A Filesharing App for Android based on Opportunistic Networks

Bachelor Thesis

by

Oliver Rohr

born in

Mettmann

submitted at

Technology of Social Networks Lab

Jun.-Prof. Dr.-Ing. Kalman Graffi

Heinrich-Heine-Universität Düsseldorf

17. February 2017

Supervisor:

Andre Ippisch, M.Sc.

Abstract

In the modern world we all carry at least one of them with us. Smartphones. Tiny computers capable enough to make calls, write messages and grant access to the Internet. But most of all, they can entertain us with pictures, music or videos. Most of us like to share these with our family and friends. Since they are often nearby, it would be a waste to use our mobile data for this, especially when the files we would like to share are huge. Opportunistic Networks provide local networks in which this sharing can be done easily and without the use of the Internet. The main motivation of this thesis is the implementation of an Android filesharing app called FileShipping. FileShipping will enter the Opportunistic Networks via Opptain, an app developed by the Technology of Social Networks Lab of the Heinrich-Heine-Univserität Düsseldorf. For the implementation of FileShipping we will have to invent a protocol that is capable to provide the communcation within the Opportunistic Networks. Throughout the implementation we will see how this protocol and FileShipping can be misused by malicious peers.

Acknowledgments

I would like to thank all the people who supported me during my work on this thesis.
A special thanks goes to my girlfriend who was always there and cheered me up, when work was overwhelming and didn't turn out as expected.
Another special thanks goes to my family, because i could always rely on them when needed.
At last I want to thank the Technology of Social Networks Lab and especially my supervisor Andre Ippisch who patiently answered all of my questions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the modern world we all carry at least one of them with us. Smartphones. Tiny computers capable enough to make calls, write messages and grant access to the Internet. But most of all, they can entertain us with pictures, music, funny cat-videos, maybe a movie or even a complete series. We get them from the Internet and show them to our friends. Later we think of the cat video we have seen on our friend's smartphone before and think that we would like to have it too and share it with others, but unfortunately our monthly mobile data is used up, the Internet is currently not accessible or we just can't find it on the Internet. But maybe exactly this file we are looking for is currently not far away on a stranger's device and he would like to share it. That is a case for Opportunistic Networks, Opptain and FileShipping.

1.1 Objectives

The main objective of this thesis is the implementation of FileShipping, an Android Filesharing-App that uses Opportunistic Networks via Opptain (Opptain is further explained in section 2.2). All peers are equal in Opportunistic Networks. In this Thesis we will refer to peers which request something from others as "Clients" and those peers which receive these requests as "Servers" or "Hosts". Filesharing is a service that allows a *Server* to passively share chosen files with the network, that can be requested by

Clients. As it is far too complicated for *Clients* to search in the network for a file by e.g., its name and actually get the file they were looking for, it should be able for the *Servers* to mark their files with Tags. The *Client* may then look for a list of Tags, instead for a files name, which have to be matched by the *Server's* files, before the *Server* offers them to the *Client*. Since FileShipping is based on Opportunistic Networks, any peer can be a *Server* for our purposes. In case we have multiple *Servers* for our requested file, we also want to have the opportunity to collect parts of the file from each *Server*. For the parts of a file we will later use the term "Partfile". FileShipping can then reconstruct the original file from these Partfiles. It is necessary to invent a communication protocol that allows multiple instances of FileShipping to exchange messages and also to invent a database that stores fundamental information. The app was implemented for Android devices with the use of Android Studio. For further explanation to Android Studio see section 2.3.

1.2 Related Work

As mentioned before, FileShipping is using Opportunistic Networks, which are explained more precisely in section 2.1. Android devices cannot use Opportunistic Networks by default, that is why FileShipping requires another service that handles the routing and bundle exchange. This service is Opptain which will be explained in more detail in section 2.2

1.3 Outline

So far the motivation for this project has been explained. In the first part of the following chapter 2 shall be explained, what Opportunistic Networks are and how messages can be routed in these. Further we will have a look at the fundamental functionalities of the Android Operating System (OS), such as how the OS is handling its running applications, how tasks can be done in the background, and the way different apps can communicate. In the 3. chapter, we will have a look at the exact functionality of FileShipping, the way the app is structured (section 3.1), what messages exist in the protocol and how they are

used (section 3.3), and the way FileShipping stores its data (section 3.2).
The next chapter “Issues“(5) shows some security problems, which FileShipping is currently facing. Here it will be explained, how FileShipping can be misused, what has been or can be done to prevent the misuse or how to make it at least more difficult for a malicious user to misuse FileShipping.
In the last chapter (6) some of the solutions from chapter 5 and other possible features for FileShipping will be discussed.

Chapter 2

Fundamentals

The communcation of FileShipping is based on Opportunistic Networks (OppNets). The access to these networks is provided by Opptain. This chapter shall offer a basic overview about Opportunistic Networks and the functionality of Opptain.
Additionally, we will have a look at some characteristics of the Android OS which will have a major influence on the implementation of FileShipping.

2.1 Opportunistic Networks

Opportunistic Networks are mobile ad hoc-networks (MANETs), in which “unpredictable and unstable topologies, prolonged disconnections and partitions can occur frequently “ [I.W13].

They are a type of Delay Tolerant Networks (DTNs). This means that the Opportunistic Networks are networks consisting of smaller networks. Technically every device can have its own local network and messages are passed from a specific device to another by passing the messages through the networks of nearby devices. A end-to-end path between two devices may never exist at a time.

In an Opportunistic Network every device has the functionality of a router. When it comes to a specific service, such as FileShipping, every device can take the role of a

Client or a *Server*.

2.2 Opptain

Opptain is developed by the Technology of Social Networks Lab of the Heinrich-Heine-Universität Düsseldorf. Opptain uses the Wi-Fi or Bluetooth connection of a device to open local networks. One device takes the role of a master and is used by the slave devices as a hotspot. Bundles are passed between the master and its slaves, but never between two slaves. To make the exchange of bundles between two slaves possible, the master looses its task every few seconds and the devices seek for a new one. Whichever device wins, becomes the new master. The new master can now exchange bundles with the devices in its own range. Through the switching of the master and the movement of the user with the device itself, a bundle can be carried over a long distance.

2.3 Android Studio

Android Studio is the official Integrated Development Enviroment (IDE) developed by Google Inc. and is based on IntelliJ IDEA. Android Studio has been published in 2013. It provides features like a Gradle-based build system, a built-in emulator, Instant-Run and Lint tools [Goo15d] which were also used throughout the development of FileShipping.

2.4 Android OS

The Android OS (Android Operating System) is an open source software developed by Google. The OS is built on top of the Linux Kernel 2.6. From release of Android 4.x upwards, it is built on top of Linux Kernel 3.x series. The Linux Kernel is responsible for the memory- and processmanagement, abstraction of the hardware and networking.

Likelihood of being killed	Process state	Activity state
Least	Foreground (having or about to get focus)	Created Started Resumed
More	Background (lost focus)	Paused
Most	Background (not visible) Empty	Stopped Destroyed

Table 2.1: Likelihood of a process to be killed by the OS [Goo15a]

The Linux Kernel is used due to its portability on many different machines. Since the Android OS is built on top of the Linux Kernel and it is an open source software itself, it can easily be customized by the manufacturer and used for many different devices [M. 15].

2.4.1 Activity Lifecycle

Android applications undergo a lifecycle in which predefined methods are being called whenever an applications activity changes its state. “The activity class provides a core set of six callbacks: onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy().“ [Goo15a]. Usually applications are started and closed by the user, in other cases the OS may shut down stopped processes, when memory is required for a running process or when the process is not responding for a certain time. When the application’s process is destroyed, all its related processes are destroyed as well. The OS is more likely to destroy an activity when its state is paused, stopped or destroyed [Goo15a] (See table 2.1).

2.4.2 Processes and Threads

Each Android application gets its own process by default, the so called “main“ process. All components of the application regarding the User Interface (UI) run on this process, it also collects and dispatches events on the Android UI [Goo15c]. Tasks that have no direct influence on the UI should be done in services or for smaller tasks in *Workerthreads*.

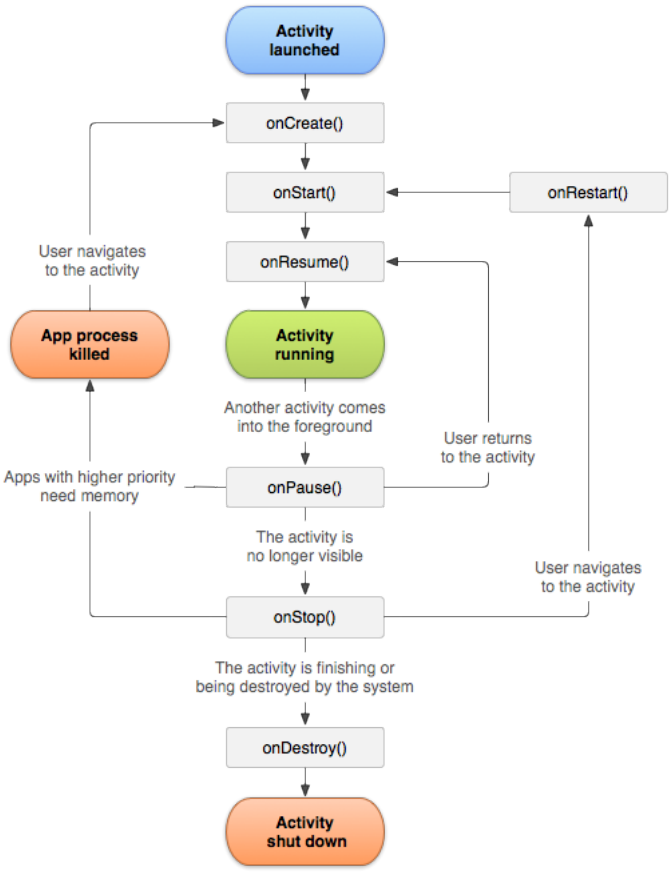

Figure 2.1: An activity's lifecycle [Goo15a]

Android provides the Thread-class and the AsyncTask as *Workerthread*.

Service

"A *Service* is an application component that can perform long-running operations in the background, and it does not provide a user interface. Another application component can start a service, and it continues to run in the background even if the user switches to another application" [Goo15e]. A service can be started by another application that might have sent a broadcast message which will be processed by the receiving application service.

AsyncTask

"AsyncTask allows you to perform asynchronous work on your user interface. It performs the blocking operations in a worker thread and then publishes the results on the UI thread (main thread) without requiring you to handle threads and/or handlers yourself" [Goo15c]. This makes AsyncTasks advantageous for accesses on databases or files within the external storage. The AsyncTask provides the method doInBackground(), where the main part of the task's work should be located. It also provides the onProgressUpdate() and onPostExecute() methods, that can be used to notify the main thread about the progress and the finish of the Workerthreads task.

2.4.3 Permissions

"To maintain security for the system and users, Android requires apps to request permission before the apps can use certain system data and features." [Goo15b]. FileShipping requires the Permissions to access the external storage and additionally the Permission to send COMMUNICATION Intents, provided by Opptain (figure 2.1).

Listing 2.1: The Permissions used by FileShipping

```
1     <uses-permission android:name=
2         "de.opptain.waitress.permission.COMMUNICATION" />
3     <uses-permission android:name=
4         "android.permission.WRITE_EXTERNAL_STORAGE" />
5     <uses-permission android:name=
6         "android.permission.READ_EXTERNAL_STORAGE" />
```

2.4.4 StrictMode

The StrictMode is a tool that observes the duration of blockings of an applications UI-Thread. The UI-Thread's main purpose is the management of the UI. Tasks that require a long time for its processing, for example an access of the database, should be done in *Workerthreads*. If the StrictMode is enabled and the UI-Thread does a costly task, a StrictMode violation is thrown.

Chapter 3

Implementation

FileShipping is a type of filesharing application, meaning that FileShipping can and is used for the hosting of files, which a user would like to share with others. Moreover FileShipping can be used to request files from other hosts, who use FileShipping too.

The first step of the implementation of FileShipping is to think of the functionalities which are required to create a usable app. One part surely comes from the definition of a filesharing app itself, but this is not enough to describe the process of the development of an application. When we take a look at FileShipping, we can divide it into three main elements. When we launch the app, the first thing we see is the UI itself. We need to provide the possibility for the user to interact with the app. We also need to display currently running queries and their already collected results, files which are usable by FileShipping and the contact list. These things are part of the UI (section 3.1).
The second element comes to mind, when we think about the storage. The running searches and their results, the picked files and their tags and contacts, are all organized in objects which have a current state. Since we don't want to loose this state whenever the process has been killed by the OS, we need to store these objects in a persistent database (section 3.2).
At last we have to think about the network. Since we want to share and collect data from other peers, we also need to implement a communication protocol that can be used to clarify our intentions in front of other devices (section 3.3).

3.1 User Interface

FileShipping provides eight different activities for the UI. The first activity, called MainActivity, is the entrypoint of the application. This activity will be shown first, whenever the application is launched. The MainActivity contains three fragments, which show basic information about the current state of the data used by FileShipping. Fragments can be replaced by other fragments at any time like an activity could be, but a user may not expect one of it to be “on top“ of another. These fragments are the QueryFragment (in section 3.1.1), the FileManagerFragment (in section 3.1.1) and the ContactFragment (in section 3.1.1). Each fragment shows a list of all current entries in the database of the corresponding table and provides the opportunity to show more detailed information about a single entry. That is how the user can reach the SingleQuery- (3.1.2), SingleFile- (3.1.3) and SingleContactActivity (3.1.4). Also each fragment contains a button that starts an activity, which provides the opportunity to add new queries, new files or new contacts. These activities are the AddQueryActivity (section 3.1.5), the AddFileActivity (section 3.1.6) and the AddContactActivity (section 3.1.7). At last the ResultActivity (3.1.8) can be reached from the SingleQueryActivity, it is used to collect a single file from multiple *Servers* at once as Partfiles, this is explained in more detail in chapter 3.3.

3.1.1 MainActivity

The MainActivity is the first activity shown, when the application is started. It has no actual own content, but it is a type of FragmentActivity, which makes it possible to show fragments. Although the fragments provide the content shown on the display, the FragmentActivity behind those is receiving performed actions, like pressing buttons. That is why the MainActivity class contains almost all logical methods, while the fragments only do tasks relating the view. The fragments can be chosen by swiping to the right which opens a DrawerMenu, and selecting the desired fragment. In the following we want to have a look on these fragments.

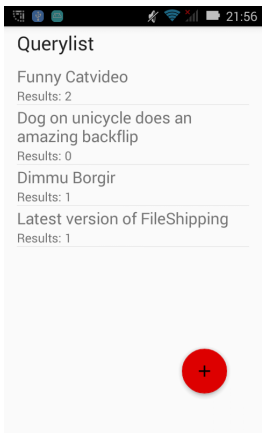

Figure 3.1: View of the QueryFragment

QueryFragment

The main purpose of the QueryFragment is to give an overview above all started queries. On first sight the user is able to see the name of the query, which the user associates with the specific search. Also the current number of received results is shown below. If the user needs to view the detailed information about the query, the item can be selected and the SingleQueryActivity is started for the chosen query. In case the user wishes to start a new query, the button on the upper right corner starts the AddQueryActivity. A screenshot of the QueryFragment can be seen in 3.1.

FileManagerFragment

In the FileManagerFragment the user can see a list of the files, which are currently useable for FileShipping. To add another file, the user may click the button on the top right corner, which then starts the AddFileActivity. To view the details of the file, the user can select a single file from the list and start the SingleFileActivity.

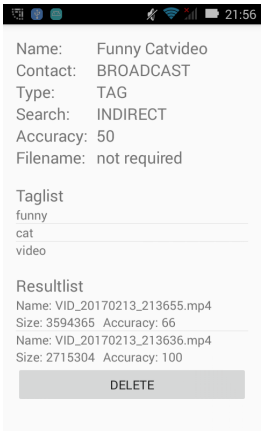

Figure 3.2: View of the SingleQueryActivity

ContactFragment

The ContactFragment provides the ability to have a look at all currently saved contacts and their names associated by FileShipping. The user can start the AddContactActivity by clicking the upper right button and also launch the SingleContactActivity by selecting a contact from the list.

3.1.2 SingleQueryActivity

The SingleQueryActivity shows detailed information of a certain query including all of the already received results. The information of a query cannot be edited here, because the query has already been wrapped in a bundle and handed to Opptain, which also might have already sent it to the network. After the query has returned with collected metainformation for files from the *Servers*, the SingleQueryActivity can be used to request the corresponding files. Also the user can decide to delete the query together with its results and tags here. Figure 3.2 shows an example for this activity.

3.1.3 SingleFileActivity

The SingleFileActivity shows all detailed information about the selected file from the FileManagerFragment. The information about the file cannot be changed at this point, but the corresponding TagList may be edited. Also the file can be deleted from the database at this point. Note that the deletion does not actually delete the file from the external storage, it just removes the reference from the database and makes the file unaccesible for FileShipping.

3.1.4 SingleContactActivity

The SingleContactActivity shows the same information as the ContactFragment, but their values may be edited in this activity. This activity provides space for further features, like starting a query from here.

3.1.5 AddQueryActivity

In the AddQueryActivity the user can set up a new query. FileShipping asks for a name which the user wishes to identify the query with. Additionally, the user can decide whether to send this query to a specific device by entering a name from the Contact-Fragment. FileShipping then looks up the entered name in its database and sets the corresponding DeviceID as the destination device. If the contact field is left blank, File-Shipping will assume that no particular device shall be asked, therefor the destination address will be set to the default broadcast adress. The user can also decide if File-Shipping shall collect headerinformation about files which match the query first, or if matching files should be returned „directly“. In case FileShipping shall look for a specific file with a known filename, the user can choose to start the search for a specific filename. But the most common way will be the search by a list of tags. These tags can be entered on the third field and be added to the list. In most cases, the entered tags won't match the desired file perfectly, that is why FileShipping offers the setting of an accuracy

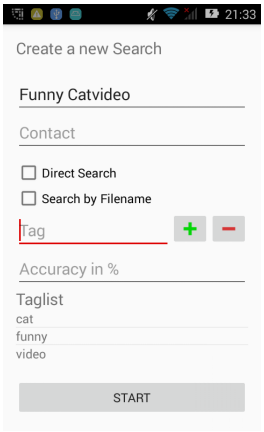

Figure 3.3: View of the AddQueryActivity

value, which is used to expand the number of matches. In Figure 3.3 a screenshot of the AddQueryActivity is shown.

3.1.6 AddFileActivity

With the AddFileActivity the user can pick files from his device, which may be used by FileShipping. AddFileActivity starts the Android built-in FileChooser which the user can choose a file from. FileShipping then takes all required values from the FileChooser's result. Additionally, the user has to set up a TagList which should describe the file.

3.1.7 AddContactActivity

In this activity a new contact can be defined. All that is needed is a DeviceID and its associated name. The typing of the DeviceID is quite error-prone, that is why FileShipping also offers the opportunity to add contacts from a queries list of results, where the source's DeviceID is shown.

3.1.8 ResultActivity

This activity can be started from the result of a query. Here, the user can decide to collect the file from multiple *Servers* at once as Partfiles. Also the user has the opportunity to add a new contact by long-clicking on one of the *Servers* in the attached *Serverlist*.

3.2 Database

The database is mainly designed to store whole objects which are required by GUI and protocol. Both GUI and protocol can read and write seperately in the database. This is the only way for the protocol to have an effect on the GUI.

3.2.1 ER-Model

Since we create queries and store them in query-objects which are also required to display the QueryFragment, the database should have a query-table. The same thoughts for each list in the GUI lead to this ER-Model (3.4). For more efficient storage handling, the file-table and query-table share the same tag-table. Since many files can be tagged to describe their usecase, for example as video or picture, also many queries will use the same tags and it would be pointless to have the same tag twice in seperated tables. The downside of this is that the deletion of a query or file and their tags is difficult, because the cost of the calculation whether the other table is still using the tag is quite high. This is why tags are never deleted from the database, only their references in “queryHasTag“ and “fileHasTag“ are removed. This is not memory efficient, but could be used for a future feature (see section 6.4). The fileresult-table is kind of special, because it is neither used by the GUI nor by the protocol. Its usecase is to collect information about Partfiles and to find out if all parts for a certain file have been collected and so the whole file can be rebuild.

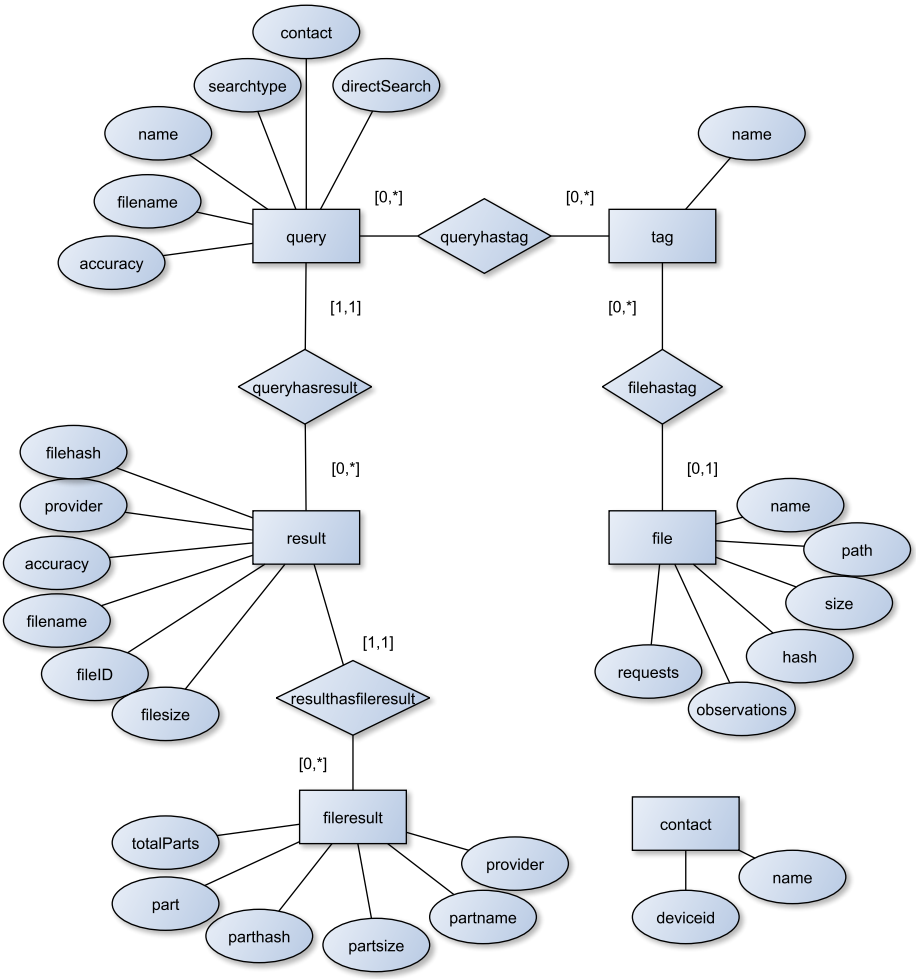

Figure 3.4: ER-Model

3.2.2 Relational Model

The ER-Model in figure 3.4 shows that there are possible merges of the tables “query“ and “result“ and of the tables “result“ and “fileresult“. The Relational-Model 3.1 is the result of the merge of those tables.

The tables “query“ and “result“ can be merged, as each instance of the Result-Object must be a result for one Query-Object. The tables “result“ and “fileresult“ can be merged, since a resulting file can have multiple parts, but each part can only be assigned to a single result. The tables “queryHasTag“ and “fileHasTag“ are used to connect the tables “query“ and “tag“ and accordingly “file“ and “tag“.

3.3 Protocol

The protocol used by FileShipping is primarily inspired by HTTP and has been supplemented with the functionality to support requests with the usage of tags.

3.3.1 Connection to Opptain

Opptain uses bundles to pass messages between devices. These bundles are seperated into IncomingBundles which are constructed by Opptain and handed to FileShipping, and OutgoingBundles which are constructed by FileShipping and handed to Opptain. The following sections show the contents of these bundles.

OutgoingBundle

BundleID - The BundleID is an identifier for the bundle which is set by FileShipping. When Opptain receives a bundle with the same BundleID of an already existing valid bundle, Opptain assumes these bundles to be identical and discards the new

Table	Primary Key	Foreign Keys	Attributes	Datatype
query	id		name contact filename accuracy directsearch searchtype	String String String Integer String String
tag	id		name	String
file	id		name path size hash requests observations	String String Integer String Integer Integer
result	id	queryid	provider accuracy filename fileid filesize filehash	String Integer Integer String Integer Integer String
contact	id		name deviceid	String String
fileresult	id	resultid	provider partname partsize parthash part totalParts	String Integer String Integer String Integer Integer
queryhastag	id	queryid tagid		Integer Integer
filehastag	id	fileid tagid		Integer Integer

Table 3.1: Relational Model

one. FileShipping BundleIDs are set by the hash of origin's DeviceID concatenated with the message itself.

DeviceID - The DeviceID is a byte array which is used to address a destination device. By Opptain's convention the broadcast address given when all bits are set to 1.

TTL - The **T**ime **t**o **L**ive long value specifies the duration of the validity of the bundle for Opptain. When a bundle's TTL runs out, Opptain will discard the bundle. This has no effect for FileShipping, since responses should have a different BundleIDs than their requests.

ApplicationPackage - The ApplicationPackage is used by Opptain to identify the targeted application. It is set to FileShipping, so Opptain can hand an Incoming-Bundle back to the FileShipping app at the target device.

PayloadData - The PayloadData is a freely accessible byte array. This field is used by FileShipping to transport Message-Objects. The destination instance of FileShipping can reconstruct the message from this field and identify its purpose.

PayloadFile - The PayloadFile is a field that can hold any file, that should be transferred between two devices. In case the bundle contains a SENDF message the attached file is packed in this field. Otherwise this field contains an empty file.

PayloadInfo - The PayloadInfo contains fields for information about the file stored in PayloadFile. It contains the file's name, the file's size and its hash.

AdditionalInformation - The AdditionalInformation is a field used by Opptain to collect information about the state of the network.

Since only the Payload fields can be written freely by FileShipping and the PayloadFile and PayloadInfo are defined by the file which is carried in this bundle, the whole protocol has to be managed within PayloadData. The finished bundle is handed to Opptain via an Intent.

```
1   public static void sendOutgoingBundle(Context context,
2      OutgoingBundle outgoingBundle) {
3       Intent intent = new Intent()
4               .setAction(IntentConstants.Action.BUNDLE)
5               .addCategory(IntentConstants.Category.OPPTAIN)
6               .setPackage(IntentConstants.Package.OPPTAIN)
7               .addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES)
8               .putExtra(IntentConstants.Extra.BUNDLE_CONTENT,
9                       outgoingBundle);
10      context.startService(intent);
11  }
```

Figure 3.5: A bundle sent to Opptain by FileShipping

IncomingBundle

The IncomingBundle is similar to the OutgoingBundle. It contains the origin's DeviceID, the BundleID, the Payload fields and AdditionalInformation.

In the following the intended protocol and its keywords shall be explained.

3.3.2 Intended Protocol

The first idea of the protocol was to have a HTTP like protocol for both, searches by Filename and searches by TagList. Also it should be possible to stop the running query at any point of the protocol. The protocol does not use any kind of acknowledgement message, because through the characteristics of the Opportunistic Network, even if a message has been sent successfully, the following acknowledgement might not be able to reach the origin device. This idea contained the following keywords.

HEAD - Used to receive a list with meta information about files that fulfill the requirements related to the TagList and the accuracy value which are attached to the message.

GETF - Used to request a specific file by its filename. For this message the *Client*

needs to know the exact file including its extension.

GETR - The functionality of GETR is the same as the HEAD request, but instead of returning a list with metainformation, the *Server* will reply with all files which match the requirements, each wrapped in its own SENDF response.

SENDF - This is a response to a GETF or GETR message. It contains a single attached file which has to be returned to the origin. This message can only be used by a *Server* in a communication cycle.

SENDH - This is the proper response to a HEAD message. It does not contain a file, but carries a list of metainformation which match the requirements of the foregone HEAD message.

TBNT - When the origin has collected some metainformation about files that match its requirements, this message can be send to all *Server* devices which files will not be requested by the origin.

ABORT - In case the origin has send a GETR or GETF request and does not need the responses anymore, this message may be sent to stop the provider from responding.

3.3.3 Final Version of the Protocol

The final version of the protocol is used as described before, but does not use the TBNT and ABORT messages, because of the following reasons.

TBNT

Since the protocol has to be used in an Opportunistic Network, it has to deal with the situation of a device randomly disappearing. Because of this there is no use for the provider to store any information about a device that has been served, as it might disappear anyway. The TBNT message was meant to notify the provider that its service is not needed,

but since the provider does not store any information, there is no reason to send a TBNT message.

ABORT

The ABORT message was meant to stop the transmission of a file to a destination device. But because of the Opportunistic Network, the whole file has to be sent at once. If a file was divided into multiple pieces, it might occur that a requesting device has received a few parts of the whole file and keeps waiting for the missing parts that may never appear. When a file is sent to the request's origin, the instance of FileShipping does not know about its arrival until it is notified by Opptain. This message contains the whole file, so it is too late to abort the transmission.

3.3.4 Usage

In the Opptain-Bundle the fields PayloadData, PayloadFile and PayloadInfo can be used by FileShipping. The fields PayloadFile and PayloadInfo will only be set when the bundle carries a response file, otherwise it contains a dummy file without content. The PayloadData always carries a Message-Object, which then contains a specific message, relating the request type it is used for. The protocol can be used as follows:

HEAD - In case of a HEAD request, PayloadData contains a HEADMessage-Object within its message. The HEADMessage holds the origin's queryID, the expected accuracy of the result and an attached TagList. PayloadFile and PayloadInfo are set by the dummy file. AddQueryActivity can construct a HEADMessage when packing an OutgoingBundle which is shown in listing 3.1.

GETF - The GETFMessage-Object contains besides the queryID and the filename, also a resultID, a part value and totalParts value. When the user is sending a GETF request from the AddQueryActivity, the activity constructs the GETFMessage, the resultID is set to -1, part und totalParts have the value 1. That is because FileShipping has no metainformation collected from the *Servers* yet, so the returned

Listing 3.1: AddQueryActivity constructs a message

```
1  private OutgoingBundle createOutgoingBundle(
2                          @NonNull Query query,
3                          DeviceId dest) {
4      Long ttl = 10L * 60 * 1000;
5      String applicationPackage = getApplicationContext()
6                  .getPackageName();
7
8      de.opptain.fileshipping.model.message.Message message = null;
9
10     if (query.getDirectSearch().equals("INDIRECT") &&
11                 query.getSearchType().equals("TAG")) {
12         message = new Message(new HEADMessage(
13                         query.getID(),
14                         query.getAccuracy(),
15                         query.getTagList()));
16     } else if (query.getDirectSearch().equals("DIRECT") &&
17                 query.getSearchType().equals("FILENAME")) {
18         message = new Message(new GETFMessage(
19                         query.getID(),-1,
20                         query.getFilename(),
21                         1,1));
22     } else if (query.getDirectSearch().equals("DIRECT") &&
23                 query.getSearchType().equals("TAG")) {
24         message = new Message(new GETRMessage(
25                         query.getID(),
26                         query.getAccuracy(),
27                         query.getTagList()));
28     }
29     File file = new File(getExternalFilesDir(null), "nofile");
30
31     PayloadData mPayloadData = new PayloadData(message.toByteArray());
32     PayloadFile mPayloadFile = new PayloadFile(file);
33     PayloadInfo mPayloadInfo = new PayloadInfo(file);
34     AdditionalInformation additionalInformation =
35                 new AdditionalInformation();
36     BundleId bundleID = new BundleId(
37                 NetworkUtils.concatenateByteArrays(
38                         FileShipping.sDeviceId.toBytes(),
39                         message.toByteArray()));
40
41     return new OutgoingBundle(
42                 bundleID, dest, ttl,
43                 applicationPackage,
44                 mPayloadData,
45                 mPayloadFile, mPayloadInfo,
46                 additionalInformation);
47     }
```

file cannot be assigned to a result. Also FileShipping does not know if any or how many *Servers* offer this specific file. So when a *Server* has a matching file, the complete file is returned. In other situations the resultID, part and totalParts can have different values.

GETR - The GETR-Message is constructed like the HEAD-Message. The response by the *Server* has to be a single complete file, which is matching the attached TagList.

SENDF - SENDF messages contain a SENDFMessage-Object, which holds the receiver's queryID, because the device needs to be able to identify the corresponding query. Also it contains the attached file's name and the reached accuracy regarding the TagList in the foregone request. In case the SENDF response is returned to a GETF request, the accuracy is set to 100. In case the foregone message has been a GETR request, or the GETF request has been sent by the *Client* without collecting metainformation first, the resultID will be set to -1, and the part and totalParts to 1. In case the SENDF transports the part of a file, the part value identifies which part is attached in this response and the totalParts value identifies in how many parts the file has been divided. These files or partfiles are set in the PayloadFile field, PayloadInfo then contains the file's metainfo.

SENDH - The SENDHMessage is the response to a HEAD request, it contains the origin's queryID and a list of files, taken from the *Servers* database, which match the TagList from the HEAD request. PayloadInfo and PayloadFile are set by the dummy file.

From the description it would be expected that SENDF is a necessary response to a GETF or a GETR request and SENDH messages are appropriate responses to HEAD requests. But since the *Client's* and *Server's* device never build a connection and the *Client* never awaits a response, these messages could also be considered one-way communications. It is more like the *Client* is telling the network that it is looking for a file that is matching either its filename or the TagList. The *Servers* are responding with their files when they have what the *Client* is looking for or in case of the HEAD request with their files' metainfo. Because of this at least HEAD and GETR requests are meant to be broadcasted, while only GETF, SENDF and SENDH should address a single device.

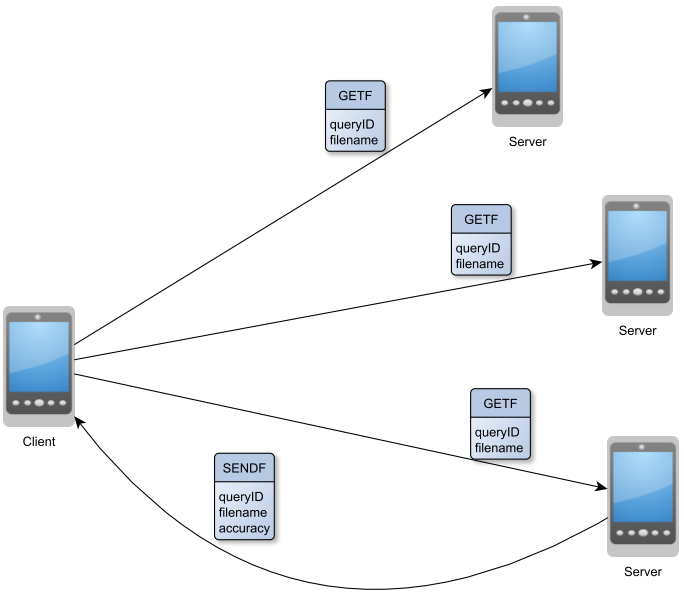

Figure 3.6: GETF Communication as expected by FileShipping

Despite this, FileShipping supports the addressing of GETR and HEAD requests to a single target and GETF, SENDF and SENDH could technically also be broadcasted.

3.3.5 Processing Bundles

As already mentioned, the bundles are passed to or received by Opptain. This is done by sending Broadcast-Intents which then start the corresponding application services and notify them that Opptain has a bundle concerning it. The sending of an OutgoingBundle is already shown in Figure 3.3.1 and the construction of that bundle is described in 3.3.4. In this section shall be shown, how IncomingBundles are processed.

ApplicationService

When Opptain sends the Intent, it starts FileShippings' ApplicationService, which then calls the method processBundleNotification() (Figure 3.2), which then tries to get the

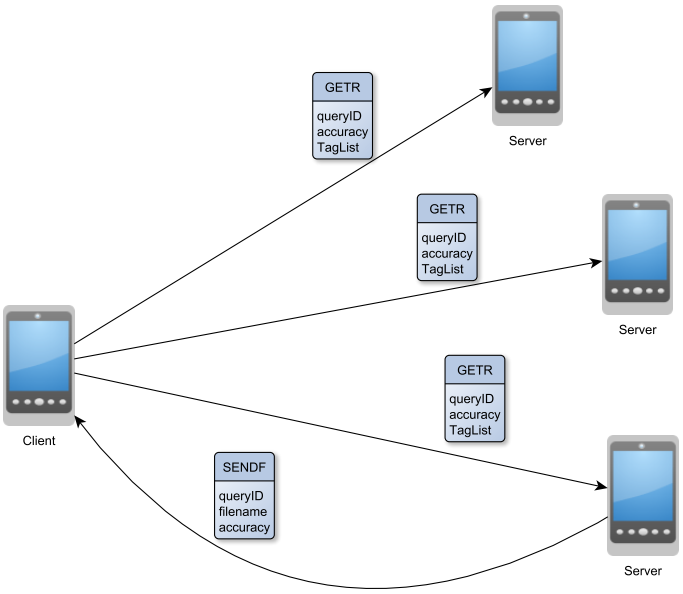

Figure 3.7: GETR Communication as expected by FileShipping

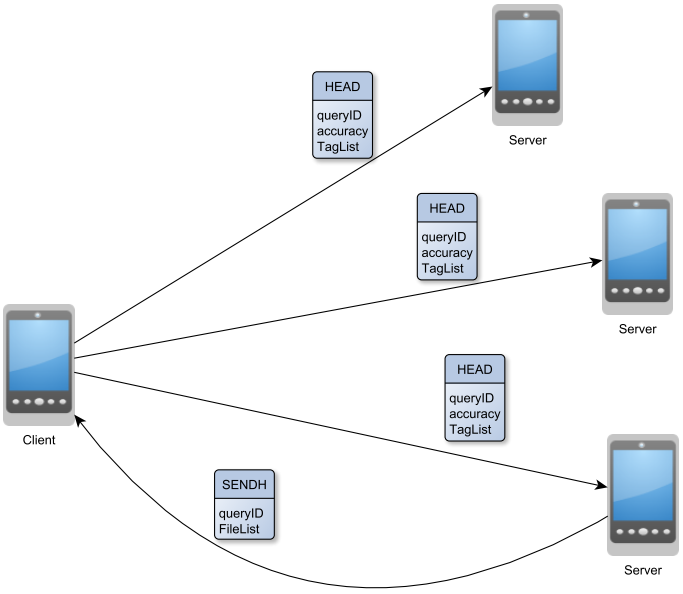

Figure 3.8: HEAD Communication as expected by FileShipping

bundle from Opptain. If it succeeds a new AsyncTask is started, which actually processes the given bundle.

The AsyncTask is necessary, because the ApplicationService would be unreachable while it is processing the IncomingBundle. That means that any Intent sent by Opptain during the procession of FileShipping's ApplicationService would get lost. But with the AsyncTask the ApplicationService can wait for new IncomingBundles.
To process the IncomingBundle, the AsyncTask calls the method handleIncomingBundle() provided by NetworkUtils. NetworkUtils takes the message from PayloadData and calls the process-method which is fitting to the requesttype taken from the message.

The methods processHEADRequest(), processGETFRequest() and processGETRRequest() are similar, since these keywords are always used to search for files. When these methods are called, the database is checked for entries that match the requirements. In case there are matches, an OutgoingBundle is constructed with a proper response message.
The method processSENDHRequest() adds a list of results, taken from the bundle to the database. This list will be shown when the user takes a look on the detailed view of the corresponding query.
ProcessSENDFRequest() tries to write the file taken from PayloadFile to the external storage and adds an relating entry to the fileresult-table in the database.
Since TBNT and ABORT are not used in the final version of the protocol their methods are still implemented, but their methodbodies are empty.
ProcessHEADRequest()-Method should be the mostly used method by FileShipping when it comes to the exchange of bundles. So we should take a more detailed look at it.

processHEADRequest()

The main purpose of this method is it to look up, if the *Server's* file-table contains any files which match the requested TagList for the noted accuracy. To do so the first step

Listing 3.2: The ApplicationService processes a received Intent

```
1  private void processBundleNotification(Intent intent) {
2      ServiceConnection serviceConnection = new ServiceConnection() {
3          public void onServiceConnected(
4                                  ComponentName name,
5                                  IBinder service) {
6              try {
7                  IOpptainRemoteService remoteService =
8                          IOpptainRemoteService.Stub
9                                  .asInterface(service);
10                 IncomingBundle bundle = remoteService
11                         .getBundle(bundleId);
12                 new AsyncTask<IncomingBundle, Void, Void>() {
13                     @Override
14                     protected Void doInBackground(
15                         IncomingBundle... params) {
16                         Log.d(ApplicationService.class.getSimpleName(),
17                                 "processBundleNotification:
18                                 handleIncomingBundle");
19                         NetworkUtils.handleIncomingBundle(params[0]);
20                         return null;
21                     }
22                 }.execute(bundle);
23             } catch (RemoteException e) { (...) }
24                           finally {
25                 FileShipping.sApplicationContext
26                                             .unbindService(this);
27             }
28         }
29
30         public void onServiceDisconnected(ComponentName name) { (...) }
31     };
32
33     Intent bindServiceIntent = new Intent().
34                 setClassName(IntentConstants.Category.OPPTAIN,
35                 IntentConstants.Class.OPPTAIN_REMOTE_SERVICE);
36     boolean success = bindService(
37                 bindServiceIntent, serviceConnection,
38                 Context.BIND_AUTO_CREATE);
39     if (success) { (...) }
40     else { unbindService(serviceConnection); }
41 }
```

is to retrieve the data from the IncomingBundle 3.3. PayloadData contains a Message-Object which is reconstructed first. This message contains a "Submessage". In this case it is a type of HEADMessage.

Listing 3.3: Retrieve information from IncomingBundle

```
1   private static void processHEADRequest(IncomingBundle bundle) {
2       HEADMessage message = Message.createFromByteArray(
3                             bundle.getPayloadData()
4                             .getPayloadData())
5                             .getHeadMessage();
6       Integer originQueryID = message.getQueryID();
7       Integer accuracy = message.getQueryAccuracy();
8       DatabaseHelper helper = new DatabaseHelper(
9                             FileShipping
10                            .sApplicationContext);
11      SQLiteDatabase db = helper.openDatabase();
12
13      TagList tagList = message.getTagList();
14      Integer leastNumberOfMatches =
15                            (accuracy * tagList.size()) / 100;
16
17  [...]
```

The HEADMessage contains the origin's queryID. FileShipping needs this information, because the response that will be constructed here needs to inform the origin which of its queries it is actually answering. The desired accuracy of the results is also saved in the HEADMessage. Then the database is opened, it will be needed soon. The attached TagList is taken from the HEADMessage. For the further handling it is advantageous to have them organized in a TagList-Object, rather than having them as single tags. The TagList is in principle an ArrayList<Tag>-Object, which it is also extending. The Tag-Objects within this list hold the tags' names as a string value, and also their corresponding ids, but those are set to the entries within the origin's database, so these cannot be used by the *Server*. With the accuracy, which is given as a percentage, and the size of the required TagList can be calculated how many tags from the TagList have to be matched by a file to be an actual match regarding the query.
In the next step FileShipping looks up which files in its database match the TagList (Figure 3.4).

The files, which match the TagList and also their accuracy, are going to be hold in an

Listing 3.4: Look up matching files

```
1        [...]
2    ArrayList<FileMatch> matches = new ArrayList<>();
3    matches.addAll(FileHasTagContract.getFileMatchesByTagList(
4                        db,
5                        tagList,
6                        leastNumberOfMatches));
7
8    if (matches.size() == 0) {
9        helper.closeDatabase();
10       return;
11   }
12   [...]
```

Listing 3.5: SQL Statement to retrieve matching files from database

```
1
2 SELECT fileID , Count( tagID ) AS matches FROM fileHasTag
3 WHERE tagID IN (
4     SELECT id FROM tag
5     WHERE name IN (
6         ?,[...],?))
7 GROUP BY fileID
8 HAVING Count( tagID ) >= leastNumberOfMatches;
```

ArrayList<FileMatch>, the FileMatch-Object just holds the id of the matching file and its accuracy regarding the TagList. To retrieve this list, the database connection, the TagList and the least number of matches regarding the tags is given to FileHasTagContract. The FileHasTagTable maps fileIDs to corresponding tagIDs. The SQL Statement executed by the database is shown in listing 3.5.

All questionmarks in the statement are placeholders for tagnames. There will be as many questionmarks as there are tags within the TagList. The statement shows, that the database has to look up the tag's id for each tag first, then maps the fileIDs to a set of tagIDs, which have to be in the foregone list. Then it removes each entry in which the set has less then the least number of matches.
With the results of this query, FileMatch-Objects are constructed and added to the match-list. In case this list is empty, the database can be closed and the processing is finished. There is no need to notify the origin about the *Server* having no matching files, since the origin is not waiting for a response.

In the third step, FileShipping looks up all files, which are noted in the matches-list and adds them to a FileList-Object (listing 3.6).

Listing 3.6: The file for each match is being looked up

```
1        [...]
2    FileList files = new FileList();
3    File currentFile;
4    for (FileMatch match : matches) {
5        currentFile = FileContract.getFileByID(db, match.getFileID());
6        files.add(currentFile);
7        FileContract.updateObservations(db, currentFile);
8    }
9
10   helper.closeDatabase();
11   [...]
```

FileList is an object that extends ArrayList<File> and contains FileShipping's File-Objects. These are not actual File-Objects, but rather objects which hold references to these, their hashes and other metainformations. For each file the observation counter is being incremented. When this is done, the database connection can be closed, since it is not required anymore.
The last step is the construction of the OutgoingBundle and its handing back to Opptain (see listing 3.7). For this purpose a new message is being created. The processHEADRequest-Method handles requests for metainformation, that is why the response contains a SENDHMessage. The SENDHMessage itself holds the origin's queryID, and the FileList from before.

In the following the OutgoingBundle is constructed. PayloadData contains the message written as an byte-Array, that has just been created before. PayloadFile and PayloadInfo are set by a dummy file, since the response does not actually carry a file. The new OutgoingBundle is then send to Opptain. The origin's DeviceID can be taken from the IncomingBundle itself. ApplicationPackage is set up with FileShipping's packagename, this is necessary, since the Opptain application of the receiving device needs to know which app the bundle has to be assigned to. The BundleID is set as the hash value over the concatenation of the origin's DeviceID and the message as a byte-Array.

Listing 3.7: The response is constructed and handed to Opptain

```
1
2         [...]
3         Message response = new Message(
4                                 new SENDHMessage(
5                                 originQueryID,
6                                 files));
7
8     PayloadData mPayloadData = new PayloadData(response.toByteArray());
9     DeviceId deviceId = bundle.getOrigin();
10    long ttl = 10L * 60 * 100;
11    String applicationPackage = FileShipping
12                        .sApplicationContext
13                        .getPackageName();
14    java.io.File file = new java.io.File(FileShipping
15                        .sApplicationContext
16                        .getExternalFilesDir(null),
17                        "nofile");
18    PayloadFile mPayloadFile = new PayloadFile(file);
19    PayloadInfo mPayloadInfo = new PayloadInfo(file);
20    AdditionalInformation additionalInformation =
21                        new AdditionalInformation();
22    BundleId bundleID = new BundleId(mPayloadInfo.getPayloadHash());
23    sendOutgoingBundle(FileShipping.sApplicationContext,
24                        new OutgoingBundle(bundleID,
25                        deviceId,
26                        ttl,
27                        applicationPackage,
28                        mPayloadData,
29                        mPayloadFile,
30                        mPayloadInfo,
31                        additionalInformation));
32    }
```

3.3.6 Accuracy Value

In the forgone sections the accuracy value has already been mentioned. The accuracy is a percantage between 0 and 100 that can be set up by the user when he is creating a new query with. These queries must either be HEAD or GETR requests. The accuracy decides how many of the tags in the TagList have to be matched. With the accuracy and the TagList the *Server* can calculate the "leastNumberOfMatches". To get the leastNumberOfMatches, the number of tags in the TagList have to be multiplied with the accuracy, the result is then divided by 100 and casted to Integer. For example, let's assume that

a *Client* is sending a HEAD request with an attached TagList with the size of 4. The accuracy is set to the default value of 75 percent. From the calculation we will see that the least number of matches is 3. In the listing 3.5 we can see that all files which have at least one tag from the TagList are grouped by their fileID and mapped to the number of the matched tags. With the "HAVING Count (tagID) > leastNumberOfMatches" statement only fileIDs that match the requirements are returned from the database.

3.3.7 Partfile

When using FileShipping, many users will collect meta information about possible results before the actual files is requested. By sending the HEAD request, FileShipping collects these information from nearby *Servers*, including the *Server's* DeviceId and the file's hash. FileShipping assumes that files with the same hash value are identical. With the knowledge of the number of identical files and their *Servers*, the idea of the Partfile suggests itself. The idea and the usage of the Partfile is that instead of requesting a single *Server* for a specific file, we could request all known *Servers* of the file for a part of the file. When all Partfiles are collected, the original file can be reconstructed from these. FileShipping provides the collecting of these Partfiles, whenever a HEAD request is answered. The user can select a specific result from the list. The called ResultActivity will show which device offers the same file. The user may then decide to start the collection of these parts.

Chapter 4

Evaluation

For development and testing purposes we have been provided with four Android Smartphones which are listed in the table 4.1. For the development of internal structures of FileShipping we mainly used the Y360 devices.

4.1 GUI

The illustration of the GUI often requires multiple accesses to the database. Since these accesses are slow and keep blocking the UI-Thread, they should be delegated to AsyncTasks. When a new activity is created, the UI-Thread starts the AsyncTask to get the required information from the database. For the AddFileActivity the Android built-in FileChooser is started as an “activityForResult“. Activities of this type are started with an Intent and are meant to do a single task. In this case its the picking of a file. When a file has been chosen the activity gets destroyed and the foregone activity is resumed. The

Name	Model	Manufacturer	Version
Galaxy Nexus	I9250	Samsung	4.3
HUAWEI Y3	Y360 - U61	HUAWEI	4.4.2
HUAWEI Y3	Y360 - U61	HUAWEI	4.4.2
Galaxy S5	SM-G900F	Samsung	6.0.1

Table 4.1: Testing devices

calling activity then gets a resulting Intent that contains the results from the activityForResult. Since the structure of FileShipping is developed and mainly tested with the Y360 devices, this mechanism uses Intents which support these devices. During the testing we got to know that these Intents cause exceptions on the Galaxy S5 device which are not handled yet.

4.2 Exchanging Bundles

4.2.1 NullPointerExceptions with the Galaxy Nexus

During the testing of the protocol, we noticed that exchange of bundles causes NullPointerExceptions on the destination device when the Galaxy Nexus is involved. To be sure, we have tested this event in different situations.
In the first test, the Galaxy Nexus had Opptain's ServerTask running, while the Y360 devices took the slave roles. Whenever a bundle has been passed from one of the slaves to the Galaxy Nexus, its ApplicationService reacted, but threw a NullPointerException when it tried to retrieve the IncomingBundle from Opptain.
For the second test we had the same setting as before, but this time the bundles had been passed from the Galaxy Nexus to the slaves. The result was that the targeted device's ApplicationService started, but threw a NullPointerException while trying to retrieve the bundle from Opptain.
To be sure that this problem did not occure because the Galaxy Nexus was running the ServerTask, we switched the roles of the devices. This made one of the Y360 running the ServerTask while the Galaxy Nexus took the role of the slave. Then we did the same tests as before and got the same results. When the Galaxy Nexus sent the bundle, the FileShipping instance of the HUAWEI device threw the NullPointerException. When the Galaxy Nexus received the bundle, its own FileShipping ApplicationService threw the exception.

4.2.2 ServiceConnectionLeaks

Since FileShipping was unstable on the Galaxy S5 and the Galaxy Nexus kept causing NullPointerExceptions whenever a bundle had been exchanged, we were left to the HUAWEI Y3 devices for further testing. To differentiate them devices we will refer to one of these as “W“ and to the other one as “X“.
During the tests for the protocol, another issue occured. As already mentioned before, Opptain notifies FileShipping via an Intent, when Opptain has received a bundle which is assigned to FileShipping. This Intent starts FileShipping's ApplicationService which then tries to connect to Opptain's ApplicationService and then request the IncomingBundle from it. Sometimes this ServiceConnection seems to fail for an unknown reason, and the connection is “leaked“. A detailed description of the situation and the meaning of the leaking of a ServiceConnection is done in section 5.3. These ServerConnectionLeaks seem to occure uninfluenced by the type of the transported message, that is why we observed the reaction of the ApplicationServices with the use of HEAD requests. The implementation of the processHEADRequest-Method forces the receiving device to respond with a SENDH message, if the HEAD request matches any files.
In the following test *W* took over the Opptain ServerTask while *X* was its slave. *W's* instance of FileShipping was set up with a requestable file. *X's* then started to send the exact same request multiple times. The TagList in *X's* requests was set up to match *W's* file, so *W* had send a response whenever one of *X's* message was processed. Each attempt was done after the foregone bundle's TTL ran out.

Attempt 1 - Bundles were processed correctly

Attempt 2 - *W* had a ServiceConnectionLeak

Attempt 3 - *X* had a ServiceConnectionLeak

Attempt 4 - *W* had a ServiceConnectionleak

Attempt 5 - Bundles were processed correctly

Attempt 6 - Bundles were processed correctly

Attempt 7 - Bundles were processed correctly

Attempt 8 - *W* had a ServiceConnectionLeak

Attempt 9 - *X* had a ServiceConnectionLeak

Attempt 10 - Bundles were processed correctly

Since the sent requests were the same in each attempt, we can see from the results that the leaking of ServiceConnection is not influenced by the type of the message. We repeated the test with the same settings for Opptain, but this time *X* holds a file that is being requested by *W*.

Attempt 1 - Bundles were processed correctly

Attempt 2 - *W* had a ServiceConnectionLeak

Attempt 3 - *W* had a ServiceConnectionLeak

Attempt 4 - Bundles were processed correctly

Attempt 5 - *X* had a ServiceConnectionLeak

Attempt 6 - Bundles were processed correctly

Attempt 7 - Bundles were processed correctly

Attempt 8 - Bundles were processed correctly

Attempt 9 - *W* had a ServiceConnectionLeak

Attempt 10 - *W* had a ServiceConnectionLeak

During the first test cycle 17 messages were sent of which 5 messages caused a Service-ConnectionLeak, this makes a success ratio of about 71%. During the second test cycle 14 of 19 messages were successfully processed by the ApplicationService, this makes a

success ratio of about 74%.
These results lead to a changing in the implementation of the ApplicationService. In the first implementation the ApplicationService started an AsyncTask which would then try to build the connection to Opptain's remote service. From the ServiceConnection another AsyncTask is started which does the actual processing of the IncomingBundle.
After the changes the ApplicationService builds up the ServiceConnection on its own. The ServiceConnection would the proceed as before.
We did a similar test after these changes. In this test another 100 messages were exchanged, of which 90 didn't cause a ServiceConnectionLeak. However the ApplicationService now causes StrictMode violations (For an explanation see section 2.4.4).

4.3 Protocol

4.3.1 HEAD and SENDH

During the testing for the ServiceConnection leaks some cases of the HEAD and SENDH messages have been tested automatically. These tests have been successful besides the ServerConnection leaks, but as we already know, these are not due to the protocol. For the testing of the other messages we have built similar test cases. For these tests we used the HUAWEI devices *W* and *X*. Since we wanted to collect responses before, we haven't tested yet how the *Server* reacts, if he cannot reply to the request, yet. For this case we had set up a file with a certain TagList on *W*. *X* sent a request with an attached TagList which's entries are completly different then these on *W's* file. With the logs we could see that *W* started the processing of the HEAD request, but since no matches could be found the processing stopped and no message was returned. For the next test we prepared the same settings like before, but this time the file's and the query's TagLists matched partially. As expected the *Server W* responded depending on the queries accuracy value. In this case the file had a single tag named 'a' and the querie's TagList contained the tags 'a' and 'b'. When the accuracy value was set to 100, *W* did not respond, with any value below 100 the *Client* received a SENDH message with the meta information about the file. Going from this we tested another special case. The *Server's* database contained

multiple files with different TagLists. The query sent from *X* contained a single tag and the accuracy value was set to 0. Theoretically it could be expected that each file from *W's* database is mentioned in the response, but due to the SQL-Statement (see listing 3.5) each file from the list of matches has to contain at least one tag from the queries TagList. In the test case the *Client* received a response that contained all files that had the queries tag within their TagList.

4.3.2 GETR and SENDF

The same test cases as for the HEAD request had been used for the GETR request. For these cases the processGETRRequest()-method on the *Server*side had been succesful. If the *Server* had a matching file a SENDF message was returned to the *Client*.

4.3.3 GETF and SENDF

For the testing of the GETF message the *Server* was set up with one file. In the first test the *Client* requested the exact name of the *Server's* file. As result the *Client* received the expected SENDF response. In the second test the *Client* requested a different file that was not existing. In the logging we could see that the *Server* tried to look up the filename in its database, but as he couldn't find any matches, he stopped the processing and returned no response.

4.3.4 Partfiles

The support of Partfiles is one of FileShipping's unique and most important features. Unfortunately, because of the lack of time we cannot perform a satisfying set of tests regarding the Partfiles, but we look forward to do so in the future.

Chapter 5

Issues

FileShipping in its current state has to deal with some more or less critical issues. First of all, there is the trust between *Client* and *Server*. Since the transmission protocol is stateless and does not involve a permanent connection, the *Client* always has to add enough information to a request, so the *Server* can create a response that is identifiable and uniquely assignable to the *Client's* request. In this case it is the *Client's* queryID, or the attached TagList. The other way around, the *Server* has to reveal things like fileIDs in its database.

5.1 Client-side: FileFishing

The main part of the protocol requires TagLists. When a *Server* receives such a TagList, the ApplicationService simply answers with any file which's TagList contains at least one of the tags from the request. In this case it does not matter if the *Server's* file may contain more tags which are not mentioned in the *Client's* request. If the *Client* decides to send an empty TagList, any file will match it. FileShippings GUI checks the input given by the user and does not allow TagLists without any entries; the protocol does not check this yet.
If the *Client* cannot send empty TagLists anymore, he might then send some with a random tag and then set the preferred accuracy of the response to zero. This way an

empty TagList could be simulated. To prevent this, a default minimum accuracy could be implemented.
Another attempt from the *Clientside* then could be to attach a TagList with a common tag and set a low accuracy to create a search space as huge as possible.

5.2 Server-side: FileSpreading

Assuming we have a *Server* that wishes to spread unwanted or even malicious data, FileShipping in its current state offers a lot of potential. The *Client* always has to set a lot of trust into the *Server* about the reliability of its offered files since the *Server* can pick any tag for each file. If the *Server* decides to attach an unmatching TagList, it can do so without any other influences. For the database or the protocol it is quite difficult to prevent a malicious *Server* from spreading its files without also diminishing the action options of reliable users. A possible solution could be the implementation of a blacklist, which is discussed in section 6.1.

5.3 Service connection leaks in ApplicationService

Currently the ApplicationService of a receiving instance of FileShipping sometimes throws an exception on the ServiceConnection. The corresponding codefragment has already been shown in listing 3.2.
Whenever Opptain receives a bundle which is assigned to FileShipping, Opptain broadcasts an intent with a “Bundle-Notification“. This intent starts FileShipping’s ApplicationService with this notification. FileShipping’s ApplicationService will then try to open a ServiceConnection to Opptain’s ApplicationService. The instance of Opptain’s ApplicationService then has to be “bound“ to FileShipping’s ApplicationService. Because of this binding, the remote service (Opptain’s ApplicationService) will be kept running until FileShipping’s ApplicationService has done its task and then be destroyed automatically. If this binding fails, the processing of the IncomingBundle won’t be started. If otherwise the binding succeeds, FileShipping’s ApplicationService will retrieve the

Listing 5.1: ServiceConnectionLeak

```
1  Service de.opptain.fileshipping.aidl.ApplicationService
2  has leaked ServiceConnection
3  de.opptain.fileshipping.aidl.ApplicationService$3@423536e8
4  that was originally bound here
5
6  android.app.ServiceConnectionLeaked:
7  Service de.opptain.fileshipping.aidl.ApplicationService
8  has leaked ServiceConnection
9  de.opptain.fileshipping.aidl.ApplicationService$3@423536e8
10 that was originally bound here
11
12 at android.app.LoadedApk$ServiceDispatcher.<init>(LoadedApk.java:1021)
13 at android.app.LoadedApk.getServiceDispatcher(LoadedApk.java:915)
```

IncomingBundle from the remote service and start its processing in an AsyncTask. After this the service will be unbound.

Apparently, when the binding of the remote service succeeds, sometimes the task of the ServiceConnection fails without calling the unbind-method. This keeps the remote service running and using resources, while the ServiceConnection is closed. Then an exception relating the “leaking“ of the ServiceConnection is thrown (shown in Figure 5.1).

Chapter 6

Future Work

FileShipping in its current state may be running, but offers a lot of space for expansions.

6.1 Blacklist

As already mentioned before, a blacklist may be a possible additional feature to FileShipping, when it comes to the handling of malicious *Servers*. But a local blacklist may be too weak, since every user potentially has to detect every malicious *Server* on their own, what is not a satisfying solution at all, because the user has to receive harmful data in the first place. The other option is a distributed blacklist, in which the devices share and broadcast their own local blacklists, but this mechanism may be used to block harmless users from the service.

6.2 Rating System

Besides the blacklist, a rating system would also be a possible solution to handle malicious peers. With a rating system, each peer would have the possibility to rate the service

of other peers. Then this rating could be used by other peers to determine, if they want to set their trust in one's service or not. The downside of this system is that the information about each peer has to be organized in a distrubuted manner, since a malicious peer could prettify his own rating at will if he was the only one who kept track of it.
Another proposal for a rating system would be one that rates single files. Then the user could decide by the file's rating, if he wants to get it from the network. FileShipping is already keeping track of the observations and requests for each file. These information are not published in the network yet, but surely they could be used for a simple rating system in a similar way.

6.3 Encryption

Opptain already provides a symmetric cryptosystem for bundles which are passed between two specific peers. Broadcasted bundles are sent in plain text. This is not satisfying from FileShipping's point of view, since most of its bundles are broadcasted. A possible solution to allow the encryption of broadcasts would be that FileShipping uses a "public" key for encryption, which is known to all FileShipping users. This way at least only other FileShipping user's could read broadcasted messages.

6.4 Tagsuggestions

Since Tags are never deleted from the database, suggestions can be shown by FileShipping when the user is entering a new Tagname. This would increase the usercomfort, because complicated tags don't have to be typed all over again.

6.5 Unique Storagedirectories for each Result

Currently FileShipping saves all received files in its own public directory on the SD-Card. If the user requests multiple files with equal names, the old version will be replaced by the new one. To prevent this, files relating different results could be stored in unique subdirectories.

6.6 Used Memory

Since FileShipping keeps collecting data in larger quantities, a tool that helps the user to keep track of the memory that is used by files collected by FileShipping could be helpful. Additionally, a tool with which the user can set a maximal amount of usable memory for FileShipping might increase the usercomfort.

6.7 Annotation

Currently FileShipping is doing all its tasks in the background. When a response to a query arrives, it is added to the database or the file is written to the memory. The usercomfort could be increased by adding annotations to FileShipping which inform the user about received results.

Bibliography

[Goo15a] GOOGLE INC.: Activity Lifecycle. http://developer.android.com/guide/components/activities/activity-lifecycle.html1, January 2015. Last checked: January 30th, 2017

[Goo15b] GOOGLE INC.: Permissions. http://developer.android.com/guide/topics/permissions/index.html, January 2015. Last checked: January 30th, 2017

[Goo15c] GOOGLE INC.: Processes and Threads. http://developer.android.com/guide/components/processes-and-threads.html, January 2015. Last checked: January 30th, 2017

[Goo15d] GOOGLE INC.: Services. http://developer.android.com/studio/intro/index.html, January 2015. Last checked: February 14th, 2017

[Goo15e] GOOGLE INC.: Services. http://developer.android.com/guide/components/services.html, January 2015. Last checked: January 30th, 2017

[I.W13] I.WOUNGANG, S.K. DHURANDHER, A. ANPALAGAN, AND A.V. VASILAKOS: Routing in Opportunistic Networks. http://link.springer.com/chapter/10.1007/978-1-4614-3514-3_6, May 2013. Last checked: February 2nd, 2017

[M. 15] M. ZINOUNE: Why is Android built on Linux Kernel? http://www.

unixmen.com/why-is-android-built-on-linux-kernel/, March 2015. Last checked February 14th, 2017

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 17..February 2017 Oliver Rohr

Diese CD enthält:

- eine pdf-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Bachelorarbeit erstellten Software FileShipping
- die Websites der verwendeten Internetquellen