Migrating Android applications towards service-centric architectures with Sip2Share

Alessandro Borriello, Fabio Melillo and Gerardo Canfora
Department of Engineering, University of Sannio, viale Traiano 1, 82100 Benevento, Italy
Email: alessandro.borriello@unisannio.it, fabio.melillo@unisannio.it, gerardo.canfora@unisannio.it

***Abstract*—Mobile devices are rapidly becoming a key means to access the Internet and to discover and consume on-line services. Currently, most mobile applications are essentially clients to interact with remote services. However, the increasing power of devices is enabling a new class of applications that overcome the traditional models of desktop applications and web browsing, and embrace entirely new ways of computing. In this scenario, mobile devices are no longer intended as a means to access server-side data and functionality, but as a source of services that other devices can discover and invoke.**

This calls for new technologies and tools to help migrating existing mobile applications towards the emerging service-centric scenario. In this paper we illustrate Sip2Share, a middleware that supports the creation of peer-to-peer networks of services on the Android platform, and discuss how the middleware can be used to migrate existing Android applications. The paper discusses on two scenarios: a re-architecting scenario, where the application is changed to make it aware of the middleware, and a wrapping scenario, where the application remains unchanged and the connection to the middleware is created in a wrapper.

***Keywords*-Mobile device programming, service oriented computing, software migration, wrapping, peer-to-peer, Android**

I. Introduction

Mobile devices, such as smartphones and tablets, are experiencing a growing popularity. Many users, especially young users, see mobile devices as their standard pathways to connect to the Internet, and *apps* as the primary medium for entertainment, to find relevant product and services, to help their purchasing decisions and to keep in touch over networks of personal or professional contacts [1][2].

While most current mobile applications are essentially clients to interact with remote services, the increasing power of devices, in terms of computation, data storage, visual displays, on-board sensors, network connectivity, and battery autonomy, is enabling a new generation of pervasive applications where handled devices behave not only as clients of back-end services, but also as providers of services.

As an example, mobile device users could share contact records and calendar entries over a trusted network of professional partners without the need for uploading the information on a server. In such a network, when Bob would need the contact record of Alice, and the record is not available on his device, a query would be broadcasted to peers in the network, each acting as a small server.

Mobile devices are also a preferred means to implement human-based services, i.e. services that harness human knowledge, capabilities and expertise with service-oriented infrastructures [3]. As an example, the same trusted network of professional partners could exchange real-time answers to specific problems that arise during a working tasks, with a dispatcher broadcasting a specific query to a set of peers selected based on their expertise profiles.

Finally, mobile devices could be used for occasional remote sensing; for example, during a fire on a large area, the crisis management unit could localize one or more firefighters in a specific area of operation and use their cameras to remotely take pictures of the scene that help making informed decisions.

All these examples require that devices be provided with the ability to act as peers in a network, being able to discover other peers and both provide and request services. As with other technology scenario evolutions in the past, the rapid diffusion of pervasive services-centric applications on mobile devices depends upon the ability to define development technologies and tools that hide developers from the complexity of underlying infrastructures and preserve past investments by supporting the migration of existing applications. To this aim, we have developed Sip2Share[4], a middleware that supports the creation of peer-to-peer networks of services on the Android platform.

A point of strength of our middleware is that services are advertised, discovered and called using the same native mechanisms of the Android platform, i.e. intents, manifests and broadcast receivers. This has two main benefits: developers who are familiar with Android application development will need very little training to be productive with our middleware; the middleware can be exploited to easily migrate existing Android applications that have not been designed for exchanging services with other devices.

In this paper we briefly describe the Sip2Share middleware, and its features, and show how the middleware can be used to migrate an existing Android application. Two migration scenarios are explored, a re-architecting scenario, where the application is changed to make it aware of the middleware, and a wrapping scenario, where the application remains unchanged and the link to the middleware is created in a wrapper. Of course, the first approach is more flexible, but it is only applicable when the source code is available, while the second approach does not require that source code be accessible. Sip2Share is available at http://sip2share.googlecode.com/svn/trunk/Sip2Share.

II. An overview of Sip2Share

An android application is a federation of loosely-coupled components – activities, services, content providers and broadcast receivers – dynamically bind to each other at run time by means of asynchronous messages, named intents. A component that needs to interact with another component prepares an intent, describing the desired action and the associated data, and passes it to the Android platform. The platform resolves the intent, based on the capabilities that components have declared, and delivers it to the target component. This may entail that the target component needs to be preliminarily activated.

The basic idea behind Sip2Share is to transparently extend this model to pervasive applications, where components are deployed on the nodes of a peer-to-peer service network and intents are resolved based on the capabilities advertised by the networked nodes and are sent across devices.

The middleware uses a two phases mechanism for resolving intents. Devices can declare their own capabilities to a super-peer, advertising themselves as a provider for some services. A device that needs to interact with another peer, to consume a service, sends a message with the required action to a super-peer, obtaining a set of addresses of nodes that have the capabilities to manage the request. The purpose of the super-peer(s) is to find the remote peers that match the requested action. In the second phase, an intent is broadcasted towards the retrieved peers and, on the remote devices, it is resolved locally by the standard Android matcher.

The communication stack of Sip2Share comprises four layers. At the bottom layer there is the IP network, on which is placed a SIP layer[5] used for identifying the nodes over the network. On this, a Sip2Peer [6] layer is used to send data using the SIP protocol. Our middleware is positioned on the top and has the responsibility for creating and managing an overlay network of services, and offering facilities to deploy, publish, discover and invoke services on mobile devices, with a full duplex communication.

An overlay network comprises peers and super-peers, the latter used to registering and seeking services. Super-peers can implement different strategies to match service requests with the available services. As an example, a simple registry-like strategy entails that devices register their offered services to a super-peer, and these are retrieved by other devices by querying the super-peer (pull strategy). Alternatively, a publish-subscribe strategy can be implemented: devices can register on a super-peer either service needs or service offerings. Whenever a new match is found between a need and an offered service, the super-peer notifies the requesting device (push strategy). In both cases, matching can be either syntactic or based on semantics.

A. *Sip2Share API*

Fig. 1 shows the main classes of the middleware; its boundaries are marked by the `Activity` and the `Peer` classes, the first belongs to the user application, the second one to the

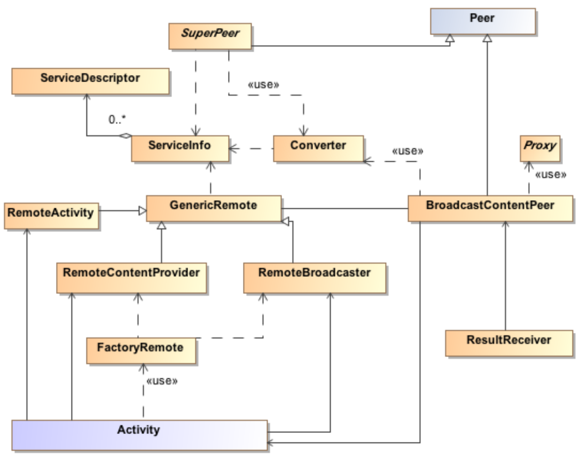

Fig. 1: Sip2Share class diagram

Sip2Peer layer. The developer is provided with three main classes:

- `RemoteBroadcaster`, that has the methods for sending intents across the networks of peers
- `RemoteActivity` that allows for starting activities on a remote peer
- `RemoteContentProvides` that exposes the methods for querying content providers on a remote peer

All the methods offered by these classes use the same interface as the local Android counterparts. To consume services on a remote peer, a developer only has to instantiate a `FactoryRemote` from which instances of the above classes are then built. Using the same Android's signature and patterns for remote communications among devices, as for communication among applications on the same device, is helpful both to migrate existing code and to leverage investments: code changes are reduces to a minimum and Android developers can be productive with Sip2Share without a need for learning another technology.

B. *Resolving Intents among devices*

Sip2share uses a two phases mechanism for resolving intents, with one or more super-peers that identify, based on the advertised services, the devices to which an intent must be broadcasted.

Independently of the response strategy, e.g. push or pull, and matching logic, syntactic or semantic, the signature for communicating with a super-peer is: `sendToSuperPeer(ServiceInfo myServiceInfos, String kindOfRequest)`, where `myServiceInfos` is a generic object that stores IP and port of the device and information describing the service advertised or searched for. Actual information will depend upon the type of matching (e.g. actions for syntactic matching, schemas or ontologies queries for semantic matching) and the component to be invoked (e.g. broadcast receiver, activity or content provider).

The `kindOfRequest` string defines the kind of message: `PUBLISH`, `SUBSCRIBE`, `RETRACT` and `UNSUBSCRIBE`. If the super-peer interaction is modeled with a *pull* strategy, then, for instance, a `SUBSCRIBE` request is a query that returns the

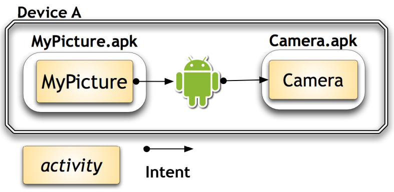

Fig. 2: Local scenario

result of the matching; otherwise, in a *push* scenario, the same request is stored for future matching with newly published peers, and in that case the requester peer is asynchronously notified with a list of new matching peers.

The middleware comprises an abstract `SuperPeer` class that developers can extended to implement their own matching strategies; an example implementation is available at http://sip2share.googlecode.com/svn/trunk/SuperPeer.

III. Using Sip2Share for migration

In this section we show how Sip2Share can be used to migrate an existing application to a peer-to-peer network of devices. We use a simple existing app, named MyPicture, depicted in Fig. 2. MyPicture requires a picture from the camera application using a generic intent with the IMAGE_CAPTURE action invoking `startActivityForResult` from the Android API. The intent is delivered to the Camera app, the photo is shot and the control is then returned to MyPicture.

Our aim is to realize a "new" app that can require a picture from any another device that has registered as a provider of remote live pictures. Two scenarios are explored, one in which the source code is available and one in which the source code is not accessible. In the first case, Fig. 3, the developer re-architects the code for broadcasting the intents to other devices; in the second case, Fig. 4, she has to implement a wrapper that will capture the intents from MyPicture and will exploit Sip2Share to interact with the remote devices. The code of the examples is available at http://sip2share.googlecode.com/svn/trunk/ExampleCode/

```
Intent newIntent = new Intent("android.media.
    action.IMAGE_CAPTURE");
newIntent.putExtra(MediaStore.EXTRA_OUTPUT,
    fileUri);
this.startActivityForResult(newIntent,
    reqValue);
```

Listing 1: Starting camera

A. Re-architecting scenario

Listing 1 shows the original code for local invocation of the camera. In the re-architecting scenario, we preliminarily transform the `startActivityForResult` invocation into a `sendOrderedBroadcast`, as shown in listing 2.

```
Intent newIntent = new Intent("android.media.
    action.IMAGE_CAPTURE");
newIntent.putExtra(MediaStore.EXTRA_OUTPUT,
    fileUri);
this.sendOrderedBroadcast(newIntent, null, new
     MyResultReceiver(), null, 0 , null, new
    Bundle());
```

Listing 2: Starting camera re-architected

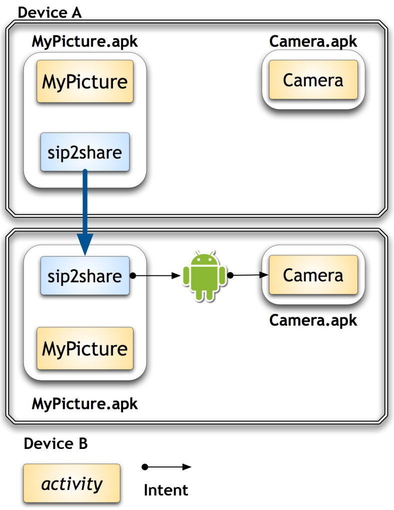

Fig. 3: Re-architecting scenario

The original intent is captured by a local receiver which starts the camera. Likewise, in the remote scenario, the original intent, passed to the remote device using Sip2Share (Fig. 3), is caught from its local receiver which starts the camera. On return, the path is followed in the reverse way; in our example an external server is used for storing temporally the picture shot. Listing 3 shows the code to create the middleware object.

```
FactoryRemote factory = new FactoryRemote();
broadcaster = factory.getRemoteBroadcaster(
    MyPicture.this);
```

Listing 3: RemoteBroadcaster initialization

Listing 4 shows the code to set the super-peer address, and the publish and subscribe messages sent to the super-peer, for the same (i.e `ServiceInfo`).

```
broadcaster.setServer(server);
ServiceInfo sinfo = new ServiceInfo();
sinfo.addDescriptor(URI, null, null,
    ServiceInfo.BROADCAST_RECEIVER);
remoteActivity.sendToSuperPeer(sinfo,
    Converter.PUBLISH);
remoteActivity.sendToSuperPeer(sinfo,
    Converter.SUBSCRIBE);
```

Listing 4: SuperPeer setting

Finally, local broadcasting (listing 2) is transformed into remote broadcasting, as shown in listing 5.

```
broadcaster.sendRemoteOrderedBroadcast(
    newIntent, null, new MyResultReceiver(),
    null, 0 , null, new Bundle());
```

Listing 5: Starting remote camera re-architected

B. Wrapping scenario

In this scenario, Fig. 4 we create an additional application, a WrappedCamera, that subscribes the same intent action of the original app, in our case the IMAGE_CAPTURE action. When MyPicture requires a picture, a pop-up is shown to the user, to select either the WrappedCamera or the local camera application. The wrapper camera implements the interaction

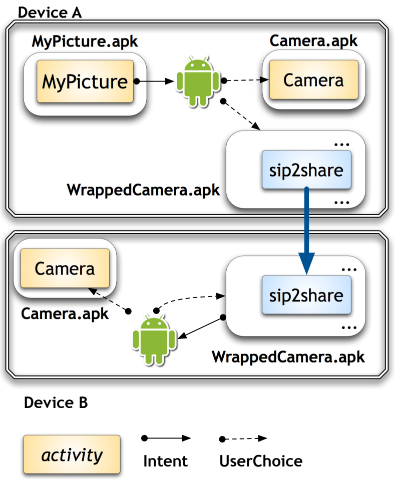

Fig. 4: Wrapping scenario

with the middleware to broadcast the intent to remote devices; on the remote device, the WrappedCamera catches the intent and sends it locally: again, the user can choose to use the local camera to further broadcast the request. Eventually, a Camera takes the photo and returns it to the invoking device. Listing 6 shows the creation of a middleware object in the wrapper. The remote call is similar to the original one, as shown in listing 7.

```
FactoryRemote factoryRemote = new
    FactoryRemote();
remoteActivity = factoryRemote.
    getRemoteActivity(WrappedCamera.this);
```

Listing 6: RemoteActivity initialization

```
remoteActivity.startActivityForResult(
    newIntent, reqValue);
```

Listing 7: RemoteActivity utilization

IV. Conclusions and Future Works

In this paper, we have described a middleware to help designers to create a peer-to-peer system with little programming effort and illustrated how the middleware can be used to migrate exiting applications. The paper has discussed two scenarios: migration and wrapping. In both cases, little code has to be developed either to change the original application and make it aware of the middleware, or to wrap the application into an additional component that creates the connection with the middleware.

While several authors have faced the problem of migrating legacy code toward service oriented architectures (see, for example, references[7][8][9]) and to ease access from mobile devices [10], little work exists on porting existing mobile applications from the client-server model to a service-centric pervasive architecture.

Reference [11] presents CloneCloud, a system that automatically transforms mobile applications to benefit from the cloud. The work [12] describes MicroMAIS, an integrated platform for supporting the execution of Web service-based applications natively on a mobile device. Sip2Share differs from [11] because the architecture is lighter, and is suitable for invoking different types of remote components, not only computational services. Unlike [12], Sip2Share does not use the WebService communication stack, but extends the Android pattern for inter-component communication. Nakao and Nakamoto [13] use a similar approach, extending the intent class at the OS level to implement transparent remote invocation of Android services without any modification to the application code.

Currently, our middleware supports remote activities, broadcast receivers and content providers; as a future work, we are developing mechanisms for remote interaction with Android services.

An open issue is the frequent disconnection of the devices, currently managed at application level using the Sip2Share API. However we intend to explore approaches, similar to AmbientTalk[14], which consider the disconnection as an integral part of the programming system.

Further, we plan to experimentally evaluate the middleware in terms of developer effort to port existing application in a distributed scenario.

References

[1] I. comScore, "comScore Reports September 2012 U.S. Mobile Subscriber Market Share - comScore, Inc," 2012. [Online]. Available: http://www.comscore.com/Insights/Press_Releases/2012/11/comScore_Reports_September_2012_U.S._Mobile_Subscriber_Market_Share
[2] ——, "Small Screens Make a Big Impact Across Europe," 2012. [Online]. Available: http://www.comscoredatamine.com/2012/09/small-screens-make-a-big-impact-across-europe/
[3] D. Schall, S. Dustdar, and M. B. Blake, "Programming human and software-based web services," *Computer*, vol. 43, pp. 82–85, 2010.
[4] G. Canfora and F. Melillo, "Sip2share - a middleware for mobile peer-to-peer computing," in *ICSOFT*, 2012, pp. 445–450.
[5] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "RFC 3261 - SIP: Session Initiation Protocol." in *Internet Engineering Task Force IETF Request for Comments*. Internet Engineering Task Force, 2002, no. June 2002, pp. 1–268. [Online]. Available: http://www.ietf.org/rfc/rfc3261
[6] M. Picone, "sip2peer Tutorial Android Example Outline," 2011. [Online]. Available: http://code.google.com/p/sip2peer/wiki/sip2peerTutorial
[7] J. C. A. Almonaies and T. Dean, "Legacy system evolution towards service-oriented architecture," in *SOAME*, 2010, pp. 53–62.
[8] M. Athanasopoulos and K. Kontogiannis, "Identification of rest-like resources from legacy service descriptions," in *WCRE*, 2010, pp. 215–219.
[9] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana, "A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures," *Journal of Systems and Software*, vol. 81, no. 4, pp. 463–480, 2008.
[10] M. El-Ramly, E. Stroulia, and H. Samir, "Legacy systems interaction reengineering," in *Human-Centered Software Engineering*, 2009, pp. 316–333.
[11] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *EuroSys*, 2011, pp. 301–314.
[12] P. Plebani, C. Cappiello, M. Comuzzi, B. Pernici, and S. Yadav, "Micromais: executing and orchestrating web services on constrained mobile devices," *Softw., Pract. Exper.*, vol. 42, no. 9, pp. 1075–1094, 2012.
[13] K. Nakao and Y. Nakamoto, "Toward remote service invocation in android," in *UIC/ATC*, 2012, pp. 612–617.
[14] J. Dedecker, "Ambient-oriented programming in ambienttalk: combining mobile hardware with simplicity and expressiveness," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. ACM, 2005, pp. 196–197. [Online]. Available: http://doi.acm.org/10.1145/1094855.1094932