Integrating Application-Oriented Middleware into the Android Operating System

Julian Kalinowski and Lars Braubach

Computer Science Department, University of Hamburg
Distributed Systems and Information Systems
Hamburg, Germany
Email: {kalinowski|braubach}@informatik.uni-hamburg.de

***Abstract*—As mobile devices are becoming more advanced in technology, the type of software they are able to process develops from simple apps to complex applications. Fortunately, a main area in software engineering research is dedicated to examining the handling of complexity. The common approach of adding abstraction layers is embodied in various middleware solutions, including application-oriented middleware that feature generic abstractions for decomposition and distribution as well as support for non-functional criteria and higher-level concepts in programming. However, embedding middleware into a mobile operating system environment bears many challenges. The several attempts of porting a middleware to Android have only been partially successful, as they either require developers to use an uncommon programming language or abandon the well-proven Android design principles. We propose a universal architecture for integrating middleware into the Android operating system while maintaining the core features of the Android application framework. The presented architecture provides the shared use of middleware libraries during runtime as well as a middleware execution platform for shared use of different apps and an event-based mechanism for middleware/android component coupling.**

Keywords–Android; Middleware; Mobile Applications; Software Agents.

I. INTRODUCTION

The possibilities of mobile applications increase with the advent of faster hardware, bigger screens and more stable broadband connections. This ongoing evolution of mobile computing leads to larger applications and increases the need for methods reducing software complexity [1]. While reducing complexity has played a central role in software engineering right from the beginning, there is still no silver bullet; complexity can only be coped with abstraction [2]. Several types of middleware can aid software developers by providing some of the abstractions that are needed to create nowadays programs.

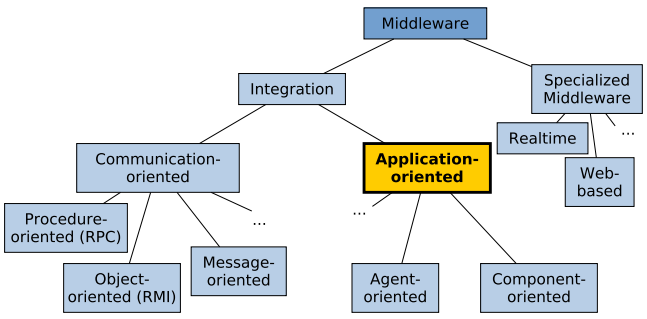

Figure 1. Middleware taxonomy, based on [3].

A helpful categorization of middleware is shown in Figure 1. This paper focuses on a subset which we call *application-oriented middleware*. This subset includes component and agent-oriented middleware. In addition to communication, they provide support for decomposition or other generic programming abstractions; thus supporting application development in multiple aspects.

Android, as well as other mobile operating systems, has been developed with limited resources in mind. Processing performance, available memory, and battery capacity have been considered at system level. In consequence, the system was designed to run rather small, self-contained applications or *apps*, which are executed in separate virtual machine (VM) environments for security and safety reasons [4]. App developers are restricted to fixed design principles to make sure their application integrates properly with the operating system. For example, apps have to be split up in activities and services, depending on whether the given part represents user interface (UI) or executes background tasks. Activities and services are subject to a specific life cycle, which is executed by Android. Acting as a framework, Android is allowed to start, stop, pause and resume apps if it needs to for various reasons such as limited resources, user interaction or even an incoming phone call.

The strict requirements for application developers lead to limitations in software architecture design and the integration of middleware in particular. As applications are executed in different VM processes and the developer cannot influence application loading, it is not possible to share libraries in a convenient and secure way [4]. Nevertheless, as apps get bigger and use more libraries, the possibility of two apps sharing some code increases. Middleware, in contrast, is built to handle more than one running application and thus supports access to common functionality by design.

Furthermore, since the Android system determines the way applications are loaded, instantiated and started, there is little chance for the application to influence mechanisms like class and resource loading. Application-oriented middleware, however, form an abstraction layer between operating system and applications as shown in Figure 2. This usually requires the use of specific classloading or startup mechanisms, as middleware provides a runtime environment called *platform* to control the life-cycle execution of runtime application components [5][6].

This paper presents an architecture that deals with these challenges and integrates a middleware platform within an Android application; to be used and accessed by *client applications* and achieving a higher level of abstraction during application development. As the presented architecture is independent

 ISBN: 978-1-61208-418-3

of a specific middleware implementation, it is conceptually applicable to various middleware.

The article is structured as follows: Section II introduces the requirements we would like to fulfill. Section III provides an overview on related work. Section IV explains the challenges of the Android operating system regarding middleware integration. Section V describes the architecture we propose to cope with these challenges. Section VI presents a prototypical implementation of the architecture, which is then evaluated in Section VII. Finally, Section VIII concludes the paper and gives an outlook on future work.

II. Requirements

In contrast to embedding a middleware into an Android *application* [7][8], the goal of this paper is the integration of middleware into the Android operating system as an abstraction layer to be used by *other applications* as shown in Figure 2.

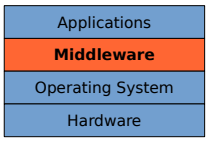

Figure 2. Middleware layer between applications and operating system, based on [9, p. 23].

This goal leads to the following functional requirements:

1) *Independent deployment:* The middleware is deployed independently of other applications and can be used by *client applications* (which in turn do not need to include middleware libraries).
2) *Multi-client capable:* Multiple applications can run on a single instance of the middleware.

As outlined in the introduction, the Android application framework introduces several design principles to simplify application development in the context of mobile devices. These principles should obviously still be applicable when developing applications using an integrated middleware. Application-oriented middleware can provide additional programming approaches, such as agent-orientation, which should be available, too. Furthermore, components developed with middleware concepts should be able to interoperate with Android application components. This leads to the following criteria:

3) *Concept integration:* Concepts of both the integrated middleware and the Android operating system should be available to the developer.
4) *Component coupling:* Components running on the middleware and components based on Android paradigms must be able to communicate easily.

As non-functional requirements cannot generally be fulfilled by an integration architecture, but are determined by the concrete middleware and application, they will not be considered here.

III. Related Work

In this section, other work regarding the use of middleware on mobile operating systems is reviewed with respect to the requirements given in the previous section.

A. Component-Oriented middleware

Component-oriented middleware realizes the idea of interchangeable and reusable software components. They implement a component model, which defines syntax and semantics of component definitions and their relations [10]. Several approaches, all based on OSGi [11], have been proposed for the use on Android devices.

Equinox was originally developed to provide a plugin-based architecture for the Eclipse IDE. In the progress of evaluating the application of Equinox on Android devices, necessary changes were added by Hargrave and Bartlett in 2008 [12]. For the time being, Equinox does not provide a concept for integrating UI. In consequence, it is uninteresting for many real-world scenarios.

The Apache OSGi implementation *Felix* supports execution on Android since version 1.0.3. It is possible to use the Felix command line shell to add bundles and run console applications, just as with Equinox. Furthermore, Felix can be embedded in Android Apps and executed during the initialization of an app [7]. Based on this approach, Escoffier showed how to create Android apps that dynamically load *.jar* bundles. Felix uses a special *ViewFactory* Interface for creating application UI from within any bundle [13].

The commercial OSGi implementation *ProSyst mBS* was designed for embedded hardware, and features explicit support for Android devices. As in Felix, application components are deployed as *.jar* bundles and can contain UI, which has to be implemented using the interface *ApplicationFactory* instead of activities. As opposed to the previously described OSGi implementations, the ProSyst platform is deployed inside a standalone Android application. To launch an individual application, a dummy app is installed on the device, instructing the platform application to load a specific application bundle and display its UI [14]. This execution model enables sharing of the middleware platform between applications, while keeping the original user experience.

B. Agent-Oriented middleware

Software agents provide a high-level approach to implement complex and concurrent software systems. In order to use such an abstraction, a runtime environment (*platform*) is required to provide services, e.g., for executing or discovering agents.

JaCa-Android [8] was specifically developed for Android and combines the *Agents & Artifacts* paradigm with an agent runtime called *CArtAgO* [15]. Agents are implemented using *Jason*, an AgentSpeak implementation [16]. The runtime model is based on embedding the runtime platform into applications, including a central *JaCa-Middleware* application, which provides several artifacts to enable using services like contact management, localization or SMS from within agents. User interfaces are developed using default Android activities and are represented to the agents as *artifacts* to enable communication between them. The Agents & Artifacts approach allows for an elaborated integration of agent and Android design principles, but introduces an implementation language that is very different from traditional languages.

Another agent-oriented middleware is *JADE*, which also features an Android version. *Jade-Android* can either integrate with a back-end or be executed as standalone platform. In any case, the runtime platform is included in applications;

 ISBN: 978-1-61208-418-3

increasing the application sizes and loading times. Agents can communicate with Android activities using the *Object-to-Agent Interface (O2A)*. O2A utilizes Android intents sent by agents and received by activities as well as Java interfaces, which are used by activities to call agent methods [17].

TABLE I. FEATURE OVERVIEW.

	Independent deployment	Single instance	Concept integration	Component coupling
Equinox	+	+	-	-
Felix	-	-	o	-
ProSyst	+	+	o	o
JaCa	-	-	+	+
JADE	-	-	+	+

Legend: +: supported, o: partly supported, -: not supported.

In Table I, all previously described approaches are compared in respect to the requirements stated in Section II. It can be seen that no approach is able to fulfill all requirements to a satisfactory degree.

IV. CHALLENGES OF THE ANDROID OPERATING SYSTEM

Several properties of Android prevent middleware developers from simply porting and using a Java SE middleware. In order to fulfill all the requirements mentioned in Section II, integrating the middleware into the Android operating system has to be done with great care. We will focus on three of the most critical characteristics of Android that have to be considered.

First, Android implements an effective way to run every application on its own virtual machine. To avoid loading core libraries twice, a central VM process called *Zygote* is used to load them into memory. This process is forked each time a new VM is needed, providing every running VM access to the previously loaded core libraries [18]. This works fine for sharing Android core libraries, but the process separation prevents applications from sharing common libraries at runtime. For the integration of middleware, this is turning into a problem: As per requirement #1, we want middleware and client applications to be deployed independently; but at the same time run multiple applications on one middleware instance (#2). To achieve this, every running client application must have access to classes which are included in the middleware application.

Second, Android applications, specifically their activities and services, utilize a preset life cycle. While dynamic library sharing is impossible due to the above-mentioned process separation, this also complicates static sharing, i.e., using the same filesystem copy of a library. An Android application's entry point is an activity or application object [19, p. 75], which is instantiated before developers gain control of execution. In particular, replacing or modifying the class loader that is used to load the application's activities and services is not possible, which renders statical sharing of libraries a hard problem.

In Figure 3, the control flow of an application with two activities is shown. The first activity requests startup of the second activity in its `onCreate()` method; passing control to the system. This is why some implementations from Section III don't allow the use of activities to implement UI, but rather provide special interfaces to be implemented by the application

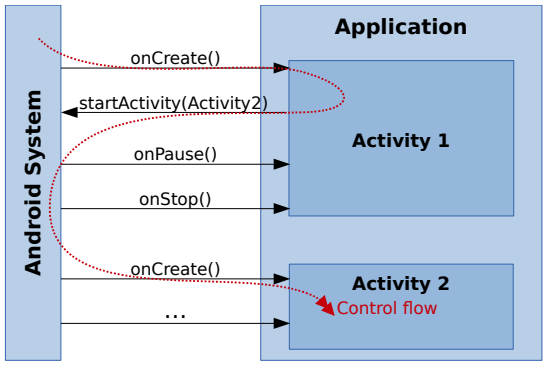

Figure 3. Control flow during application startup and activity change.

developer. This way, they can ensure the right class loader is used to call the interface methods.

Third, Android introduces a simple way to access *Resources* from code. Resources can be images, UI layouts, string values and arbitrary binary files such as sounds. At compile time, every resource is assigned an id, which is stored in the generated class `R`, pointing to the correspondent file. For resolving resources at runtime, Android automatically uses the `R` class belonging to the current application context. In consequence, loading an application-external class that contains resource ids (e.g., UI), which is what middleware has to do to show application-specific UI, will usually fail. This is mirrored by the fact that in current implementations which allow sharing of middleware libraries, it is not possible to use resources such as XML-UI layouts (see Section III).

V. ARCHITECTURE

Our middleware-embedding architecture is based on separate Android applications with no additional .jar deployment, as this would differ from default Android concepts (requirement #3). One application provides the middleware runtime and contains all middleware-specific classes and libraries. This *middleware app* can then be called from *client apps*, providing access to middleware libraries and functionality. While keeping both application types separated at deployment time, our architecture executes client apps in the middleware process to allow for sharing of middleware libraries at runtime. To preserve Android and middleware concepts, the proposed architecture does not allow to create or modify UI inside of middleware components. Instead, middleware and Android components are created separately, while a convenient way to communicate between them is part of the architecture.

The following subsections will describe the startup procedure as well as three important architectural details: *UI instantiation*, *service binding* and *communication* between Android and middleware components.

A. Startup phase

The interaction on startup of a client app is illustrated in Figure 4. First, the client app has to send a special startup *intent* adressed to the middleware app, which is started on demand. This intent must contain information about the client app: the full path to the installed android application package (APK) file and the name of the main activity to launch. This behavior can be extracted to a base activity class that can be extended by the application developer.

 ISBN: 978-1-61208-418-3

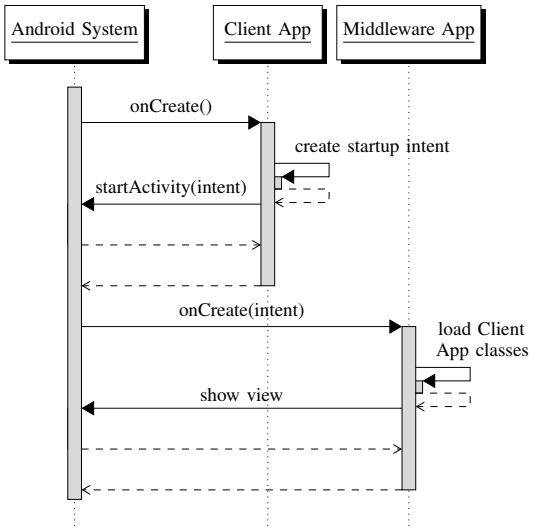

Figure 4. Interaction on application startup.

Upon reception of the intent, the middleware app will access the client app's classes by creating a class loader that points to the correspondent APK. Interestingly, it is possible for applications to read other applications' compiled code, if it is not explicitly forward-locked [20, p. 79]. After loading the classes, the middleware app will instantiate the main activity, which has been specified in the startup intent. The mechanism that is used to display the client app's UI will be introduced in the next section.

B. UI instantiation and management

As discussed earlier, app developers are not able to influence activity instantiation and presentation themselves. Fortunately, since the release of Android 3.0, there is a way to manage sub-views, which are called *Fragments*, from inside an application. Fragments implement their own life cycle, which is derived from the system-executed activity life cycle, but executed by a *Fragment Manager* instead [21]. Contrary to activities, this enables the use of fragments that are instantiated by application-own code, which in turn is the requirement for using a custom class loader.

In consequence, we are not allowing activities inside a client app, but instead use fragments as top-level replacement. As fragments were introduced to allow the partition of user interfaces, they also provide all capabilities to implement Android user interfaces (including fragments in fragments) and thus provide a viable replacement for activities. Three essential elements are needed in order to make the replacement work:

- A basic `FragmentActivity` inside the middleware app which will contain the client application's layout.
- A mechanism that implements switching between shown fragments and used resource contexts, depending on active client application.
- A base class extending `Fragment` that will (possibly transparently) replace activities in the client app. This class will be called `ClientAppFragment` and its instances are referred to as *client fragments*.

To focus on architectural design, we omit discussion of these elements here. They are discussed in detail in [22].

C. Service binding

When executing a client app inside the middleware process, the Android service binding mechanism needs special attention. Generally, binding an Android service returns the control flow to the Android system, which then determines which service to call and whether it has to be started or is already running. As this procedure is similar to starting an activity, it does not allow the developer to influence class loaders beforehand. This makes it impossible to build services that in turn use middleware libraries or communicate with the middleware. Since this is obviously undesirable, we propose an alternative way services are bound by client apps.

The used approach was inspired by the way Android handles fragments. Instead of letting the system create, bind and destroy services, a single service is always bound to the middleware app's main activity. All service calls originating from client fragments are handled by this *universal service*, which manages the (quite simple) service life cycle and maintains all existing service instances and connections.

The binding process is shown in Figure 5. In step (1), a binding is requested by a client fragment. The request, containing the classname of the targeted service, is handled by the universal service. In step (2), the service is instantiated and its life cycle is executed until `onBind()` is called, returning a binder object in step (3). The binder object is passed to the corresponding client fragment by calling `onServiceConnected()` in step (4), finally establishing the connection between client fragment and client service.

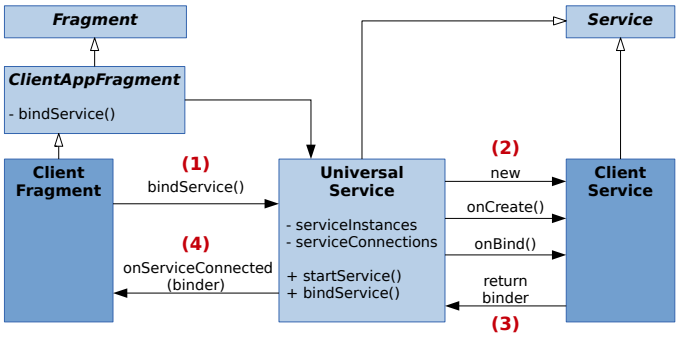

Figure 5. Universal service handles binding of client services.

Calls to external services, e.g., from other third-party apps, are still handled by the original service binding mechanism. Its possible to differentiate between internal and external services by the type of intent used: *Explicit* intents always refer to application-internal classes, while *implicit* intents can address external services [21].

D. Communication/Coupling

Besides displaying UI and binding services, at some point, the client app has to interact with the middleware in terms of starting the platform or using an already running instance, starting or stopping a middleware-supported runtime element, or looking up middleware services and components. Since interactions of this kind generally take place during the whole application lifetime, they should not be executed inside short-living activities/fragments, but rather in long-living Android services. For the sake of convenience, we only allow one service inside an application to communicate with the middleware directly; we will call this service *platform service*.

 ISBN: 978-1-61208-418-3

In the requirements section, we further demanded easy communication between relevant Android components, which we cut down to services and middleware components. The latter can be software components, agents, or any other runtime elements that are supported by the used middleware. For coupling between the platform service and middleware components, we use the observer pattern to allow for loose coupling of the components: a middleware component can be called from the platform service through interface methods or using the middleware's service model if available. The platform service can in turn register for typed events that are thrown inside middleware components; either by calling a static method or by integrating an appropriate event service into the middleware itself, which can be used by runtime components.

E. Overview

Figure 6 shows an overview of the architecture containing the elements described above. Green dots represent connection endpoints, embodied on Android by implementations of the classes `ServiceConnection` and `Binder`.

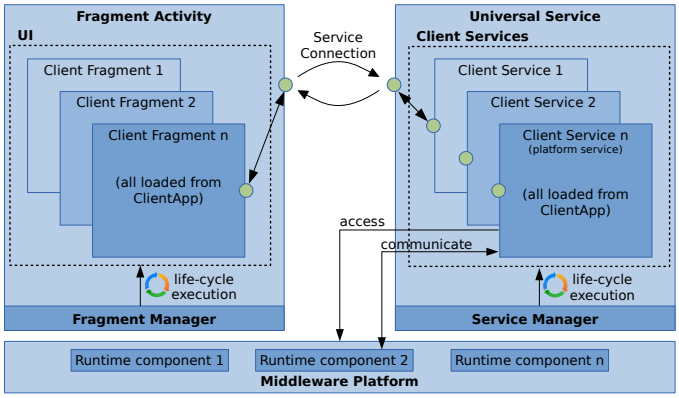

Figure 6. Overview of the middleware integration architecture.

On the left side, the *fragment activity* provided by the middleware app is shown. It loads all needed client fragments and displays them according to their life cycle, which is executed by the fragment manager. Client fragments communicate with client services on the right side, which are contained in the *universal service*. Their life cycles are executed by a service manager and the *platform service* is able to access the underlying *middleware platform* for managing and communicating with runtime components. Everything that is shown runs inside the same process and application context.

VI. Implementation

While the Android-specific implementation is rather universal and could be generalized, the used middleware might need some extensions. For example, it must provide a class loading mechanism that supports dynamic adding of class loaders. This is required for running a shared middleware platform, as it is unknown in advance which client application classes will be needed in the future. Upon requests by applications, the middleware must to be able to distinguish between individual applications and their classes to make sure classes can be loaded and the correct runtime components are managed. Since middleware may already handle dynamic class loading, this is a modest requirement. In addition, the aforementioned class loading mechanism has to support the Android class loaders; supporting the android-specific class formats. The presented integration architecture was implemented using the active component middleware *Jadex* [23].

To use the middleware platform, developers can extend the a special Android service class providing methods for configuring platform options, such as platform name, sharing of the platform, and other Jadex-specific options. After the platform was started in shared mode, other client apps can access it using their own implementations of the platform service. Each app-specific platform service can now start Jadex components on the shared platform and register for their events. The platform takes care of correlating apps with their Jadex components by using the corresponding class loader for each client app. Also, loading resources, layouts and assets from the right client app package is done transparently.

On the UI side, an app has to define a class inheriting from `JadexClientLauncherActivity` and implement a method returning the class name of the app's default fragment. After initiatiing the startup procedure described in Section V, this activity will show the default fragment. When the developer would like to implement further activities, he has to write a new Fragment instead. To launch these, `startActivity()` is enhanced to support Fragments and display them as if they are contained in a new Activity. Similarly, `startService()` is modified to use the universal service from Section V. Enhancing the methods specified by the Android framework keeps differences in the programming model as small as possible. In the case where fragments are required to split application layouts, implementing `ClientAppFragment` enables them to gain access to the top-level fragment for layouting purposes, such as adding/removing fragments.

VII. Evaluation

For evaluation, two versions of a demo application were compared against each other regarding startup time and application size. The first app is a client app and thus capable of using a shared platform, while the second embeds the Jadex platform, as it would have to without the presented architecture. For this evaluation, the applications just contain one fragment and bind itself to a platform service which starts up a platform component. In consequence, they model the smallest possible Jadex applications in each setup and the total startup time heavily depends on the platform startup time.

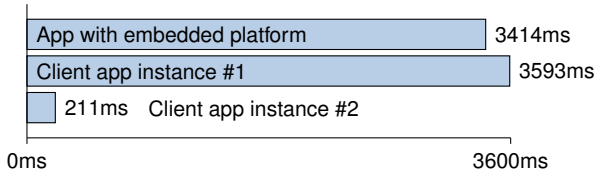

Figure 7. Application startup time comparison.

Figure 7 compares application startup times, showing the average of ten measurements on a Samsung Galaxy Nexus i9250. We differentiated between the first client app starting a shared platform and the case of a second app instance using an already running platform. As the chart indicates, the second client app starts up much faster, as the platform is already running. In comparison to embedding the platform, starting the first app using the integration architecture does

 ISBN: 978-1-61208-418-3

not significally influence startup time. Additionally, Table II shows corresponding application sizes. As splitting application and middleware results in only 132 KB overhead compared to embedding the platform, the benefit of using the integration architecture is confirmed again.

TABLE II. APPLICATION SIZE COMPARISON.

	Own classes	*DEX size*	*APK size*
Embedded	64 KB	3.516 KB	1.548 KB
Clientapp	58 KB	274 KB	136 KB
Middleware app	49 KB	3.532 KB	1.544 KB

Nevertheless, the proposed architecture comes with some limitations, which are mostly induced by circumventing the process separation of Android and running multiple apps in one process. First, client-side manifest declarations are ineffective, as the client app is only started by itself to initiate the startup phase. Afterwards, only the manifest of the middleware app is respected. This is especially critical for permissions, while intent receivers, or styles, could be handled by passing them through for the middleware app to handle. For evaluation purposes, our middleware app was given all necessary permissions. While app internal services are already handled by the architecture, externally visible services cannot be declared.

Second, access to the Android storage options, such as databases, preferences and private files is shared between all client apps running on the middleware. Since they use the same application context, applications also have access to the same storage resources. This could be prevented to a certain extent by enhancing the middleware with access control features.

Last, running multiple apps in one process affects runtime security and safety. Running in the same process means sharing memory with each other, possibly causing sensible data to be exposed. With big apps, one might also reach resource limits such as heap space sooner, which in the worst case might lead to termination of the middleware including all client apps.

VIII. CONCLUSION AND FUTURE WORK

This paper presented an approach that enables the integration of middleware into the Android system. In particular, the architecture allows different apps to use one middleware platform jointly at runtime. It further provides full access to the sophisticated Android design principles, which are unavailable on most current middleware implementations for Android. An event-based mechanism ensures smooth interaction between application components running on the middleware platform and Android application components. Most of the problems handled in the presented architecture arise from the fact that Android itself provides an extensive framework and runtime environment, complicating the integration of middleware.

With the presented architecture, other middleware can be integrated into Android, allowing developers to program using alternative programming principles, decomposition features and non-functional criteria of modern application-oriented middleware. Future work may include removing limitations where possible, evaluating the architecture using other middleware, as well as implementing a generic integration solution that abstracts from the concrete middleware platform.

REFERENCES

[1] J. Dehlinger and J. Dixon, "Mobile application software engineering: Challenges and research directions," in Workshop on Mobile Software Engineering, 2011, pp. 27–30.

[2] F. P. Brooks, Jr., "No silver bullet essence and accidents of software engineering," Computer, vol. 20, no. 4, Apr. 1987, pp. 10–19.

[3] T. A. Bishop and R. K. Karne, "A survey of middleware," in Procs. 18th Int. Conf. Computers and Their Applications, 2003, pp. 254–258.

[4] D. Ehringer, "The dalvik virtual machine architecture," Tech. Rep., 03 2010.

[5] R. H. Bordini et al., "A survey of programming languages and platforms for multi-agent systems," Informatica (Slovenia), vol. 30, no. 1, 2006, pp. 33–44.

[6] W. Emmerich, "Software engineering and middleware: a roadmap," in Proceedings of the Conference on The future of Software engineering. ACM, 2000, pp. 117–129.

[7] The Apache Software Foundation, "Apache felix framework and google android," 05 2009, [retrieved: 2015-06-10]. [Online]. Available: http://felix.apache.org/documentation/subprojects/apache-felix-framework/apache-felix-framework-and-google-android.html

[8] A. Santi, M. Guidi, and A. Ricci, "Jaca-android: an agent-based platform for building smart mobile applications," in Languages, Methodologies, and Development Tools for Multi-Agent Systems. Springer Berlin Heidelberg, 2011, pp. 95–114.

[9] P. Naur and B. Randell, Eds., Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, 1969.

[10] G. T. Heineman and W. T. Councill, Component-based software engineering: putting the pieces together. Addison-Wesley Longman Publishing Co., Inc., 2001.

[11] The OSGi Alliance, OSGi Service Platform Core Specification, Release 5, Jun. 2012.

[12] B. Hargrave and N. Bartlett, "Android and osgi: Can they work together?" [retrieved: 2015-06-10]. [Online]. Available: http://www.eclipsecon.org/2008/sub/attachments/Android_and_OSGi_Can_they_work_together.pdf

[13] C. Escoffier, "ipojo on android," 10 2008, [retrieved: 2015-06-10]. [Online]. Available: http://ipojo-dark-side.blogspot.de/2008/10/ipojo-on-android.html

[14] ProSyst Software GmbH, "Osgi runtime on android," [retrieved: 2015-06-10]. [Online]. Available: http://dz.prosyst.com/pdoc/mBS_SDK_7.3.0/modules/framework/common/android/introduction.html

[15] A. Ricci, M. Viroli, and A. Omicini, "Carta go: A framework for prototyping artifact-based environments in mas," in Environments for Multi-Agent Systems III, ser. Lecture Notes in Computer Science, D. Weyns, H. Parunak, and F. Michel, Eds. Springer Berlin Heidelberg, 2007, vol. 4389, pp. 67–86.

[16] R. H. Bordini, J. F. Hübner, and M. Wooldridge, Programming multi-agent systems in AgentSpeak using Jason. John Wiley & Sons, 2007, vol. 8.

[17] F. Bergenti, G. Caire, and D. Gotta, "Agents on the move: Jade for android devices," in Procs. Workshop From Objects to Agents, Sep. 2014.

[18] D. Bornstein, "Dalvik vm internals," Google I/O conference presentation video and slides, 2008, [retrieved: 2015-06-10]. [Online]. Available: http://sites.google.com/site/io/dalvik-vm-internals

[19] C. Collins, M. Galpin, and M. Käppler, Android in Practice. Manning Publications Company, 2011.

[20] N. Elenkov, Android Security Internals: An In-depth Guide to Android's Security Architecture. No Starch Press, 2014.

[21] Open Handset Alliance, "Android Developer Docs," [retrieved: 2015-06-10]. [Online]. Available: http://developer.android.com/

[22] J. Kalinowski, "Analysis and integration of application-oriented middleware into mobile devices in context of the android operating system," Master's thesis, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, Apr. 2014, in German.

[23] A. Pokahr and L. Braubach, "The active components approach for distributed systems development," International Journal of Parallel, Emergent and Distributed Systems, vol. 28, no. 4, 2013, pp. 321–369.

 ISBN: 978-1-61208-418-3