Cross-Device Integration of Android Apps

Dennis Wolters[1(✉)], Jonas Kirchhoff[1], Christian Gerth[2], and Gregor Engels[1]

[1] Department of Computer Science, Paderborn University, Paderborn, Germany
{dennis.wolters,engels}@uni-paderborn.de, jonaskir@mail.uni-paderborn.de
[2] Faculty of Business Management and Social Sciences, Osnabrück University of Applied Sciences, Osnabrück, Germany
c.gerth@hs-osnabrueck.de

Abstract. Integrating apps on mobile devices into applications running on other devices is usually difficult. For instance, using a messenger on a smartphone to share a text written on a desktop computer often ends up in a cumbersome solution to transfer the text, because many applications are not designed for such scenarios. In this paper, we present an approach enabling the integration of apps running on Android devices into applications running on other devices and even other platforms. This is achieved by specifying adapters for Android apps, which map their services to a platform-independent service interface. For this purpose, we have developed a domain-specific language to ease the specification of such mappings. Our approach is applicable without the need to modify the existing Android apps providing the service. We analyzed its feasibility by implementing our approach and by specifying mappings for several popular Android apps, e.g., phone book, camera, and file explorer.

Keywords: Cross-Device · Integration · Android · Adapter · DSL

1 Introduction

Android devices like smartphones or tablets incorporate a high degree of interaction between apps. For instance, users usually have two choices when changing the profile picture in a messenger app: they can either select an existing picture from the gallery app or capture and insert a new picture with the help of the camera app. In this service-oriented interaction the messaging app is acting as the service requestor and the gallery/camera app is acting as service provider, who provides a service returning a picture. Furthermore, it is very easy for the user to add new integrable services by simply installing additional apps using an app store. However, this integration of another app's services is only available to apps on the same Android device. An integration of apps running on different devices is usually not supported, forcing users to find workarounds in order to integrate these apps to some extent [16]. For example, when users want to update their profile picture using their desktop computer with a picture taken on their smartphone, they have to transfer the picture somehow, e.g., manually or by using file synchronization tools like DropBox.

Q.Z. Sheng et al. (Eds.): ICSOC 2016, LNCS 9936, pp. 171–185, 2016.
DOI: 10.1007/978-3-319-46295-0_11

It would be highly beneficial from a user's perspective if functionality provided by apps running on mobile devices could be integrated and accessed by applications on devices like desktop computers as well. However, these apps usually do not provide a public interface that is accessible by applications on other devices, e.g., via the internet. Therefore, application developers wanting to integrate apps running on other devices are missing both integrable services and an infrastructure to access these services from their applications. But as we have described earlier, apps on Android devices already provide services that can be integrated by other apps on the same device. In fact, the Android platform encourages app developers to program their apps in a way such that they can be integrated by other apps. Thus, integrable services already exist but until now they cannot be accessed from other devices. Hence, a solution is required that enables developers to integrate those services into their application, providing users a seamless cross-device experience as envisioned by Satyanarayanan [17].

In this paper, we present an approach that enables the cross-device integration of Android apps into applications running on different devices. In order to realize this without requiring any changes to the implementation of the Android apps providing the services, our approach allows the specification of adapters which map these services to platform-independent service interfaces, e.g., a RESTful interface. Thereby, services can be integrated by applications running on other devices and on other platforms. To ease the specification of such adapters, we present an extensible domain-specific language (DSL). Using this DSL it is possible to develop adapters for existing apps with only a fraction of the complexity that native code would require.

Additionally, we have implemented a prototype of our approach, which allows to find and integrate services provided by apps on Android devices. Along with our prototype we specified adapters using our DSL for several standard Android apps like the camera, phone book, or the file explorer. Thereby, we demonstrate the capabilities of our approach as well as the expressiveness of our DSL, and enable developers to integrate these services into their own applications.

The remainder of this paper is structured as follows: In Sect. 2, we explain fundamental concepts of the Android platform needed to understand our approach. Section 3 gives an overview of our approach and in Sect. 4 we present our DSL to specify adapters, followed by Sect. 5 describing our implementation. In Sect. 6, we discuss the feasibility of our approach and its limitations. Related work is presented in Sect. 7. Finally, Sect. 8 concludes the paper and gives an outlook on future work.

2 Fundamentals of the Android Platform

Since our approach focuses on the Android platform, some basic knowledge of the Android platform is needed. In particular, we explain how Android apps can provide services for other apps on the same device.

We first have a look at the internal structure of Android apps. Android apps are written in Java and are divided into several components, where each component is an instance of the following component types: *Activity*, *Background*

Service[1], *Broadcast Receiver*, or *Content Provider*. An app may contain arbitrary many instances of each component type. An Activity represents one screen of an app, e.g., a phone book app would contain an Activity for the contact list as well as one for displaying the details of a single contact. Activities are used to interact with the user. In contrast, Background Services are used in the background, usually invisible for the user, to perform certain long-running operations like downloading a file. Broadcast Receivers can react on events, e.g., when the device is plugged into a docking station, and invoke corresponding logic, mostly by starting an Activity or a Background Service. Content Providers encapsulate access to data stores like relational databases. For instance, Android's phone book app has a Content Provider storing all contacts.

In order to start components of type Activity, Background Service, or Broadcast Receiver, Android uses a message data structure called *Intent*. Such Intents can be sent by the Android system or by app components. In case of Activities and Background Services, an Intent can explicitly specify which Activity or Background Service should be started. In addition, an Intent can contain data which is passed to the started component. For Activities, it is also possible to implicitly define which Activity should be started by specifying only the action to be performed, e.g., "Take Picture". At runtime the Android system resolves these implicit Intents by offering the user a list of apps capable of performing this action. For this purpose, developers have to declare at design time which Activities can handle certain actions. By only implicitly defining the providing app, a loose coupling between the requesting and providing app can be realized.

In addition to this unidirectional style of communication, where only one Intent is sent to the providing app, the Intent-based communication with Activities can also be bidirectional. For instance, if an Intent is used to start an Activity in order to get a certain result, e.g., a phone number from the phone book, the result is encapsulated in a second Intent and sent back to the component from where the request originated. Additionally, Intents can be used to broadcast events, which are received by Broadcast Receivers that subscribed to receive those events.

In contrast to other component types, Content Providers cannot be accessed via Intents. Instead, apps have access to predefined system functions in order to query or manipulate data items encapsulated in a Content Provider. Even though Content Providers can be used as front ends for other kinds of data stores, their whole interface is inspired by interfaces of relational databases. Furthermore, some parts of RESTful interfaces are adopted as well, like using Uniform Resource Identifiers (URIs) to identify Content Providers as well as every single data item being stored in those providers.

By default, interaction between components is limited to components within an app. However, apps can declare their components as public. By doing so, these public components become entry points into the app and other apps on the device

[1] A Background Service component is actually just called Service but it is not necessarily a service being provided for other apps. To avoid confusion, we call this component Background Service.

can integrate these components. For instance, a public Content Provider can be queried or manipulated by other apps. In the case of Activities, Background Services, or Broadcast Receivers, it means that they accept Intents sent by other apps. From a requesting app's perspective, public components are distinct units of logic, and therefore, they can be seen as services provided by the corresponding app. In this sense, Intents can be seen as Android-specific service request. Since Intents are also used to encapsulate the result that is sent back to the service requestor, it is also the Android-specific response format.

3 Cross-Device Integration of Android Apps

As explained in the previous section, services offered by Android apps can be integrated by other apps on the same device. In this section, we introduce our approach that enables the integration of apps running on Android devices into applications running on other devices. First, we present requirements for such an approach, and subsequently, we provide a general overview of our approach.

3.1 Requirements

We have identified the following requirements for an approach to enable the cross-device integration of Android apps:

- **R1 Applicable on existing apps**: The approach shall enable the cross-device integration of existing Android apps without having to alter them in any form. Thereby, it enables the reuse of a huge number of existing services, provided by currently available apps.
- **R2 Usage of platform-independent service technologies**: Intents are an Android-specific concept which is barely used outside of the Android domain. Hence, it must be possible for requestors running on different platforms to use platform-independent standards like RESTful or SOAP-based web services to access the services provided by apps on Android devices.
- **R3 Service registry and device management**: It must be possible to find and select devices offering certain services.

3.2 Overview of Our Approach

In this section, we provide an overview of our approach and explain how it addresses requirements R1 to R3. A visualization of our approach is given in Fig. 1. As stated in Requirement R1, the approach shall be applicable on existing apps and services offered by those apps shall be provided in a platform-independent manner (see Requirement R2) to requesting applications on other devices. Consequently, we chose to build adapters for existing Android apps, similar to using adapters to integrate legacy systems into service-oriented architectures [14]. In particular, we use adapter apps installed on the same device as the app providing the service. Adapter apps utilize Android's inter-app communication and map services provided by Android apps to platform-independent service technologies. For the remainder of this paper, we call the services provided by apps on Android devices *internal services*, because they are only available for apps on the same device and have an Android-specific interface. Adapter apps convert these internal services to *external services*, which have a platform-independent service interface and can therefore be integrated by applications running on other device/platforms.

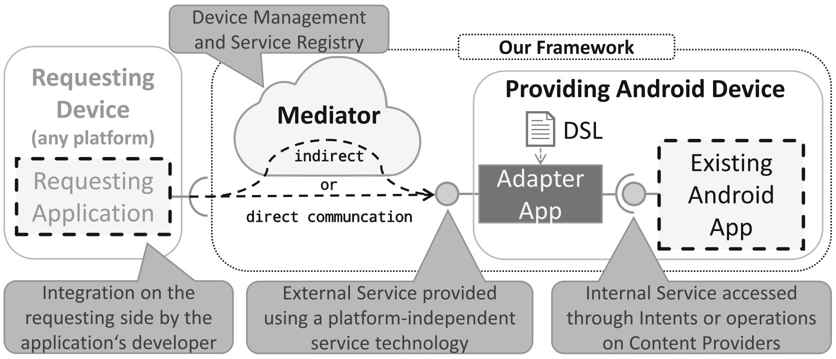
Device Management and Service Registry
Our Framework
Requesting Device (any platform)
Requesting Application
Mediator
indirect
or
direct communcation
Providing Android Device
DSL
Adapter App
Existing Android App
Integration on the requesting side by the application's developer
External Service provided using a platform-independent service technology
Internal Service accessed through Intents or operations on Content Providers

Fig. 1. General overview of our approach

Integrating the external services into the requesting application is not part of our approach. However, since we support the provisioning of external services using well-known service technologies like RESTful web services, developers of requesting applications can use well-supported, mature client implementations of those technologies to integrate these services. Additionally, if a requesting application already supports the integration of other applications via an import interface, an external service can be defined in such a way that it is directly usable by the requesting application. Alternatively, existing adapter-based approaches, as discussed in Sect. 7, can be utilized to deal with interface mismatches on the requesting side.

In addition to adapter apps, our approach includes a central entity called mediator, which addresses Requirement R3. A mediator serves as a service registry [14] and as a registry for devices. Moreover, when dealing with heterogeneous communication technologies, e.g., when a requestor wants to use HTTP and a provider is only reachable via push notification, the mediator can bridge between these different technologies. In order to deal with connectivity problems, the mediator can be used for indirect communication between requestor and provider, which includes buffering of requests and responses if either side is temporarily not available, which is common when using mobile devices.

To find devices and services, users must first register them at the mediator. If this has been done, we provide several ways to find devices providing a certain service: (i) select the provider from a list of accessible devices on the requesting device, (ii) send a broadcast to all accessible devices and claim the request on the providing device, or (iii) perform an indirect binding based on a PIN or QR code. By default, users can only see and access services offered by their own devices. However, the indirect binding also allows to use services offered by devices owned by other users.

Figure 2 visualizes how an adapter app converts an internal to an external service. Applications on other devices send a service request to an external service interface (see ❶). Details on the representation of external requests/responses are discussed in Sect. 5. An external request is received by an adapter which delegates the service request to internal services by converting the request to an equivalent Intent (see ❷). Subsequently, this Intent is sent explicitly or implicitly to an existing app on the Android device. In order to use our approach for existing apps, we must provide the data as expected by the existing app. Hence, if the app requires input data to be stored in a Content Provider, we must copy the data provided in the request to the corresponding Content Provider and must add a data reference to the Intent (see ❸). We must proceed in the similar manner if the app expects data to be provided within files. Mapping external requests to internal requests is sufficient to allow unidirectional communication with existing apps, e.g., sharing information via a messenger app.

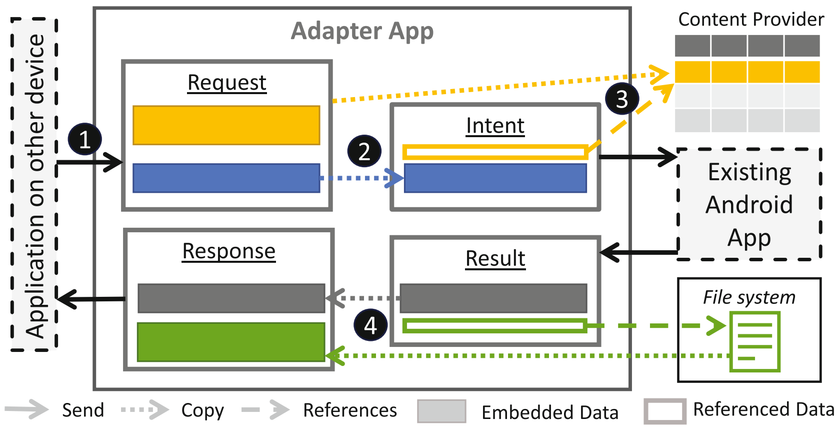

Fig. 2. Internal view of an adapter app

If the communication is bidirectional, we must map the internal result returned by the existing app to an external service response that is understood by the requestor on the other device. Thus, if the result references data in a Content Provider or as a local file, we must resolve these references and include the data in the service response (see ❹), or alternatively, create external services granting access to the corresponding Content Provider or the file system. While the former is privacy preserving since only necessary data is exchanged, the latter exposes more data than needed and is not advisable if privacy is a concern.

4 A DSL to Specify Adapter Apps

Initially, we implemented adapter apps for various common services offered by Android apps, e.g., selecting a file, taking a picture, and sharing text or images. Even though these adapters work, their source code is not easily understandable, because a lot of boiler plate code is needed, e.g., for error handling. Additionally, the Android API as well as the asynchronism of certain actions lead to the distribution of coherent code fragments across various methods and classes. Furthermore, extensive knowledge about the Android platform is needed to build these adapters. To ease the applicability of our approach, we decided to develop a DSL to specify adapters.

In the following, we present requirements as well as the definition of our DSL, which includes a discussion of the metamodel, the concrete syntax, the interpretation of our DSL, and an example mapping.

4.1 Requirements for a DSL to Specify Adapter Apps

Based on prototypical implementations of adapters for various internal services, we have identified the most common steps an adapter performs and defined them as requirements for a DSL to specify mappings performed by adapter apps:

- **D1 Specification of Intents**: The DSL must allow to specify Intents, which are sent explicitly or implicitly to other apps. This includes the specification of data embedded into an Intent.
- **D2 Specification of CRUD[2] operations on files and Content Providers**: In order to provide the data as needed for the providing app and abstract from local data stores for external requesting applications, the DSL must provide means to specify CRUD operations on the local file system and on Content Providers.
- **D3 Specification of a response**: The DSL must allow the specification of a response, which is sent back to the requestor (see Fig. 2).
- **D4 Value passing**: Data being received in a request or as part of a result shall be usable as input for other operations. For instance, a service provided by an Activity might return a reference to a data item being stored in a Content Provider. To resolve this reference, we need to use it as an input for a read operation on this Content Provider. The return value of the read operation might again be included into the response sent to the requesting application.

4.2 Language Definition

Based on the Requirements D1 to D4, we developed the DSL presented in this subsection. We start by explaining the metamodel and how mappings are being interpreted, followed by an explanation of the concrete syntax.

[2] Create, Read, Update, and Delete.

Metamodel. Figure 3 shows the metamodel of our DSL as an UML class diagram. A mapping document can describe multiple external services by defining mappings to one or more internal services offered by existing Android apps. Each external service specified in the mapping document has a unique name. A mapping from an external to an internal service consists of an ordered sequence of instructions. Based on the requirements D1 to D3 we have defined instructions for the most common steps which need to be performed during a mapping. The attributes of these instructions can be set to values provided with the request or values created by other instructions, e.g., the result of a bidirectional Intent. Additionally, constant values can be used.

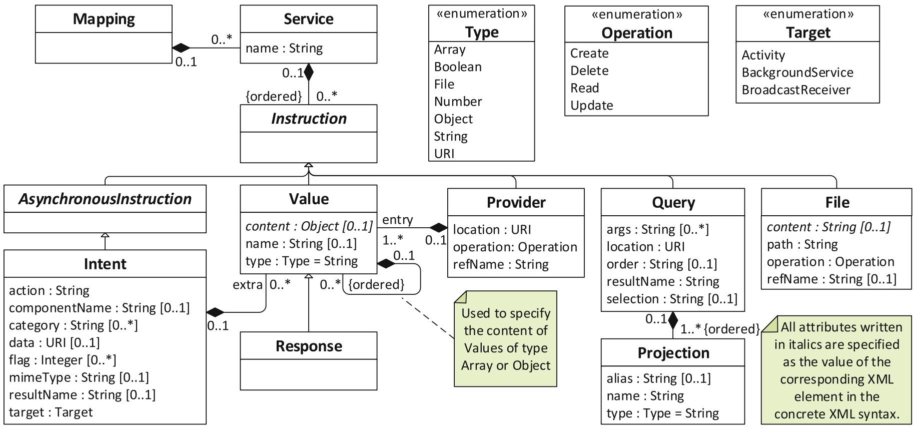

Fig. 3. Metamodel of our DSL

In order to work with values provided in the request or by other instructions (Requirement D4), we allow to define variables. Each variable is a `Value`. A `Value` is a key-value object structure. The `type` attribute of a `Value` specifies the type of the `content` attribute. For instance, if the type is `String`, the `content` attribute is interpreted as a string. If the type is `Object` or `Array`, the `content` attribute is not used, instead the `Value` has further `Values` as children. For items of an array `Value`, consecutive ordinal indices are used as names of the respective items. A variable starts with the $ sign followed by the name of the variable. To access members of a variable, we use the operators commonly known from Java. In particular, accessing a subproperty is done using the dot operator and accessing a specified array item is done via an index enclosed by square brackets. For instance, if a request from an external application contains a property `items` with an array of strings, the variable reference for the first item would be `$request.items[0]`. The `$request` variable always holds the information being passed in the service request. In addition to serving as a data structure for variables, a `Value` can also be used as an instruction to specify complex objects inside a mapping. In this case, the described object is available as a variable with the name specified in the `name` attribute.

The `Intent` instruction is used to create and send an Intent within the Android system (see Requirement D1). The `target` attribute specifies whether the Intent is directed towards an Activity, a Broadcast Receiver, or a Background Service. If the target is an Activity, the presence of the attribute `resultName` indicates whether a result Intent is expected. The result Intent is stored in a variable with the name specified in the `resultName` attribute. The recipient is either defined explicitly using the `componentName` attribute or implicitly using the attributes `action` and `category`. The payload of an Intent can be specified using the attributes `data`, `type`, and `flag`. Additional payload can be specified in a key-value-map called `extras`. A detailed description of these fields is given in the documentation of the Intent data structure in [7]. Using the `Intent` instruction, we are already able to specify simple mappings which only rely on a single Intent and unidirectional communication.

The `File` instruction provides CRUD operations on files (see Requirement D2). The affected file is specified by the attribute `path`. A reference to the affected file is available as a variable with the name specified in the `refName` attribute. The instruction can be used to create blank files and to create or update existing files using static content or values provided by variables. This content has to be specified using the `content` attribute. Furthermore, the instruction can be used to delete files and to read the content of a file into a variable.

CRUD operations on Content Providers (see Requirement D2) are realized using the `Provider` and `Query` instruction. The former instruction is used for create, update, and delete operations on Content Providers. The data items being manipulated are specified as entries, e.g., an entry might specify the content "555-1234" with the name "phonenumber". Such an entry could be inserted into a Content Provider, similar to adding a row to a table of a relational database, where "555-1234" would be the value of the cell of the "phonenumber" column. The URI of an inserted item is stored in a variable with the name specified in `refName` attribute. The `Query` instruction is used to perform queries on Content Providers. The result of the `Query` instruction is an array stored under a variable with the name specified in the `resultName` attribute. A `Projection` specifies which fields are retrieved. The `type` attribute of the `Projection` instruction specifies the type of the retrieved data field. The content of the field is available under the field's name in the corresponding list item. If the `alias` attribute is set, the field's content is available under the alias. Selection criteria are specified using the `selection` and `args` attribute. The result list sortation can be changed using the `order` attribute.

The `Response` instruction addresses Requirement D3 and allows to specify a `Value` being sent as a response to the requestor. Since it is a specialized form of the `Value` instruction, it can be used to create complex types as well. We allow only one `Response` instruction per mapping.

Based on our experience in developing adapter apps, the instructions which are already part of our DSL allow to describe mappings for most services offered by Android apps (see Sect. 6). However, there will be cases where the mapping is more complex and current instructions do not suffice. To address this issue, we provide means to add new instructions to our language. More information on the extensibility of our DSL is given in the Sect. 5.

Interpretation. Figure 4 visualizes the interpretation of a mapping in terms of a UML activity diagram. Upon receiving a request for a certain service, the corresponding mapping is loaded and the instructions are executed in the order they are listed in the mapping-document. Most instructions represent synchronous actions and after they have been processed, we continue with the next instruction. However, for asynchronous instructions like the `Intent` instruction, the processing of the next instruction is postponed until the result has been received and processed. Thereby, we can define mappings in sequential manner even if asynchronous instructions are used (see Sect. 4.3).

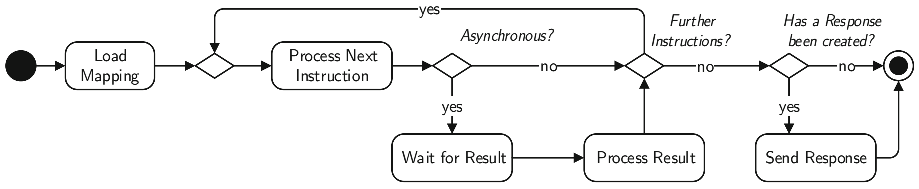

Fig. 4. Interpretation of a mapping visualized as a UML activity diagram

Since there might be additional instructions after a `Response` instruction, e.g., instructions to remove temporarily created files or Content Provider items, the response is sent after all instructions have been processed. Thereby, we can add error messages to the response if those operations fail or discard the response completely.

Concrete Syntax. To formulate instances of our language, we defined a concrete XML syntax. The mapping between the metamodel and the concrete XML syntax is rather straightforward: Every class represents an XML element. Compositions describe parent-child relations. For instance, a `<query>` element can have `<projection>` elements as children. If role names are specified, these are used as element names instead of the name of the class, e.g., extras of an Intent are specified using `<extra>` instead of `<value>`. Attributes of classes are specified as attributes of the XML element but there are two exceptions from this rule: First, those attributes in the metamodel which are written in italics are specified between the opening and closing tag of the corresponding XML element, e.g., `<value>foobar</value>`. Second, attributes with an upper bound greater than 1 are specified as children of the corresponding XML element. For example, categories are specified as XML elements and not as attributes. If an attribute is omitted, the default value specified in the metamodel is assumed.

4.3 Example

This subsection presents a concrete example for an external service, which allows to pick a contact from a smartphone's phone book. The corresponding mapping in shown in Listing 1. In Line 1, the name of the service is specified. To select a contact from the phone book, a bidirectional Intent is sent with the action `ACTION_PICK` and a reference to the Content Provider storing the contacts (Lines 2–3). Once the Intent has been sent, the phone book is being started on the Android device and the user has to select a contact. The user's selection is returned as an Intent stored under the variable name *$contact* (Line 3) and the `data` attribute contains an URI referencing a concrete contact in the phone book's Content Provider. Further contact information like the contact's name and phone number are retrieved by querying the Content Provider on the referenced item (Lines 4–7). Thereby, we resolve the local data references and are able to build a response containing the selected contact information (Lines 8–11), which is sent back to the requestor.

Listing 1. Mapping for a service to pick a contact from the user's phone book

```
 1 <service name="pickContact">
 2  <intent target="Activity" action="ACTION_PICK"
 3    data="content://contacts/data/phones" resultName="contact"/>
 4  <query location="$contact.data" resultName="result">
 5   <projection alias="name" name="display_name"/>
 6   <projection alias="no" name="data1"/>
 7  </query>
 8  <response name="contact" type="object">
 9   <value name="name">$result[0].name</value>
10   <value name="number">$result[0].no</value>
11  </response>
12 </service>
```

5 Implementation

We have built a framework implementing the approach described in the previous sections. Figure 5 gives an overview of our framework, which includes an implementation of the mediator as a node.js web server as well as a generic adapter app that can be specialized through mappings defined using our DSL. We outsourced the provisioning of the platform-independent interface into separate interface apps to enable different interfaces for the external services. Thereby, we are able to reuse our adapter for different kinds of external service interfaces, e.g., a RESTful interface or one using XML-RPC.

Interface and adapter apps use the `Value` data structure (see Fig. 3) as an intermediate format to exchange requests and responses. Interface apps have to perform a bidirectional transformation between the `Value` data structure and the actual representation format. We implemented such a transformation for JSON and XML and are confident that we can support other formats as well.

Currently, we have implemented an interface app providing a RESTful interface via the mediator. The mediator uses push notifications or, if available, a web socket connection to contact the device providing the service. Other interface apps, e.g., one providing an XML-RPC interface, are possible but have not yet been implemented. Also, services do not necessarily need be offered via the mediator, interface apps could directly provide access to the service, e.g., over a socket connection.

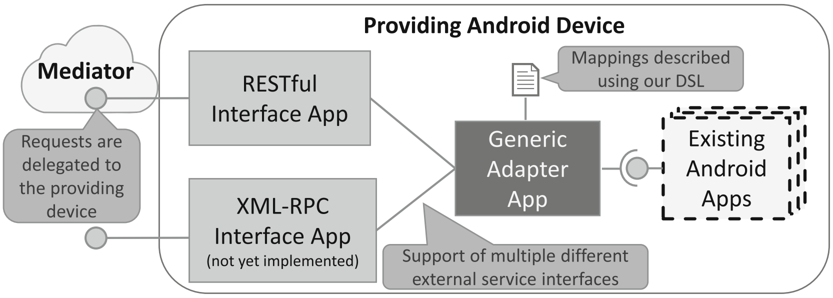
Providing Android Device
Mediator
Requests are delegated to the providing device
RESTful Interface App
XML-RPC Interface App (not yet implemented)
Mappings described using our DSL
Generic Adapter App
Existing Android Apps
Support of multiple different external service interfaces

Fig. 5. Overview of our implementation

We implemented a generic adapter app that includes an interpreter for our DSL. This app can be specialized by defining a mapping document using our DSL and by requesting the necessary permissions to perform these mappings. Specialized versions of this generic adapter app can be deployed on devices which shall offer the services defined by the mapping document.

The generic adapter app allows to extend our DSL by introducing further instructions. For this purpose, one must define a new instruction handler, which is a Java class that interprets the XML code for a single instruction and has access to the variable scope. Hence, it is possible to access existing variables from an instruction handler and to make results available for other instructions by defining them as variables.

6 Feasibility Study and Limitations

Based on our implementation, we tested the feasibility of integrating apps running on Android devices into applications running on other devices. For this purpose, we identified relevant services offered by current Android apps and converted them to external services by specifying mappings using our DSL. In addition, we implemented a demo web application that allows to invoke all of the defined external services from another device[3].

[3] A demo video can be found at http://xdai.dwolt.de.

We mainly focused on services offered by Activities, since we are interested in services which contain user interaction on the Android device. As mentioned in Sect. 2, Activity-based services are accessed by sending and receiving Intents. The Android documentation lists over 20 *Common Intents*[4], which are supported by Activities of various apps, and we were able to define mappings for all of these Common Intents. An example is the mapping discussed in Sect. 4.3.

With regard to services accessible through Common Intents, there are limitations we need to mention: If the input data items for a service are already stored on the device providing the service, we need to know the URIs referencing these data items. There are two options to handle those situations: (i) the requestor has to know the appropriate URIs from previous requests, e.g., from insert actions, and has to include them in the request, or (ii) the user has to choose the data item before the respective service can be used. This can be realized by orchestrating multiple Intents, e.g., the URI that was returned as the result of a pick Intent can be used as an input for a second Intent.

Intents can include objects of self-defined data types as payload. Even though our approach supports the definition complex key-value object structures with the `Value` instruction, we do not support the usage of self-defined data types. However, this is not a problem regarding the before mentioned Common Intents as none of them require any self-defined data types. Additionally, most Intents do not use any other data types than those supported by our language, since using custom data types often results in a tight coupling between the requesting and providing app as both have to know this data type. Nevertheless, to support such scenarios, our language can simply be extended by specifying a new instruction to construct objects of the respective data type (see Sect. 5).

In addition to defining mappings for Activity services, we also defined mappings for services offered by Content Providers and Broadcast Receivers. In particular, we defined external services to query the phone book and the calendar without user interaction on the Android device, e.g., to enable the browsing of the phone book on another device. Furthermore, we created a service to control the media playback on Android devices.

Our language allows to define mappings aggregating multiple internal services offered by apps on the same Android device into a single external service. As an example for such an aggregation, we defined a service that utilizes the camera app to take a picture, which is then passed to a second app, where the user can crop this picture before it is sent back to the requestor. All mappings created during our feasibility study can be used out-of-the-box with our implementation. Furthermore, they show that the current instructions suffice to define external services based on the most common services offered by Android apps.

7 Related Work

Languages to deal with interface mismatches are presented in [4,5] and Autili et al. developed an approach to synthesize mediators for heterogeneous

[4] http://developer.android.com/guide/components/intents-common.html.

networked systems [2]. All of these approaches assume that the existence of an external interface on the providing side, which is not the case in our scenario. For the same reason, the approach presented in [12] is not applicable, since it uses adapters on the requestor's side. Nonetheless, this can be beneficial to integrate external services created with our approach.

Conductor [8] is framework to enable cross-device integration between Android apps running on different devices. In contrast to our approach, Conductor is an invasive framework requiring that both providing and requesting app use this framework. For the Android platform, Iyer et al. [10] build the counter part for our approach. If an existing Android app is loosely coupled to services of other Android apps, their approach allows to substitute these services with web services like external services created using our approach. Thus, combining these two approaches enables cross-device interaction between two Android apps without having to alter any of them. Lee et al. [11] developed an approach to wrap Android activities into OSGi bundles. Thereby, they enable that these activities are composed along with other services by an extended BPEL engine. With our approach, any kind of Android component can directly be exposed as a SOAP-based web service and be composed by standard BPEL engines.

Non-intrusive approaches exist that enable the cross-device usage of single web applications [6] and for mashups which combine multiple web applications [9]. Those approaches leverage the fact that web applications can be channeled through a proxy and that the user interface is interpreted by a browser. Both is not the case for Android apps.

The feasibility of providing web services on mobile devices is analyzed in [1,13], but none of these approaches mention the reuse of existing applications, and thereby, offer their services beyond the device boundaries. Web Intents [3] was an approach trying to expand the idea of Android Intents to a mechanism to integrate any kind of application. It would have been logical to transform Android Intents to Web Intents, but the development has been ceased. Therefore, we decided to abstract completely from Intents and support the usage of widespread technologies for the external interface.

Paulheim [15] developed an approach for UI level integration, but only for applications running on the same device. We also enable cross-device UI level integration since we allow to define external service for Activity-based services, which involve user interaction on the Android device providing the service.

8 Conclusion and Future Work

In this paper, we present an approach to enable the integration of services offered by existing Android apps into applications running on other devices/platforms. Our approach is non-intrusive with regard to existing apps, since we use adapter apps to map the existing Android-specific interfaces to platform-independent service interfaces. We introduced a DSL to specify such mappings and described an architecture supporting the cross-device integration of existing Android apps. We implemented a framework demonstrating the feasibility of our approach.

Along with our framework we already created mappings for various common services offered by standard Android apps.

As part of this paper, we analyzed which current Android services are supported by our approach. In the future, we want to analyze the impact on the user's performance by comparing cross-device interactions using our automated approach vs. manually coordinated cross-device interactions in a user study. Additionally, we are exploring how we can extend the ideas presented in this paper to cover further platforms like iOS or Windows.

References

1. AlShahwan, F., Moessner, K.: Providing SOAP web services and RESTful web services from mobile hosts. In: ICIW 2010, pp. 174–179. IEEE (2010)
2. Autili, M., Inverardi, P., Mignosi, F., Spalazzese, R., Tivoli, M.: Automated synthesis of application-layer connectors from automata-based specifications. In: Dediu, A.-H., Formenti, E., Martín-Vide, C., Truthe, B. (eds.) LATA 2015. LNCS, vol. 8977, pp. 3–24. Springer, Heidelberg (2015)
3. Billock, G., Hawkins, J., Kinlan, P.: Web Intents (2013). http://www.w3.org/TR/web-intents/
4. Cavallaro, L., Di Nitto, E.: An approach to adapt service requests to actual service interfaces. In: SEAMS 2008, pp. 129–136. ACM (2008)
5. Dumas, M., Spork, M., Wang, K.: Adapt or perish: algebra and visual notation for service interface adaptation. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 65–80. Springer, Heidelberg (2006)
6. Ghiani, G., Paternò, F., Santoro, C.: Push and pull of web user interfaces in multi-device environments. In: AVI 2012, pp. 10–17. ACM (2012)
7. Google Inc.: Android Developers (2016). http://developer.android.com/
8. Hamilton, P., Wigdor, D.: Conductor: enabling and understanding cross-device interaction. In: CHI 2014, pp. 2773–2782. ACM (2014)
9. Husmann, M., Nebeling, M., Pongelli, S., Norrie, M.C.: MultiMasher: providing architectural support and visual tools for multi-device mashups. In: Benatallah, B., Bestavros, A., Manolopoulos, Y., Vakali, A., Zhang, Y. (eds.) WISE 2014, Part II. LNCS, vol. 8787, pp. 199–214. Springer, Heidelberg (2014)
10. Iyer, A., Roopa, T.: Extending android application programming framework for seamless cloud integration. In: MS 2012, pp. 96–104. IEEE (2012)
11. Lee, J., Lee, S.J., Wang, P.F.: A framework for composing SOAP, non-SOAP and non-Web services. IEEE Trans. Serv. Comput. **8**(2), 240–250 (2015)
12. Lin, B., Gu, N., Li, Q.: A requester-based mediation framework for dynamic invocation of web services. In: SCC 2006, pp. 445–454. IEEE (2006)
13. Mohamed, K., Wijesekera, D.: A lightweight framework for web services implementations on mobile devices. In: MS 2012, pp. 64–71. IEEE (2012)
14. Papazoglou, M.P., van den Heuvel, W.J.: Service oriented architectures: approaches, technologies and research issues. VLDB J. **16**(3), 389–415 (2007)
15. Paulheim, H.: Ontology-Based System Integration. Springer, Heidelberg (2011)
16. Santosa, S., Wigdor, D.: A field study of multi-device workflows in distributed workspaces. In: UbiComp 2013, pp. 63–72. ACM (2013)
17. Satyanarayanan, M.: Pervasive computing: vision and challenges. IEEE Pers. Commun. **8**(4), 10–17 (2001)