VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNCHRONIZATION AND BACKUP OF DATA UNDER ANDROID OS

SYNCHRONIZACE A ZÁLOHOVÁNÍ DAT PRO OS ANDROID

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAROMÍR KARMAZÍN
AUTHOR

VEDOUCÍ PRÁCE Ing. PAVEL OČENÁŠEK, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce zkoumá existující synchronizační aplikace a další nástroje vhodné pro tvorbu synchronizačního nástroje, z nichž následně vybírá vhodné metody pro tvorbu nového nástroje pro operační systém Android. Vytvořený nástroj je schopen synchronizovat uživatelské soubory v peer-to-peer sítích nad IPv4 i IPv6, přičemž spolupracuje s analogickými nástroji v operačních systémech Linux a Windows. Nástroj využívá sledování verzí založené na Gitu a zabezpečení komunikace pomocí TLS.

Abstract

This thesis analyzes existing synchronization tools and other tools suitable for creating a synchronization tool, which are then used to select appropriate methods for creating a new tool for the Android operating system. The newly created tool is able to synchronize user files across peer-to-peer networks over IPv4 and IPv6, while cooperating with analogical tools for the Linux and Windows operating systems. The tool uses Git-based version tracking and TLS communication security.

Klíčová slova

Android, bezpečnost, decentralizovaný model, Git, multiplatformita, peer-to-peer, TLS, synchronizace dat, zálohování dat.

Keywords

Android, data backup, data synchronization, decentralized model, Git, multiplatformity, peer-to-peer, security, TLS.

Citace

Jaromír Karmazín: Synchronization and Backup of Data under Android OS, bakalářská práce, Brno, FIT VUT v Brně, 2013

Synchronization and Backup of Data under Android OS

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Pavla Očenáška, Ph.D., a že jsem uvedl všechny literární prameny, publikace a internetové zdroje, ze kterých jsem čerpal.

. . . .
Jaromír Karmazín
May 13, 2013

Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Pavlu Očenáškovi, Ph.D. za to, že mi poskytoval konzultace během řešení projektu a že mi umožnil publikovat související článek na světové konferenci HCI International 2013.

Contents

Chapter 1

Introduction

The number of electronic devices like computers, tablets, and cell phones that a single person uses has been increasing in the past years, giving rise to problems with data availability and consistency. Unless specialized tools are used, users who wish to read or modify their data on multiple devices have to manually transfer the data among their devices, which is both time-consuming and error-prone. Commercial services like Dropbox[1], Microsoft SkyDrive[2], Google Drive[3], and others have emerged to solve this problem. Their software automatically replicates all user data to all the user's devices and keeps it up to date, using a process called data synchronization.

Because replicating data from device to device is a core mechanism of data synchronization tools, they can also be used to backup data. With a basic setup, each synchronized device acts as a mirror of the others. In addition, a device with enough storage space (usually a server or a cloud) may archive older versions of the data.

In this bachelor's thesis, we will describe the design, implementation, and testing of a system tool able to synchronize user data among multiple devices. While we will focus on the synchronization process, we will keep in mind the inherent potential usage for data backup, hence the title of the thesis.

This technical report documents a solution, implemented for the Android operating system, that aims to be non-commercial, fully decentralized, not requiring cloud storage, and using a network protocol compatible with implementations for the Linux and Windows operating systems.

1.1 Cooperation

To be able to communicate with Linux and Windows devices, the tool's network protocol and parts of the design are shared with the following two bachelor's theses:

- Adam Marťák: Synchronizace a zálohování dat pod OS Linux, bakalářská práce, Brno, FIT VUT v Brně, 2013.

[1] Dropbox: http://www.dropbox.com/

[2] SkyDrive: http://skydrive.live.com/

[3] Google Drive: https://drive.google.com/

- Jakub Slováček: Synchronizace a zálohování dat pod Windows, bakalářská práce, Brno, FIT VUT v Brně, 2013.

Chapter 2

Requirements analysis

A handful of data synchronization and/or backup tools have already been created. Furthermore, there are various technologies, tools, and libraries that are useful for data synchronization and/or backup, even if they are not created solely for such purposes.

In this chapter, we will research existing synchronization software, tools responsible for certain aspects of data synchronization, and options of the Android platform.

From the research, we will deduce a list of requirements a user might place on a synchronization and backup tool. We will also consider the benefits and drawbacks of using particular technologies in the tool of our design.

2.1 Existing solutions

2.1.1 Cloud synchronization service – Dropbox

Dropbox is a commercial tool for data backup and synchronization. It is a centralized solution with client-server access, utilizing the Amazon 3S cloud storage.

Part of the Dropbox distribution is a system daemon which performs synchronization in the background and monitors local file system changes using `inotify`. [15] It is also able to detect changes that happened while it was shut down.

On the target system, a special directory named "Dropbox" is created for synchronization. The user has the option to synchronize this directory as a whole, or any subset of its subdirectories. It is however not possible to have multiple of these directories on one system, or to use symbolic links for connecting to directories located elsewhere. The subdirectory selection only affects data downloads, not uploads. [14]

One of the extended functions of Dropbox is "LAN Sync", which allows two devices to transfer data over the local network, given that they are able to locate each other using IPv4[1] broadcast. The transfer is still coordinated by the server, so both devices require an Internet connection. Files have to be uploaded to the server first, so "LAN Sync" reduces download times only.

[1] Internet Protocol, version 4

Dropbox utilizes the `librsync` library for efficient transmission of small changes in large files.[2]

User files are transferred over server ports 80 and 443 [16], encrypted using the SSL[3] protocol [16] during transfer and using the AES[4] cipher during storage. The AES key is available to Dropbox's employees, but not to the user. [13]

For third-party developers, Dropbox provides a "Sync API" and a "Core API" that enable using Dropbox's synchronization in their applications.

Main features:

- cloud storage
- easy to use interface
- changed file detection
- secure file transfer
- transfer efficiency
- support for multiple platforms
- selective synchronization
- LAN device discovery and data transfer

2.1.2 Secure command and data transfer – SSH

This section uses [23] as its source.

Secure Shell (SSH) is a tool and a client-server protocol for secure remote connections over an insecure network. Despite being primarily designed for terminal command-line access, it can be also used for forwarding arbitrary network traffic and for file transfer.

The SSH protocol consists of three layers:

SSH-TRANS
Transport layer protocol. Ensures connection security, server authentication and stream compression.

SSH-USERAUTH
User authentication protocol.

SSH-CONNECT
Connection protocol. Splits the encrypted channel into several logical channels.

A server should provide a cryptographic key, called a "host key", to confirm its identity. This is verified by the client either by a lookup in its local database (which has the form of a server-key map), or by referring to a certification authority. The client can save the

[2] The library is included in the Dropbox distribution as of version 1.4.17.
[3] Secure Sockets Layer
[4] Advanced Encryption Standard

host key automatically when connecting to a server for the first time[5], which is used, for example, by the OpenSSH implementation.

Client authentication is performed after server authentication. The client can authenticate using a password, a cryptographic key, or a combination thereof. The key can be further protected by a password, which is more secure than a password authentication.

Known cryptographic algorithms are used for this security, usually DSA[6] and RSA[7]. The protocol allows negotiation of the cryptographic algorithms to be used.

Algorithms and methods are named using a string which is either registered by the IETF[8] and does not contain the at-sign "`@`", or which is not registered and has the format `name@domain`.

The key exchange protocol does not ensure perfect forward secrecy.

Despite having a client-server architecture, the tool can be used as both a server and a client on each terminal device, allowing their interconnection in a peer-to-peer fashion.

Main features for data synchronization:

- secure file transfer
- identity checking
- support for multiple platforms

2.1.3 Distributed file version control – Git

Git[9] is a distributed system for source code management (SCM). It allows tracking changes in files, storing previous versions, and exchange of all data with another device in the network without the presence of a central server.

The user is responsible for creating snapshots, called "commits", of the data to track. A new "commit" contains locally up-to-date versions of all tracked files and can be restored at any time in the future.

A single Git project may be tracked by multiple repositories. Git can transfer data among those repositories and resolve arising conflicts.

[11, ch. 9.2] describes how the data and metadata are organized in a Git repository. The organization scheme is based on a distributed database, consisting of three types of objects:

blob

Stores the contents of a file at a given point in history. Does not contain any metadata.

tree

Stores the contents of a directory at a given point in history. Consists of entries that

[5] This technique is called TOFU – Trust on First Use.

[6] Digital Signature Algorithm

[7] Rivest-Shamir-Adleman, an encryption and signature algorithm

[8] Internet Engineering Task Force

[9] Homepage: http://git-scm.com/

specify a file name, a type (file or directory), and a reference to either a “blob” (for files) or another “tree” (for directories). Directory hierarchies can be stored using the recursive property of “trees”.

commit

Stores the state of the project at a given point in history. Contains, among other things, the name of the author, the time of creation, a comment, a reference to the top-level “tree”, and optionally a reference to a previous “commit” in history. Creating “commits” is the responsibility of the user.

Each object is addressed by its SHA-1[10] hash. Collision between hashes is possible, but highly improbable, and so Git makes no attempt to resolve it. [11, ch. 6.1]

All objects in a Git database are immutable – changes are always represented by adding new objects. [11, ch. 1.3] This allows the synchronization of two repositories by merely copying the missing objects over. On the down side, this means that a Git database always grows in size.

Information can be transferred between two Git repositories using the local file system, HTTP, SSH, or the custom “Git” protocol. Secure transfer requires the use of SSH or HTTPS because Git itself does not handle transfer security. All methods of transfer are described in [11, ch. 4.1].

Main features for data synchronization:

- file change detection
- version tracking
- decentralized storage and synchronization
- support for multiple platforms

Drawbacks for data synchronization:

- object accumulation – every device must keep the complete history of metadata and data
- non-transparency – the user is required to manually “commit” data
- “commits” are atomic – either all files are updated at once, or the update is aborted

2.1.4 Efficient transfer of changed data – librsync

librsync is a free software library that implements the *rsync remote-delta algorithm*. This algorithm allows efficient remote updates of a file, without requiring the old and new versions to both be present at the sender. [9] The algorithm reduces the amount of data necessary to synchronize a file in a network.

[10] Secure Hash Algorithm

The basis of the algorithm is as follows: When machine B requests a new version of a given file from A, it splits its version of the file into blocks, creates checksums these blocks, and sends them to A. Machine A then searches for matches of these checksums within its own version of the file. It then sends B instructions to reconstruct its version of the file, sending only those parts of the file contents which did not match any of the checksums. [22]

Main features for data synchronization:

- efficient data transfer

2.1.5 Secure communication – GnuTLS

GnuTLS is a secure communications library implementing the SSL[11], TLS[12] and DTLS[13] protocols and technologies around them. It provides a C language API[14] and is licensed under the GNU Lesser General Public License version 3. [8].

Besides using X.509 certificates (which are typically used for SSL authentication), it also allows the use of OpenPGP certificates[7], simple password authentication (SRP [2, heading 4.3.1]) or pre-shared keys (PSK [2, heading 4.3.2]). These methods do not require a central authority like X.509 does, making them appropriate for decentralized networks.

Main features for data synchronization:

- transfer security
- identity checking

Drawbacks for Android:

- missing native Java interface

2.1.6 Specific data synchronization – SyncML

This section cites information from [1].

SyncML is a standard used primarily for synchronization of mobile data, e.g. contact information and calendar events.

The network architecture of SyncML is client-server. Synchronization is performed in a one-shot fashion. Data are transferred between client and server in so-called “packages” which contain lists of changes since the last synchronization. During the usual synchronization process, the client sends its package to the server first. Conflicts are usually resolved at the server’s side.

[11] Secure Sockets Layer

[12] Transport Layer Security

[13] Datagram Transport Layer Security

[14] Application Programming Interface

The synchronization process is checked using so-called “sync anchors” which are strings saying when the synchronization events happened. At the beginning of a synchronization process, two anchors are transferred – one marking the last finished synchronization, the other marking the current synchronization.

Synchronization over SyncML usually happens incrementally, but in case of problems, a complete data refresh (called “slow sync”) can be triggered.

Main features:

- basic synchronization control principles
- support for multiple platforms

Drawbacks for our project:

- designed for specific data types only
- fixed client-server architecture

2.2 Protocol Buffers – data interchange format

This section is based on information from [17] and [19].

Protocol buffers are Google’s binary data serialization format. They can be used to define, encode and decode network protocols, similarly to ASN.1[15].

A protocol buffer format is defined using a text file with the `.proto` extension, which is then compiled using the `protoc` tool into classes for C++, Java or Python. These classes can be used by applications to directly read and write serialized data. When compared to XML, Protocol Buffers also offer extensibility, but are more space- and computation-efficient to use. The binary serialization format is not self-describing.

The data types usable in he protocol buffer language are integers, floating-point numbers, ASCII[16] or UTF-8[17] strings, booleans, arbitrary sequences of bytes, and enumerations. Nested structures can be created by nesting messages within one another, and arrays can be created using the `repeated` modifier, which allows a field to occur zero or more times within a message.

Protocol buffers can be used together with an RPC[18] system. This works by defining a `service` inside the `.proto` file, which the compiler then translates into an RPC stub. The user must provide a channel to handle RPC calls.

The following functions are missing from protocol buffers and must be handled by the programmer: [18]

[15] Abstract Syntax Notation One

[16] American Standard Code for Information Interchange

[17] Universal Character Set (UCS) Transformation format – 8-bit

[18] Remote Procedure Call

Message delimiting.
Serialized messages are not self-delimiting, so in order to use them in a stream, the programmer must provide a means for separating individual messages.

Unions.
Fields may be declared optional, but there is no way to ensure that at most one field from a group is set.

Inheritance.
A message cannot be defined as a specialization of another.

Message discrimination.
When more than one message type is used, the programmer must determine which type to expect when parsing a stream.

2.3 Options of the Android platform

This section cites information from [21] and from the developers' portal `http://developer.android.com/`.

Android runs on the Linux kernel, which means, among other things, that file system paths use a forward slash ("/") as a delimiter and store the whole directory tree under a single root.

Applications for Android are mostly written in the Java programming language. The platform provides most of the Java SE libraries, excluding ones like AWT, Swing, and CORBA. External libraries can be added to a project in the form of JAR files, although this brings some complications. [21, ch. 24] In addition, parts of Android applications can be implemented in C or C++ using a toolkit called Android NDK, allowing existing libraries written in these languages to be added to a project as well, although not seamlessly. [3]

2.3.1 Application architecture

An Android application consists of several application components: *activities*, *services*, *broadcast receivers*, and *content providers*.

Activities are components that create the user interface of the application. They roughly correspond to windows in classical windowed GUIs[19]. The main difference is that when an *activity* is not visible to the user, it can be removed by the operating system to save resources, and later created again when the user wants to display it again. This means that an *activity* has limited lifetime and should only be used for interacting with the user. [4, Activities]

Services, on the other hand, can run persistently, even without user interaction. They are used to perform work in the background, using separate threads. They do not have a user interface, but they can present information to the user using *toasts* (transient messages), system notifications, or broadcast messages to other components. [4, Services] The way services are started and stopped will be analyzed in 2.3.2.

[19] graphical user interfaces

Broadcast receivers enable application to receive events generated by the system or other applications, even if the application is not running when the event is generated. [5, `android.content.BroadcastReceiver`]

Content providers allow sharing data among applications. They will be further analyzed in 2.3.3

2.3.2 Service lifetime

This section cites [4, Services].

Android controls the lifetime of a *service* depending on how it is started. There are two basic forms of a service regarding to its lifetime – a *bound* and a *started* form. These forms can be used selectively or combined.

Bound services are created when a client (for example, an *activity*) first binds to them, and they are destroyed after the last client unbinds. The client can communicate with such services either using direct calls, which only works in the same process, or using an interface defined using *Android Interface Description Language (AIDL)*, creating what is a form of inter-process communication (IPC). This type of *service* is thus well suited for controlling a remote process.

Started services are started and stopped by an explicit intent[20], which may come from the *service* itself or from another application component. They provide no means of communicating with the client other than passing extra data when starting the *service*, so their output is usually limited to displaying notifications or *toasts*. This type of *service* is well suited for one-time time-intensive tasks, such as downloading files.

Android starts *services* on the same thread as the user interface (i.e. *activities*) of the same application. Because the Android design mandates that no time-consuming work be done on the UI thread (in order to keep the user interface responsive), a *service* must create its own thread or threads to offload all time-consuming work to.

2.3.3 Storage

The Android platform allows applications to use several forms of storage. [6]

Applications can use *external storage*, which can be "*a removable media (such as an SD card) or an internal (non-removable) storage*". The user can read from and write to the external storage, as can all application that have the necessary permission. This means that this form of storage can be accessed by a synchronization tool, even without system privileges.

Changes in the external storage can be monitored using the class `android.os.FileObserver` in the Android API. This class uses the kernel subsystem called `inotify` to monitor events such as creation, modification, movement, and deletion of files in the file system. [5, `android.os.FileObserver`]

[20] *Intents* are Android-specific facilities for describing operations to be performed. These operations generally include launching *activities*, starting and stopping *services*, and broadcasting information to applications in the system. See [5, `android.content.Intent`]

For internal purposes, applications can use *shared*[21] *preferences* (simple key-value pairs), *internal storage* (files), and *SQLite databases*. All these forms of storage are, in most cases, private to the application and not accessible to other applications or the user. This means that an application without system privileges cannot synchronize this form of data, unless it is accessible through a *content provider*.

Content providers are application components that allow an application to share some of its internal data with other applications and to define data security. Two standard content providers are part of the Android platform: a *calendar provider* and a *contacts provider*. Other applications can define their own providers. Each content provider has its own data structure. [4, Content Proviers] This means that this form of storage can be synchronized, but an adapter would need to be written for each content provider.

2.3.4 Security

Android includes the standard Java security API.

The general security package `java.security` contains classes that allow parsing X.509[22] certificates and PKCS#8[23] private keys.

The package `javax.net.ssl`, which is part of the *Java Secure Socket Extension (JSSE)*, allows using the SSL and TLS protocols for network communications. Certificate validation can be overridden using the `X509TrustManager` interface. [5, `javax.net.ssl`]

2.4 Requirements

Combining the assignment with knowledge from our research, we will now list the requirements that we place on the tool of our design:

Security.
None of the transmitted data may be susceptible to forgery or diversion to other than authorized devices.

Efficiency.
Data should not be transmitted repeatedly, unless necessary.

Integrity.
The synchronization process must not damage the file system on any of the devices.

Transparency.
The synchronization process should require minimum user input.

Selective transfer.
The user should be able to define which parts of their data should be synchronized, and which should not be.

[21] These are called “shared” because they can be accessed by multiple application components, not because they would be shared with other applications.

[22] See RFC 5280.

[23] Private-Key Information Syntax Standard. See RFC 5208.

Platform independence.

The synchronization protocol should allow working with devices not running the Android operating system.

In addition, we will place one extra requirement on our tool in order to make it fulfill the original goal:

Decentralization.

The entire solution should be functional without the presence of a central, always-on element.

Chapter 3

Technical analysis

This chapter will sum up the previous findings with focus on our application. The aforementioned tools and technologies will be compared from a technical point of view and, based on their properties, they will be chosen for use in our application.

3.1 Detection of file changes

A synchronization tool must be able to detect when a local file was changed and, if necessary, distribute the changes to other devices. File changes can be classified as:

On-line – the synchronization service is running when the change happens. The change can then be detected using system events.

Off-line – the synchronization service is not running when the change happens. It is then necessary to scan the file system and find what changes happened when the service starts.

Although on-line change detection provides smoother operation, it only works when the data is synchronized when the detection starts. This requires either that off-line changes be forbidden, or that this mechanism be used together with off-line change detection. In cases where it is undesirable or impossible to lock the storage while the synchronization tool is not running, the tool must unconditionally contain a mechanism for off-line change synchronization.

There is a Linux API for on-line change detection called `inotify`, its wrapper for Android is called `android.os.FileObserver`.

Off-line changes can be detected using file attributes (size, data and time of change) or contents (bitwise comparison, checksum comparison). It is necessary to store a copy of the data that we wish to compare.

Bitwise comparison is the most reliable, but it requires storing a complete copy of every tracked file (like Git does). This is inconvenient for limited storage devices, such as cell phones. It is more convenient to store a file's checksum, e.g. an SHA-1 hash, instead of its whole contents.

Recalculating hashes on every service startup may become tedious when data grow large and/or when on devices with limited computational power. If we can assume that local changes cause the alteration of at least one of its attributes (size, modification time, metadata change time), all files with their attributes unchanged can be skipped from the check. This allows a faster off-line change scan, although in special cases, a file can be falsely considered to be unchanged.

3.2 Metadata storage

There are two types and two purposes of file metadata that we need to store:

File metadata
Name, directory path, size, modification time etc. This is the type of metadata that the off-line change detection mechanism, described in 3.1, depends upon.

Version metadata
Identification of previous versions and dependent objects. This type of metadata is required for history tracking and conflict resolution during synchronization.

As was described in 2.1.3, Git stores metadata as three types of objects: `blob`, `tree`, and `commit`. Out of all file metadata, only the file size (in `blob`s), file type, file name and directory hierarchy (in `tree`s) is stored. Version metadata are only stored in `commit`s.[11, ch. 9.2] This is illustrated in figure 3.1.

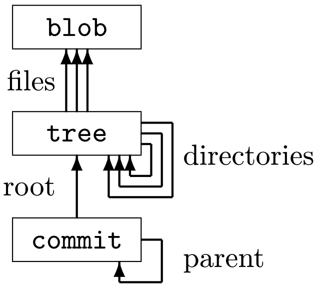

Figure 3.1: Object relationships in Git.

Git's architecture was designed to track changes of source code. However, it is not appropriate for tracking changes in arbitrary content. For instance, the fact that versions are only stored within "commits" forbids versioning files and directories individually.

The object hierarchy can be modified so that versioning can be done on the file level. We can do so by inserting a new object type between the `tree` and the `blob`.

Using this new object type, not only can we track versions of a single file, but also its metadata (and changes thereof), so we choose to name it `metadata`.

This modification is illustrated in figure 3.2.

The `metadata` object can also store file name and type, which removes the need for having this information stored in a `tree`. If we do not require directory-level change monitoring (we

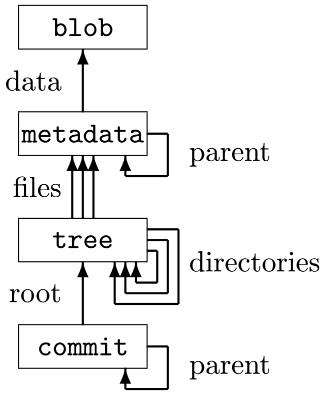

Figure 3.2: Modified Git object hierarchy.

already have file-level monitoring), we can omit the `tree`. The resulting object hierarchy is illustrated in figure 3.3.

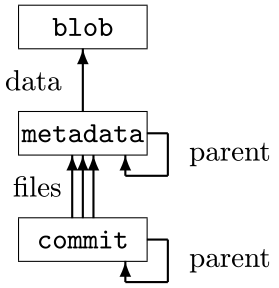

Figure 3.3: Modified Git object hierarchy, with `tree` omitted.

The downside of this solution is that the `commit` must contain references to current versions of all files, which may amount to large numbers. This can be solved in a number of ways:

- **Abandon `commits`.** This gives us a greatly simplistic architecture using two object types, `metadata` and `blob`, which meets our requirement of storing both file and version metadata. However, it is not suitable for cumulative synchronization: Two devices that need to inform each other of all changes that happened while they were off-line have no other ways of doing so than to exchange a complete list of current `metadata` objects – which is essentially the same sending a `commit` object, except neither side stores it permanently.

- **Store `commits` as deltas.** Each `commit` can be designed to only refer to those `metadata` objects which have changed from the parent `commit` – i.e. those `metadata` representing an added, modified or deleted file. This is more suitable for cumulative synchronization because two devices can, upon coming on-line, find a common ancestor of their `commits` and skip the listing of files that have not changed since. Alternatively, each device can create its own commits and, upon synchronization, combine the metadata in its own commits with metadata from foreign commits.

3.3 Network communication

The supposed use case of our application is synchronization over a computer network. It is therefore necessary to consider the method of communication between two instances of this application on a network. We will use the reference OSI[1] model for network communication analysis.

We will use Internet as the network for the use of our application. We will therefore discuss IP and upper layer protocols.

Network layer

The IPv4[2] and IPv6[3] protocols are the ones almost solely used on the Internet, so we will not consider other protocols.

Transport layer

We can make no assumptions about the nature of user data, so we need to ensure its reliable transfer. This is where TCP[4] is more suitable. UDP[5] can be eventually used for non-critical informative messages or, if there is another mechanism for reliable delivery, even user data.

Session layer

On this layer, we need to discuss communication security. User data might be sensitive, so it should be protected against eavesdropping. Because a synchronization tool has access to the terminal device's file system, it must be protected from unauthorized access my the means of peer identity verification. This type of application does not have a critical need for availability, so we do not need to have denial of service (DoS) protection on this layer.

The requirements suggest using well-known secure channels such as SSL or TLS. These protocols can be found as a security element in other protocols such as HTTP, SMTP or XMPP. There are numerous libraries supporting SSL and TLS, for example OpenSSL, NSS, JSSE (for Java) or GnuTLS.

SSL and TLS were not designed for a peer-to-peer architecture. During connection establishment, the client and server roles are clearly separated[6] and certificates are verified using a trust chain with a central authority, the X.509 Public Key Infrastructure (PKIX). A certificate is always required from the server, but it can be required from the client too [12, heading 7.4.4]. The use of a certification authority can be avoided using self-signed certificates; however, it then becomes critical to securely transfer certificate fingerprints between the devices, otherwise the certificates could be forged and used for a man-in-the-middle attack.

[1] Open Systems Interconnection

[2] Internet Protocol, version 4

[3] Internet Protocol, version 6

[4] Transmission Control Protocol

[5] User Datagram Protocol

[6] This can be seen, for instance, in the distinction between Server Hello and Client Hello in [12].

There is an alternative in using OpenPGP certificates within TLS. [20] OpenPGP uses a decentralized identity verification model, which is suitable for a peer-to-peer application. Unfortunately, GnuTLS is the only one of the aforementioned libraries to support it.

Presentation layer

Above the secure channel, it is necessary to define the format of messages that will be exchanged by the devices. We will consider using a text, a binary and a combined format for the presentation of our protocol.

Text formats can be human-readable, which simplifies debugging. They are however not suitable for the transfer of user data, because these can be binary in general. Such data would have to be escaped or recoded within a text protocol, which would reduce the protocol's efficiency. A text protocol could be built on popular formats such as XML, YAML, JSON, or popular simple text protocols like HTTP, SMTP, or SIP.

Binary formats allow a more concise representation of a message and do not pose problems for binary data storage. It is however more difficult to decode the communication for debugging purpose, especially when using a non-standard protocol with no software analyzers available. A binary protocol could be built on popular encodings such as ASN.1 BER/CER/DER/PER[7], XDR[8], or using Google's protocol buffers.

Combined formats use text for control messages and binary sequences for data. Such a format is used by the Git protocol.[9] They combine the advantages of text and binary formats, but they cannot be handled as simple text and are not well standardized.

Application layer

Two devices should be able to exchange the following types of messages:

- Negotiation of optional protocol extensions (for forward compatibility).
- Authentication (unless handled in the session layer handshake).
- Negotiation of objects to transfer.
- Object transfer itself, ideally using an efficient algorithm.
- User control, e.g. requests to stop a running synchronization.
- State change notifications and error messages.

The protocol should take the peer-to-peer architecture into account, so both sides should be able to send requests. This can be realized by creating another connection in the opposite

[7] Basic Encoding Rules, Canonical Encoding Rules, Distinguished Encoring Rules, and Packed Encoding rules, respectively

[8] External Data Representation, a serialization format developed by Sun Microsystems

[9] As described in Git's technical documentation (see `https://www.kernel.org/pub/software/scm/git/docs/v1.7.0.5/technical/`, the protocol is composed of so-called *pkt-lines*, which are variable length binary strings. They often contain textual data (and their length indicator is, in fact, encoded in hexadecimal ASCII text rather than binary), but they may contain binary data, typically when transferring file contents.

direction (which might fail in IPv4 networks using NAT), or by not distinguishing the server and client role in the protocol design.

We can decide whether to designate certain messages as “push” (sending unsolicited data to the other side) or “pull” (soliciting data from the other side).

“Push” messages are suitable for messages arising from asynchronous events and/or needing immediate propagation, e.g. on-line changes in the local file system or unexpected errors.

“Pull” messages are, conversely, suitable for situations where either side has a knowledge of the information it is missing. Usage may include user-initiated file downloads from the other side, or iterative filling of an incomplete object database without requiring the other side to know in advance exactly what information is missing.

Chapter 4

Design

This chapter will detail the design of the core synchronization mechanisms, choices of suitable tools, network protocol definition, and finally the design of the Android application.

We will begin by creating a model of the data that we want our application to operate on. Then, we will design a way to store these data and to transfer them over the network. Lastly, we will design the user application that will control these data and processes.

4.1 Organizational model

We will introduce five terms for the entity sets of our model – *devices*, *volumes*, *snapshots*, *metadata*, and *data* – and design a distributed database based on the analysis of Git in 3.2.

Devices

By definition, data synchronization is only useful when used by more than one device. We will therefore use the term *device* to refer to a networked device where our application is installed. We assume that each device has only one instance of our application installed and that the user is able to connect certain pairs of their devices. Since we are aiming for a decentralized design – that is, a design without a central, always-on device – we assume that a device may be off-line at any time.

Volumes

From the analysis of Dropbox (2.1.1), we recall that one of its weaknesses is being able to only have one synchronized directory per user account. We choose to overcome this weakness by allowing the user to define multiple directories for synchronization and distinguish them by design.

We will use the term *volume* to refer to a user-specified directory whose contents are to be used for synchronization. Each volume has its own files, directories, history, and settings, and is synchronized separately in the network protocol. The *volume root* is a directory containing the entirety of the volume and is not itself synchronized, but its contents are.

We choose to allow the user to define different sets of volumes on different devices – each pair of devices will then only be able to synchronize those volumes that they have in common.

Snapshots

We follow up from the idea of "storing commits as deltas" in 3.2 and define a *snapshot* to be a set of changes (references to metadata objects – see below).

The "delta" property requires that each snapshot contains a reference to the previous snapshot, unless it is the first snapshot. This means that snapshots form a tree.

We want to avoid branching and merging of snapshots because this brings complexity to the design and is, in fact, unnecessary in a design without atomic snapshots. Instead, we let each device generate its own snapshots. A single device cannot create diverging change sets (unless it returns to an older state and begins new work), so instead of a tree of snapshots, our design can be thought of as a set of parallel linked lists.

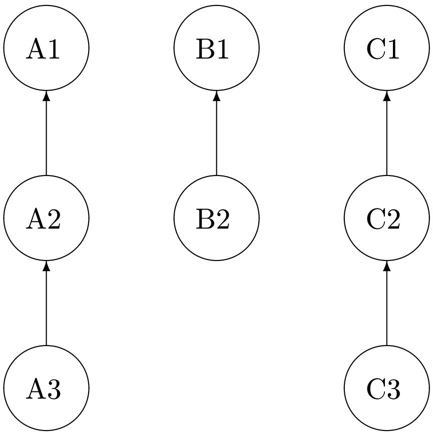

Figure 4.1: Example of a snapshot graph.

The first step in synchronizing a volume is the identification of the latest snapshots – that is, identifiers of the head of each snapshot chain. From this information, a peer can determine what snapshots it needs to retrieve in order to get the complete list of changes.

The example in figure 4.1 shows a situation with three hosts and three chains of snapshots. Hosts A and C have created three snapshots each, while host B has created two snapshots. When synchronizing, the peer having this graph in its database would report that the heads are A3, B2, and C3. Because each snapshot contains a reference to the previous snapshot in its chain, knowing the heads allows any peer to reconstruct the complete graph.

Metadata

Metadata objects are taken directly from the final proposal in 3.2, including their name. Each metadata object represents one version of a file or directory at a given point in time. It contains its location in the file system, its name, a reference to its contents (data – see below), a reference to its previous version (if any), and a reference to a conflicting version that was abandoned (if any).

The graph of metadata relationships (previous version – next version) forms a graph. If you also count the conflict resolutions, it forms a directional acyclic graph (DAG).

Metadata may change along with data (when file contents change) or independently of them (when file attributes change). They may also represent deleted files.

Data

Data objects follow the concept of blobs from 3.2. A data object represents the physical contents of a non-empty file. In the Android implementation, it is stored in the file system under the volume's root, but in implementations supporting backups, a copy should be stored separately.

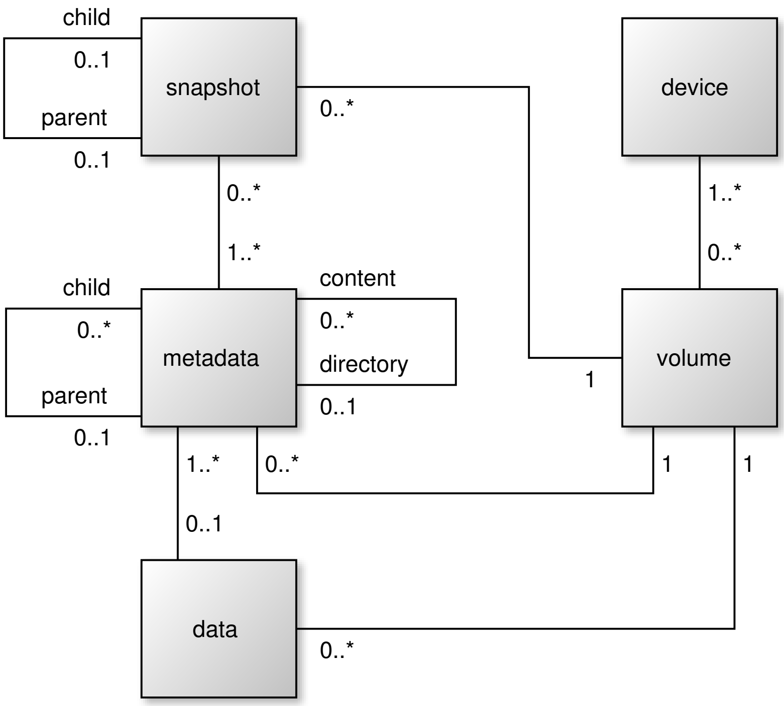

Figure 4.2: Entity-relationship diagram of the design.

Summary

In this section, we have introduced the following terms representing the key entity sets in our design:

Device – a peer where our application is installed.

Volume – a file system tree intended for synchronization.

Snapshot – a collection of references to metadata objects, used for negotiation of changes.

Metadata – the notation of a version of a file or directory within a volume.

Data – the representation of the contents of a file.

Figure 4.2 illustrates the relationships among the entity sets we have defined.

4.2 Distributed database

In 3.2, we modified Git's object hierarchy to be able to update files separately. We then designed *data*, *metadata*, and *snapshots* as the object types in this hierarchy. It follows that these three object types will form the entity sets in the distributed database of our design. Neither *devices* nor *volumes* fit in this hierarchy, so they will not be parts of the distributed database.

We recall from 2.1.3 that Git identifies objects in the distributed database by their SHA-1 hash. While this provides an extremely low risk of collision, integrity checking, immutability assurance, and inherent deduplication[1], it requires that the hash be computed before an object is inserted into the database. This may become a performance issue on Android devices because they usually come with limited computational power, and in general because it requires large files to be completely read each time a change in them is detected, even if this version of the file is never going to be transmitted. For that reason, we design our database to use universally unique identifiers (UUIDs), rather than SHA-1 hashes, for identifying objects. UUIDs also have an extremely low risk of collision, but they can be generated by a secure random number generator, independently from the data they identify.

The distributed database contains the following entities and attributes:

snapshot
UUID, previous version UUID, array of metadata UUIDs;

metadata
UUID, previous version UUID, abandoned conflicting version UUID, containing directory UUID, type (file or directory), name, size, data UUID, deletion flag;

data
UUID, binary contents.

These are reflected in the local database (see below) as well as in the synchronization protocol (see appendix A).

We assume that the user does not keep the clocks of all their devices synchronized. Therefore, the distributed database does not use timestamps for comparing file versions, as it uses previous version UUIDs for this purpose. In fact, timestamps are not even stored in the distributed database, in the same way Git manages its database.

4.3 Local database

As was mentioned in 2.3.3, Android provides SQLite as one of the options for storing data. Since SQLite is is a relational database management system, it is well suited to store the

[1] Two objects with the same contents have the same SHA-1 hash, so they can never exist as duplicates in the database.

objects we mentioned earlier.

Our design uses a single SQLite database to store its image of the distributed database. However, the local SQLite database differs from the distributed database in certain ways.

Most importantly, the local database does not store *data* objects. One reason is that SQLite is not suited for large rows. Another reason is that storing all *data* objects is only useful for storing history. Since Android is a platform for mobile devices, storage space tends to be limited. Our design refrains from storing more data than necessary and only treats *data* objects as arbitrary identifiers, linked to an actual file in the file system. It is still possible to backup data from an Android device, but the history must be stored on a Linux or Windows machine.

Secondly, the local database contains a table for *volumes* in order to map volumes to their locations in the file system and to save their human-readable names. To stay consistent with the distributed database, volumes are also identified by UUIDs.

Thirdly, the local database extends the *snapshot* and *metadata* tables with local fields. Some of these fields are mutable, i.e. they can be modified even after the object was inserted into the database. None of the local fields are part of the network protocol.

- The *snapshot* table is extended by two immutable fields – *device* UUID and *volume* UUID – and two mutable fields – "current" flag and "pending update" flag. The *device* UUID is used to distinguish between local snapshots and foreign ones. The *volume* UUID allows distinguishing *snapshots* for different *volumes* in the same table. The "current" flag is a redundant piece of information, signifying that the given *snapshot* is the last in its chain, i.e. no parent reference points to it. The "pending update" flag is set on *snapshots* received over the network and is cleared once the *snapshot* is fully processed, to allow recovery from failed synchronization attempts.

- The *metadata* table is extended by the immutable *volume* UUID and mutable "pending update" flag for the same reasons as above. In addition, it contains a "checked out" flag to determine whether the object represents an actual file or directory currently in the file system, and finally the mutable field *mtime* stores the last modification time of the file to allow checking for changes.

These fields are the only fields that may be modified after an object is saved to the database. They serve local purposes only, as they are neither part of the distributed database nor part of the network protocol.

Lastly, the *temporary snapshot* table accumulates references to newly created *metadata* objects until a *snapshot* is requested, at which point the contents of the table are converted to a permanent *snapshot* and made part of the distributed database. This helps minimize the number of *snapshots* required.

4.4 Internal storage and shared preferences

Android provides other storage options, as was mentioned in 2.3.3. These include *internal storage* and *shared preferences*.

The *internal storage* provides a private space for storing arbitrary files. Our design uses this storage to store cryptographic certificates of the trusted peers and the local device, as well as the private key of the local device. Using plain files has proven to be much faster than using a specialized key store, while the security implications are the same: an entity with root or physical access to the device can retrieve the certificate, but other applications cannot.

The *shared preferences* are well suited for storing simple, application-global key-value pairs. Our design uses them for storing the preferred server port, the local device's identifier (a UUID generated when the application is first started, not modifiable), and its human-readable name.

4.5 External storage

User data is located on external storage (typically the SD card). Other types of storage (Android's content providers) will not be covered by this application.

Because detecting off-line changes is essential, as was explained in 3.1, our application detects off-line changes only, while leaving the protocol design open for further extension using on-line change detection.

A file change is detected by comparing the size and last modification time of a file to its respective values in the local database.

This method of detection cannot detect changes name of location. All renames and moves are interpreted as deletions followed by additions.

4.6 Communication protocol

The application layer protocol, also called the *synchronization protocol*, is used to exchange data between two devices. It uses Protocol buffers for presentation of data and TLS for security. The protocol is designed to be compatible with the works of Marťák and Slováček, as noted in 1.1.

The following table illustrates how the synchronization protocol stands in terms of the OSI reference model:

Application layer	synchronization protocol
Presentation layer	Protocol buffers
Session layer	TLS
Transport layer	TCP
Network layer	IPv4, IPv6
Data-link layer	managed by the platform
Physical layer	managed by the platform

Chapter 3.3 lists the required message types. In this section, specific messages and their roles will be designed. The protocol is described in detail in Appendix A.

4.7 Connection security

Considering the options outlined in 3.3 for maintaining the authenticity, confidentiality, and integrity of the network protocol, we choose to use TLS because of its wide availability. To mitigate the costs and complexities arising from the need to have signed certificates, our design uses a simplified trust model using self-signed certificates.

Each device requires a self-signed X.509 certificate, a private key, and a database of trusted X.509 certificates. (These are stored in the application's internal storage, as mentioned in 4.4.) Provided that the certificates of all trusted devices are securely transferred to the devices where the trust is checked, the risk of man-in-the-middle attacks, which normally arises from using untrusted self-signed certificates, is mitigated.

The standard trust chain verification using certification authorities is disabled, so as not to allow any user holding a valid, CA-signed certificate, to be falsely identified as trusted.

4.8 Service architecture

To allow background operation, our design focuses on creating an Android *service*. The ways Android offers for creating services were discussed in 2.3.2.

We require the service to be available for the user to control it, and to keep it running as long as it has work to do. However, to preserve resources, the service should shut down once its work finishes and the user is not actively controlling it. We generally do not need other applications to be able to control the service.

To achieve this type of lifetime, we combine the concepts of a *bound service* and a *started service* by letting clients bind to the service, use same-process method calls, and have the service extend its own lifetime by sending intents to itself. The service explicitly starts itself when it starts work in the background, allowing it to persist even after all user activities are stopped. When all background work finishes, the service calls `stopSelf`, which reverts its behavior back to *bound*, i.e. it stops after all clients unbind.

As a result, the service's life cycle follows this rough pattern:

- The service is usually started when the user launches an *activity* of our application, which creates a binding to the *service*.
- The service is usually stopped when the user exits all *activities* **while also** having no background work running.

Because the service's work is not trivial, it is split among the following *service components*:

Server
Runs in a single thread for the whole application and listens for incoming connections.

Connection
Creates three threads – one for sending data over the network, one for receiving data, and one for handling requests and generating responses – and manages the exchange of messages with another device.

Volume monitor
Runs in a thread for a given volume and traverses its contents, looking for files and directories that have changed and registering these changes in the database.

Based on the concept of service components, we can think of the *service* as being in one of three states:

Destroyed or not yet created.
The service is in this state when no clients are bound and no service components are active.

Idle.
The service is in this state when there are bound clients, but no service components are active.

Doing background work.
The service is in this state when at least one service component is active.

The service components have their own life cycle, which will be described in the following section.

4.8.1 The life cycle of service components

A service component is started for the purpose of performing some sort of background work. For `Servers`, the work is listening for incoming connections, for `Connections`, it is exchanging messages with another device, and for `VolumeMonitors`, it is searching for changes in the local file system. A service component stops when its work finishes, when it encounters an error, when the user requests it to be stopped, or when the service is forced to shut down.

4.9 Use cases

From a user's point of view, the main use case of a synchronization tool is simply synchronization. A minimalist approach would have the user specify the directories to synchronize and the devices to synchronize with, and perform all the work automatically.

However, since we aim our application at advanced users, we choose to expose more of the application's internal workings. Based on the design described previously in this chapter, we can decide what parts of the application's functionality should be exposed to the user.

Apart from the user, we can identify two more actors in the system: the network, through which other devices send queries to the local device, and the system, which can force the service to shut down when running out of memory or when the device is turned off.

Figure 4.9 shows the actors and use cases deduced from the design. We will use these use cases to form the user interface and to aid in implementation.

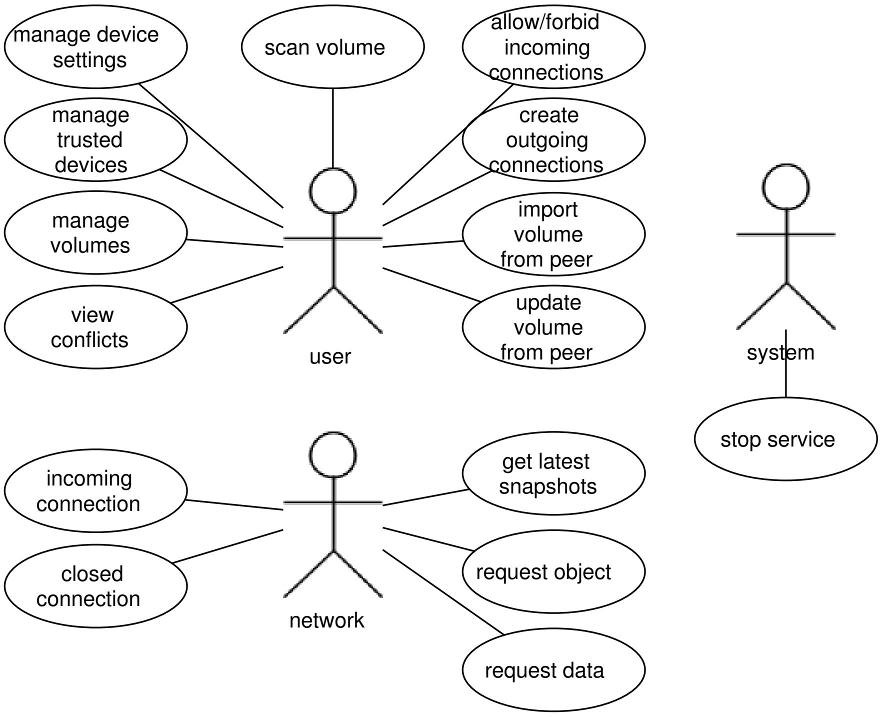

Figure 4.3: Use case diagram for the application.

Chapter 5

Implementation

In this chapter, we will describe the specifics of of implementing the Android application that we designed in the previous chapter.

The source code, communication protocol, and user interface is realized in English.

5.1 Android application properties

Because our tool enables synchronization between devices in a peer-to-peer fashion, it was dubbed “PeerSync”. Android actually requires that an application have a fully qualified name, so we used the domain used by the Faculty of Information Technology for students’ accounts, `stud.fit.vutbr.cz`, together with the login name that the faculty provided me, `xkarma06`, to form the package name `cz.vutbr.fit.stud.xkarma06.peersync`.

As of writing this bachelor’s thesis, the most recent version of Android was 4.2 and the oldest version in use by 99.9% of all users was 1.6, as documented in [10]. In order for PeerSync to be usable on most devices, it declares the minimal required version to be Android 1.6, the target version to be Android 4.2, and no maximal required version.

The application declares two permissions: Internet access (for creating network sockets) and writing external storage.

The application consists of a front-end interface, allowing the user to configure and control the application, and a background service. For better orientation, the classes are divided into four packages:

- `cz.vutbr.fit.stud.xkarma06.peersync`, containing user interface elements like *activities* and *list adapters*, as well as a few generic helper classes;
- `cz.vutbr.fit.stud.xkarma06.peersync.db`, containing classes that enable high-level interaction with the SQLite database;
- `cz.vutbr.fit.stud.xkarma06.peersync.security`, containing classes that enable working with cryptographic certificates, private keys, and secure network sockets; and finally

- `cz.vutbr.fit.stud.xkarma06.peersync.service`, containing all the necessary classes to run the background service.

5.2 User interface

The main *activity* of the application has the form of a menu that provides access to six different *activities* representing the various aspects of the application:

Server
This enables the user to start or stop listening on a TCP port, view the server status, and configure the port number.

Connections
This lists all clients currently connected to the Android device, allows viewing their states and published volumes, and provides a method to create new connections to specified IP addresses and TCP ports.

Conflicts
This provides methods to resolve conflicts between local and received versions of files and directories.

Volumes
This allows the user to configure which parts of the file system should be synchronized.

Keys and certificates
This allows importing X.509 certificates and private keys for the proper functionality of TLS channels.

Device settings
This shows the UUID generated for the device and allows the user to change the human-readable device name advertised in the network protocol.

The menu items are ordered by decreasing assumed frequency of use. Server, Connections, and Conflicts are useful during everyday operation of PeerSync, while Volumes, Keys and certificates, and Device settings are configuration-oriented and should require little intervention once they are set up.

All activities extend a common abstract superclass, *PeerSyncActivity*, whose main role is providing a simple interface for binding to the background service, since the binding interface is asynchronous. Subclasses may override the `onBound`, `onUnbound`, and `onBroadcast` methods to execute code when the service is bound, is unbound, or sends a broadcast *intent*, respectively. They can also use the method `runWhenBound` to schedule some code for running after the service is bound.

5.3 Volume management

To allow the user to manage volumes without running a server component, the `VolumeManager` class was implemented to be available globally. It provides a static interface to allow accessing it and keeps a reference count to automatically clean up once it is no longer needed.

All operations on the local set of volumes must be done using `VolumeManager`. This makes it possible for application-wide listeners to be set up to react whenever a volume is added or removed.

Internally, `VolumeManager` stores information about volume in the SQLite database.

5.4 Settings management

To access the application's shared preferences (server port number, device UUID, and device name) the helper class `SettingsManager` is used. Because manipulating these settings is cheap, this class is instantiated every time it is needed and discarded after performing its operation.

5.5 Certificate and private key management

The user imports their X.509 certificate, their PKCS#8 private key, and trusted peer's X.509 certificates into the application through the Keys and certificates *activity*. The `PeerSyncKeyStore` class, located within the `security` package, stores these keys and certificates as ordinary files inside the application's local storage.

The `security` package also contains a class named `SecureSocketFactory` that creates server-side and client-side TLS sockets, using the keys and certificates provided by the user.

5.6 Database access

The classes within the `db` package provide the functionality for all database operations required by the application. The package may be accessed concurrently – synchronization is performed using SQLite transactions.

There are classes representing all the database tables – namely, `Volume`, `Snapshot`, `TempSnapshot`, and `Metadata`. All these classes have a notion of which fields of the table are immutable (i.e. parts of the distributed database), and which ones are mutable (i.e. local extensions). The immutable fields can only be set when constructing a new object. The classes handle automatic creation of UUIDs for new objects, when necessary.

Because our design does not keep a separate table for data objects, the `Data` class only provides auxiliary functionality to locate the contents of a file using information from the metadata table.

5.7 Background service

The `service` package provides all the classes for running the background service and its components. The class diagram of the service can be seen in figure 5.1.

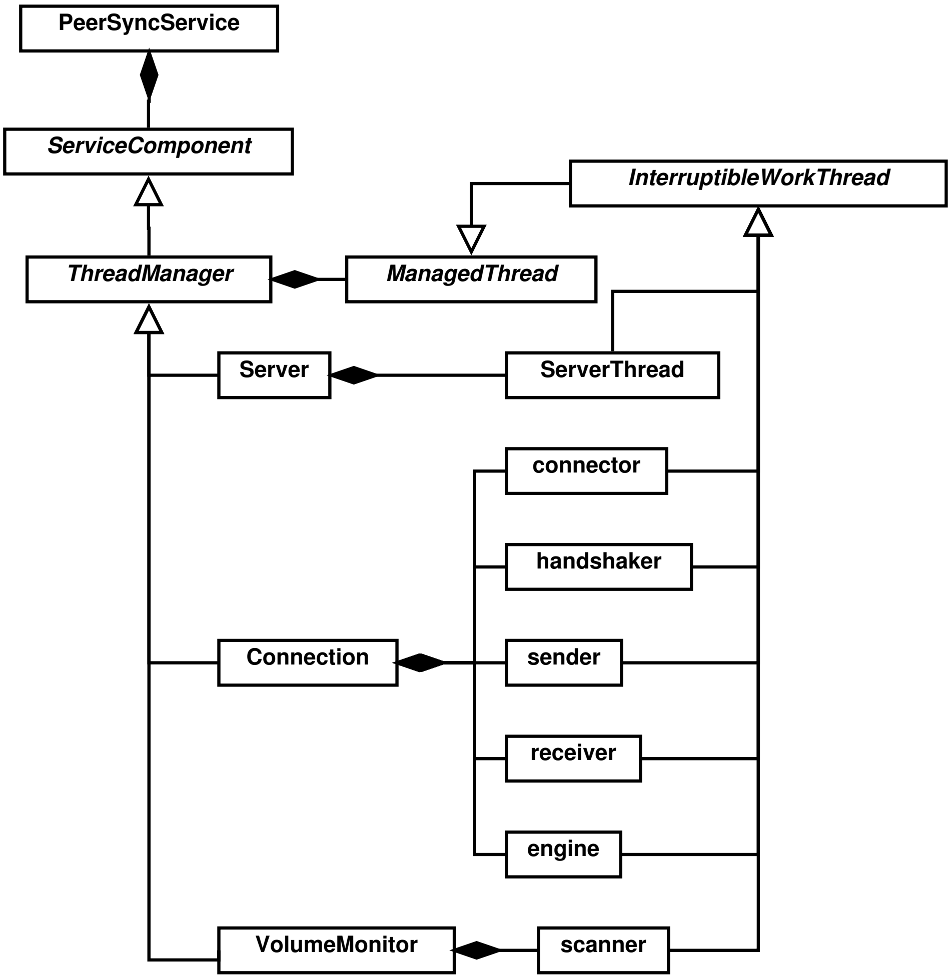

Figure 5.1: Class diagram of the background service.

The service itself, implemented in the class `PeerSyncService`, provides an interface for binding to same-process clients. Once a client requests a service component to start, the service calls `startService` on itself, extending its lifetime for the time necessary to host the service component. When all service components finish, the service calls `stopSelf` to revert to its “bound” life cycle, that is, being automatically stopped when all clients unbind.

The service broadcasts *intents* whenever a service component starts, stops, or needs to alert the service's clients of a change in its state (such as a connection finishing the handshake process). *Activities* use these broadcast *intents* to refresh the information they display.

All server components extend the `ThreadManager` abstract class, which in turn extends the `ServiceComponent` abstract class. These abstract classes are separate because each focuses on a separate problem. `ServiceComponent` provides a common interface for the service to start the components, monitor them, request them to stop, and receive a callback once they stop, all while making sure these methods are not accidentally called from another thread. `ThreadManager` allows each service component to start an arbitrary number of threads and automatically finish once all these threads are finished.

Similarly, all the used threads extend the abstract class `InterruptibleWorkThread`, which in turn extends the abstract class `ManagedThread`. The `ManagedThread` abstract class only defines an external interface that invokes callbacks when the thread starts and when it stops, plus it allows an optional implementation of an asynchronous stop request. The `InterruptibleWorkThread` abstract class implements the asynchronous stop in a way that is thread-safe.

The `Server` service component only runs a single thread that listens for incoming connections.

The `VolumeMonitor` service component also runs a single thread, called `scanner`

The `Connection` service component defines five threads, with three at most running at the same time. The `connector` handles establishing the TCP link, the `handshaker` waits for the TLS handshake to finish, the `receiver` blocks on the socket's input and reads data, the `sender` reads messages from a synchronized queue and dispatches them to the socket's output, and the `engine` handles the communication protocol and data synchronization.

5.8 File system access

To ensure the atomicity of file updates, the class `TempData` within the `service` package implements the functionality necessary to:

1. create temporary files for storing downloaded data;
2. once the download is complete, rename the files to distinguish them from incomplete ones; and
3. atomically replace a file in the user's volume by the temporary file after all the dependencies are satisfied.

The mechanism relies on the assumption that moving a file within the same file system is atomic on Linux-based systems. The temporary files are stored in a directory named `.peersync-tmp` inside the volume's root. This directory is always excluded from scanning.

Chapter 6

Testing

In this section, we will define a test suite to check if the implemented application performs its work correctly. We will then perform the test and report the results.

6.1 The goals

We wish to test the following:

- Basic functionality.
- Empty directories.
- Deep directory structures.
- Empty files.
- Large files (over 1 MiB).
- Deleting directories.

6.2 The setup

We used three different Android emulator instances for the course of our tests:

- Device A: Android 1.6 (oldest supported version).
- Device B: Android 2.1.
- Device C: Android 4.2 (latest version as of 2013-05-11).

6.3 The test

This test case was designed to test all the named goals in a sequence.

The test shall begin with all three devices connected to each other, having an empty database. Device A shall create a volume named “test-volume” with the following contents:

```
(volume root)
├─ small-file (100 bytes)
├─ empty-file (0 bytes)
├─ large-file (4 MiB)
├─ empty-dir/
├─ normal-dir/
│   └─ shallow-file (100 bytes)
└─ deep-dir/
    └─ level2/
        └─ level3/
            └─ deep-file (100 bytes)
```

The files shall be filled with arbitrary data (for example, repetitions of a single character).

The following tests shall be performed in a sequence, always using the results from the previous ones.

6.3.1 Initial scan

Steps:

1. Create the initial structure described above on device A.
2. Perform a scan of the volume on device A.

Expected results:

- 10 new metadata objects are created.

Observed results: The expectations were met.

6.3.2 Basic replication

Steps:

1. Import the volume onto device B.
2. Perform a pull of the volume from device A to device B.

Expected results:

- 1 new snapshot is created on device A.
- The snapshots and metadata objects are replicated exactly (with the exception of mutable fields).
- The directory structure, file names, and file contents of B’s image of the volume match A’s image exactly.

Observed results: The times of last modification in the database differ between the device, which is intentional. The volume on device B contains the directory `.peersync-tmp`, which was intentionally created by the tool. The *size* fields for directory metadata have changed from `NULL` to `0`, but this poses no problem as these values are ignored for directories. Otherwise, all expectations are met. Additionally, the Metadata were replicated in preserved order, which was not required.

6.3.3 Modifications

Steps:

1. On device A, perform the following modifications:
    - Create a new file named `new-file`, 100 bytes long, in the volume's root.
    - Change the contents of `empty-file` to be 100 bytes long.
    - Change the contents of `small-file` to be 0 bytes long.
    - Change the contents of `large-file` to be 4 MiB long, but differ from the original contents.
    - Delete the file `deep-file` in `deep-dir/level2/level3`.
2. Perform a scan of the volume on device A.
3. Perform a pull of the volume from device A to device B.

Expected results:

- 1 new snapshot and 5 metadata objects are created on device A. The snapshot refers to the 5 metadata objects and its parent reference points to the previous snapshot.
- The snapshots and metadata objects on devices A and B match exactly (with the exception of mutable fields).
- The directory structure, file names, and file contents of A's image of the volume match B's image exactly.

Observed results: Similarly to the previous test, times of last modification are not preserved, the directory `.peersync-tmp` is present, and *size* is changed from `NULL` to `0`. Otherwise, all expectations are met. The ordering of database objects is preserved as well, although this was not required.

6.3.4 Replication through an intermediary

Steps:

1. Import the volume from device B onto device C.
2. Perform a pull of the volume from device B to device C.

Expected results:

- The snapshots and metadata objects on devices A, B, and C are exactly the same (with the exception of mutable fields).
- The directory structure, file names, and file contents of A's, B's, and C's images of

the volume match exactly.

Observed results: Similarly to the previous two tests, times of last modifications are not preserved (and are in fact set to `NULL` for old versions), the directory `.peersync-tmp` is present, and *size* is changed from `NULL` to `0`. Otherwise, all expectations are met. In this case, the ordering of database objects is not preserved.

6.3.5 Directory tree deletion

Steps:

1. Delete the directory `normal-dir`, including the file `shallow-file` it contains.
2. Perform a scan of the volume on device A.
3. Perform a pull of the volume from device A to device B.

Expected results:

- 1 new snapshot and 2 metadata objects are created on device A. The snapshot refers to the 2 metadata objects and its parent reference points to the previous snapshot. The metadata objects have the *deleted* flag set to `1` and refer to `normal-dir` and `shallow-file`, respectively.
- The snapshots and metadata objects on devices A and B match exactly (with the exception of mutable fields).
- The directory structure, file names, and file contents of A's image of the volume match B's image exactly.

Observed results: As with all the tests, times of last modifications are not preserved, the directory `.peersync-tmp` is present, and *size* is changed from `NULL` to `0`. Otherwise, all expectations are met.

Chapter 7

Possible extensions

The implemented application provides a working basis for a decentralized system synchronization tool. Still, there are many areas where the application could be improved or given extended functionality. Some of these areas will be briefly mentioned in this chapter.

Address book.
To make connecting to other peers easier, the *New connection* dialog could present to the user a list of known host names and IP addresses to select from. The list could be filled manually from a configuration activity, automatically from the addresses of connected clients, and/or automatically from server addresses entered in the *New connection* dialog.

Automatic connection.
To make synchronization more transparent to the user, options could be provided to connect to hosts automatically when their presence in the network is detected. The detection could be based on polling known IP addresses for responses and/or on sending broadcast or multicast messages in the local network segment.

Automatic reconnection.
If an error occurs in a connection or the application is shut down, the connection is not restored when the condition is remedied. The application could be extended to try to re-establish connections that had previously been broken to provide a more seamless use.

Automatic synchronization.
Instead of requiring the user to start the synchronization process manually, the application could be set to start synchronizing upon connecting to a peer or discovering a common volume.

Backup.
Although backup facilities were intentionally omitted from the Android implementation, we cannot rule out that they might be useful to someone. To facilitate backup, the application would have to store copies of files outside of the volumes' locations.

Certificate generation.
To ease device setup, the application could be extended to generate X.509 certificates and private keys itself, instead of relying on an external generator and an import

facility. This would require including external libraries or creating custom classes for encapsulating certificates and private keys.

Database pruning.
As time progresses, the synchronized database increases in size, reducing the amount of available space on the device. Using algorithms to search for outdated entries, these entries could be removed from the database, freeing space.

Delta encoding.
Transferring large files could be greatly optimized by employing the `librsync`'s remote delta algorithm. The network protocol supports this. However, the Android NDK is required in order to use a library written in C, making the build process more complicated. Alternatively, an implementation of `librsync` could be created in Java.

Encrypted storage.
The protocol could be extended to allow having file data (and possibly metadata) encrypted using a shared key. This would enable zero-knowledge setups where certain peers (possibly file hosts) store encrypted data without having access to the decrypted information.

Interactive certificate validation.
Instead of requiring the user to import all trusted certificates beforehand, the application could present the fingerprints of unknown certificates to the user when connecting to or accepting connections from new hosts, allowing the user to verify the fingerprint visually and then have the certificate imported into the local trust database automatically.

Private key encryption.
When importing private keys to the Android device, they have to be decrypted, creating a window of opportunity for rogue applications to compromise the key. This could be avoided by using encrypted private keys. This would, however, require external libraries or a custom implementation of key encryption.

Real-time synchronization.
Instead of performing scans on the local file system, a file monitor could be created to watch all changes to the file system, update the database, and distribute the changes to connected peers in real time. This would also allow detection of removals a renames, operations which are interpreted by a naïve scan to be a deletion followed by an addition.

User friendliness.
Although the application was designed for advanced users, some improvements could be made to make it more attractive and usable for intermediate users, by providing automated actions, helpful messages, easier setup and a cleaner user interface.

Chapter 8

Conclusion

In this bachelor's thesis, we have successfully demonstrated that a synchronization tool that does not require a central element to operate can be implemented using concepts similar to the database of objects used by Git.

We have analyzed Dropbox, a centralized synchronization tool; we took from the concepts of SSH and Git; we analyzed the workings of SyncML, the ways to enable Transport Layer Security (TLS), and the way our synchronization tool could be extended to transfer large files efficiently using `librsync`.

From these findings, we picked the concepts to use in our application, named PeerSync, which we subsequently designed and implemented for the Android operating system.

The implemented application demonstrates the ability to manage multiple volumes, to connect to multiple devices at once, and to be able to synchronize basic file and directory operations such as addition, modification, and removal.

Our application provides security using TLS connections and user-defined trust databases. Efficiency is ensured by using incremental snapshots. The user can select what data to transfer using multiple volumes.

Due to different implementation schedules and operational states of the related bachelor's theses for the Linux and Windows implementations, the application was not fully tested in a multi-platform environment. However, initial tests showed that the network communication was working.

The working basis created in this bachelor's thesis can be further extended in many aspects, as was described in chapter 7. These may provide easier use, more functionality, more control, and other features.

An article describing the basic concepts of this application was submitted to the conference *HCI International 2013.*

Bibliography

[1] Syncml sync protocol, version 1.1. `http://www.syncml.org/docs/syncml_sync_protocol_v11_20020215.pdf`, 2002-02-15 [cited 2013-01-28].

[2] Gnutls 3.2.0. `http://www.gnutls.org/manual/html_node/index.html`, 2013-04-24 [cited 2013-05-11].

[3] Android developers: Android ndk. `http://developer.android.com/tools/sdk/ndk/index.html`, [cited 2013-01-28].

[4] Android developers: App components. `http://developer.android.com/guide/components/index.html`, [cited 2013-05-11].

[5] Android developers: Package index. `http://developer.android.com/reference/packages.html`, [cited 2013-05-11].

[6] Android developers: Storage options. `http://developer.android.com/guide/topics/data/data-storage.html`, [cited 2013-05-11].

[7] Gnutls openpgp key support. `http://gnutls.org/openpgp.html`, [cited 2013-05-11].

[8] The gnutls transport layer security library. `http://gnutls.org/`, [cited 2013-05-11].

[9] librsync. `http://librsync.sourceforge.net/`, [cited 2013-05-11].

[10] Android developers: Dashboards. `http://developer.android.com/about/dashboards/index.html`, [cited 2013-05-12].

[11] Scott Chacon. *Pro Git*. Apress, 2009. ISBN 978-1430218333.

[12] Tim Dierks and Eric Rescorla. Rfc 5246. *The Transport Layer Security (TLS) Protocol Version 1.2*. `http://tools.ietf.org/html/rfc5246`, 2008 [cited 2013-05-12].

[13] Dropbox, Inc. Can i specify my own private key for my dropbox? [online]. `https://www.dropbox.com/help/28/en`, [cited 2013-05-11].

[14] Dropbox, Inc. How do i find and change my desktop application preferences? [online]. `https://www.dropbox.com/help/166/en`, [cited 2013-05-11].

[15] Dropbox, Inc. Why aren't certain files on one computer syncing to another? [online].

https://www.dropbox.com/help/145/en, [cited 2013-05-11].

[16] Dropbox, Inc. Why can't i establish a secure connection? [online]. https://www.dropbox.com/help/159/en, [cited 2013-05-11].

[17] Google Inc. Protocol buffers. developer guide. https://developers.google.com/protocol-buffers/docs/overview, 2012-04-02 [cited 2013-05-11].

[18] Google Inc. Protocol buffers. techniques. https://developers.google.com/protocol-buffers/docs/techniques, 2012-04-02 [cited 2013-05-11].

[19] Google Inc. Protocol buffers. language guide. https://developers.google.com/protocol-buffers/docs/proto, 2013-03-05 [cited 2013-05-11].

[20] Nikos Mavrogiannopoulos. Rfc 5081. *Using OpenPGP Keys for Transport Layer Security (TLS) Authentication.* http://tools.ietf.org/html/rfc5081, 2007 [cited 2013-05-12].

[21] Mark L. Murphy. *Android 2: Průvodce programováním mobilních aplikací.* Computer Press, a.s., 2001. Vydání první. ISBN 978-80-251-3194-7.

[22] Andrew Tridgell and Paul Mackerras. The rsync algorithm. http://rsync.samba.org/tech_report/, 1998-11-09 [cited 2013-05-11].

[23] Tatu Ylonen and Chris Lonvick. Rfc 4251. *The Secure Shell (SSH) Protocol Architecture.* http://www.ietf.org/rfc/rfc4251.txt, 2006 [cited 2013-05-11].

Appendix A

The synchronization protocol

This appendix defines the syntax and semantics of the protocol used to exchange information between two peers. It builds on top of Google's protocol buffers.

A.1 Syntax

These are the contents of the `peersync.proto` file, whose exact copy is used in both the Linux and the Windows implementation.

Note. Some white space was removed to make the definition more suitable for printing. This has no effect on the semantics of this file.

Note. The actual version used in the Android implementation was altered by adding Java-specific options, which are not shown here. They have no effect on the protocol itself.

```
package sync;

option optimize_for = LITE_RUNTIME;

enum MessageType {
    ERROR          =  2;
    HELLO          =  3;
    ADDED_VOLUME   =  4;
    REMOVED_VOLUME =  5;
    UPDATE         =  6;
    SYNCHRONIZED   =  7;
    SUCCESS        =  8;
    PULLED         =  9;
    SNAPSHOT       = 10;
    METADATA       = 11;
    DATA           = 12;
    PULL           = 13;
    STOP_UPDATES   = 14;
    GET_OBJECT     = 15;
    GET_DATA       = 16;
    DATA_SIGNATURE = 17;
    CANCEL         = 18;
```

```
}

// wrapper for all message types
message Message {

    // enum values equal type tags, so value 1 is omitted
    required MessageType   type            =  1;

    // hybrid messages
    optional Error         error           =  2;

    // notifications
    optional Hello         hello           =  3;
    optional AddedVolume   addedVolume     =  4;
    optional RemovedVolume removedVolume   =  5;
    optional Update        update          =  6;
    optional Synchronized  synchronized    =  7;

    // responses
    optional Success       success         =  8;
    optional Pulled        pulled          =  9;
    optional Snapshot      snapshot        = 10;
    optional Metadata      metadata        = 11;
    optional Data          data            = 12;

    // requests
    optional Pull          pull            = 13;
    optional StopUpdates   stopUpdates     = 14;
    optional GetObject     getObject       = 15;
    optional GetData       getData         = 16;
    optional DataSignature dataSignature   = 17;
    optional Cancel        cancel          = 18;

}

// ----------------
// HYBRID MESSAGES
// ----------------

// an error, either transactional or not
message Error {
    enum ErrorCode {
        INTERNAL_ERROR      =  1;  // problems not related to the protocol
        SYNTAX_ERROR        =  2;  // error in the protocol, will disconnect
        UNKNOWN_MESSAGE     =  3;  // message not recognized, will be ignored
        OBJECT_NOT_FOUND    =  4;  // UUID not found in database
        DATA_NOT_FOUND      =  5;  // file contents not stored
        USER_CANCELLED      =  6;  // transaction terminated at user's will
        PEER_CANCELLED      =  7;  // transaction ended because of Cancel
        INVALID_REQ_ID      =  8;  // reference to a non-existing txn ID
        VOLUME_NOT_FOUND    =  9;  // volume UUID not found
        BAD_REQUEST         = 10;  // interchanged GetObject with GetData
    }
    optional uint32    response_to = 1;  // sender's transaction ID, if any
    required ErrorCode code        = 2;  // generic code of the error
    optional string    message     = 3;  // English description for debugging
}

// ----------------
// NOTIFICATIONS
// ----------------

// mandatory initial message
message Hello {
    repeated string options         = 1;    // list of extended capabilities
    optional bytes  peer_id         = 2;    // peer UUID
    optional string peer_name       = 3;    // human-readable name
}

// a volume is made available for sync
message AddedVolume {
    required bytes  volume_id       = 1;    // volume UUID
    optional string volume_name     = 2;    // human-readable name
}

// a volume is no longer available for sync
message RemovedVolume {
    required bytes volume_id        = 1;    // volume UUID
}

// announcement about a new metadata object
message Update {
    required bytes  volume_id       = 1;    // the volume containg the metadata
    required bytes  metadata_id     = 2;    // metadata UUID
}

// announcement about having finished a synchronization
message Synchronized {
    required bytes  volume_id       = 1;    // the volume having been synced
    optional bytes  snapshot_id     = 2;    // optionally the resulting snapshot
}

// ----------------
// RESPONSES
// ----------------

// a simple affirmative response
message Success {
    required uint32 response_to     = 1;  // recipient’s transaction ID
}

// a response with the latest snapshot UUID
message Pulled {
    required uint32 response_to     = 1;  // recipient’s transaction ID
    repeated bytes  snapshot_ids    = 2;  // snapshot UUID
}

// a snapshot object
```

```
message Snapshot {
    required uint32 response_to     = 1;  // recipient's transaction ID
    optional bytes  parent_id       = 2;  // parent snapshot UUID
    repeated bytes  metadata_ids    = 3;  // metadata UUIDs in this snapshot
}

// a metadata object
message Metadata {
    enum Type {
        FILE       = 1;  // ordinary file
        DIRECTORY  = 2;  // aka folder, has no size and no data_id
        PSEUDOFILE = 3;  // not saved in the filesystem, used for special purposes
    }
    required uint32 response_to = 1;  // recipient's transaction ID
    optional bytes  parent_id = 2;  // parent metadata UUID
    optional bytes  resolves  = 3;  // conflicting branch metadata UUID
    optional bytes  dir_id    = 4;  // parent directory UUID, unless in root
    required Type   type      = 5;  // type of the filesystem entry
    required string file_name = 6;//name of the file or directory (UTF-8, no path)
    optional uint64 size      = 7;  // size of the file in bytes
    optional bytes  data_id   = 8;//file contents (null if deleted or directory)
    optional bool   deleted     = 9 [default = false];
}

// response to GetData
message Data {
    required uint32 response_to     = 1;    // recipient's transaction ID
    required bool   final           = 2;    // true iff this is the last chunk
    required bytes  chunk           = 3;    // chunk of data
    // format of the chunk depends on the download method (raw or librsync)
}

// ----------------
// REQUESTS
// ----------------

// request for the latest volume snapshot + optionally new updates
message Pull {
    required uint32 request_id      = 1;    // sender's transaction ID
    required bytes  volume_id       = 2;    // volume UUID
    optional bool   start_updates   = 3 [default = false];
}

// request to stop receiving new updates if previously started
message StopUpdates {
    required uint32 request_id      = 1;    // sender's transaction ID
    required bytes  volume_id       = 2;    // volume UUID
}

// download an object (metadata or snapshot)
message GetObject {
    required uint32 request_id      = 1;    // sender's transaction ID
    required bytes  object_id       = 2;    // metadata or snapshot UUID
}
```

```
// download/synchronize a file
message GetData {
    enum Method {
        FULL            = 1;    // download the entire file (start immediately)
        REMOTE_DELTA    = 2;    // use librsync (DataSignature must follow)
    }
    required uint32 request_id      = 1;    // sender's transaction ID
    required bytes  data_id         = 2;    // data UUID
    required Method method          = 3;    // whether to use deltas or not
}

// additional information for GetData with method = REMOTE_DELTA
message DataSignature {
    required uint32 request_id      = 1;    // sender's existing transaction ID
    required bool   final           = 2;    // true iff this is the last chunk
    required bytes  chunk           = 3;    // chunk of librsync file signature
}

// cancel an existing transaction
message Cancel {
    required uint32 request_id      = 1;    // ID of the cancellation itself
    required uint32 to_cancel       = 2;    // ID of the transaction to cancel
    // both IDs are sender's transaction IDs
    // (recipient's transactions must be cancelled by sending an Error)
}
```

A.2 Message encapsulation

Each message in the protocol is preceded by a 4-byte, big-endian unsigned integer, denoting its length in bytes. The maximum length of a message is 327680 bytes.

The message itself must be an encoded form of the message named `Message`. Exactly one of its optional fields must be set, namely the one denoted by the value of the `type` field.

From now on, we will refer to the optional fields of `Message` as “inner messages”. Whenever an inner message is mentioned, it will be assumed that proper encapsulation is done before sending the message over the network, and that proper decapsulation is done after receiving the message.

Messages where the proper inner message is not set must be dropped by the recipient. If an additional (improper) inner message is set, the recipient must ignore the inner message and may drop the whole message.

A.3 Message sequence

Once the lower layers of the connection are established, both parties must send a `Hello` message to identify themselves and present a list of optional extensions to the protocol that they support. These extensions are identified by short strings and designed to allow forward compatibility.

After the Hello exchange is completed, both parties may start participating in an asynchronous exchange of messages other than Hello. Due to the asynchronous nature of the protocol, a response to a request may not be sent immediately; in fact, other messages may be sent before the response. The recipient must not assume any particular order of independent messages.

Errors (Error) can be transactional or non-transactional, depending on whether the response_to field is set or not. If the field is set, the sender's transaction with the given number must be considered unsuccessful, but the connection need not be terminated. If the field is not set, the error is non-transactional and the connection should be terminated.

Notifications (AddedVolume, RemovedVolume, Update, Synchronized) do not expect any response.

Requests expect a response whose response_to field matches the request_id field in the request. The response must be one of the messages denoted in the following table or an Error:

Request type	Response type
Pull	Pulled
StopUpdates	Success
GetObject	Snapshot, Metadata
GetData, DataSignature	Data
Cancel	Success

With the exception of GetData, all transactions consist of exactly of one request and one response. The GetData request must be followed by one or more DataSignature messages if its method field is set to REMOTE_DELTA, and the Data response may be sent in multiple messages if the requested data does not fit into a single message.

The connection is terminated by simply closing the communication channel. The implementation must be robust enough to be able to recover from the channel breaking at any given time.

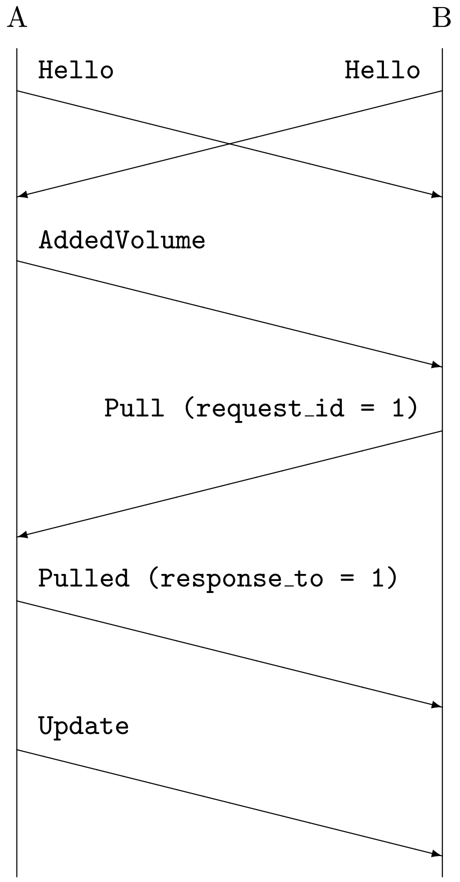

Figure A.1: Example message sequence.

Appendix B

User manual

B.1 Overview

PeerSync is a system tool for advanced users that allows the synchronization of files and directories without relying on a central server. It allows permanent or intermittent connections among pairs of devices in arbitrary IP networks, made of devices running PeerSync on Android, Linux, or Windows, attempts to negotiate the latest versions of files in a shared file system structure, and updates data over the network.

The PeerSync network protocol relies on Transport Layer Security (TLS) for maintaining the authenticity, confidentiality, and integrity of network operations. Each installation of PeerSync requires its own key-pair for authenticating itself, as well as a database of trusted certificates to authenticate other peers. The certificates may be self-signed.

Each device running PeerSync can maintain a number of distinct *volumes*, which are structures representing a directory and all of its contents within a local file system. A pair of devices can synchronize the contents of their common subset of *volumes*.

This manual documents the Android implementation of PeerSync.

B.2 Installation

The CD contains the file `PeerSync.apk` that can be directly installed onto any Android device, provided its Android version is 1.6 or higher. The usual process of installation is as follows:

1. Install the Android SDK from `http://developer.android.com/sdk/index.html` onto your computer.

2. If your computer is running Windows, acquire and install the USB driver for your device. A driver for certain devices can be downloaded at `http://developer.android.com/sdk/win-usb.html`, other devices require specific drivers provided by the hardware manufacturer.

3. On your Android device, go to *Application Settings* and enable installing software

from unknown sources.

4. From your computer, run the Android Debug Bridge (ADB) tool with the following command line: `adb install path/to/PeerSync.apk`.

Alternatively, you can copy the APK file onto your Android device's SD card and then install it using an application such as *ASTRO File Manager*.

B.3 Cryptography setup

To start using PeerSync, you first need to create a key pair to use for TLS. PeerSync requires a certificate in the X.509v3 format and a corresponding private key in PKCS#8 format. These cannot be created in the Android application itself, so you have to create them externally and then import them into PeerSync. To create the key pair, you can use the OpenSSL utility and the following commands:

```
openssl req -new -x509 -outform DER -out certificate.der -utf8 \
    -subj '/CN=Your name/' -newkey rsa:2048 -keyout raw_key.pem -nodes \
    -days 730 -set_serial 1
openssl pkcs8 -topk8 -inform PEM -in raw_key.pem -nocrypt -outform DER \
    -out pkcs_key.der
```

This example will create a self-signed X.509v3 certificate for the name "Your name", valid for two years, saved into the file `certificate.der` using Distinguished Encoding Rules (DER), and a corresponding 2048bit private RSA key, saved into the file `pkcs_key.der`. You can adjust the commands' arguments to suit your needs.

Warning: The private key will be stored unencrypted. Due to the current limitations of PeerSync, encrypted private keys cannot be imported. You should take care to prevent the private key from being compromised when creating it and when transferring it to your Android device.

After you create a key pair for each device, you have to launch PeerSync on that device, locate its *Keys and certificates* activity, and import both the certificate and the unencrypted private key.

You also have to import each certificate onto all the other devices you wish to use, in order for the certificate's owner to be trusted. The certificates may be self-signed because they are checked against a trust database, not by validating their signatures. When you are done, each device should contain all other devices' certificates in its trust database.

B.4 Testing the connection

After you set up the certificates and a private key, you should test if you have set them up properly. This can be most easily done by attempting to connect two devices.

PeerSync can operate both in server mode, by allowing other peers to connect to it, and in client mode, by connecting to the peers by itself.

If you wish to test PeerSync in server mode, determine the device's IP address, then launch

PeerSync, open its *Server* activity, enable the server, and try to connect to the device's IP address from another device, using the port number configured on the server.

If you wish to test PeerSync in client mode, start PeerSync on another device in server mode, then open the *Connections* activity on your Android device and try to create a connection to the server.

However you created the connection, try to locate it in the *Connections* activity. If you can see the connection, tap it to see the details. If the *Connection state* field contains the word *online*, you have successfully connected the two devices.

Once you have tested your connection, you can begin setting up *volumes* to synchronize among your devices.

B.5 Volume setup

To create a synchronized *volume*, you must create the files and directories on one device only and let the other devices download it. PeerSync does not support synchronizing pre-existing file system structures.

If your Android device is the one you wish to create a *volume* on, launch PeerSync, locate its *Volumes* activity, and tap *New volume* (either in the action bar or in the context menu, depending on your Android version). You will be prompted for the *volume*'s name and location. Use a sensible, short, human-readable name for the *volume* – this will be used for your convenience only, and is not important for the synchronization itself. As the location, choose the directory whose contents you wish to synchronize.

If you wish to import a *volume* into your Android device, first connect to another device, as described in the previous chapter. Then, locate the device in the *Connections* activity, and tap its item in the list. Then, from the action bar or the context menu, select *Import volume*, choose the *volume* you wish to import, and pick a location for it. (Do not choose any directories with existing files; PeerSync will create an empty directory.)

When you have a *volume* set up, you can start synchronizing it.

B.6 Synchronizing

The synchronization process in PeerSync is controlled manually. In addition, each device can only download changes from another device, not vice versa.

First, PeerSync needs to scan the *volume* to determine what has changed. If you want to scan the volume on your Android device, locate the *volume* in the *Volumes* activity and select *Scan* from the action bar or context menu. You should see the *Volume state* change from *idle* to *scanning* and, after a while, back to *idle*. If the state changes to *idle* and no error notification appears, the scan has finished successfully.

After the scan is done, you need to pull the changes to the other device. If the other device is an Android device, connect it to the device where the scan was done, then locate the connection in the *Connections* activity, select *Pull changes* from the action bar or context menu, and choose the *volume* that you wish to synchronize.

If you wish to synchronize the devices both ways, reverse their roles and repeat the process.

B.7 Conflict resolution

At times, there may be cases when a file or a directory in a particular *volume* is changed on two or more devices before the devices can synchronize the change. When the synchronization finally happens, the changes would try to overwrite each other, which is why this is called a *conflict*. To avoid damaging your data, PeerSync will show a notification alerting you of the presence of a conflict. By tapping the notification or opening the *Conflicts* activity in PeerSync, you can view the files and directories in conflict.

To resolve a conflict, delete or move away the file in question on one device, then synchronize again. The conflict should disappear.

B.8 Backup

Android was designed to be an operating system for devices with limited storage and computational power, such as mobile phones and tablets. With this in mind, the Android implementation does not include facilities to be used as a backup storage, as this was deemed unsuitable for Android devices. PeerSync for Android will respond to requests for older versions of synchronized files, but will always report an *Object not found* or *Data not found* error to the requester.

Still, it is possible to backup data from the Android device onto Linux or Windows devices running PeerSync, provided that they have enabled storing old versions of synchronized files.

B.9 Caveats

The Android implementation of PeerSync synchronizes the *external storage*, which is typically an SD card. Since the capacity of SD cards is limited, it is not advisable to have large *volumes* (spanning hundreds of megabytes or more) from desktops or servers synchronized with Android devices.

Keep in mind that since PeerSync can perform potentially heavy network transfers, it may both significantly reduce your device's battery life and increase charges for data, depending on your carrier and data plan. It is recommended that you only run PeerSync when plugged in a power source and connected over a Wi-Fi network.

Attempting to synchronize a *volume* that has been out of sync for a long time may cause a large number of conflicts. Consider whether wiping the out-of-sync *volume* and importing it again might be a better option.

Avoid using PeerSync on directories that change content extremely often, such as log file directories or temporary file directories, to prevent flooding the network and the internal database.

Make sure you re-generate and re-distribute your certificates and private keys when their expiration date comes close.

Keep your private keys safe. If a malicious party compromises your private key, it can read all of your data (which might contain confidential information) and/or flood your device with large amounts of garbage data, causing denial of service. If you suspect that a private key has been compromised, remove its corresponding certificate from all of your devices and create a new key pair for the device in question.

Older versions of Android contain outdated Java libraries which do not support newer versions of TLS, leaving the connections vulnerable to known attacks. For best security, upgrade your Android OS whenever possible.