.org/derby/> last visited on 9/14/2012

4 Synchronization Evaluation

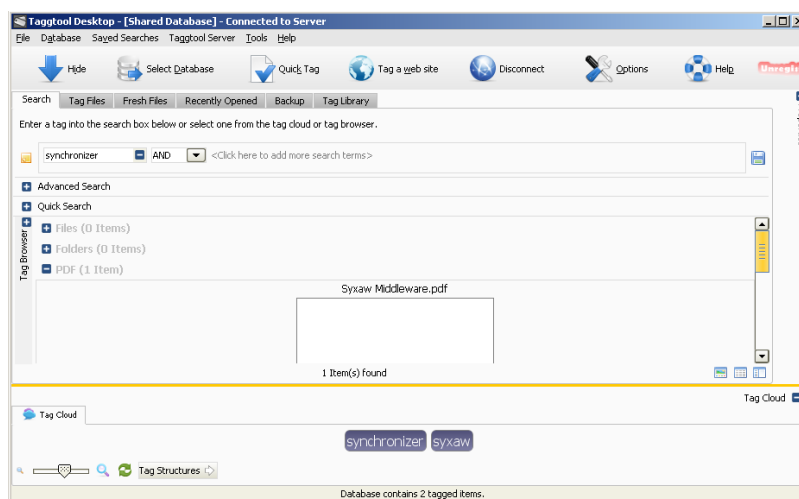


Figure 4.3: This figure shows the search screen of Taggtool Desktop. The screen supports searching of files by their used tags, modification date range, or a content type filter. The figure shows the search result for the tag *synchronizer*. The file *Syxaw Middleware.pdf* was found.

Desktop provides a method to manage moved files. Users can provide a folder or volume path, which contains the moved files. When a file in that location is found with the same name, Taggtool Desktop updates the path of the file. Figure 4.3.1 displays this dialog. In contrast to removed files, Taggtool Desktop detects renamed files and automatically updates the references. However, it requires that Taggtool Desktop is running while the file is renamed.

```
"OBJECTTYPEID", "PATH", "VOLUMELABEL", "FILE", "CSVTAGS", "RATING", "DESCRIPTION", "ALIAS"  
"0", "C:\test\storage", "SYS(C:)", "Syxaw Middleware.pdf", "synchronizer,syxaw", "0", "", ""  
"0", "C:\test\storage", "SYS(C:)", "cartoon.jpg", "cartoon,jpg", "0", "", ""
```

This is a sample file, which was exported by Taggtool Desktop. Taggtool Desktop supports importing and exporting the database in a csv style format. The first line declares the export fields. The first field called *OBJECTTYPEID* defines the type of the entry. Files use the identifier “0”, whereas bookmarks have the identifier “2”. The next field defines the path, followed

4 Synchronization Evaluation

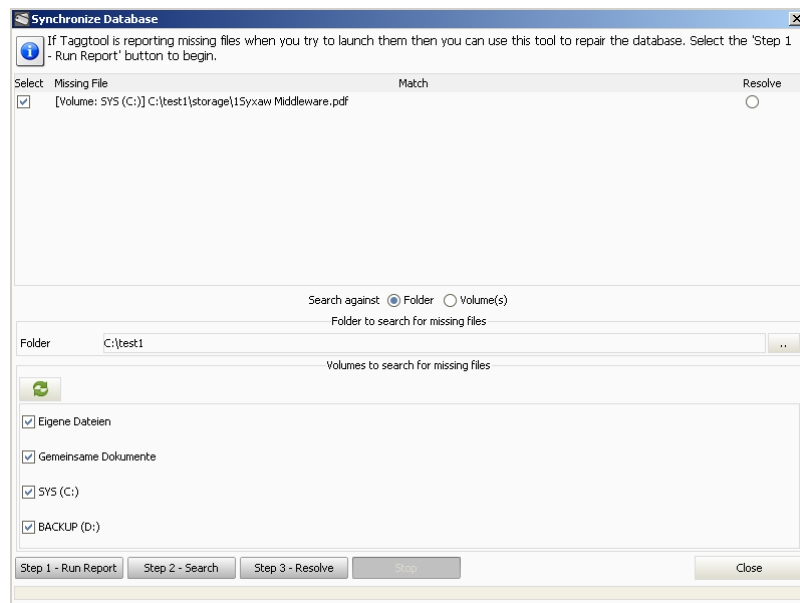


Figure 4.4: This figure shows the synchronize database screen of Taggtool Desktop. The figure shows that one file was detected to be missing. The dialog lets the users provide a folder or volume path to search for the moved file.

4 Synchronization Evaluation

by the volume label. The field labeled *FILE* contains the name of the file or the URL of the bookmark. The other fields are self-explanatory. The sample file contains two entries. The first entry is the file *Syxaw Middleware.pdf*, which is associated with the tags *pdf*, *syxaw*, and *synchronizer*. The other entry is the file *cartoon.jpg* which is associated with the tags *cartoon* and *jpg*.

As already stated Taggtool Desktop supports tagging of bookmarks. Bookmarks can be tagged with the *Tag a web site* button. Furthermore, Taggtool Desktop can import bookmarks from a Del.icio.us⁵⁴ account. In order to import bookmarks, Taggtool Desktop needs the user name and password. Del.icio.us uses HTTP authentication and passes the results in JSON format. Taggtool notifies the users with a dialog saying that the import is complete.

Taggtool Desktop provides the ability to backup the files. However, this is no sufficient solution for file synchronization. Therefore, it requires again an online synchronization system like SMB network protocol. In terms of meta-data synchronization, only an one-way synchronization is performed.

4.3.2 Taggtool Business Server

The Taggtool Business Server is intended for multi-user environments inside a LAN. Taggtool Desktop supports connecting to the Taggtool Business Server. By connecting to Taggtool Business Server, all Taggtool Desktop users have access to the same database.

Taggtool Business Server is deployed as a service in Microsoft Windows. The application installer configures the service to start automatically during system boot. The service supports TCP/IP connections and accepts requests on port port 1527. Taggtool Business Server uses the Distributed Relational Database Architecture (DRDA) protocol for encapsulating SQL commands and responses. The current standard is released in version 5 (The Open Group, 2012).

⁵⁴<http://delicious.com> last visited on 9/14/2012

4 Synchronization Evaluation

When Taggtool Desktop is connected to the Taggtool Business Server, it requires tagged files to be in a SMB share. The reason is that all users need to be able to access the file, which is not possible otherwise.

Since Taggtool Desktop users all access the same database, the meta-data is always updated for all users. In addition, Taggtool Desktop relies on the SMB network file system, which eliminates the file synchronization problem. Although in fact the Taggtool software truly does not perform synchronization, it achieves a similar functionality as the tagstore synchronization system. However, it must be stressed Taggtool software is currently available on Microsoft Windows platforms. Furthermore, it provides no support for mobile clients. Finally, it only works for users residing in the same LAN.

4.4 Evaluation Summary

In this chapter a selection of tagstore alternatives is represented. Table 4.1 summarizes characteristics of tagstore and its alternatives. Full support for a given category is displayed with the character ("x"), minimal support is shown with ("*"), and no support with a blank.

There are three main categories. The category *Tagging* presents features of the corresponding application. The ability to tag files, folders, and URLs are displayed. In addition, the *TagTrees* category determines, whether the TagTree is visualized.

The category *Synchronization* deals with the synchronization abilities. The sub-category *SMB Sync* determines if synchronization is supported with a SMB share. The *Cloud Sync* sub-category shows the ability to synchronize with a cloud storage provider such as Dropbox.

The category *Platform* shows the availability of the application on the corresponding operating system. The PC platforms Windows, Mac OS X, Linux as well as the mobile platform Android are shown.

Table 4.1 shows a comparison of tagstore and its alternatives. The features of the mobile tagstore and desktop tagstore are combined. However, it is

4 Synchronization Evaluation

Systems	Tagging				Synchronization			Platform			
	Files	Folders	URLs	TagTrees	SMB Sync	Cloud Sync	Meta-data	Windows	Mac OS X	Linux	Android
TaggedFrog	x	x	x	x	*		x	x			
Tabbles	x	x	x		*		x	x			
Taggtool Desktop	x	x	x		*		x	x	*	*	
Taggtool Business Server	x	x	x		x		x	x	*	*	
mobile & desktop tagstore	x	x		x	x	x	x	x	x	x	x

Table 4.1: Comparison of tagstore characteristics concerning tagging support, synchronization features, and platform availability. Full support for a given category is displayed with the character ("x"), minimal support is shown with ("*"), and no support with a blank.

important to note that tagging of folders is not supported on the mobile tagstore as it requires an external file browser.

As already stated, Taggtool Desktop and Taggtool Business Server are written in Java programming language. Since Java is supported on all popular platforms, the Taggtool Desktop and the Business Server application have been successfully run in XUbuntu 12.04.1 using OpenJDK 1.6.0.24. However, as Taggtool Desktop targets Microsoft Windows, the functionality is reduced. Therefore users of Linux and Mac OS X need to wait for official support of their platform.

The mobile and desktop tagstore perform well in the area of tagging support as well as platform availability. In addition, the synchronization implementation provides capabilities to synchronize files and associated tags. Furthermore, the synchronization can be performed networked. Users now have a method to manage and synchronize personal files on the Android platform in combination with the desktop tagstore research software.

5 Conclusion

“Now this is not the end. It is not even the beginning of the end.
But it is, perhaps, the end of the beginning.”

(Winston Churchill)

This master thesis analyzed the requirements and platform aspects for a mobile tagging application. It also described the mobile tagging application, which was developed during this master thesis. In addition, a synchronization algorithm was implemented in two independent systems.

Another focus of the master thesis laid on the synchronization of files and associated meta-data. The characteristics of synchronization systems as well as the different types synchronization systems were elaborated. Afterwards, the requirements of a synchronization system for a mobile tagstore were gathered. Derived from the requirements, a synchronization algorithm was designed. Later the developed synchronization algorithm was described in detail. Finally, the tagstore synchronization system was evaluated with other tagstore alternatives and its strengths and weaknesses were highlighted.

The mobile tagstore is a prototype for managing personal files on a mobile phone. In addition, the implemented synchronization system helps to synchronize personal files on the fly. However, it is important to note that the mobile tagstore is reduced version of the desktop tagstore due to elaborated platform requirements and characteristics.

The mobile tagstore can be extended by providing a decent tag recommender, which extracts tags from the file name, meta-data, or file content. The recommended tags are then displayed when a new file is tagged or a file is re-tagged. Another useful improvement is the ability to tag folders,

5 Conclusion

which is already supported in the desktop tagstore. Concerning a synchronization back-end for another network file system such as NFS or a distributed file system like AFS can improve the mobile tagstore usability. Furthermore, the synchronization algorithm can be improved. The algorithm currently only processes files. Synchronization of tagged folders is eligible.

Bibliography

- Andrea Arpaci-Dusseu, Remzi Arpaci-Dusseu (2012). *Operating Systems: Three Easy Pieces*. Arpaci-Dusseu Books. URL: <http://www.lulu.com/shop/remzi-arpaci-dusseu-and-andrea-arpaci-dusseu/operating-systems-three-easy-pieces/paperback/product-20340734.html> (visited on 09/24/2012) (cit. on p. 29).
- ANSI/NISO (2005). *Guidelines for the Construction, Format, and Management of Monolingual Controlled Vocabularies*. Tech. rep. Z39.19-2005. Bethesda, MD, USA. (Visited on 08/07/2012) (cit. on p. 5).
- Axel Kossel Markus Stöbe, Ragni Zlotos (June 2012). "Aktuelle Daten immer parat mit Dropbox & Co." In: *c't* 13, pp. 78–83. URL: <http://www.heise.de/artikel-archiv/ct/12/13/083/> (cit. on p. 41).
- Bai, Songlin and Hao Wu (Oct. 2011). "The Performance Study on Several Distributed File Systems." In: *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, pp. 226–229. DOI: [10.1109/CyberC.2011.45](https://doi.org/10.1109/CyberC.2011.45) (cit. on p. 29).
- Binder, Gerulf (2012). *Marktübersicht von Tagging-Werkzeugen und Vergleich mit tagstore*. Tech. rep. Graz, Austria: Graz University of Technology (cit. on p. 62).
- Borgmann, Moritz et al. (2012). *On the Security of Cloud Storage Services*. Tech. rep. (cit. on pp. 36–38, 41).
- Bzoch, P. and J. Safarik (Sept. 2011). "Security and reliability of distributed file systems." In: *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on*. Vol. 2, pp. 764–769. DOI: [10.1109/IDAACS.2011.6072873](https://doi.org/10.1109/IDAACS.2011.6072873) (cit. on p. 29).
- Fielding, Roy Thomas (2000). "Architectural styles and the design of network-based software architectures." PhD thesis. ISBN: 0-599-87118-0 (cit. on p. 39).
- Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung (2003). *The Google File System* (cit. on p. 43).

Bibliography

- Justice, Department of (2001). *USA Patriot Act*. URL: <http://www.gpo.gov/fdsys/pkg/PLAW-107publ156/pdf/PLAW-107publ156.pdf> (visited on 08/07/2012) (cit. on p. 38).
- Lagerspetz, Eemil, Sasu Tarkoma, and Tancred Lindholm (2010). "Dessy: Search and Synchronization on the Move." In: *Eleventh International Conference on Mobile Data Management, MDM 2010, Kanas City, Missouri, USA, 23-26 May 2010*. Ed. by Takahiro Hara et al. IEEE Computer Society, pp. 215–217. ISBN: 978-0-7695-4048-1. DOI: <http://doi.ieeecomputersociety.org/10.1109/MDM.2010.18> (cit. on p. 33).
- Levy, Eliezer and Abraham Silberschatz (1990). "Distributed File Systems: Concepts and Examples." In: *ACM Comput. Surv.* 22.4, pp. 321–374. URL: <http://dblp.uni-trier.de/db/journals/csur/csur22.html#LevyS90> (cit. on p. 28).
- Lindholm, Tancred (2004). "A three-way merge for XML documents." In: *Proceedings of the 2004 ACM symposium on Document engineering. DocEng '04*. Milwaukee, Wisconsin, USA: ACM, pp. 1–10. ISBN: 1-58113-938-1. DOI: [10.1145/1030397.1030399](http://doi.acm.org/10.1145/1030397.1030399). URL: <http://doi.acm.org/10.1145/1030397.1030399> (cit. on p. 33).
- Lindholm, Tancred, Jaakko Kangasharju, and Sasu Tarkoma (Dec. 11, 2009). "Syxaw: Data Synchronization Middleware for the Mobile Web." In: *MONET* 14.5, pp. 661–676. URL: <http://dblp.uni-trier.de/db/journals/monet/monet14.html#LindholmKT09> (cit. on p. 33).
- Lutz, M. (2001). *Programming Python*. O'Reilly Series. O'Reilly. ISBN: 9780596000851. URL: http://books.google.at/books?id=37_AJD1EytEC (visited on 07/26/2012) (cit. on p. 4).
- Microsoft Technet (2005). *Distributed File System overview: Remote File Systems; File and Storage Services*. URL: <http://technet.microsoft.com/en-us/library/cc738688%28v=ws.10%29.aspx> (visited on 09/17/2012) (cit. on p. 29).
- Rittinghouse, J. (2009). *Cloud Computing: Implementation, Management, and Security*. Taylor & Francis. ISBN: 9781439806807. URL: <http://books.google.at/books?id=YRIeASgVUJoC> (cit. on p. 39).
- Rivadeneira, A. W. et al. (2007). "Getting our head in the clouds: toward evaluation studies of tagclouds." In: *Proceedings of the SIGCHI conference on Human factors in computing systems. CHI '07*. San Jose, California, USA: ACM, pp. 995–998. ISBN: 978-1-59593-593-9. DOI: [10.1145/1294247.1294346](http://doi.acm.org/10.1145/1294247.1294346)

Bibliography

- 1240624 . 1240775. URL: <http://doi.acm.org/10.1145/1240624.1240775> (cit. on p. 20).
- Satyanarayanan, M. et al. (Apr. 1990). "Coda: a highly available file system for a distributed workstation environment." In: *Computers, IEEE Transactions on* 39.4, pp. 447–459. ISSN: 0018-9340. DOI: 10.1109/12.54838 (cit. on p. 29).
- Schütt, Thorsten (2002). "Synchronisation von verteilten Verzeichnisstrukturen." MA thesis. URL: <http://opus4.kobv.de/opus4-zib/files/1007/SchuettDiplom.pdf> (visited on 07/30/2012) (cit. on pp. 26–28, 32).
- Shepler, S. et al. (2003). *Network File System (NFS) version 4 Protocol*. United States. URL: <http://www.networksorcery.com/enp/rfc/rfc3530.txt> (visited on 09/25/2012) (cit. on p. 29).
- The Open Group (2012). *Distributed Relational Database Architecture standard*. URL: <https://collaboration.opengroup.org/dbiop/> (visited on 09/17/2012) (cit. on p. 67).
- Tridgell, A. (1999). *Efficient Algorithms for Sorting and Synchronization*. Australian National University. URL: <http://books.google.at/books?id=YH3btwAACAAJ> (cit. on p. 30).
- Tridgell, Andrew (2012). *The rsync algorithm*. URL: http://rsync.samba.org/tech_report/node2.html (visited on 08/01/2012) (cit. on p. 31).
- Uppoor, S., M.D. Flouris, and A. Bilas (Sept. 2010). "Cloud-based synchronization of distributed file system hierarchies." In: *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pp. 1–4. DOI: 10.1109/CLUSTERWKSP.2010.5613087 (cit. on p. 34).
- Voit, Karl (May 2012). *tagstore — Project home page*. URL: <http://tagstore.org> (visited on 05/10/2012) (cit. on p. 3).
- Voit, Karl, Keith Andrews, and Wolfgang Slany (Nov. 2011). "TagTree: Storing and Re-finding Files Using Tags." In: *Proc. 7th Conference of the Austrian Computer Society Workgroup: Human-Computer Interaction (Usab 2011)*. Vol. 7058. LNCS. Graz, Austria: Springer, pp. 471–481. ISBN: 3642253636. DOI: 10.1007/978-3-642-25364-5_33 (cit. on p. 3).
- Wang, Haiyang et al. (2012). "On the impact of virtualization on Dropbox-like cloud file storage/synchronization services." In: *Proceedings of the 2012 IEEE 20th International Workshop on Quality of Service. IWQoS '12*.

Bibliography

- Coimbra, Portugal: IEEE Press, 11:1–11:9. ISBN: 978-1-4673-1298-1. URL: <http://dl.acm.org/citation.cfm?id=2330748.2330759> (cit. on p. 40).
- Wikipedia (2012a). *Rsync*. URL: <http://en.wikipedia.org/wiki/Rsync> (visited on 08/01/2012) (cit. on p. 30).
- Wikipedia (2012b). *Smartphone*. URL: <http://en.wikipedia.org/wiki/Smartphone> (visited on 08/01/2012) (cit. on p. 1).
- Wu, Jiye et al. (2010). “Cloud Storage as the Infrastructure of Cloud Computing.” In: *Proceedings of the 2010 International Conference on Intelligent Computing and Cognitive Informatics*. ICICCI '10. Washington, DC, USA: IEEE Computer Society, pp. 380–383. ISBN: 978-0-7695-4014-6. DOI: 10.1109/ICICCI.2010.119. URL: <http://dx.doi.org/10.1109/ICICCI.2010.119> (cit. on p. 35).
- Zhang, Jiaran et al. (Dec. 2011). “HadoopRsync.” In: *Cloud and Service Computing (CSC), 2011 International Conference on*, pp. 166–173. DOI: 10.1109/CSC.2011.6138515.