# Service Oriented Architecture for Mobile Cloud Computing

by

Xinyi Dong

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved May 2012 by the
Graduate Supervisory Committee:

Dijiang Huang, Chair
Partha Dasgupta
Yinong Chen

ARIZONA STATE UNIVERSITY

August 2012

# DEDICATION

To my family.

ABSTRACT

The Open Services Gateway initiative (OSGi) framework is a standard of module system and service platform that implements a complete and dynamic component model. Currently most of OSGi implementations are implemented by Java, which has similarities of Android language. With the emergence of Android operating system, due to the similarities between Java and Android, the integration of module system and service platform from OSGi to Android system attracts more and more attention. How to make OSGi run in Android is a hot topic, further, how to find a mechanism to enable communication between OSGi and Android system is a more advanced area than simply making OSGi running in Android. This paper, which aimed to fulfill SOA (Service Oriented Architecture) and CBA (Component Based Architecture), proposed a solution on integrating Felix OSGi platform with Android system in order to build up Distributed OSGi framework between mobile phones upon XMPP protocol. And in this paper, it not only successfully makes OSGi run on Android, but also invents a mechanism that makes a seamless collaboration between these two platforms.

TABLE OF CONTENTS

Page

CHAPTER

# Chapter 1

# INTRODUCTION

Recently, along with the increasing complexity of applications in mobile devices, it's necessary to reduce coding complexity when developing mobile applications [1]. As the trend of mobile cloud computing, keeping the code reusable [2] and manageable becomes a concern. Among current mobile device systems, Android [3] system takes the biggest portion in market [4]. To achieve scalability and reusability in mobile cloud computing, service oriented architecture (SOA) [5] is regarded as a new paradigm for Android mobile devices' application development. OSGi [6][7] (Oriented Service Gateway initiate) is a dynamical service framework, which is implemented in Java, designed for fulfilling SOA. OSGi framework is a container that contains components or services in forms of bundles. A bundle is an independent module of program codes, which can be used as basic service components to build mobile applications running on mobile devices. So it is reasonable to integrate an Android process, which comes in form of application or service, with OSGi framework containing bundles running in it, and some of the tasks that belong to Android application or service can be assigned to OSGi bundles respectively. This approach not only increases software's scalability [8] since bundle's decoupled and independent but also enhances software's reusability because an independent bundle could be easily transplanted to other application. And another notable feature of OSGi is dynamicity [9], which involving dynamic control over bundles along with dynamic bundle update without reboot. These features ensure that OSGi serves as an ideal platform for

Android SOA architecture [10]. However, OSGi is not originally designed to provide an SOA platform for mobile devices but for PCs [11]. There are some problems preventing us from integrating OSGi to Android. Firstly, OSGi is not designed for Android. It is difficult to run OSGi on Android; moreover it is not possible for current solutions allowing OSGi service modules running in Andriod and Android non-OSGi processes to communicating with each other in bundle level. Secondly, incorporating OSGi into Android must be able to work with existing OSGi bundles for PCs. Some existing solutions attempted to solve these problems, however none of them can address both above two problems effectively and simultaneously. For example, EZdroid [12] can run OSGi framework on Android, but it cannot support the intercommunication between OSGi bundles and Android internal processes; mBS [13] devised by ProSyst rebuilt OSGi on Android to support the intercommunication between OSGi bundles and Android internal processes, but the new OSGi bundles cannot work with original OSGi bundles. Thirdly, to build an SOA framework among mobile devices and PCs, it is necessary to allow OSGi frameworks to interact with each other among multiple mobile devices. None of any existing solutions provide this capability.

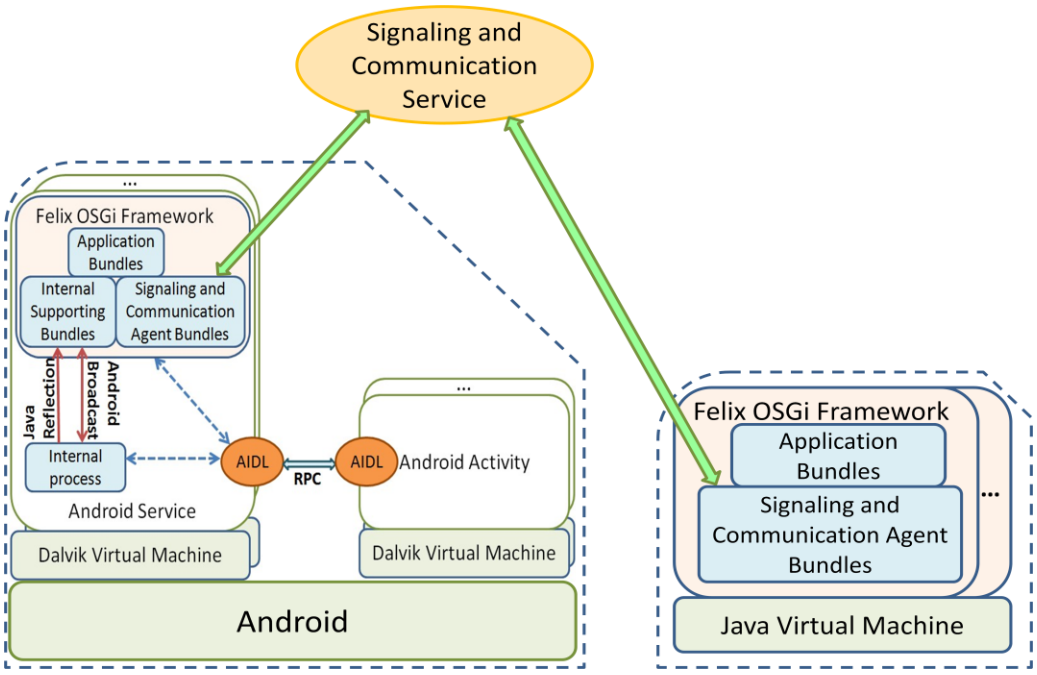



**Figure 1-1: SOA-MCC Model**

To solve the above described problems, we propose SOA-MCC model. In Figure 1-1, SOA-MCC is consisted by three parts: Android, signaling and communication service, and PC. Signaling and communication service is used to dispatch data among signaling and communication agents.

In Android side, Felix [14] OSGi framework is used as the container to support OSGi bundles running on Android. Felix OSGi is a light weight implementation of OSGi on an Android device to support mobile applications. Based on it, we developed an internal support bundle to enable the communication between OSGi bundles and Android internal processes. The internal supporting bundle utilizes both java reflection and Android broadcast mechanism to achieve the communication between OSGi bundles and Android internal processes. Since Felix OSGi does not change the original OSGi implementations, this approach work with any OSGi bundles running on both mobile devices and PCs.

To address the inter-OSGi framework communication issues, SOA-MCC adopts an XMPP (Extensible Messaging and Presence Protocol) based solution, in which an XMPP server is used as a signaling and communication service between different OSGi frameworks. To enable this service within Android, a signaling and communication agent bundle needs to be developed within Felix OSGi.

To make SoA-MCC model work in cloud environment, we put this model in MobiCloud [15][16] system, which is a cloud system with VMs serving for dedicated mobile phone users. As Figure 1-2 shows, each mobile phone interacts with its dedicated VM [17] located in Mobicloud VM pool on top of OSGi framework by signaling and communication service.

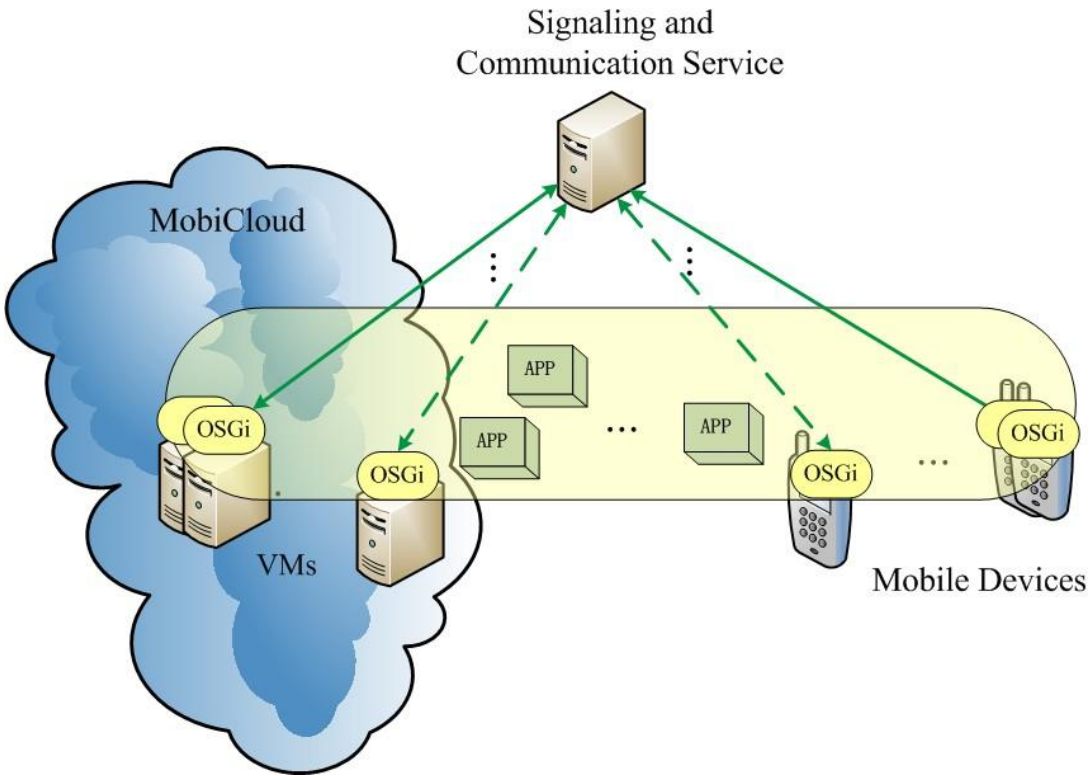



**Figure 1-2: SoA-MCC model in Mobicloud System**

In summary, SOA-MCC has the following contributions: (1) it implements the communication in bundle level between OSGi framework and Android native processes; (2) it does not change OSGi original framework and thus OSGi bundles can run in Java Running Environment (JRE) without compatibility issues; (3) it support mobile cloud computing in that allowing OSGi bundles as cloud services to be reused among OSGi frameworks; (4) it is a lightweight implementation and consumes limited memory and power on mobile devices.

In the rest of this paper, we described the related work and compared with our solution in Section Related Work. In Section Systems and Models, we introduced Android, OSGi and XMPP to explain our architecture; also the description of technique mechanism in these systems would be introduced in this section. We would describe the overall architecture and technique details in section "SoA-MCC". And section "Performance and Evaluation" states the experimental results and figures. The conclusion is made in section "Conclusion".

# Chapter 2

## RELATED WORK

To completely integrate OSGi framework with Android, there are two major problems pending to be solved: The first is to let OSGi framework run in Android, and the second is to enable communication between OSGi framework and Android. Furthermore, whether existing bundles is compatible with integrated OSGi in Android should be taken into consideration as well. And following projects: mBS from ProSyst, EZdroid from Luminis and Bundroid+Fedroid from Samia Bouzefrane [18] tried different approaches to tackle these two problems.

To let OSGi run on Android, mBS implemented their own design of OSGi as an Android application, which could run on Android just like other Android application. While EZdroid converted Felix OSGi into Dalvit VM readable machine code, then used ADB (Android Debug Bridge) tool to push and run Felix OSGi on Linux kernel of Android using shell command. Meanwhile Bundroid+Fedroid wrapped Felix OSGi as java package into an Android application by importing it, and then created an OSGi class instance inside Android application, which is the same idea as we utilized.

To enable communication between Android and OSGi framework, since mBS rebuilt the OSGi framework as an Android application, the communication between Android and OSGi is actually very intuitive; they could simply call each other's API directly. But EZdroid didn’t have a way to communicate between Android and OSGi; their approach isolates OSGi from Android because they run OSGi on Linux kernel of Android than on Android. Meanwhile

Bundroid+Fedroid didn't figure out a solution for communication between Android service and OSGi framework yet. In our solution, it utilizes java reflection mechanism to set Android context instance inside OSGi bundles to send Android broadcast, as well as register broadcast listener inside OSGi to enable to receive Android broadcast from Android to achieve mutual interaction between OSGi and Android.

For compatible issue between existing bundles and integrated OSGi in Android, mBS is not compatible with OSGi in PC because mBS has rebuilt the OSGi framework into Android and redefined bundles. Existing bundles compatible with original OSGi have to re-implement to run on mBS. But EZdroid is compatible since they keep the origin of Felix OSGi running on Android, thus previous bundles could run on integrated OSGi on Android as long as they've converted into Dalvik VM readable machine code. And Bundroid+Fedroid and our solution are the same as EZdroid since OSGi used for integration remains unaltered.

To conclude, mBS successfully solved integration OSGi into Android but compatible issue with existing bundles remained. Although EZdroid and Bundroid+Fedroid did comply with existing bundles, the integration with Android was incomplete due to the lack of communication between Android and OSGi framework. While our solution not only fulfills complete integration from OSGi to Android but also complies with existing bundle.

# Chapter 3

# SYSTEMS AND MODELS

## 3.1 OSGi

In this section it will introduce OSGi architecture, OSGi alliance, Felix OSGi, and explain bundle, bundle life cycle and bundle manifest.

### 3.1.1 OSGi architecture

The Open Services Gateway initiative framework is a module system and service platform for the Java programming language that implements a complete and dynamic component model. An important concept in OSGi is bundle, which is a form of applications or components in OSGi framework. Each bundle is a tightly-coupled, dynamically loadable collection of classes, jars, and configuration files that explicitly declare their external dependencies (if any). And bundles can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot. Also, APIs of OSGi can mange application life cycle (start, stop, install, uninstall, etc.), the remote downloading of management policies are allowed. The service registry allows bundles to detect the addition of new services, or the removal of services.

The OSGi Service Platform facilitates the componentization of software modules and applications and assures interoperability of applications and services over a variety of network devices. Building systems with OSGi modules will enhance development productivity and makes them much easier to modify and evolve. The OSGi Service Platform is delivered in many Fortune Global 100 company

products and services and in diverse markets including enterprise, mobile, home, consumer, etc.

OSGi is a framework that can be implemented in many frameworks, which providing environment for the modularization of applications into bundles. The architecture of OSGi is like this:

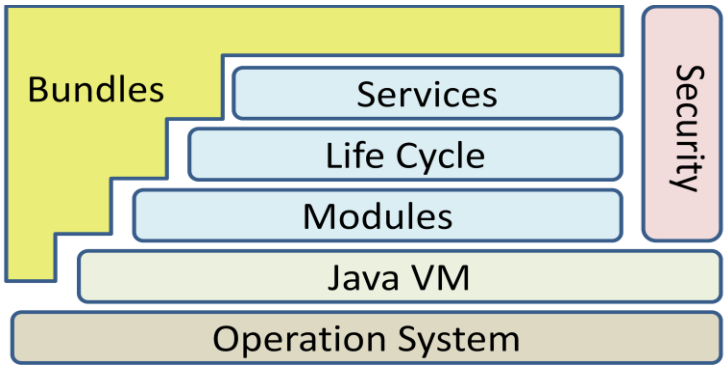



**Figure 3-1: OSGi architecture**

Bundles: A form of applications or components in OSGi framework, usually are jar components with extra manifest headers.

Services: The services layer connects bundles in a dynamic way by offering a publish-find-bind model

Life Cycle: The API for life cycle management to install, start, stop, update, and uninstall bundles.

Models: The layer that defines encapsulation and declaration of dependencies. Ex, how a bundle can import and export code.

Security: The layer that handles the security aspects by limiting bundle functionality to pre-defined capabilities.

### 3.1.2 OSGi Alliance

The OSGi Alliance [19] is a worldwide consortium of technology innovators that advances a proven and mature process to create open specifications that enable the modular assembly of software built with Java technology. Modularity reduces software complexity; OSGi is the best model to modularize Java.

The alliance provides specifications, reference implementations, test suites and certification to foster a valuable cross-industry ecosystem. Member companies collaborate within an egalitarian, equitable and transparent environment and promote adoption of OSGi technology through business benefits, user experiences and forums. The alliance also promotes collaboration among important ecosystem players within and outside the OSGi Alliance in order to provide the market with innovative solutions based on open standards.

### 3.1.3 Felix OSGi

Apache Felix is a mature OSGi framework implemented by Apache and Is the most compact one, suitable for embedded systems. It implements Release 4.x. Apache license. It has been successfully applied to projects like Apache servicemix. It provides full set of services, covering all the specifications defined in OSGi 4.2. It’s a light weight version of OSGi comparing to other frameworks. And moreover, when integrating to Android system, it’s not necessary to hack the root permission to run it.

### 3.1.4 Bundle explanation and bundle life cycle explanation

As stated above, we choose Felix implementation as our OSGi framework, and it is a component framework for Java. In this framework, the basic component/unit

is called bundles which can be remotely installed, uninstalled, started, and stopped. A bundle is a group of Java classes (compiled java code) and additional resources files (.xml files, .jpeg files, etc) equipped with a detailed meta-data file (manifest.mf) on all its contents, as well as additional services needed to give the included group of Java classes more sophisticated behaviors, to the extent of deeming the entire aggregate a component.

Bundles can export services or run processes, and have their dependencies managed, such that a bundle can be expected to have its requirements managed by the container. Each bundle can also have its own internal classpath, so that it can serve as an independent unit. All of these are standardized such that any valid OSGi bundle can theoretically be installed in any valid OSGi container. The bundles in Felix OSGi follow this standard and could be migrated to other OSGi framework as long as it has java virtual machine and implemented in Java.

In OSGi architecture, bundle lifecycle is a layer to manage bundles to install, uninstall, start and stop, etc. The life cycle layer introduces dynamics that are normally not part of an application. Extensive dependency mechanisms are used to assure the correct operation of the environment. Life cycle operations are fully protected with the security architecture.

And in a bundle's life cycle, it has status INSTALLED, RESOLVED, UNINSTALLED, STARTING, ACTIVE and STOPPING.

INSTALLED: The bundle has been installed into OSGi container, but some of the bundle's dependencies have not been met. The bundle requires packages that have not been exported by any currently installed bundle.

RESOLVED: The bundle is installed, and the OSGi system has connected up all the dependencies at a class level and made sure they are all resolved. The bundle is ready to be started. If a bundle is started and all of the bundle's dependencies are met, the bundle skips this state.

STARTING: A temporary state that the bundle goes through while the bundle is starting, after all dependencies have been resolved. BundleActivator.start method will be called and this method has not yet returned.

ACTIVE: The bundle is running.

STOPPING: A temporary state that the bundle goes through while the bundle is stopping. The BundleActivator.stop method has been called but the stop() method has not yet returned.

UNINSTALLED: The bundle has been removed from the OSGi container.

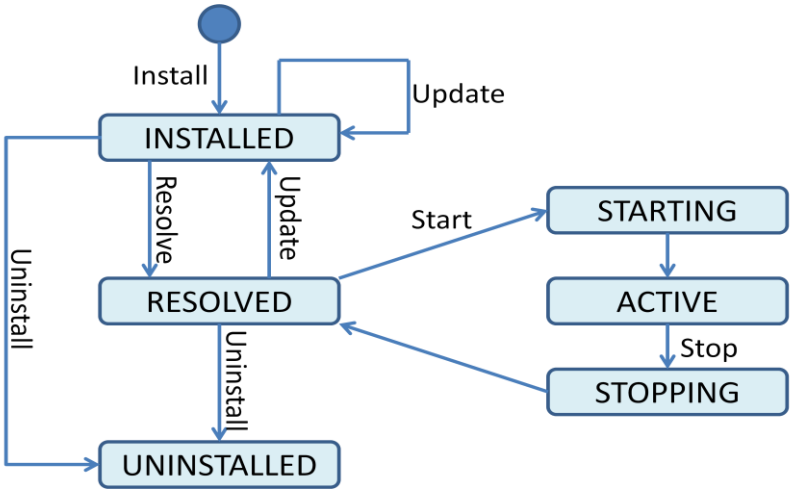



**Figure 3-2: Bundle Life Cycle**

The Life Cycle of a bundle inside the OSGi Platform is defined as follows. The bundle must first be installed. When it is required to start, the package-level dependencies with other bundles are resolved. When all dependencies are resolved, the bundle activator is launched: the start**()** method is called, and the related code is executed. Typically, these operations consist in configuration and publication of services.

After start() is called, the bundle is in the STARTED state. Updating, stopping and uninstalling build the last possible operations for bundle management.

### 3.1.5 Bundle manifest

An OSGi bundle, which can be a JAR or web application archive (WAR) file, contains a bundle manifest file. The bundle manifest file contains additional headers to those in the manifest for a JAR or WAR file that is not an OSGi bundle. The metadata that is specified in these headers enables the OSGi Framework to process the modular aspects of the bundle. The OSGi R4.3 Framework specification defines a set of manifest headers such as Export-Package and Bundle-Classpath, which bundle developers use to supply descriptive information about a bundle.

Bundle's manifest file, which is named as META-INF or MANIFEST.MF, contains bundle's information which would be checked when the bundle is called or exported.

Take one of bundles in this project as an example:

```
Manifest-Version: 1.0
Export-Package: asu.snac.android.broadcast.sender;uses:="asu.snac.
android.broadcast.api,org.osgi.framework,android.content",asu.snac.
android.broadcast.test;uses:="asu.snac.android.broadcast.api,org.osgi
.framework,android.content"
Tool: Bnd-1.15.0
Bundle-Name: broadcastsender
Created-By: Apache Maven Bundle Plugin
Bundle-Version: 1.0.0.SNAPSHOT
Build-Jdk: 1.6.0_21
Bnd-LastModified: 1323126754077
Bundle-ManifestVersion: 2
Bundle-Activator: asu.snac.android.broadcast.sender.Activator
Import-Package: android.content,asu.snac.android.broadcast.api,org.osgi.
framework;version="[1.3,2)"
Bundle-SymbolicName: asu.snac.android.broadcast.sender.broadcastsender
```

There are many contents in the example, and the meaning of some important contents in the example is as follows:

*Export-Package*: Expresses what Java packages contained in a bundle will be made available to the outside world.

*Import-Package*: Indicates what Java packages will be required from the outside world, in order to fulfill the dependencies needed in a bundle.

*Bundle-Name*: Defines a human-readable name for this bundle, Simply assigns a short name to the bundle.

*Bundle-SymbolicName*: The only required header, this entry specifies a unique identifier for a bundle, based on the reverse domain name convention

*Bundle-Description*: A description of the bundle's functionality.

Bundle-ManifestVersion: This little known header indicates the OSGi specification to use for reading this bundle.

*Bundle-Version*: Designates a version number to the bundle.

*Bundle-Activator*: Indicates the class name to be invoked once a bundle is activated.

## 3.2 Android System

Android is an operating system initially designed for mobile phones by Google. It ’s an open source project based on Linux and is organized around multiple layers. Each Android application runs on an independent virtual machine called Dalvit V M to prevent from ectangling to other application once an application crashes.

### 3.2.1 Android Architecture

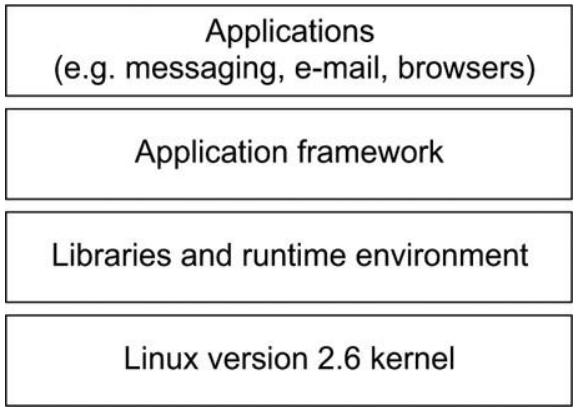



**Figure 3-3: Android Architecture**

As the architecture shows above, Android Architecture has various layers. It has Linux kernel as basis, then there is Android runtime provides runtime environment for application framework. The application framework facilitates the application development based on several components such as life-cycle management (activity manger), GUI management (View system), data sharing (Content providers), resource access (Resource manager), etc. And on top of application framework, Android applications, for example, messaging, browser, emails, runs on it.

Here are several concept explanations in Android system we used in this project: *activities*, *services*, *activity life cycle*, *service life cycle* and *remote service*.

### 3.2.2 Activity

An activity is an application component that provides a screen with which users can interact in order to do something, such as view a map, take a photo, and dial phones, etc. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows. An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an

application is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack. When a new activity starts, it is pushed onto the back stack and takes user focus.

### 3.2.3 Activity Lifecycle:

An activity can exist in essentially three states:

Resumed: The activity is in the foreground of the screen and has user focus.

Paused: Another activity is in the foreground and has focus, but this one is still visible. That is, another activity is visible on top of this one and that activity is partially transparent or doesn't cover the entire screen. A paused activity is completely alive, but can be killed by the system in extremely low memory situations.

Stopped: The activity is completely obscured by another activity. A stopped activity is also still alive. However, it is no longer visible to the user and it can be killed by the system when memory is needed elsewhere.

If an activity is paused or stopped, the system can drop it from memory either by asking it to finish, or simply killing its process. When the activity is opened again, it must be created all over.

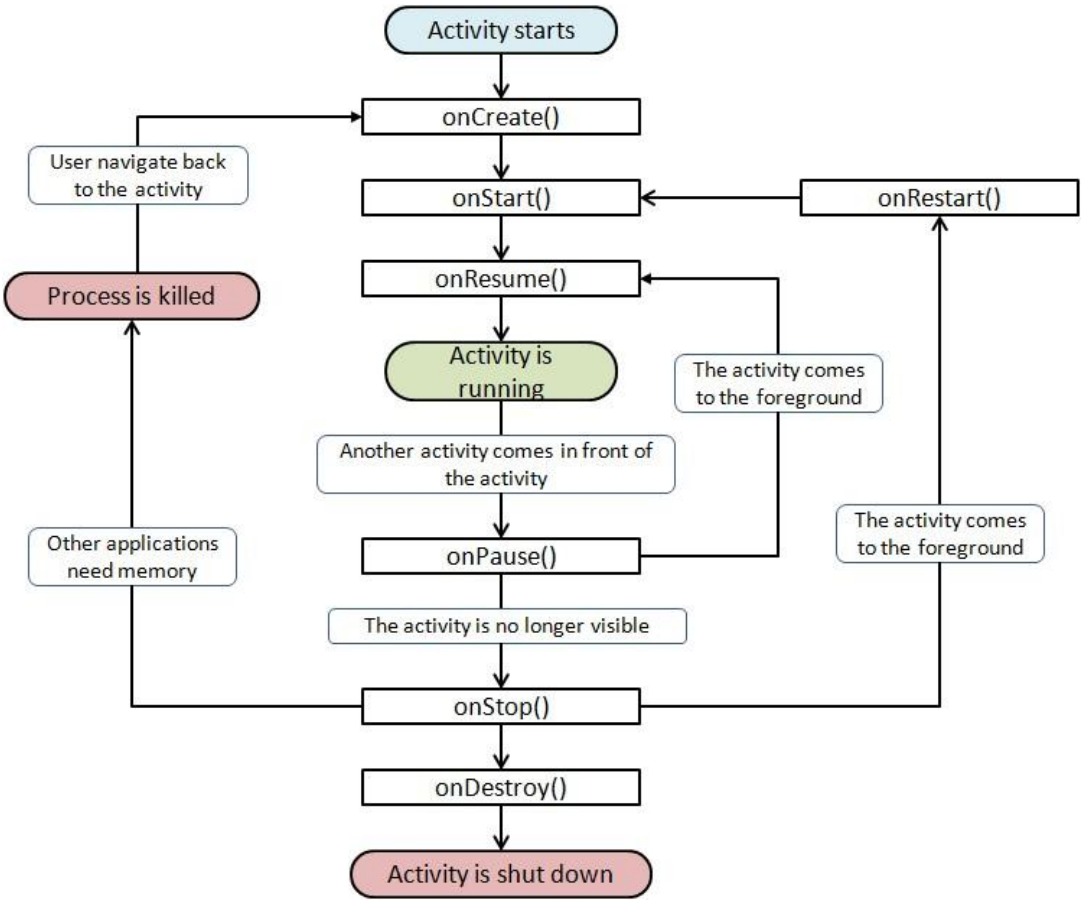



**Figure 3-4: Android Activity Lifecycle**

As the figure above, there are several lifecycle's callback methods:

*onCreate()*: Called when the activity is first created. This is where you should do all of your normal static set up — create views, bind data to lists, and so on. This method is passed a Bundle object containing the activity's previous state, if that state was captured.

*onRestart()*: Called after the activity has been stopped, just prior to it being started again.

*onStart()*:Called just before the activity becomes visible to the user.

*onResume()*:Called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it.

*onPause()*: Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns.

*onStop()*: Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it.

*onDestroy()*:Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing, or because the system is temporarily destroying this instance of the activity to save space.

Services: A Service is an application component representing either an application's desire to perform a longer-running operation while not interacting with the user or to supply functionality for other applications to use. Service facilities for the application to tell the system about something it wants to be doing in the background; and it facilities for an application to expose some of its functionality to other applications.

### 3.2.4 Service Lifecycle

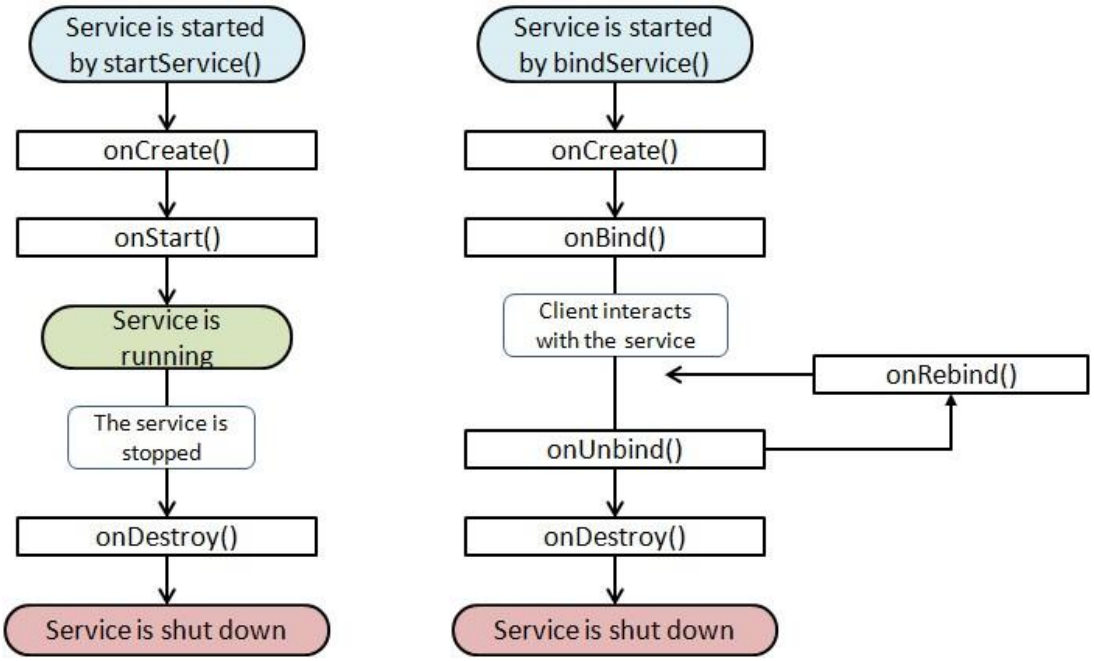



**Figure 3-5: Android Service Lifecycle**

It can be started and allowed to run until someone stops it or it stops itself. In this mode, it's started by calling Context.startService() and stopped by calling Context.stopService(). It can stop itself by calling Service.stopSelf() or Service.stopSelfResult(). Only one stopService() call is needed to stop the service, no matter how many times startService() was called. Or it can be operated programmatically using an interface that it defines and exports. Clients establish a connection to the Service object and use that connection to call into the service. The connection is established by calling Context.bindService(), and is closed by calling Context.unbindService(). Multiple clients can bind to the same service. If the service has not already been launched, bindService() can optionally launch it.

The two modes are not entirely separate. You can bind to a service that was started with startService().

### 3.2.5 Remote service

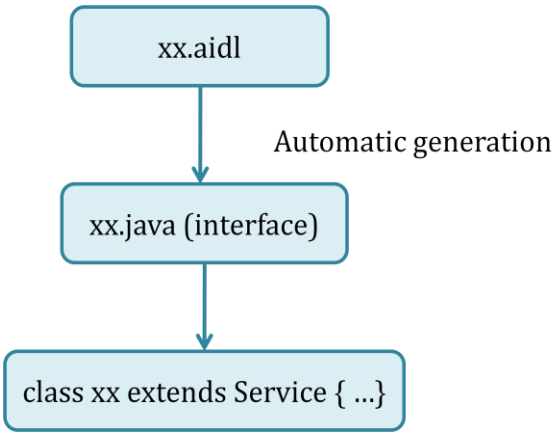



**Figure 3-6: AIDL Mechanism**

Android remote service is a Android service implemented AIDL (Android Interface Definition Language). In our design, we choose to implement Felix OSGi as an Android remote service. This is because Felix has to execute as a background task and does not need any graphical resources. AIDL defines basic operations of OSGi bundles, like start, stop, install, uninstall bundles in OSGi framework.

In Android platform, a service is generally a component that runs in background. It is defined with an ".aidl" specification called Android Interface Definition Language (AIDL) from which a Java interface is automatically generated with an abstract Stub class. The service class is the implementation of a predefined Service class that contains an internal class extending the Stub class. To make the service available to other applications, the service-class name appears in the

Service entry of the Manifest file. Additionally, Android introduces the activity notion to define a treatment associated with a physical view to interact with a user. The activity is the entry point of the application since it has a GUI. Moreover, an Intent mechanism allows the communication between applications.

## 3.3 XMPP

In this section it will introduce what is XMPP, what XMPP server we are using and XMPP APIs.

### 3.3.1 What is XMPP

Extensible Messaging and Presence Protocol (XMPP) is an open-standard communications protocol for message-oriented middleware based on XML(Extensible Markup Language).

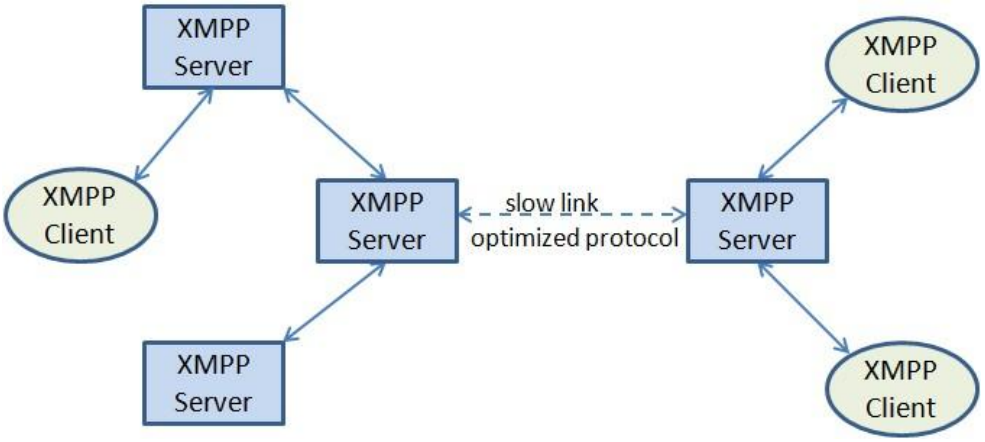



**Figure 3-7: XMPP architecture with multiple XMPP servers**

### 3.3.2 XMPP server and according API

Openfire is a real time collaboration (RTC) server licensed under the Open Source Apache License. It uses the only widely adopted open protocol for instant messaging, XMPP (also called Jabber). Openfire is incredibly easy to setup and administer, but offers rock-solid security and performance.

Smackx bundle is extended APIs for XMPP communications. This bundle exports API packages.

Smack bundle is for XMPP communication APIs. This bundle exports API packages.

## 3.4 Used mechanisms/models

In this section it will introduce the used mechanisms and models: Java reflection, Android broadcast and Android context.

### 3.4.1 Java reflection

Java reflection mechanism makes it possible to inspect and manipulate classes, interfaces, fields and methods at runtime. By accessing a particular type of metadata describing classes and objects within the JVM, Java reflection provides runtime access to variety of class information. By knowing the name of according class and name of methods, Java reflection can make it possible to call a method in a class which has been instantiated as an object.

The reason to use java reflection is that in external environment, it cannot call methods in JVM when compiling as normal function call because there isn’t access to the function. So from outside it can only try to get access to the object

instantiated from the class. And then use getmethod() and invoke to call the function.

### 3.4.2 Android broadcast

In Android system, broadcast is a widely-used mechanism for message transportation among Android applications/ services. To use Android broadcast to send broadcast, it requires wrapping data which prepares to be broadcasted, then sends the wrapped object out. To receive broadcast, a broadcast listener needs to be registered at destination. After sending the object of broadcast, all registered broadcast listener will check if the coming broadcast matches certain format, if so, the broadcast object is received.

### 3.4.3 Android context

Android context is the context of current state of the object/Android application/Android service. It is the interface to global information about an application environment. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc. An instance of Context class allows getting access to lots of managers which facilitate the applications they manage, like ActivityManager, AccountManager, WindowManager, etc. And it also has access to other services that are not named “managers”, like “Vibrator”, for interacting with hardware. It also provides access to APIs: APIs for working with permissions, cache, APIs for starting and stopping activities, etc.

# Chapter 4

# SOA-MCC

## 4.1 Overall Architecture

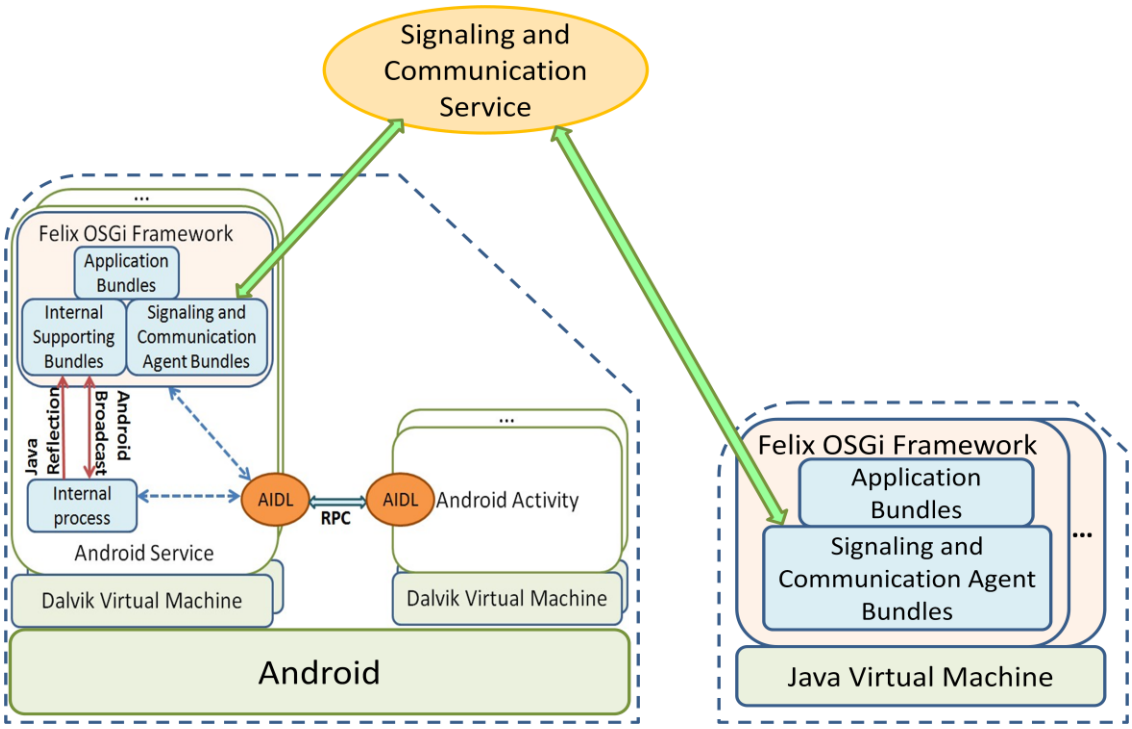



**Figure 4-1: Overall Architecture**

As figure 4-1 shows, the architecture are consisted of 3 parts: Android system, Signaling and Communication Service, and PC.

In Android system, each Android service or Android activity is on top of one Dalvik Virtual Machine. The communication between Android service and Android activity is based on RPC (remote procedure call) by AIDL. In Android service, we wrap Felix OSGi framework as part of Android service. There are two types of communication between Android service and OSGi framework: framework level communication and bundle level communication. Framework level communication allows Android service to manipulate OSGi

Framework including installing, uninstalling, starting and stopping bundles. While bundle level communication enables running bundles communicate directly with Android service by implementing internal supporting bundle, which adopts Java refection and Android broadcast mechanism to realize the communication. To send or receive message between OSGi and signaling and communication service, signaling and communication agent bundles are developed in OSGi framework.

In PC, Felix OSGi framework as a java application runs on top of Java VM. The OSGi framework contains many bundles, in which signaling and communication bundles as a signaling and communication service agent communicates with the signaling and communication service.

Signaling and communication service is a medium to dispatch message from one signaling and communication agent to another. In our development we use XMPP server located in PC as the signaling and communication service. The signaling and communication agent bundles in OSGi framework are actually XMPP client bundles, which communicate with other XMPP client through the dispatching of XMPP server.

The message transferred between XMPP client and server is in format of XML, take an example,

```
<message to='foo'>
  <body>
     Hello, World
  <body/>
</message>
```

This is a XMPP message of “Hello, World”, and the recipient of this message is ‘foo’.

In figure 4-1, if a user sends message from Android activity to VM, it will go through the following steps. After the message is input by user from user interface on Android activity side, it will be sent from Android activity to Android service through AIDL. The following paragraph describes sequence of sending a message from Android phone to PC:

User enters the message from user interface and Android activity gets the message and then sends the message to Android service. In Android service, internal process implements AIDL method to fulfill the requests from Android activity. Once receiving the message, the internal process sends the message to internal supporting bundles in OSGi framework by mechanism of Android broadcast. Then internal supporting bundles pass the message to signaling and communication service. Since signaling and communication bundles set up a communication channel with signaling and communication service, the message in XMPP message format is redirected to signaling and communication service. Once the signaling and communication service got this message, it will dispatch the message to signaling and communication bundles to other OSGi frameworks located in destination machine, which can be an Android phone or a PC.

## 4.2 Explanation of each part

In this section we introduce and explain our proposed mechanism in Android system, and describe the bundles we developed and bundles dependency. At last, the sequence diagram is present to explain the sequence of initiating and mutual communication between Android activity and OSGi.

### 4.2.1 Proposed mechanism in Android system

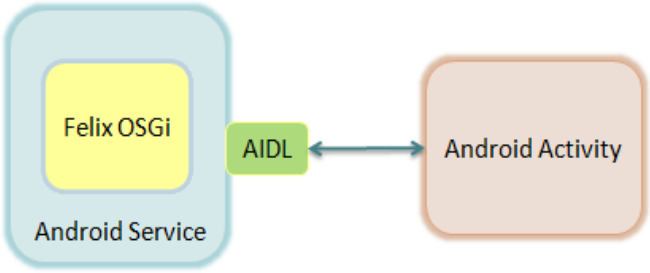



**Figure 4-2: Bundle operation from activity to service**

In our design, OSGi framework is wrapped as an object in Android service, which runs at back end of Android system consistently unless it's explicitly called to stop service. As former explanation, Android activity provides user interface and thus, user can manipulate Android service through user interface. To connect Android activity and Android service, there is an interface call AIDL between Android activity and Android service. In our design every Android service which will be accessed by Android activity has an AIDL (Android Interface Definition Language) defining the interface to manipulate bundles in OSGi framework.

As figure 4-2 shows, every function call from Android Activity goes from Android activity to Android service by RPC (remote procedure call). AIDL defines methods for manipulations of OSGi bundles, like starting, stopping, installing, and uninstalling bundles. And Android service implements the methods that AIDL defines, and calls the methods defined in Felix OSGi framework to operate the bundles. So from Android activity, it calls interface of OSGi framework in AIDL using RPC, and then Android service call the method of implemented interface in OSGi framework to complete the according manipulation.

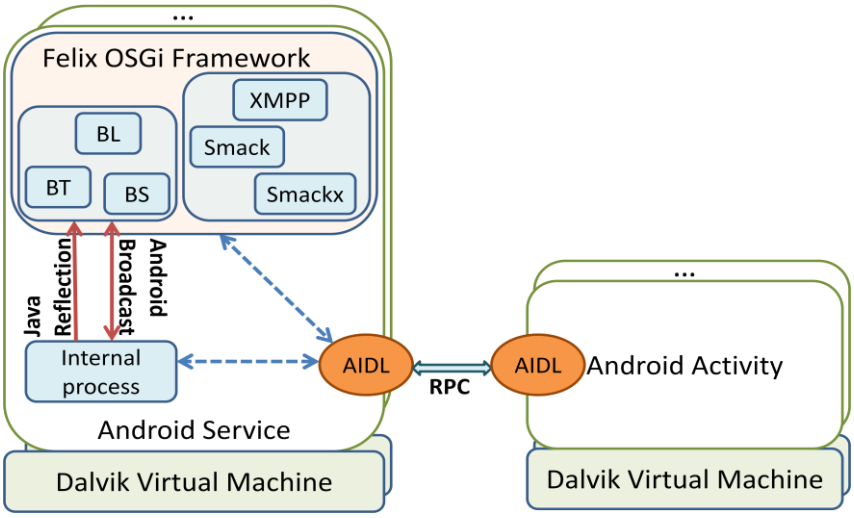



**Figure 4-3: Proposed Mechanism in Android System**

In our design, there are two types of communication between Android service and Felix OSGi framework. One type of communication is framework level communication, by which Android service calls OSGi framework APIs to manipulate OSGi bundles to start, stop, install and uninstall bundles. In Figure 4-3, the upper dotted line between AIDL and Felix OSGi framework shows

framework level communication between OSGi framework and Android service.

The other type of communication is Bundle level communication. To enable the bundle level communication between OSGi bundles and Android service, we develope internal supporting bundles with mechanism of Java reflection and Android broadcast inside OSGi framework, as well as implement internal process inside Android service. With internal supporting bundles, which include BL (BroadcastListener) bundle, BS (BroadcastSender) bundle, and BT (BroadcastTest) bundle, messages sent by BS bundle or received by BL bundle are transferred between OSGi framework and Android service using Android broadcast.

In bundle level communication, internal process inside Android service works as a bridge between internal supporting bundle in Felix OSGi framework and Android activity. Internal process is the middle point of OSGi bundles and Android activities. Once message reaches BS bundle, it will send an Android broadcast to internal process, which has an Android broadcast listener. Then the internal process will render this message to Android activity user interface using AIDL. In reverse, Android activity fetches the message from user input and calls internal process inside Android server through AIDL, after internal process gets the message, it will send an Android broadcast to internal supporting bundles, BL bundle specifically.

### 4.2.2 Bundle description

There are three bundles in internal supporting bundles: BS (broadcastsender) bundle, BL (broadcastlistener) bundle and BT (broadcasttest) bundle.

BL bundle: named as broadcastlistener bundle, this bundle is going to register an Android broadcast listener used to listen to Android broadcast which are sent to OSGi framework from internal process inside Android service. On receving broadcast, BL bundle will dispatch the message to other service running inside OSGi framework, in our case, it will be sent to XMPP client bundle. This bundle initiates listener instance and attach the listener to context to get action of context.

BT bundle: named as broadcasttester bundle, this bundle holds an Android context in it. Android context is interface to global information about an application environment. This is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc. And the Android context is set up at the moment Android service created through Java reflection mechanism. After this bundle setting up the Android context field, it provides service of getters and setters for Android context, allowing other services to access this field.

BS bundle: named as broadcastsender bundle, this bundle is used to send broadcast from OSGi framework to Android service. Before sending the broadcast to Android service, the broadcastsender bundle gets the context of destination process from broadcasttest bundle which holds the according

context. With the context, BS bundle send the broadcast to destination internal process in Android service.

There are three bundles in signaling and communication agent bundles: XMPP bundle, smack bundle and smackx bundle.

XMPP bundle sends message to XMPP server and receive message from the XMPP server. After receiving message, XMPP bundle will pass the message to broadcast sender bundle to preparing for next step message transportation. Also to send message to XMPP server, XMPP bundle gets message from broadcast listener bundle and then sends the message to XMPP server using according APIs which supported by smack bundle and smackx bundle.

Smack bundle wraps smack API, which provides XMPP instant messaging and presence APIs, as a bundle. This bundle exports API packages of XMPP client library.

Smackx bundle wraps smacks API, which is extended APIs for XMPP communications, as a bundle. This bundle exports API packages.

### 4.2.3 Bundle dependency

Dependency is the degree to which each bundle relies on each one of the other bundles. In our design, we divide application logically into independent modules and each module is served as a bundle in OSGi framework. Although each module/bundle is designed to be as decoupled as possible to others, dependency relationship between different bundles still exists. For example, it's harmless to run sole internal supporting bundles, but this will cause unfeasibility to communicate with signaling and communication service due to

the lack of signaling and communication agent bundles. Though each feature could be an independent bundle, their dependency relationships still exists regarding as whole application logic.

In our design, we separate the bundles into internal supporting bundles and signaling and communication agent bundles by functionality. For internal supporting bundles which serve for communication with internal process in Android service contains BT, BS, BL bundles, while signaling and communication agent bundles have smack, smackx and xmppclient bundles served for communication with signaling and communication service. Figure 4-4 below defines the dependency relationships in our design.

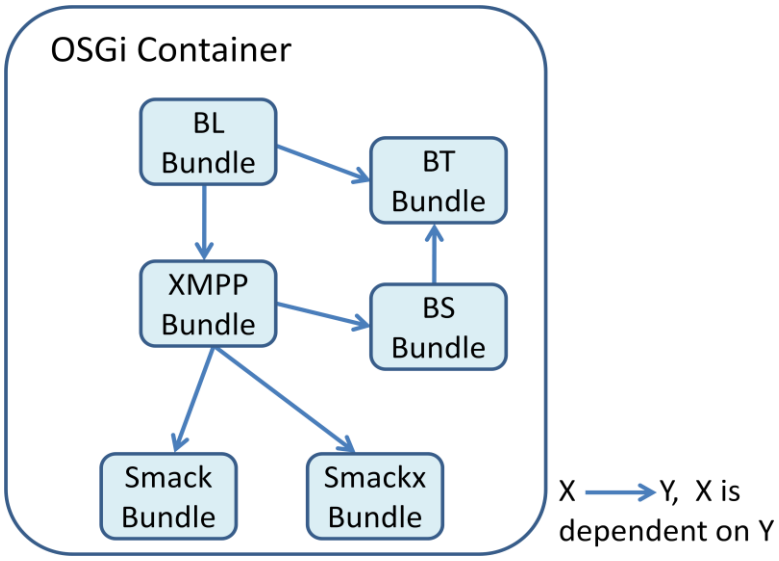



**Figure 4-4: Dependency of Bundles**

Dependency Relationship Explanation:

BL Bundle → BT Bundle: BL bundle has dependency on BT bundle. Since BT bundle holds the service to provide getter for Android context, BL bundle

needs to create an Android broadcast listener and attach this listener to Android context by using the getter service. Therefore, BL bundle is dependent on BT bundle.

BL Bundle → XMPP Bundle: BL bundle has dependency on XMPP bundle, because after receiving android broadcast from internal process in Android service, BL bundle has to dispatch the message in broadcast to signaling and communication service, while XMPP bundle serves to send message, BL bundle will need get XMPP bundle service first and using send message feature of this service.. This makes BL bundle has dependency over XMPP bundle.

XMPP Bundle → BS Bundle: XMPP bundle is dependent by BL bundle because of sending message to signaling and communication service. On the contrary, regarding to the receiving message, XMPP bundle needs to depend on BS bundle, since BS bundle holds service to send Android broadcast. On purpose to send the Android broadcast which the message is wrapped into, BS bundle should first retrieve the BS bundle service, then use it to send broadcast to internal process in Android service.

BS Bundle → BT Bundle: BS bundle can send Android broadcast to internal process using the Android context, since only the Android context can send broadcast in Android. Due to that Android context is stored in BT bundle, and

getter and setter service are provided by it, it's natural that BS bundle need BT bundle service to get Android context and use it to send broadcast.

XMPP Bundle → Smack Bundle: Smack bundle is a standard XMPP library. XMPP bundle depends on Smack bundle and utilizes it for sign-in, sign-out, sending message as well as receiving message.

XMPP Bundle → Smacks Bundle: Smackx bundle provides extra features for XMPP, and XMPP bundle relies on it.

### 4.3 Solutions for problems in communication between Android service and OSGi framework

To realize the integration for Android and Felix OSGi, it is necessary to enable the communication between Android service and Felix OSGi framework in bundle level. The following reasons explain why communication between these two sides is necessary. Firstly, some services published by OSGi bundles should be accessible to users, allowing them to consume the services. Secondly, some bundles needs to render information to users for decision, for example, if xmppbundle receives other user's message, it should be able to pass this message to Android-OSGi services.
Based on prior 2 considerations, we need bi-directional communications between Felix OSGi framework and Android-OSGi Service. From Android

Application to OSGi framework we use java reflection method, and in the reversed direction we use Android context/broadcast methods.

To enable the communication between Android service and OSGi framework, the initialization and bi-direction communication problems need to be solved. And the following solutions solve the problems respectively.

### 4.3.1 Solution to initializing Android context in OSGi framework

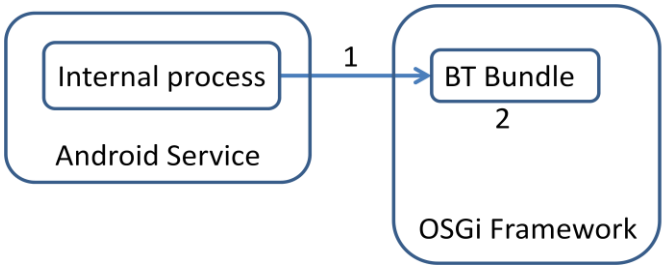



**Figure 4-5: Initialization of communication: set Android context in OSGi**

Because only by setting Android context in OSGi framework can OSGi bundles send broadcast to Android services. Thus, at the beginning, internal process start the BT bundle with communication in framework level, with the BT bundle running internal process set “context” instance to BT bundle by Java Reflection. Then BT bundle this instance and provide set() and get() method to allow other bundles access and use this instance to send or receive broadcast.

### 4.3.2 Solution to communication from Andorid service to OSGi framework

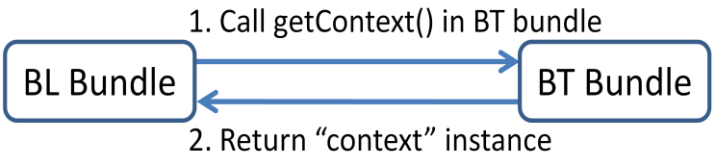



**Figure 4-6: BL bundle get context before sending broadcast**

In order to receive broadcast in Android, it’s necessary to register a broadcastlistener to a context. To register the listener:

Step (1) BL bundle call getContext() in BT bundle.

Step (2) BT bundle return “context” instance.

Step (3) In BL bundle, it registers a broadcast listener to “context” instance, the broadcast listener specifies what kind of broadcast is going to be listened to.

Once the broadcast listener is registered, the specific broadcast will go to BL bundle. Here is the flowchart for how Android service sends a broadcast to XMPP bundle:

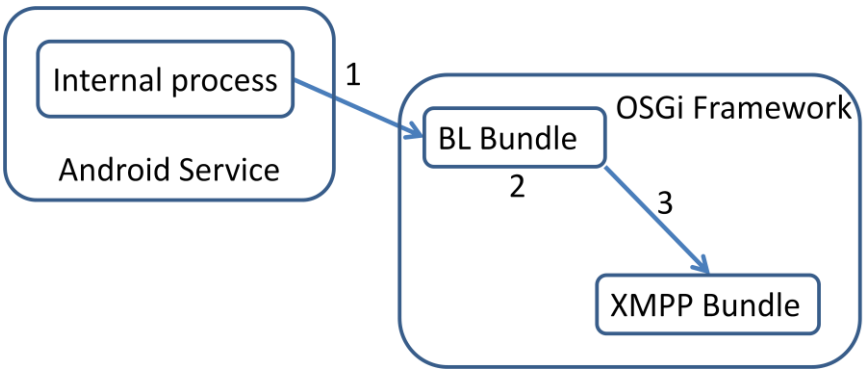



**Figure 4-7: Internal process sends a broadcast to XMPP bundle**

Step (1) Internal process in Android service sends a broadcast, and BL bundle receive this broadcast.

Step (2) BL bundle checks and know this broadcast is sent to XMPP bundle, then BL bundle dispatches the message to XMPP bundle.

Step (3) XMPP bundle extracts useful information in the broadcast and wraps the information as an XMPP message, then sends the XMPP message to XMPP server.

### 4.3.3 Solution to communication from OSGi framework to Android service

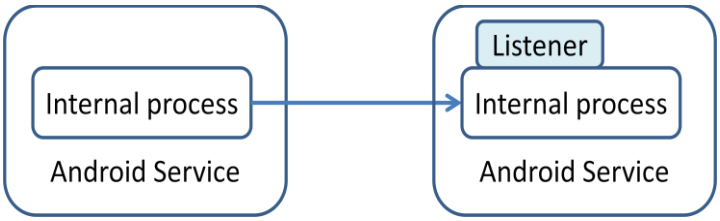



**Figure 4-8: Register listener on internal process**

In order to receive broadcast in internal process, a listener needs to be registered on it. Once the broadcast listener is registered on internal process, XMPP will use the sendBroadcast() in BS bundle. Before BS bundle sends message to internal process, BS bundle needs to get the "context" instance from BT bundle. Then BS bundle uses "context" instance to send broadcast to internal process. Here the flowchart.

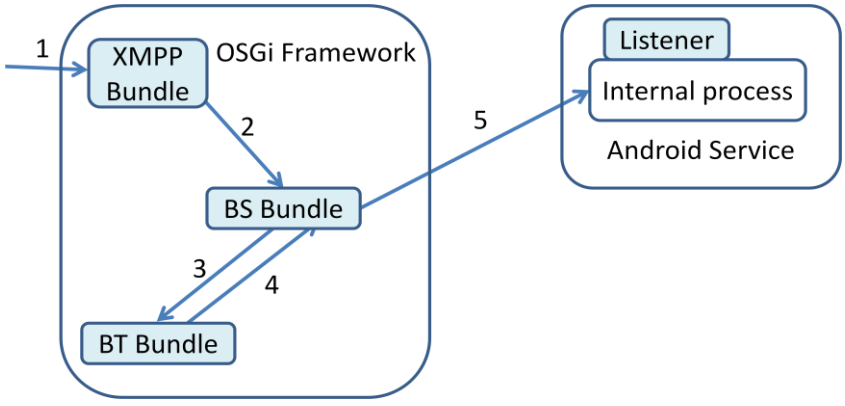



**Figure 4-9: Send broadcast from XMPP bundle to internal process**

Step (1) XMPP bundle received message from XMPP server.

Step (2) XMPP bundle wrap message as a broadcast instance and the received message is included in the broadcast instance. Then XMPP bundle call BS bundle to send this broadcast to internal process.

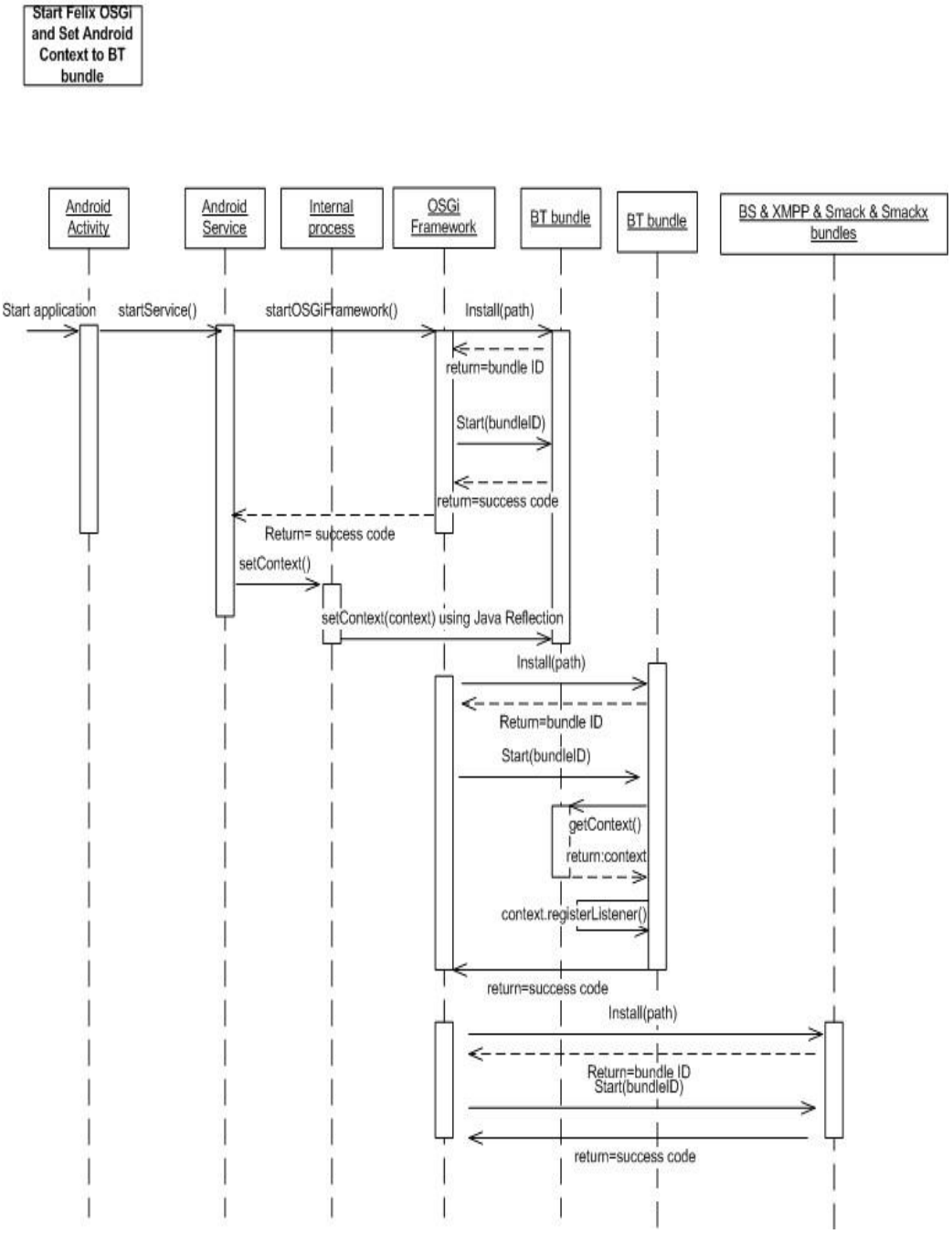
Step (3) BS bundle request “context” instance from BT bundle to get more information about the broadcast and recipient of the broadcast.

Step (4) BT bundle checked according “context” instance and return it to BS bundle.

Step (5) BS bundle use the context instance to send broadcast, and the listener located in internal process will receive it.

## 4.4 Sequence diagrams

### 4.4.1 Start Felix OSGi and set Android context to BT bundle

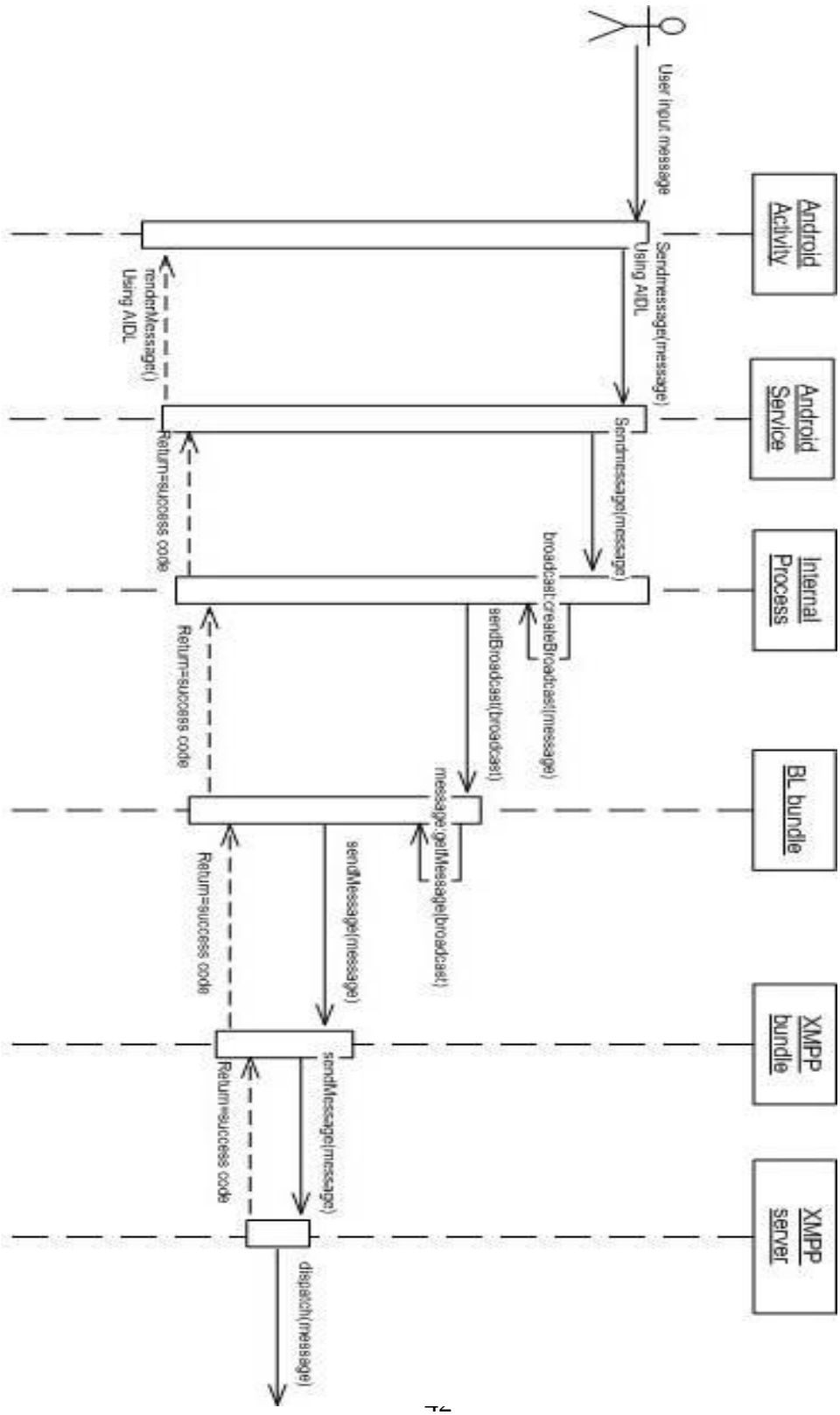


The above diagram describes the process of initializing the communication from Android service to OSGi. After the initializing, OSGi framework is started and Android context is set to Broadcast Test bundle in OSGi.

From the diagram, we can see user starts the Android application through Android activity, then Android activity starts the Android service by “startService()”. Then Android service starts OSGi framework by framework-level communication between Android service and OSGi. And next OSGi framework installs and starts broadcast test bundle. Then Android service calls setContext() function by java reflection to set its Android context to broadcast test bundle. So broadcast test bundle holds the Android context of Android service.

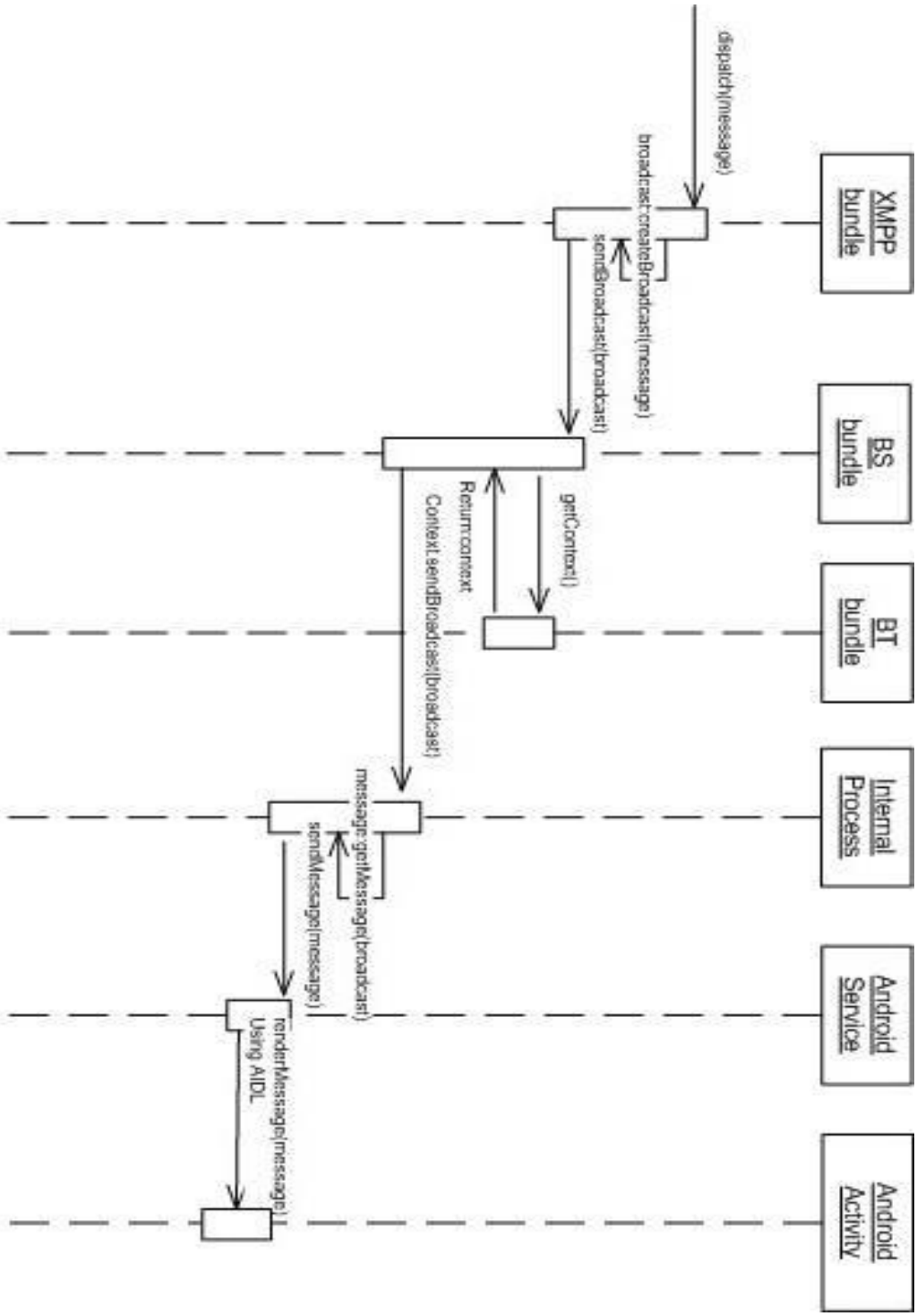
After setting context into broadcast test bundle, other bundles are started at the initialization moment. So OSGi framework also starts broadcast sender bundle, broadcast listener bundle, and XMPP bundle, etc.

4.4.2 Send message from Android activity to XMPP server



This diagram shows the process of a message sending from user to XMPP server. First user inputs a message from user interface by Android activity. Android activity sends the message to Android service through AIDL by RPC. AIDL passes this message to internal process, which locates in Android service and communicates directly with OSGi. Internal process sends the message as a broadcast, and Broadcast Listener bundle in OSGi framework listens to this broadcast and receives it. Then Broadcast listener bundle dispatches this broadcast to XMPP bundle. At last XMPP bundle wraps this broadcast into a XMPP message and send it to XMPP server.

4.4.3 Receive message from XMPP server and display it at Android activity



This diagram describes the transferring of a message from XMPP bundle to Android activity.

XMPP bundle gets the message from XMPP server and then wraps the XMPP message as a broadcast. Then XMPP bundle sends this broadcast to Broadcast Sender bundle. Broadcast Sender bundle requests Broadcast Test bundle by method getContext() in order to get context of Android service. After Broadcast sender bundle getting context of Android service from Broadcast test bundle, broadcast sender bundle sends the broadcast to Android service. Then Android service sends the broadcast to Android activity through AIDL by RPC. So user will get the message from GUI provided by Android activity.

# Chapter 5

## PERFORMANCE AND EVALUATION

To implement SoA-MCC model, we established the SoA-MCC architecture in MobiCloud system. As Figure 4-1 shows, the architecture is consisted by three parts: mobile device, signaling and communication service and PC. For signaling and communication service, we deployed an XMPP server in a computer, and for PC part, we established a OSGi framework in PC and developed an XMPP client in bundle format in OSGi framework. For mobile device part, we developed internal supporting bundles to support the fully bundle-level integration of OSGi framework and Android system. We also developed an application which runs on top of OSGi framework.

This application is a bundle transferring application based on support of internal supporting bundle and signaling and communication agent bundle. The application can send a “Helloworld” bundle which runs properly on both Android OSGi framework and PC OSGi framework.

To evaluate the performance of the application, I tested the responding time of bundle operation and application operation by embedding time measuring code in the Android application code.

The test bed for this performance evaluation is: Android Galaxy SII, Android version 2.3.6, Kernel version 2.6.35.11, memory 700 MB.

In PC, test bed is ThinkPad X220i, Intel Core i3, memory 4GB.

5.1 Bundle operation performance

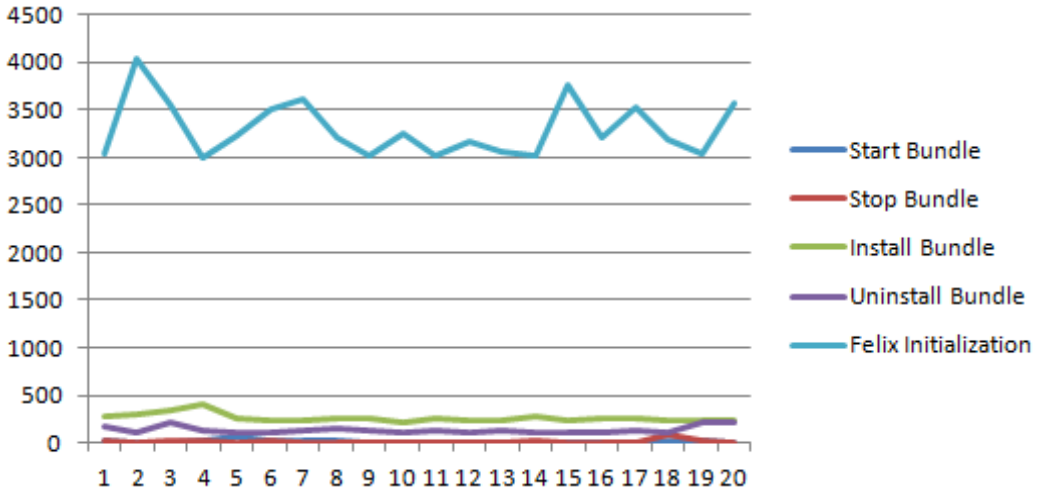



**Figure 5-1: Bundle operation time**

| **Bundle Operation** | **Average Time /ms** |
|---|---|
| Start Bundle | 14.8 |
| Stop Bundle | 170.45 |
| Install Bundle | 262.15 |
| Uninstall Bundle | 137.3 |
| Felix Initialization | 3302.65 |

**Table 5-1: Bundle operation average time**

We can see from figure 5-1 and table 5-1, Felix initialization time is about 3 seconds, other bundle operation time is much lesser. Felix initialization occurs once when Android service is started. But bundle operations are quite frequent when application runs.

## 5.2 Application Performance Evaluation

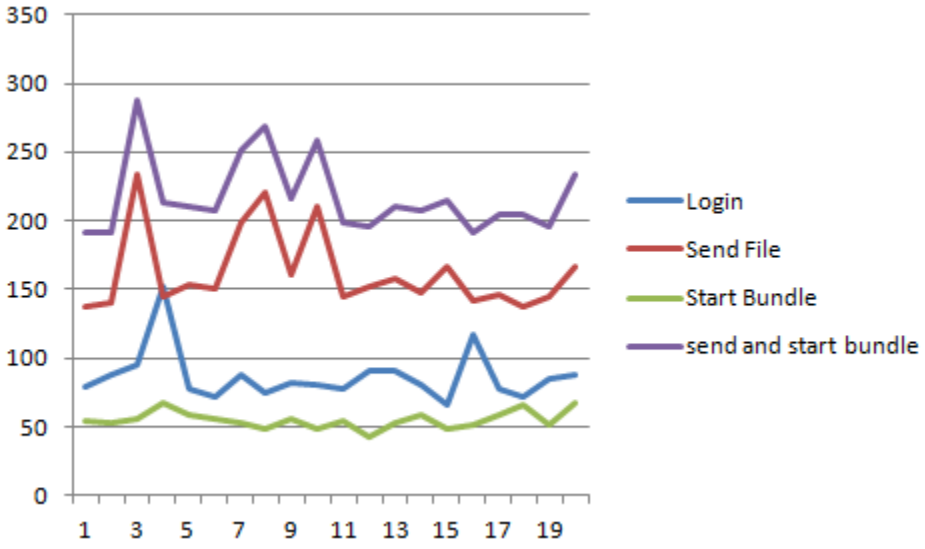



**Figure 5-2: Application operation time**

| Application Operation | Average Time/ms |
|---|---|
| XMPP user login | 86.5 |
| Send file (Size:4kB) | 162.65 |
| Start bundle | 54.9 |
| Send and start bundle | 217.55 |

**Table 5-2: Average application operation time**

The service of this application uses memory 14MB in average in Android.

From figure 5-2 and table 5-2, we can see that the time of sending bundle and starting the bundle in PC is less than 1 second. The internet in the test is local area network.

# Chapter 6

# CONCLUSION

## 6.1 Conclusion of current work

To solve the problem that there's no existing solution to bundle-level communication between OSGi bundles and Android services, we proposed a solution that realized the communication with developing several internal supporting bundles supporting this communication. Based on this solution, we established the SoA-MCC architecture among mobile devices and computers. Further, to prove that our architecture is fitful for mobile cloud environment, we put our architecture in MobiCloud system and make OSGi framework works on virtual machines which maintained by Mobicloud VM pool.

Our proposed model, SOA-MCC is consisted by three parts: Android, signaling and communication service, and PC. The contribution of this thesis can be describes as follows (1) it implements the communication in bundle level between OSGi framework and Android native processes; (2) it does not change OSGi original framework and thus OSGi bundles can run in Java Running Environment (JRE) without compatibility issues; (3) it support mobile cloud computing in that allowing OSGi bundles as cloud services to be reused among OSGi frameworks; (4) it is a lightweight implementation and consumes limited memory and power on mobile devices.

## 6.2 Further work

In further research, we will improve the current model to enable the bundle transferring among OSGi platforms which located in different operating systems. Also we will research on the XMPP server to fulfill group and individual authentication of bundles which belong to different users. And also the download of bundles from XMPP server can be considered in future work.

REFERENCES

[1] Woon Yong Kim, Seok-Gyu Park, *The 4-Tier Design Pattern for the Development of an Android Application*, Lecture Notes in Computer Science, 2011, Volume 7105/2011

[2] L. Zhang and Q. Zhou, *"CCOA: Cloud computing open architecture,"* in *Proc. IEEE Int. Conf. Web Services*, 2009, pp. 607–616.

[3] Android website, http://www.android.com/

[4] News: Android takes the largest portion in smart phone market, http://blog.laptopmag.com/report-android-claims-largest-portion-of-smartphone-market

[5] Tergujeff, R., Haajanen, J., Leppanen, J., Toivonen, S. *Mobile SOA: Service Orientation on Lightweight Mobile Devices*, icws, pp. 1224-1225, *IEEE International Conference on Web Services* (ICWS 2007), 2007

[6] About the OSGi Service Platform, Technical Whitepaper. http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf, June, 2007.

[7] OSGI Alliance. OSGi. *The Dynamic Module System for Java*. www.osgi.org.

[8] S. Pack, K. Park, T. Kwon, and Y. Choi, *"SAMP: Scalable Application-Layer Mobility Protocol," IEEE Communications Magazine*, no. June, pp. 86-92, 2006.

[9] Wolf B, Rosjat M. *A Dynamic OSGi-based Data Stream System[C]*. MDS'08: Leuven, Belgium, 2008: 31-36.

[10] A. Ennai and S. Bose, *"MobileSOA: a service oriented web 2.0 framework for context-aware, lightweight and flexible mobile applications,"* EDOCW, 12th Enterprise Distributed Object Computing Conference Workshops, pp.345-352, 2008.

[11] Erl, T. *SOA: Principles of Service Design.* Prentice-Hall, Englewood Cliffs

(2007)

[12] EZdroid website, http://www.ezdroid.com/

[13] mBs website, http://www.prosyst.com/index.php/de/html/content/49/mBS-Mobile-for-Android/

[14] Felix by Apache, http://felix.apache.org/site/index.html

[15] D. Huang, X. Zhang, M. Kang, and J. Luo, *Mobicloud: A secure mobile cloud*

*framework for pervasive mobile computing and communication*, in Proceedings of 5th IEEE International Symposium on Service- Oriented System Engineering, 2010.

[16] Dijiang Huang, Zhibin Zhou, Le Xu, Tianyi Xing, and Yunji Zhong, *Secure Data Processing Framework for Mobile Cloud Computing*, In Proceedings of IEEE INFOCOM's Workshop on Cloud Computing, 2011.

[17] Dijiang Huang, Vetri Arasan. *Email-based Social Network Trust*. Proceeding SOCIALCOM'10 Proceeding of the 2010 IEEE Second International Conference on Social Computing.

[18] S. Bouzefrane , D. Huang , P. Paradinas, *An OSGi-based Service Oriented Architecture for Android Software Development Platforms*, April 2012, Paris, France

[19] OSGi Alliance website, http://www.osgi.org/Main/HomePage