Computer Communications 34 (2011) 169–183

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Android/OSGi-based vehicular network management system

Ming-Chiao Chen [a], Jiann-Liang Chen [b,*], Teng-Wen Chang [c]

[a] *Department of Computer Science and Information Engineering, National Taitung University, Taiwan*
[b] *Department of Electrical Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan*
[c] *Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan*

ARTICLE INFO

Article history:
Available online 2 April 2010

Keywords:
Intelligent transport systems
Vehicular platform
OSGi
Android
Cloud computing

ABSTRACT

With the enormous market potential of the telematics industry and the rapid development of information technology, automotive telematics has attracted considerable attention for mobile computing and Intelligent Transport Systems (ITSs). However, as a result of varied platform standards, not all telematics services can be used in telematics terminals. The main issues are that most telematics technologies depend on vertical, proprietary, and closed Original Equipment Manufacturer (OEM) platforms. These platforms form islands of non-interoperable technology and prevent third-party service providers from creating valuable services. This study integrates the Open Gateway Service Initiative Vehicle Expert Group (OSGi/VEG) into an Android platform to generate a vehicular Android/OSGi platform that has the advantages of both original platforms. These features include remote management, rich class-sharing, proprietary vehicular applications, security policies, easy management of Application Programming Interface (APIs), and an open environment. This study also integrates a cloud computing mechanism into the Android/OSGi platform, allowing service providers to upload their telematics bundles onto storage clouds using a provisioning server. A management agent in the Android/OSGi platform can simultaneously update its application service modules using remote storage clouds and use visual intelligence to continually change the distinguishing features of applications based on context-awareness without user intervention. To assess the feasibility of the proposed Android/OSGi platform, this study presents a vehicular testbed to determine the functionalities of different telematics applications. Android/OSGi platform applications require less memory and system resources than those on the original Android platform when performing complicated operations. Additionally, the Android/OSGi platform launches telematics services more quickly than the original Android platform. The proposed platform overcomes the problem of frequent non-responsive exceptions in the original Android platform.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

In recent years, sophisticated information technology has allowed the automotive industry to mature in the field of telematics. Telematics in automobiles involves the automated convergence of telecommunications and information technology. So far, automakers and third-party service providers have developed quite a number of telematics services, including monitoring, emergency road-side assistance, navigation, driver aids, remote diagnostics, entertainment, and web browsing. The telematics market is currently quite young, and most telematics technologies are independent of each other due to multiple platform standards. This situation also makes it difficult for service providers to create value-added telematics services. Consequently, numerous vehicle manufacturers have established and developed open/standard embedded platforms for vehicles. These platforms include OSGi/VEG, AUTOSAR, AMI-C, CVIS, OSEK/VDX, and Android.

Fig. 1 illustrates an open Linux operating system ported into an embedded on-board terminal. This system does not only provide a variety of device drivers, including CAN/LIN/FlexRay buses and out-network connection modules, but also offers resources management. The open/standard telematics platforms in the telematics middleware layer mainly standardize API's and graphic/vocal HMI (human–machine interface) so that both service providers and car manufacturers can quickly deliver solutions to potential markets and simplify development. If service providers want to remotely deploy telematics services to an on-board terminal, or if road-side centers need to diagnose vehicular devices or configure telematics applications, they should use remote management services that rely on open/standard telematics platforms.

Telematics applications can be divided into four categories, including VANET embedded systems, vehicular multimedia embedded systems, intelligent driver aid embedded systems, and Urban Nomadic/Pedestrian Telematics embedded systems. First, a VANET

* Corresponding author.
E-mail address: lchen@mail.ntust.edu.tw (J.-L. Chen).

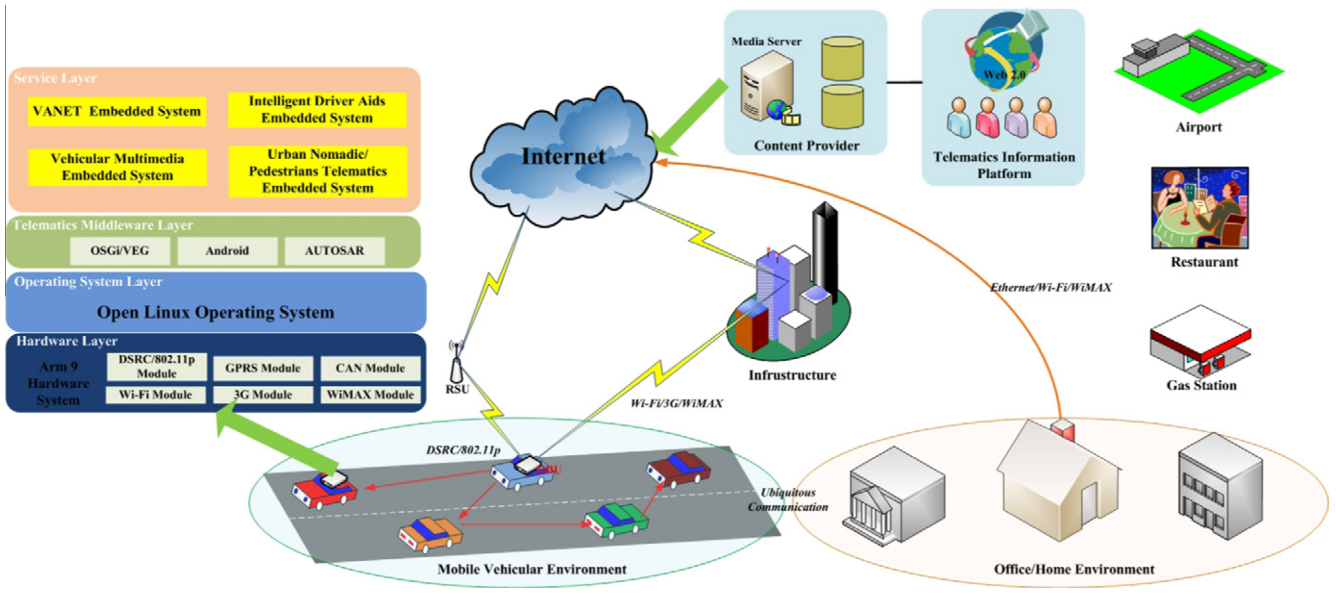


Fig. 1. Telematics architecture.

system allows vehicles to communicate with other vehicles or road-side units using DSRC/IEEE 1609. For example, if a vehicle has an accident, it can broadcast an emergency message to vehicles behind it using the VANET embedded system. The second type of telematics application is, a multimedia system that allows drivers to watch DVB programs, play on-line games, monitor home conditions, and so on. This type of multimedia embedded system takes advantage of high-performance graphics, SoC, or 3D engines when performing complicated operations. Third, driver aid systems ensure the security of drivers, economize the use of power, and so on. Finally, an Urban Nomadic/Pedestrians Telematics system provides interactive services between users and service providers in a non-automotive environment, such as finding local gas stations in an unfamiliar area. Other telematics services allow drivers to download/upload or share information to telematics information platforms using Web servers and out-network connection capabilities.

This study discusses how to combine OSGi/VEG into an Android platform to form a new vehicular Android/OSGi platform that has the advantages of both original platforms, including remote management/deployment, rich class-sharing, proprietary vehicular applications, security policies, and more.

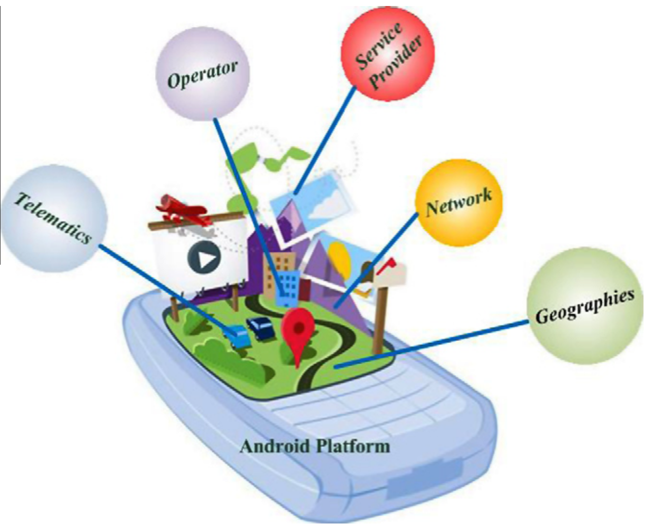


Fig. 2. Open environment for Android platform.

2. Related works

2.1. Android platform

The Android™ platform delivers a complete set of software for mobile devices: an operating system, middleware, and key mobile applications [1]. Windows Mobile and Apple's iPhone provide a richer, simplified development environment for mobile applications. However, unlike Android, they're built on proprietary operating systems that often prioritize native applications over those created by third parties and restrict communication between applications and native data. Android offers new possibilities for mobile applications by offering an open development environment built on an open source Linux kernel. Real hardware can be accessed through a series of standard API libraries, allowing the user to manage Wi-Fi, Bluetooth, and GPS devices. As Fig. 2 illustrates, the Open Mobile Alliance (OHA) [2] and Google support the Android platform and hope to reach the goal of ensuring global mobile services that operate across devices, geographies, service providers, operators, and networks.

Google has already released the open source Android platform, providing the opportunity to create new adaptive mobile platform interfaces and applications designed to look, feel, and function as desired. Consequently, the Android platform has recently been ported into mobile devices, such as notebooks, PDAs, and automotive systems. In automotive system fields, the Intel and Wind River Systems are actively integrating Android-powered infotainment operating systems for vehicles.

2.1.1. Android software stack

Fig. 3 illustrates the Android software stack [3], which consists of a Linux kernel, a collection of Android libraries, an application framework that manages Android applications in runtime, and native or third-party applications in the application layer. The following list specifies these Android software stack components:

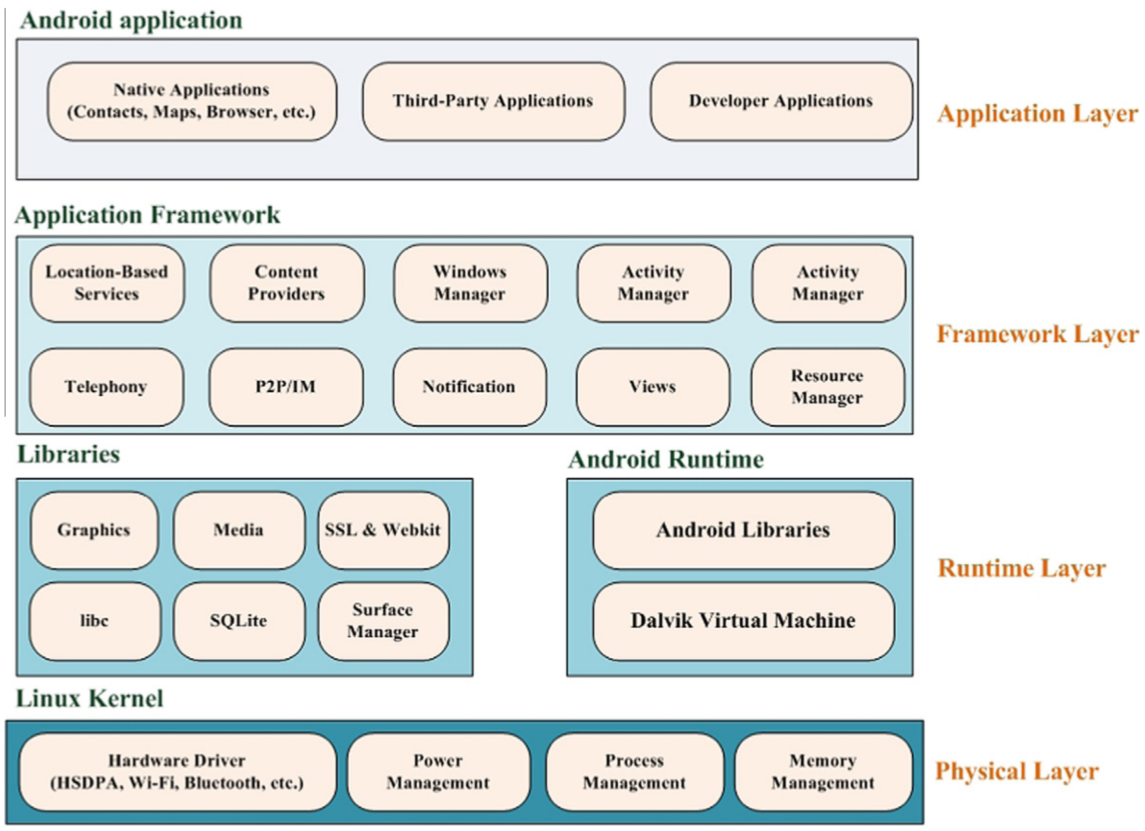


Fig. 3. Architecture of Android software stack.

- *Linux kernel:* A Linux 2.6 kernel manages core services (including device hardware drivers, process and memory management, security, network, and power management). The kernel also provides an abstraction layer between the hardware and the rest of the stack.
- *Libraries:* Android includes various C/C++ core libraries running on top of the kernel. These libraries include libc and SSL, and the following:
  - A media library for playing multimedia resources
  - A Surface manager to provide display management
  - Graphics libraries that include SGL and OpenGL for 2D and 3D graphics
  - SQLite for data storage
  - SSL and WebKit for integrated web browser and Internet security
- *Android runtime:* Including the core libraries and the Dalvik virtual machine, the Android runtime is a runtime environment for normal Android applications.
- *Core libraries:* While Android development is performed in Java/C++, Dalvik is not a standardized Java VM. The core Android libraries provide most of the functionality available in the core Java libraries and the Android-specific libraries.
- *Dalvik virtual machine:* Dalvik is a register-based virtual machine that is optimized to ensure the efficient and stable operation of multiple instances. Dalvik relies on the Linux kernel for threading and low-level memory/process management.
- *Application framework:* The application framework provides the classes used to create Android applications. It also provides a generic abstraction layer for hardware access and manages the user interface and application resources.
- *Application layer:* All applications, both native and third-party, are built on the application layer using the same API libraries. The application layer runs within the Android runtime using the classes and services available from the application framework.

2.1.2. Android platform features

Android offers several unique features that other mobile platforms do not possess:

- *Google map of navigation applications:* Fig. 4 shows that the Android platform provides a MapView widget that allows a user to navigate, display, manipulate, and annotate a Google Map within the Activities to build map-based applications using the more familiar Google Maps interface.
- *Background services and applications:* Background services allow Android applications to use an event-driven model that works silently while other applications are being used in the foreground.
- *All applications are created equal:* Android does not differentiate between core applications and third-party applications, and all applications have equal access to a platform's capabilities providing users with a broad spectrum of applications and services.
- *P2P inter-device application messaging:* Android offers a peer-to-peer messaging mechanism that supports presence, instant messaging, and inter-device/inter-application communication.
- *Breaking down application boundaries:* Android breaks down the barriers to building new and innovative applications. Users can combine information from the web with their own Android platform, including user contacts, calendar, or geographic location, to provide a more relevant user experience. Further, third-party applications can be downloaded to the user's Android platform from the Android Market for free.

2.2. Open services gateway initiative (OSGi) service platform

The OSGi™ platform standard [4] defines a standardized, components-oriented computing environment for networked services that is the foundation of an enhanced Service-Oriented Architecture (SOA). The OSGi specifications were initially targeted at

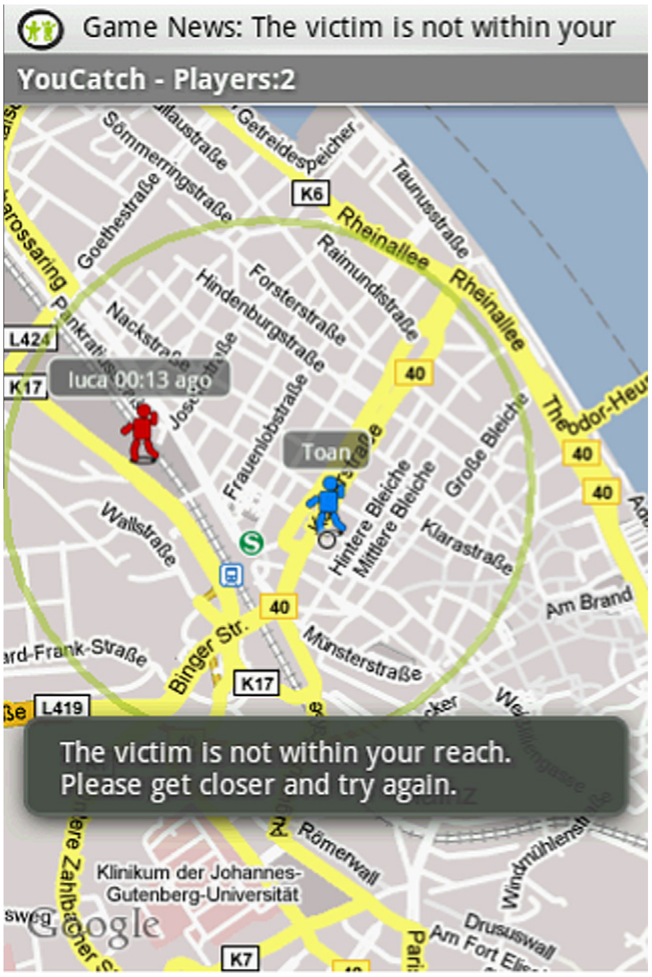

Game News: The victim is not within your
YouCatch - Players:2
luca 00:13 ago
Toan
The victim is not within your reach.
Please get closer and try again.

Fig. 4. Location-based services for Android platform.

residential internet gateways for home automation applications [11,12]. However, the standard and extensible features of OSGi make enterprises regard OSGi technology as key solutions. For example, Nokia and Motorola adopted the OSGi technology standard for next-generation smart phones, and many car manufacturers have embedded OSGi specifications into their Global System for Telematics (GST) specifications [5]. The OSGi framework provides general-purpose, secure support for deploying extensible and downloadable java-based service applications, called bundles. OSGi bundles are applications packaged in a standard Java Archive (JAR) file, and contain a manifest file that describes the relationships among bundles and the OSGi framework, a series of compiled Java classes, and native code. Fig. 5 illustrates the OSGi framework running on a Java virtual machine. The OSGi framework provides a shared execution environment that dynamically installs, updates, and uninstalls bundles without restarting the system.

2.2.1. OSGi-based telematics gateway

The OSGi Alliance whitepaper [10] indicates that the research and development of OSGi can be extended to any networked environment, including automobile applications, and implemented in complex embedded devices that require deployable services. The OSGi framework makes it possible to access, download, and use services in an automotive mobile environment, creating an enormous potential for automotive manufacturers and service providers to offer new proactive services to drivers and passengers [13]. The open standard-based, service-oriented automotive infrastructure for the telematics architecture [14,15] includes a OSGi

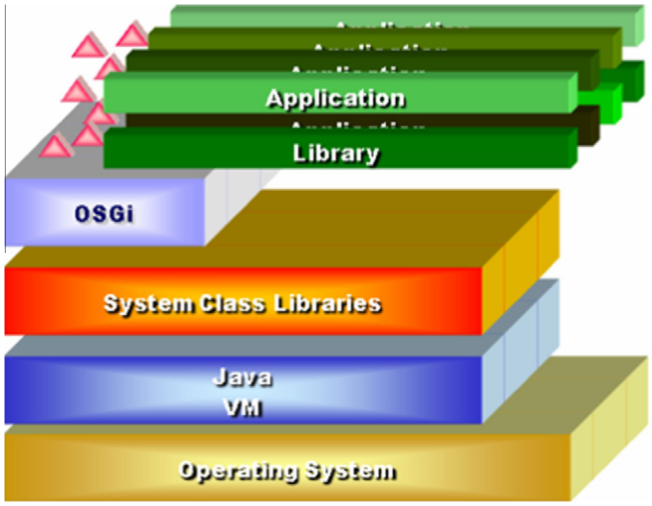


Fig. 5. Architecture of OSGi framework.

gateway, service infrastructure, and self-automobile. The central control point of the service infrastructure, which is the OSGi-based telematics gateway, can be interconnected with various internal and external networks and buses. This gateway acts as an execution environment for mobile automotive services. The following bundles provide many critical services for OSGi-based telematics gateway:

- *Monitoring bundle:* This bundle offers basic intelligent vehicle functions. By monitoring the status and performance of local vehicle devices, a driver can acquire vehicle information, such as temperature, various pressure levels, engine fluid levels, etc.
- *Navigation bundle:* This bundle provides a navigation system that offers navigational assistance to drivers. The system receives and processes GPS position information signals to determine the current latitude and longitude coordinates, and direction of travel.
- *Communication bundle:* This bundle acts as an OSGi-based telematics gateway for in-vehicle or out-vehicle network connections. This bundle extends the functionality of the OSGi-based gateway by remote deployment of OSGi bundles provided by the service provider. Through CAN, LIN, or MOST [16], the OSGi-based telematics gateway realizes local area connections among in-vehicle devices, and by using GPRS, GSM or even special ITS FM channels, this bundle allows wide area connections among vehicles, home gateways, remote service centers, and road-side systems.
- *Driver aid bundle:* This bundle provides driver assistance and offers some warning messages based on lane recognition.
- *Diagnostics bundle:* This bundle provides remote diagnostic services, allowing complicated diagnostic information to be analyzed and stored in remote systems. This technique is similar to the MYCAREVENT project [17] in that it enables a telematics gateway to have the capacities of automatic repair and self-management.
- *Information/Entertainment bundle:* Video and audio equipment are already widely applied in vehicles. Previous research discusses an infotainment server system for in-vehicle users that allows the network-enabled Information Appliances (IA) to access information and entertainment services from an infotainment server [18].

Automotive operating systems are also very significant in vehicles [19]. By supporting specific device drivers such as CAN/LIN buses, which communicate with other Electronic Control Units

(ECU) nodes for diagnostic purpose, previous studies have embedded an OSGi platform in a vASOS automotive operating system [20,21]. Fig. 6 illustrates OSGi-based middleware run on a K Virtual Machine (KVM), which is a compact, portable Java virtual machine intended for small, resource-constrained devices [5]. In addition, previous studies use an OSGi R3 implementation of Oscar to develop application bundles, including navigation and CAN/LIN/MOST access bundles. This study takes advantage of OSGi R4 implementation of Apache Felix to integrate with the Google Android platform, turning a variety of Google APIs into OSGi bundles, such as location-based, peer-to-peer communication, and network manager bundles. This approach not only allows service providers to deploy telematics services, but also enhances Android runtime layer performance and remote management functionality.

The goal of the Vehicle Expert Group (VEG) is to tailor and extend the OSGi specification in order to meet vehicle-specific requirements. To achieve this, the VEG has defined a list of topics that cover vehicle-specific issues. Working with the automotive industry, the VEG has specified telematics API's so that both service providers and car manufacturers can quickly deliver solutions to market and increase customer loyalty. The OSGi–VEG organization has the following automotive projects:

- *3GT:* The 3GT project [6] is designed to help establish OSGi-based in-vehicle telematics platforms for the European mass market by ensuring interoperability between the products of different middleware providers, terminal manufacturers, and service providers. This will be done by establishing common telematics interfaces for OSGi-based service delivery. More specifically, 3GT will develop OSGi-based specifications for the interface between vehicles and control centers and the interface between control centers and service providers.
- *ITEA EAST EEA:* This automotive project [7] is funded by the European Union, and involves major European automotive manufacturers, first-tier suppliers, and research departments. The goal of EAST-EEA is to enable hardware and software interoperability of in-vehicle ECU nodes by defining an open, middleware-based architecture.
- *Stadtinfokoeln:* Stadtinfokoeln is a Cologne-based project [8], funded by the German Ministry of Education and Science (BMBF), that focuses on parking services for the residents or visitors of Cologne, Germany. The goal of this project is to satisfy customer demand and to reduce traffic, which is related to parking space availability. An important component of this project is the rapid development of information and communication technology in a mobile environment. To fulfill these requirements, the Stadtinfokoeln project managers decided to use an OSGi service platform framework. This has enabled the rapid development and deployment of new services.
- *TOP-IQ:* The Eureka-project TOP-IQ [9] focuses on the development of a new generation of On-Board Computers (OBC) for luxury cars and trucks. This type of OBC facilitates telematics services for transport and logistics, and conducts the management, billing, and delivery of new services such as on-line navigation, trailer management/tracking, cargo tracking, and more.

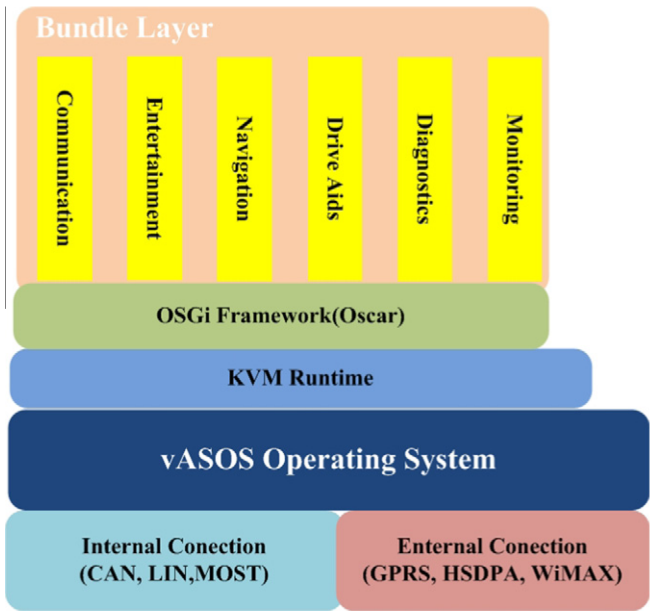


Fig. 6. Architecture of vASOS and OSGi-based middleware for vehicles.

2.2.2. Safe deployment of OSGi bundles

The security of deployment vehicular services is a crucial issue in the telematics environment. If vehicle owners install third-party applications on their on-board vehicle system, they may experience hazardous conditions if those vehicular applications have many bugs, such as sending mass messages, accessing the security sensitive areas of the vehicle, and consuming excesses system resources. Therefore, it is hard to establish meaningful trust relationships in an open environment, and even when one can, trust does not equal quality. Previous researchers have utilized Aspect-Oriented Programming (AOP) to weave a security enforcement policy into OSGi bundles [22]. Additionally, AOP provides cross-cutting functionalities in complicated vehicular applications, allowing program concerns in a software system to be captured and encapsulated into so-called aspects. Fig. 7 illustrates the weaving process scenario in a vehicular environment, which involves an in-vehicle system, control center, and third-party service providers.

Some vehicle owners install vehicular application bundles by making a corresponding request in the GST standard. Before installing this type of application bundle on an in-vehicle system over a wired/wireless network, the control center weaves the application with security policies so that the woven bundle can operate with enforcement security policies on the in-vehicle system.

2.3. Comparison of OSGi and Android

Fig. 8 shows that both OSGi and Android define a development platform for developing service-oriented computing mechanisms in mobile devices, and share similar VM technology, service models, component models and openness. The main difference between them is that the Android platform utilizes process-based isolation to enhance resource management, while the OSGi platform uses a classloader-based isolation model. This allows OSGi bundles to selectively export some of their internal packages to other consumer bundles. Bundles can declare their package dependencies and the framework is responsible for managing dependency relationships between consumer bundles and provided bundles. Due to its classloader-based isolation mechanism, the OSGi framework also has the advantage of rich class-sharing. The Android platform uses background services to perform time-consuming operations, but these services are usually heavyweight, and only can be accessed utilizing inter-process communication, which makes service access slower and more expensive. In OSGi, service components are lightweight. They are called directly when found in the service registry. Consequently, OSGi services play a significant role in keeping components lightweight and loosely coupled. The OSGi specification defines many remote management services, including SNMP, CMISE, CIM, and OMA DM [23]. Utilizing remote management services, managers can easily remotely administer an OSGi framework in many situations. These remote

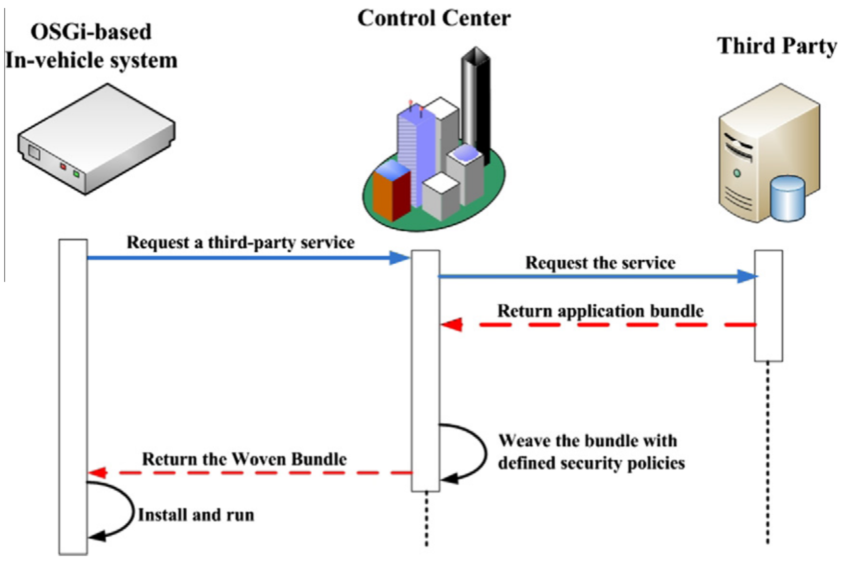

OSGi-based In-vehicle system
Control Center
Third Party
Request a third-party service
Request the service
Return application bundle
Weave the bundle with defined security policies
Return the Woven Bundle
Install and run

Fig. 7. The weaving process in vehicular environment.

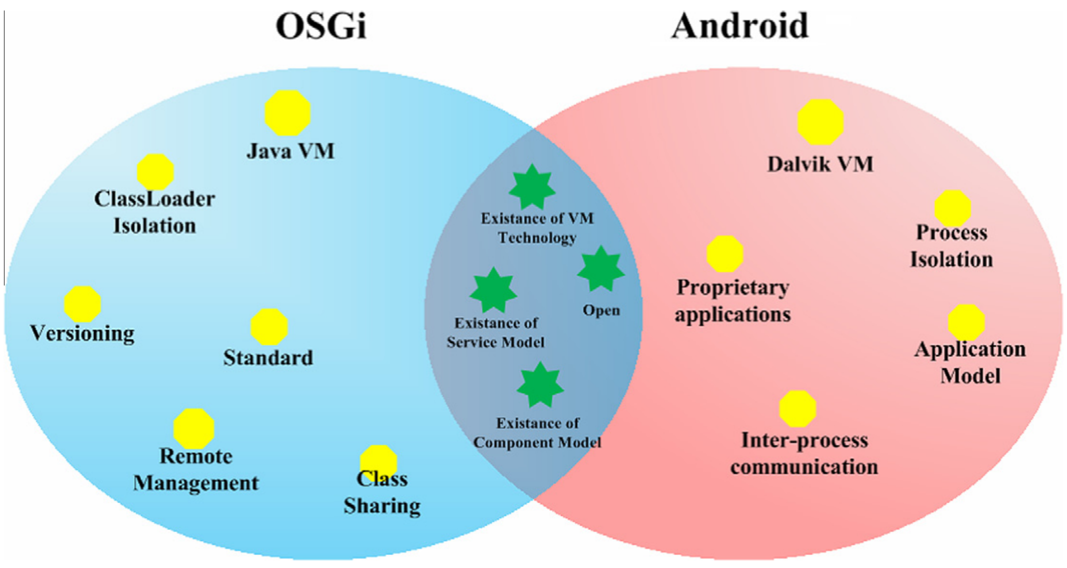


Fig. 8. Comparisons of OSGi and Android.

management services are not available on the Android platform. The Android platform's lack of class-sharing ability seriously limits the sharing of components and remote device management. The lack of versioning and management APIs is a further limitation. Presumably, OEMs will provide some solutions for these important requirements in the future.

3. Open embedded Android/OSGi platform for telematics

Fig. 9 illustrates the proposed architecture for a telematics system. Here, the Android platform adopts the latest Android 1.5 kernel, allowing the proposed Android/OSGi platform to be integrated into real Android-based hardware devices. This design also allows the OSGi R4 implementation to be operated normally in an Android Dalvik Virtual Machine (VM), developing the communication interface between users and OSGi framework in the Android application layer, bridge Android/OSGi applications, and other Android applications. This study also develops an advanced technique for embedding an OSGi instance into Android applications, to make them more adaptive and flexible. Any time-consuming operations, such as continually loading the main class of OSGi framework and updating the Android GUI interfaces, will be performed by Android background services. As a result, the proposed Android/OSGi platform has high-performance and responsiveness. In the proposed design, an iPOJO service-oriented framework is ported and run on top of Android/OSGi applications. The primary goal is to provide the components with a self-management capacity and dynamically respond to service changes in capricious environments. We also installed indispensable OSGi bundles for developing some valuable telematics applications, such as the Web management console, remote deployment interface, navigation, diagnostics, and so on. Finally, this study develops the management agent to communicate with remote cloud called Amazon EC2 through Restful API. The management agent installs/updates local services synchronously with remote cloud, and all of Android/OSGi bundles are stored in Amazon S3. This study also implements a provisioning server to remotely deploying packages or OSGi bundles to the client's Android/OSGi platforms based on Amazon S3.

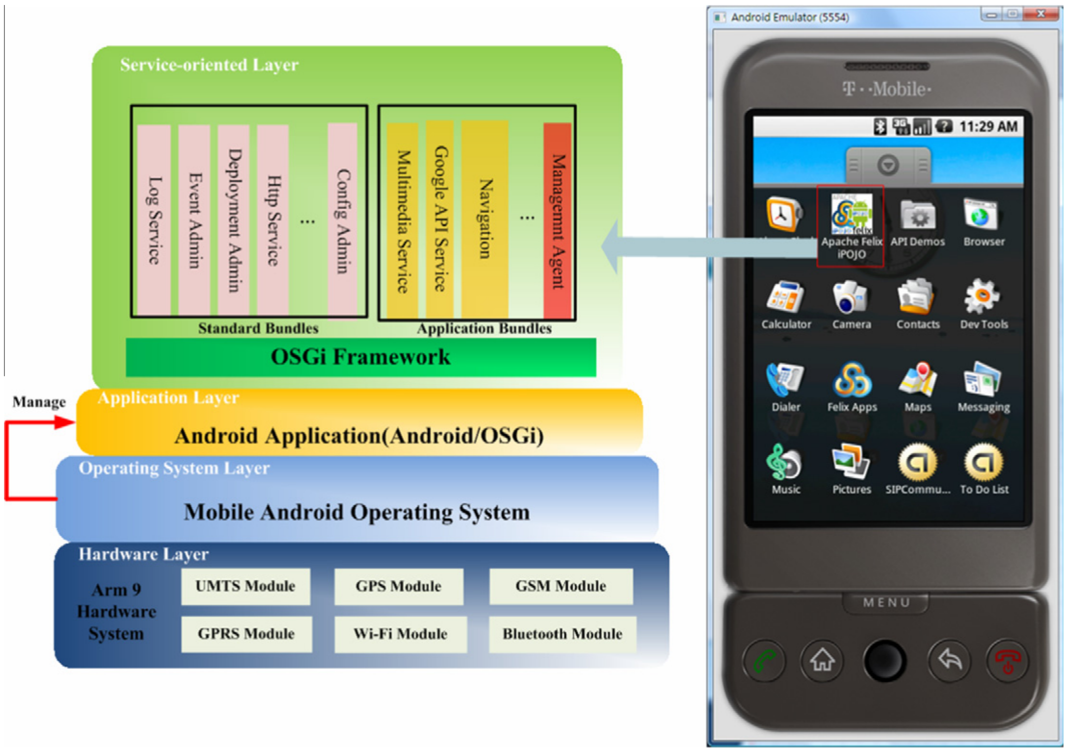


Fig. 9. Location of OSGi framework.

3.1. Location of OSGi framework in the Android platform

The Dalvik VM use the device's underlying Linux kernel to handle low-level functions, including security, threading, process, and memory management. It's also possible to read and write C/C++ applications that run directly on the underlying Linux OS. However, Dalvik VM can only execute Dalvik executable files (e.g., **classes.dex**), a format optimized to ensure minimal memory footprint. Consequently it cannot perform general Java packages and OSGi bundles, causing OSGi framework operations to fail when Android encounters OSGi functionalities in runtime. Besides, as Fig. 10 shows, if the OSGi framework successfully runs in the Android Runtime layer, it cannot directly interact with users, because users can only view Android Activities (or applications) in the application layer. For this reason, the Android/OSGi Activity must grab a report or log message from the OSGi framework, and use the Android UI to present information to the users. The proposed design uses the OSGi R4 implementation of Apache Felix to integrate with the Android platform because this container is more portable and lightweight, and better-suited to embedded hardware devices such as telematics systems.

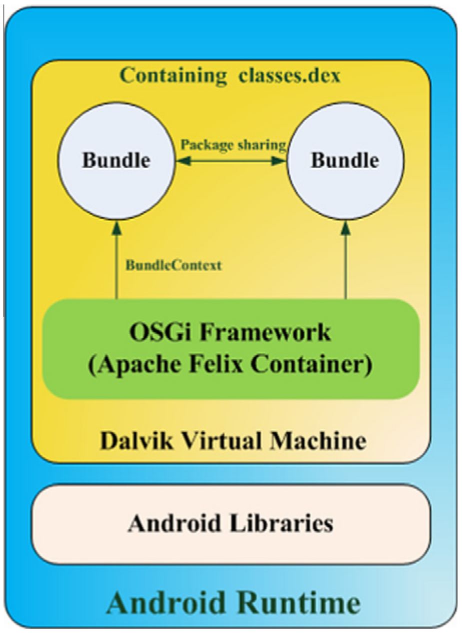


Fig. 10. Location of OSGi framework.

3.2. Android/OSGi conversion mechanism

The Android/OSGi conversion mechanism uses the OSGi framework to dynamically load bundles through the Android Davik VM. Fig. 11 shows the Android/OSGi conversion mechanism, which makes the OSGi framework perform a series of complied Java classes from OSGi bundles in the Android runtime layer using a classes.dex file to reference the OSGi bundle's general Java classes. Therefore, the classes.dex file plays a bridging role between the OSGi framework and bundles. The proposed design implements Android/OSGi conversion code through Dalvik.system.dexFile API, and imports two parameters, the package name (bundle API packages) and ClassLoader (reading the Java classes) objects. The conversion code then returns the collection of Java classes, allowing the OSGi framework to indirectly perform the functionalities provided by the OSGi bundles.

After operating Android conversion mechanism, it is necessary to ensure that all OSGi bundles or general Java packages contain classes.dex files. This allows the OSGi framework to assume that any bundle works, share API packages, and let consumer bundles query the OSGi services provided by **the OSGi framework**. This approach adopts the **DEX** conversion tool from Android SDK. This tool can easily convert Java complied files into classes.dex files, which that Dalvik VM is able to execute. Fig. 12 illustrates the conversion flow of OSGi bundles to **apk** files (Android executive extension); in

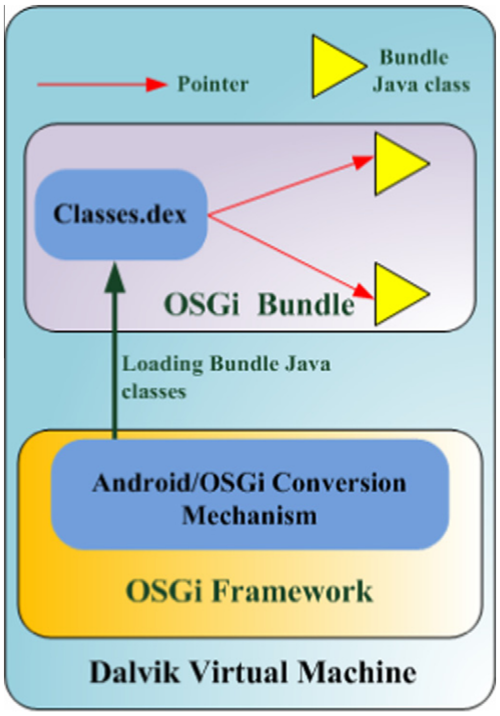


Fig. 11. Android/OSGi conversion mechanism.

original, this study had made the OSGi bundle projects to be compiled, and if no exception will generate a series of Java compiled classes. After converting into jar files, the DEX tool is used to make jar files containing classes.dex, and he **aapt** command transforms OSGi bundles into Android apk files. Finally, the **adb push** command is used to push converted the apk files to the Android platform.

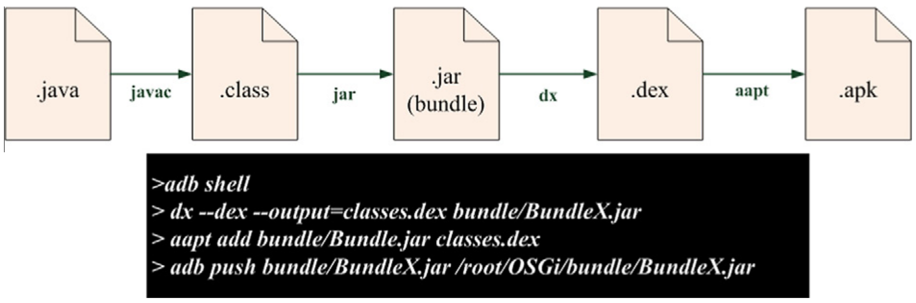
With the Android conversion mechanism and converted OSGi bundles, it is possible to port the OSGi framework to the Android underlying layer, along with other functional bundles such as telnet, deployment admin, http, and remote manager bundles etc., so that the Android platform can be operated and managed remotely. The original platform does not have this ability. Fig. 13 shows the script of starting the OSGi framework in the Android runtime layer, and using a Dalvik VM to read the main class of the OSGi framework, enter the OSGi console mode, and show a list of some installed bundles. Nevertheless, because users cannot directly interact with the OSGi framework in real hardware devices using this technique, this study implements Android/OSGi Activity and GUI interface to acquire corresponding OSGi information from the underlying layer, and also embeds the OSGi framework into an Android application to make it more powerful and adaptable.

3.3. Development of Android/OSGi application

This study had created Android/OSGi application in Android application layer, which can fetch the OSGi corresponding information from Android runtime layer, and providing interactive environment between users and OSGi framework. In first integration of procedure, this study must to implement Android Activity interface, letting Android/OSGi application to have Android Activity properties. Unlike most traditional environment, Android applications have no ability of control over their own life cycles, Instead, Android application framework must to be in charge of Android applications state, and react accordingly, taking particular care to be prepared for untimely termination. When main launch Activity which belongs to Android/OSGi application has become active state and need to operate continually, the Android application framework will pay attention to ensure that the Android/OSGi application remains responsive. Therefore Android application framework has the capacity of monitoring platform resources at any time, if necessary, will kill some empty or background processes to free resources for high-priority applications.

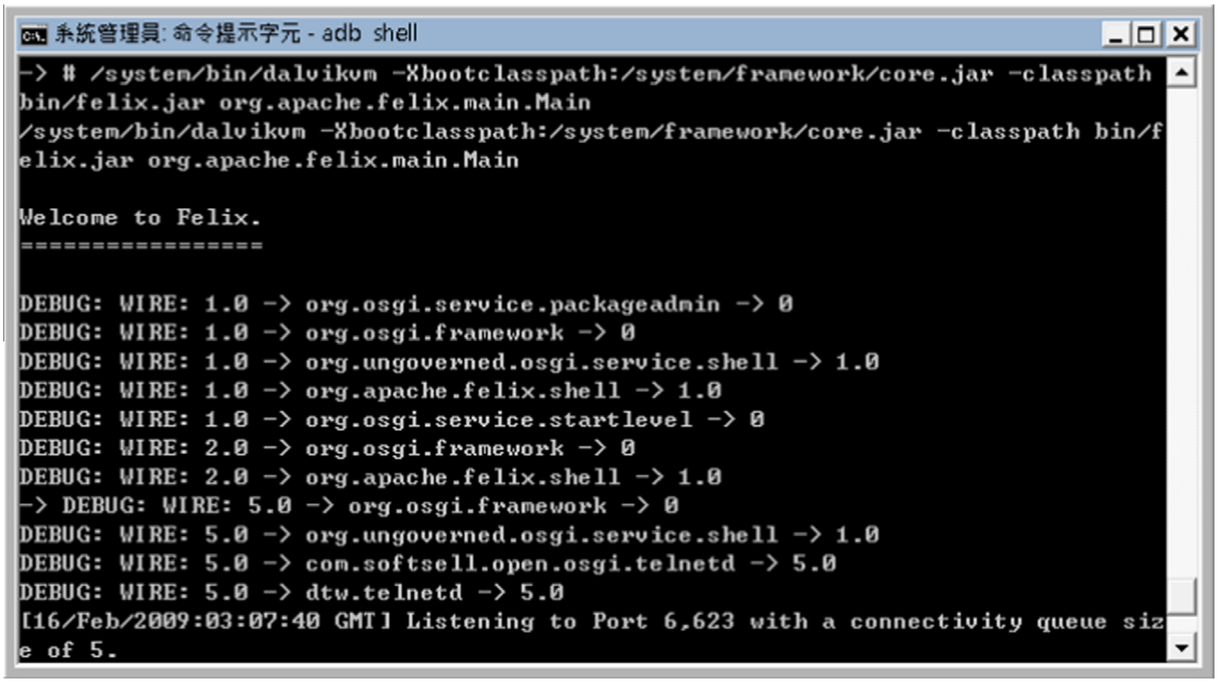
The Android application has various Activities, which represent a screen that an application can present to its users. Android Activity typically includes at least a primary interface screen that handles the main UI functionality for Android application. This is often supporting additional by secondary Activities for entering information, and providing different perspectives on user data. For example, this study had created the Android views of OSGi console, making users can enter OSGi commands and catch the sight of OSGi corresponding Information. Every Android Activity has an initializer method called ***onCreate***, by using this initializer method; Android platform can distribute the memory to perform the launcher thread which can make OSGi framework to be started. Fig. 14 illustrates the cross-communication operation that Android/OSGi Activity launches the ***FelixService*** (Android background service) which can represent the GUI interface of OSGi console and to load the main class of OSGi framework after Android/OSGi Activity onCreate method had already been initialized. Android services run without a dedicated GUI, they usually silently perform background works that users can't perceive. Before FelixService to run, they must to be attached with Android/OSGi Activity. From different point of view, the Android/OSGi Activity provides Android ***Context*** object, which makes Android application framework to be able to manage its life states, and communication between Android/OSGi application and other native or third-party applications.

To ensure that the proposed Android/OSGi application remains responsive, the proposed design moves all slow and time-consuming operations off the main Activity thread and onto a child thread. Examples of slow operations include continually reading the main class of the OSGi framework, and updating the GUI interface when the OSGi console view has been changed or GUI bundles provide



```
>adb shell
> dx --dex --output=classes.dex bundle/BundleX.jar
> aapt add bundle/Bundle.jar classes.dex
> adb push bundle/BundleX.jar /root/OSGi/bundle/BundleX.jar
```

Fig. 12. Conversion flow of OSGi bundles to become apk files.



```
-> # /system/bin/dalvikvm -Xbootclasspath:/system/framework/core.jar -classpath
bin/felix.jar org.apache.felix.main.Main
/system/bin/dalvikvm -Xbootclasspath:/system/framework/core.jar -classpath bin/f
elix.jar org.apache.felix.main.Main

Welcome to Felix.
=================

DEBUG: WIRE: 1.0 -> org.osgi.service.packageadmin -> 0
DEBUG: WIRE: 1.0 -> org.osgi.framework -> 0
DEBUG: WIRE: 1.0 -> org.ungoverned.osgi.service.shell -> 1.0
DEBUG: WIRE: 1.0 -> org.apache.felix.shell -> 1.0
DEBUG: WIRE: 1.0 -> org.osgi.service.startlevel -> 0
DEBUG: WIRE: 2.0 -> org.osgi.framework -> 0
DEBUG: WIRE: 2.0 -> org.apache.felix.shell -> 1.0
-> DEBUG: WIRE: 5.0 -> org.osgi.framework -> 0
DEBUG: WIRE: 5.0 -> org.ungoverned.osgi.service.shell -> 1.0
DEBUG: WIRE: 5.0 -> com.softsell.open.osgi.telnetd -> 5.0
DEBUG: WIRE: 5.0 -> dtw.telnetd -> 5.0
[16/Feb/2009:03:07:40 GMT] Listening to Port 6,623 with a connectivity queue siz
e of 5.
```

Fig. 13. OSGi framework run in Android runtime layer.

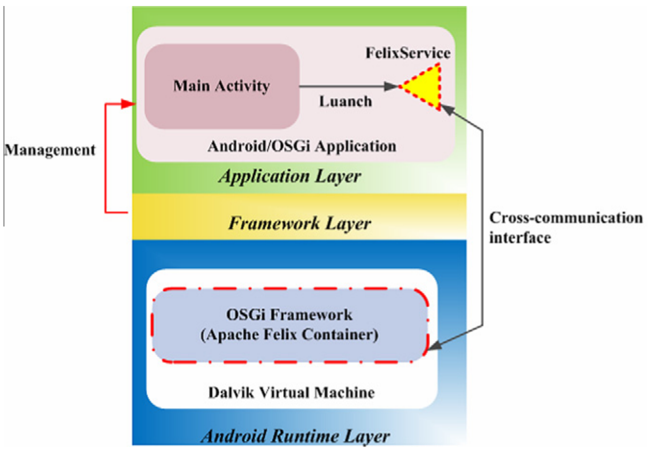


Fig. 14. Cross-communication operation.

new screen services. Consequently, this study implements the Felixservice to perform the above-mentioned operations. Fig. 15 shows that Felixservice can directly determine the location of the OSGi framework in the Android runtime layer and extract the main jar file of the OSGi framework, which is Apache Felix of R4 OSGi implementation. Using the Dalvik ClassLoader object to read the main class continually in child thread, the Android/OSGi platform can start the OSGi framework in the Android application layer. In addition, this study also implements a OSGi console view that can fetch the corresponding OSGi message and display them to users. The OSGi console view has two input/output interfaces that allow users to immediately interact with the OSGi framework.

Fig. 16 shows that how the Android/OSGi Activity can transmit the received information to the OSGi framework after finishing the above-mentioned operation. The OSGi framework regards this information as permanent properties data (key/value pairs) stored in the OSGi environment and OSGi bundles can retrieve these properties through preference service to implement advanced applications, such as network manager, navigation bundle, and so on. In the same way, when OSGi bundles become ACITVE, they can also transmit explicit or implicit intents to launch useful Activities. Based on the strength of this two-way communication mechanism, the proposed design allows Android applications to easily connect with the OSGi framework and bundles.

3.4. Embedding OSGi framework to Android application

In traditional devices, the Android/OSGi platform utilizes the OSGi services and service registry as an extensibility mechanism. This mechanism allows any application to run completely on top of the OSGi framework, but this is not always possible. In other words, system components must follow the OSGi standard to be implemented, which will result in increasing the development times and costs. Besides, the system components spend extra-time to communicate with OSGi framework in runtime layer. Therefore, if any Android application wants to use either the OSGi services or the APIs provided by bundles, it needs a complicated Android/OSGi communication mechanism. We created an embedded mechanism that allows the OSGi framework to tightly embed into Android applications in the application layer, as Fig. 17 shows. In this way, any Android application can host the instance of OSGi framework by utilizing reflection approach, and application can externally load the services which OSGi bundles provided. Nevertheless, the extender mechanism has some drawbacks in that it lack dynamic changes in OSGi bundles/services and cannot configure OSGi instances. Therefore, this study combines two mechanisms with the Android platform to create a more integrated Android/OSGi environment.

The Felix framework instance does not utilize ***ServiceReference*** array object to get referenced services, but on the contrary, it uses the ***Service Tracker*** interface to monitor services. As a result of utilizing traditional service-oriented mechanism, the producer bundles should use the ***BundleContext*** object to register services in the service registry. If referenced services have been changed, the ***ServiceChanged*** event must be implemented to ensure that no exceptions have been generated. As a result, this mechanism is more cumbersome then the service tracker mechanism. For

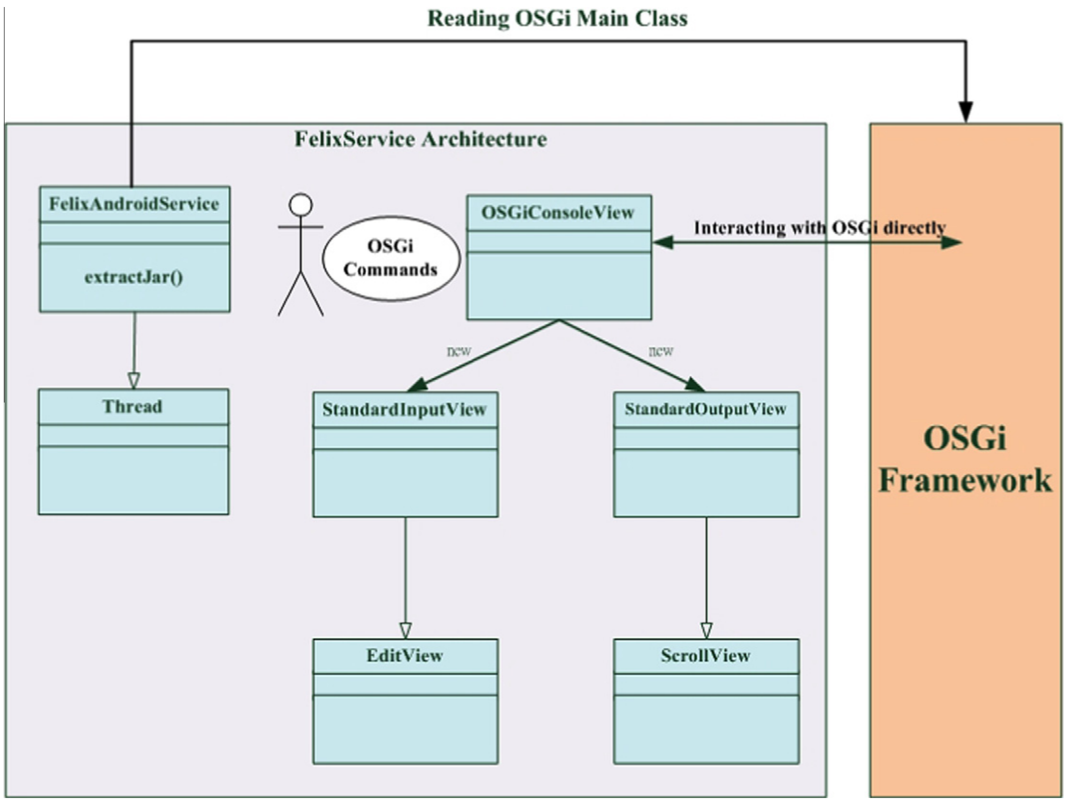


Fig. 15. Inter-architecture of the Felixservice.

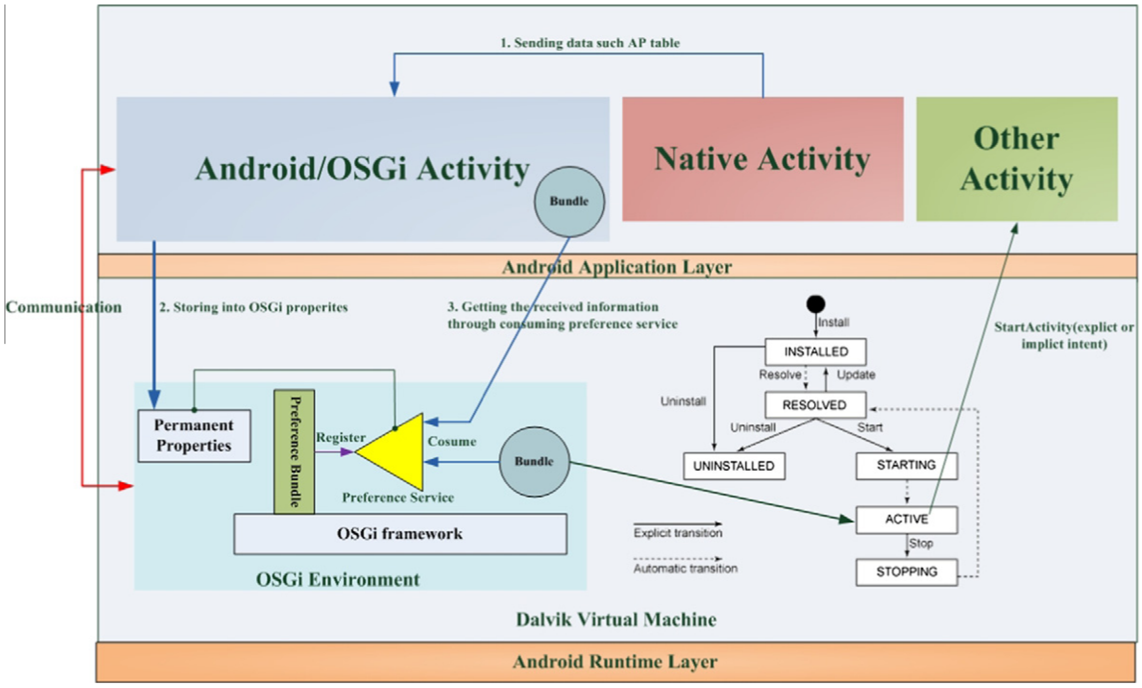


Fig. 16. Two-way communication mechanism between OSGi and Android.

example, Fig. 18 shows, if referenced services have been modified, the server bundle automatically callback the ***modifiedService*** method to rebind these services without implementing the ServiceChanged event handler. The service tracker also can take advantages of a whiteboard pattern, allowing embedded OSGi framework to utilize a server bundle to monitor listener services changes. The embedded OSGi Activity has the responsibility of tracking UI services, and if these UI services exist, the embedded

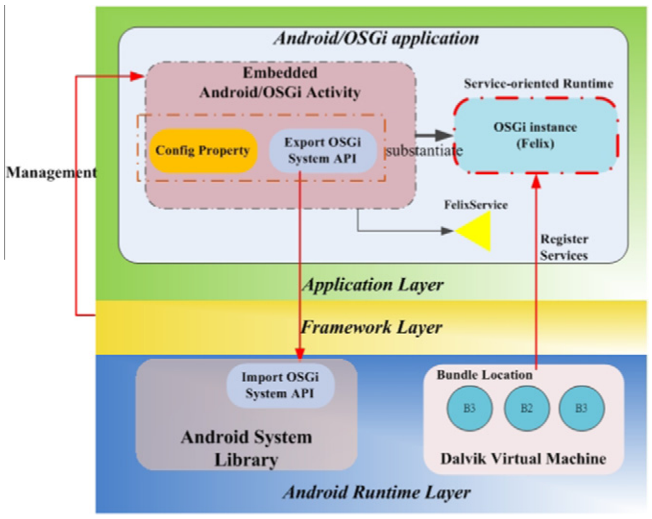


Fig. 17. Tightly integrated Android/OSGi environment.

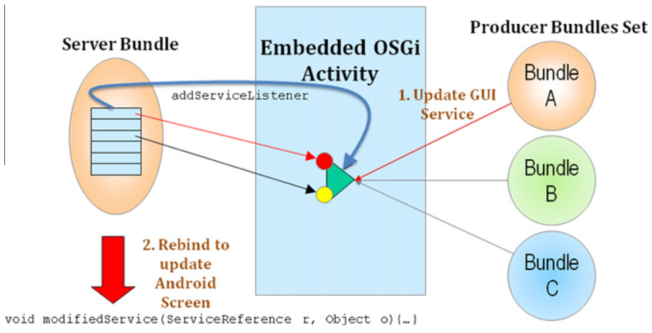


Fig. 18. Service tracker of embedded OSGi activity.

OSGi Activity utilizes a child thread to call the ***setContentView*** method to update the foreground screen for the user.

3.5. Self-managing middleware for the Android platform

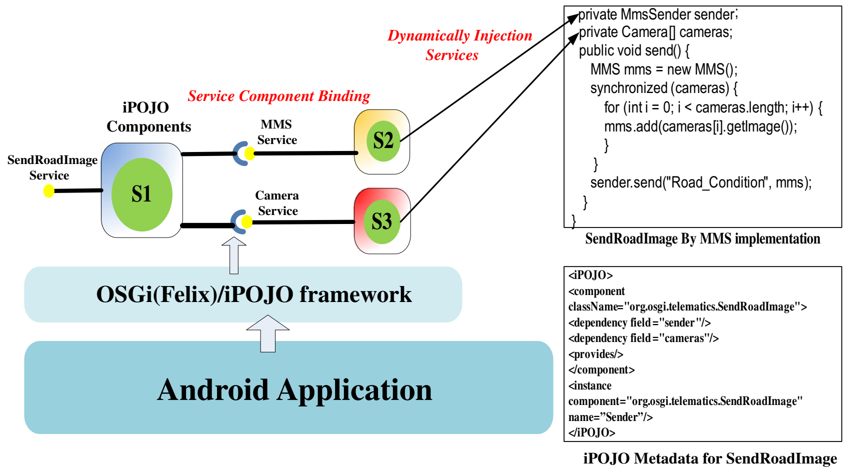
We ported the iPOJO service-oriented framework into the resource directory of the Android/OSGi application using the Fileinstall Activator, which has the main goal of making Android/OSGi applications more adaptive and flexible. This approach not only allows any component with self-managing capability to publish/discover services using iPOJO handlers, but also reduces system complexity. Besides, an Android/OSGi application can dynamically react to the changing availability of services in capricious environments. The iPOJO framework extends the ***declarative service*** concepts defined in OSGi R4 version, which also makes allows services to be defined in iPOJO metadata. Then, if the provision component becomes valid, it will spontaneously inject services to on-demand components from metadata in runtime without any interruptions from the OSGi framework. In the same way, if the provision component becomes invalid, the on-demand components will unbind the referenced services to ensure system reliability. However, the iPOJO framework has two limitations in Dalvik VM that prevent it from support dynamic generated classes and composite services. Fig. 19 illustrates the ***SendRoadImage*** component, which allows road-side center to monitor the road conditions of remote vehicle. It uses two services: the first required service is MmsService, which allows a client to send an MMS message, and the second service is Camera, which allows a client to take a monitor image using camera hardware in the Android platform. When the SendRoadImage component becomes available, it can easily use the above-mentioned services through defined iPOJO metadata. Therefore, developers only need to be concerned with the logic design of this component.

4. System design and implementation of Android/OSGi platform

Testing the proposed telematics applications in real vehicles is inconvenient and expensive. Consequently as Fig. 20 illustrates, a vehicular testbed was built to test and verify the functionalities and security of the proposed Android/OSGi applications. This study regards sensor-based LEGO robots as simulated vehicles, and makes the vehicular Android/OSGi platform act as an on-board terminal for simulated vehicles. Instead of utilizing car buses, this on-board terminal mainly controls simulated vehicles using Bluetooth, like IBM iDrive systems. Communications between different vehicles is conducted via Wi-Fi. Finally, this study designs a provisioning server responsible for storing telematics applications and deploying an admin bundle called the management agent, which assign the location of remote provision server. When this agent has been started, it installs and starts remotely downloaded applications, and also manages their packages and resources.

4.1. Vehicular Android/OSGi platform

The Android/OSGi bundles which mainly contain standardized BundleActivator interface and BundleContext object that make


Dynamically Injection Services
Service Component Binding
iPOJO Components
MMS Service
SendRoadImage Service
S1
S2
Camera Service
S3
OSGi(Felix)/iPOJO framework
Android Application
SendRoadImage By MMS implementation
iPOJO Metadata for SendRoadImage

```
private MmsSender sender;
private Camera[] cameras;
public void send() {
   MMS mms = new MMS();
   synchronized (cameras) {
      for (int i = 0; i < cameras.length; i++) {
      mms.add(cameras[i].getImage());
      }
    }
   sender.send("Road_Condition", mms);
  }
}
```

```
<iPOJO>
<component
className="org.osgi.telematics.SendRoadImage">
<dependency field="sender"/>
<dependency field="cameras"/>
<provides/>
</component>
<instance
component="org.osgi.telematics.SendRoadImage"
name="Sender"/>
</iPOJO>
```

Fig. 19. The iPOJO component in Android application layer.

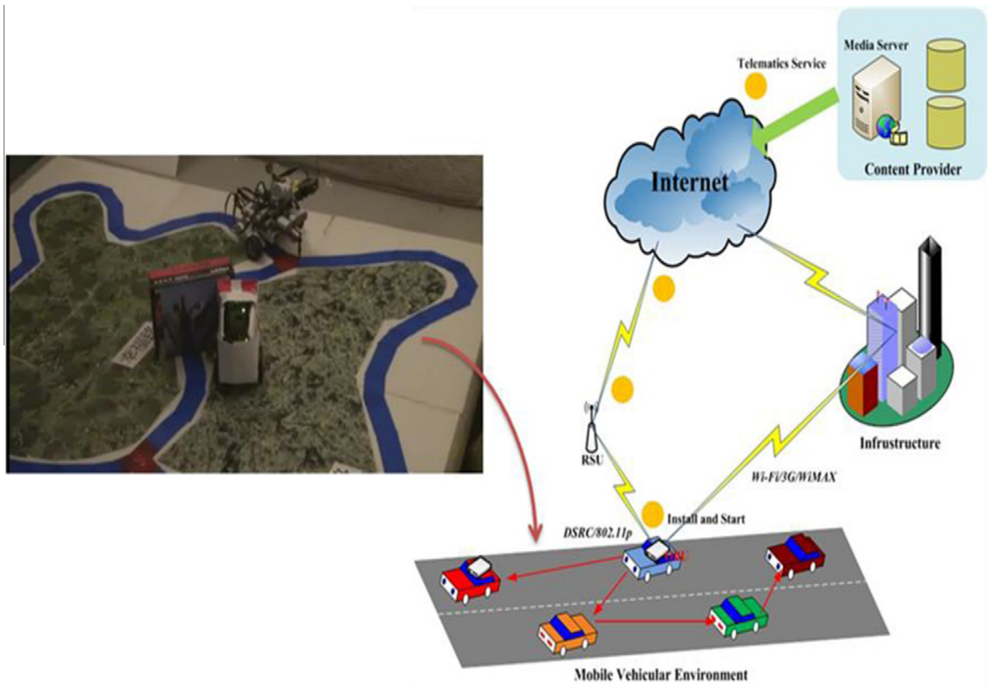


Fig. 20. Vehicular testbed.

OSGi framework can manage their life cycle and service registration or publishing. They also include required Android libraries into their descriptive Manifest files, so that OSGi framework can resolve these special packages. Fig. 21 illustrates inter-architecture of Vehicular embedded Android/OSGi platform, in first, because Android/OSGi platform can only update its foreground screen after calling setContentView method, Android/OSGi application must to provide ***ViewFactory*** objects make other Android/OSGi bundles being able to provide or update their Android GUI interfaces into main Activity such as ImageView, MapView, VideoView, and so

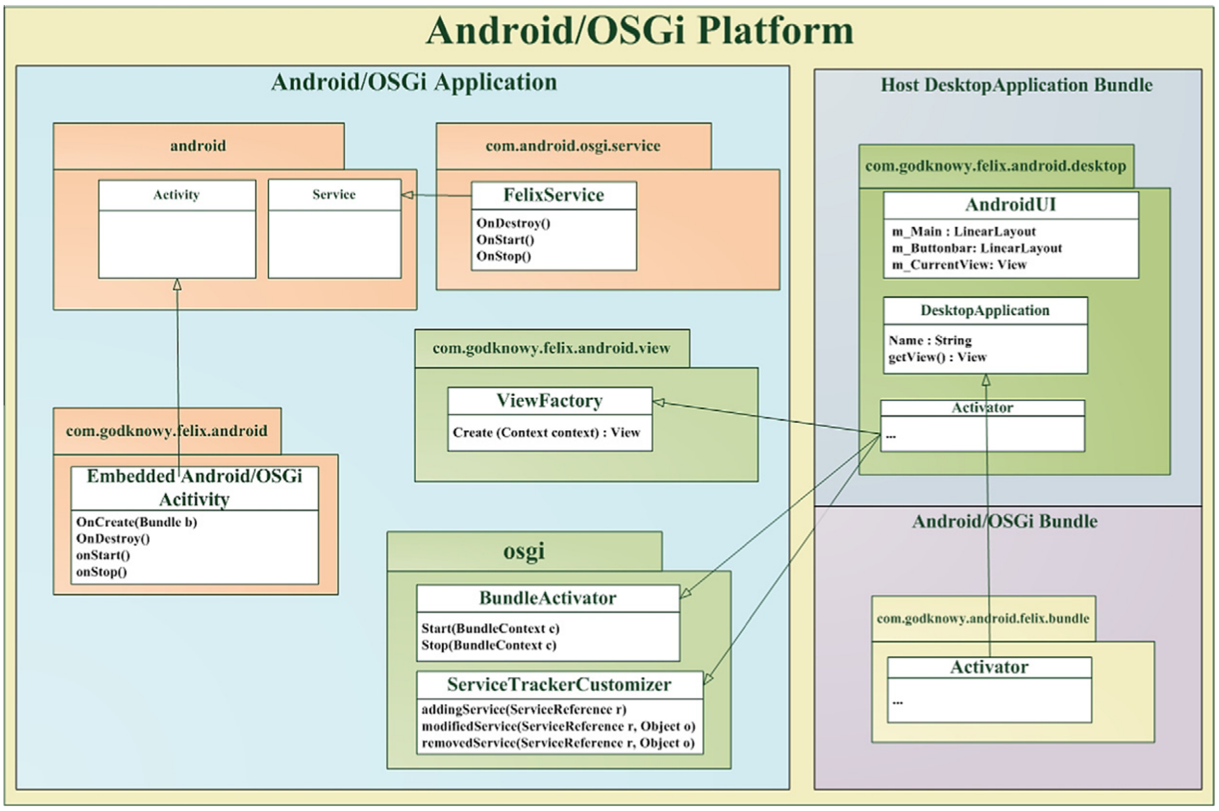


Fig. 21. Inter-architecture of Android/OSGi platform.

on. Additionally, the Android/OSGi application should register its Activity instance service, so that host bundle can fetch this main Activity, and push registered GUI interfaces into it. The host bundle implements a ViewFactory interface with the primary responsibility of providing ***DesktopApplication*** service to other consumer bundles. The second service of DesktopApplication acts as a container for any Android View in the bundles. Consequently, if any bundles want to provide some Views for the main Activity, they must implement the DesktopApplication interface, and register their DesktopApplication services with the OSGi service registry. Besides, the host bundle implements ***ServiceTrackerCustomizer*** interface, which makes host bundle being able to track primary Activity service and other DesktopApplication services simultaneously in customized way. Supposing quantity of DesktopApplication service greater than zero in OSGi registry, host bundle can call ***getView*** method belong to DesktopApplication interface to obtain container instances of Android View which GUI-based bundles provided. Finally, host bundle will add one of instances into main ***LinearLayout*** object, and return it to Android/OSGi Activity. In the proposed mechanism, only one bundle can provide GUI services to the main Android/OSGi Activity, based on user interaction.

4.2. Android/OSGi bundles

4.2.1. GUI-based bundles

The original Android platform allows developers to create map-based Activities using Google Map as a user interface element. We have full permissions to access the map, include change the zoom level, move the centered location, use overlays, provide map-contextualized information and functionality, and so on. But if service providers want to design navigation applications based on map-based Activities, they must to create many heavyweight services to perform time-consuming operations. Besides, Android platform lack of class-sharing among different applications and islands of no-interoperable processes, which often limit usage of system resources and remain non-responsive. Fig. 22 illustrates vehicular Android/OSGi platform has feature of lightweight bundles such as navigation bundle, which bundle size is less 11 KB than original Android application. But vehicular Android/OSGi platform must to spend about 7.5 s to start OSGi framework in first time, and additional memory size of OSGi framework. In tested Android hardware, it has total 990440 KB memory size, and Android/OSGi platform need to spend about 1.91 MB memory size for initiating and starting OSGi framework, and 0.25 MB memory size for starting application, however in pure Android platform same application only needs to spend about 1.21 MB memory size. In other word, if this study compares pure Android/OSGi bundles with Android applications, as Fig. 22 table shows, Android application should spend extra 0.96 MB size, but comprehensive look at total platform, Android/OSGi platform spends extra 0.95 MB memory size. However, if existence of enormous applications, Android/OSGi platform can substantially lighten its system resources based on OSGi mechanism.

Fig. 23 illustrates Android/OSGi platform spends approximately 2 MB memory size to launch OSGi framework, and when OSGi framework had been started, Android/OSGi platform can consume less memory size to start other applications. Total occupied memory size of probably lies in between 2 MB and 3 MB in Android/OSGi platform, however, pure Android platform has increased memory size, and grows up 5 MB. Consequently, proposed Android/OSGi platform can spend less memory size than pure Android platform when OSGi framework had been started.

4.2.2. Telematics services

This study presents various telematics services to establish intelligent vehicles in mobile environment, as Fig. 24 shows, which mainly consist of line follow, object detection, keep distance, and so on. The first part of service helps vehicles follows guidelines to ensure driver security and reduce route-finding complexity to lighten the use of power. The second part of this service provides

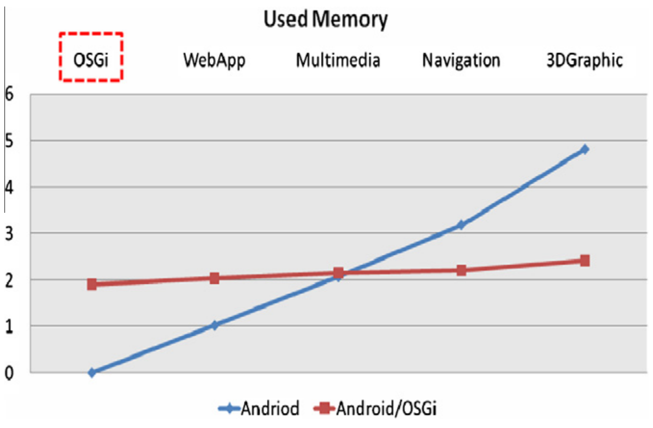
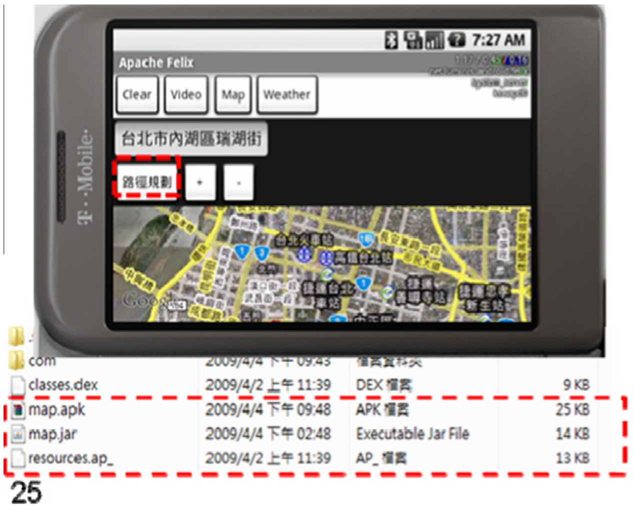
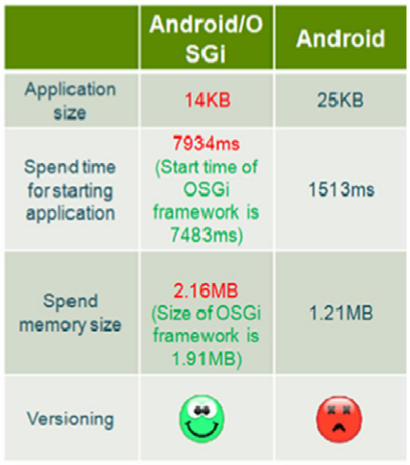


Fig. 23. Used memory analysis.

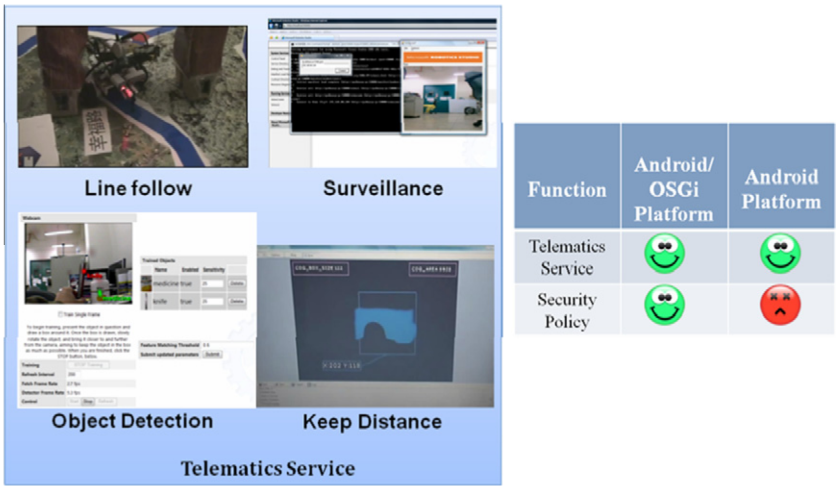


```
com              2009/4/4 下午 09:43
classes.dex      2009/4/2 上午 11:39   DEX 檔案               9 KB
map.apk          2009/4/4 下午 09:48   APK 檔案              25 KB
map.jar          2009/4/4 下午 02:48   Executable Jar File   14 KB
resources.ap_    2009/4/2 上午 11:39   AP_ 檔案              13 KB
```



	Android/OSGi	Android
Application size	14KB	25KB
Spend time for starting application	7934ms (Start time of OSGi framework is 7483ms)	1513ms
Spend memory size	2.16MB (Size of OSGi framework is 1.91MB)	1.21MB
Versioning	☺	☹

Fig. 22. Lightweight navigation application for Android/OSGi platform.



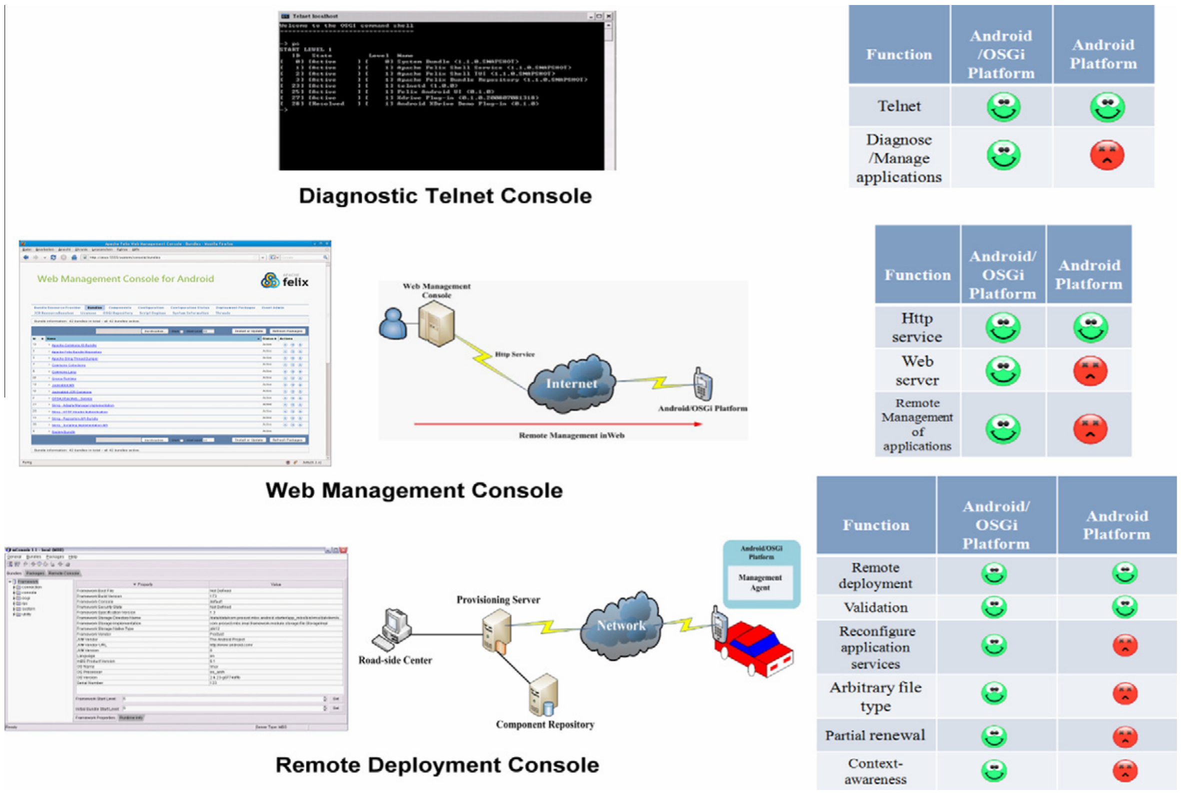
Function	Android/ OSGi Platform	Android Platform
Telematics Service	☺	☺
Security Policy	☺	☹

Fig. 24. Android/OSGi applications.

vehicles with visual intelligence to identify objects like traffic signals. The third part of the service helps vehicles maintain a set distance from vehicles in front of back. The last component of this service allows road-side centers to survey the condition of vehicles. If some of services access the significant components of vehicle, the vehicular Android/OSGi platform can enforce security policies to avoid accidents using a AOP-based OSGi weaving mechanism. For example if one service makes the vehicle travel faster than 70 km/h, this service will be rejected.

4.2.3. Remote management

Fig. 25 illustrates vehicular Android/OSGi platform mainly provide three management consoles, which include telnet, web management, and remote deployment console. The first part of the telnet console allows a remote manager to diagnose and manage Android/OSGi applications though a telnet interface, which extends the telnet bundle provided by the Oscar R3 implementation. The original Android platform cannot utilize telnet services to diagnose or manage its applications or system configuration. The second part of the web console allows remote managers to diagnose and manage Android/OSGi applications based on web techniques. This web management console relies on underlying OSGi bundles, including http service, declarative service, log service, configadmin, deployadmin, metadata, and so on. Although the original Android platform has http service that allows users to surf the internet depend on a WebKit browser, it does not have a web server, which



```
Welcome to the OSGI command shell
=================================

-> ps
START LEVEL 1
   ID   State         Level  Name
[   0] [Active     ] [    0] System Bundle (1.1.0.SNAPSHOT)
[   1] [Active     ] [    1] Apache Felix Shell Service (1.1.0.SNAPSHOT)
[   2] [Active     ] [    1] Apache Felix Shell TUI (1.1.0.SNAPSHOT)
[   3] [Active     ] [    1] Apache Felix Bundle Repository (1.1.0.SNAPSHOT)
[  23] [Active     ] [    1] telnetd (1.0.0)
[  25] [Active     ] [    1] Felix Android UI (0.1.0)
[  27] [Active     ] [    1] Xdrive Plug-in (0.1.0.2008087081318)
[  28] [Resolved   ] [    1] Android XDrive Demo Plug-in (0.1.0)
->
```

Function	Android /OSGi Platform	Android Platform
Telnet	☺	☺
Diagnose /Manage applications	☺	☹

Function	Android/ OSGi Platform	Android Platform
Http service	☺	☺
Web server	☺	☹
Remote Management of applications	☺	☹

Function	Android/ OSGi Platform	Android Platform
Remote deployment	☺	☺
Validation	☺	☺
Reconfigure application services	☺	☹
Arbitrary file type	☺	☹
Partial renewal	☺	☹
Context-awareness	☺	☹

Fig. 25. Remote management of vehicular Android/OSGi platform.

allows remote users to log into or our out the platform. The rest of the remote deployment console enables applications to be deployed on a remote vehicular Android/OSGi platform. This is similar to the Android market deployment mechanism, but has some differences. For example, the Android/OSGi platform uses a management agent to request deployment packages from a provisioning server, while a pure Android platform uses a web server to directly download Android applications from the Android Market. This management agent communicates with remote cloud called Amazon EC2 through Restful API. Android/OSGi bundles are stored in Amazon S3, and Android/OSGi platform uses management agent to automatically install/update bundles with remote cloud same as active sync. way. Consequently, the Android/OSGi platform can dynamically reconfigure its application services by communicating with a provisioning server in runtime based on context-awareness. Besides, the Android/OSGi platform supports partial renewal of applications, reducing system delay time and enhancing system performance. The Android Market should also take advantage of OSGi remote deployment mechanism using the above-mentioned advantages.

5. Conclusion

This paper introduces the principles of an embedded OSGi framework in a Google Android platform. The proposed vehicular Android/OSGi platform provides various unique features:

- Rich class-sharing
- Loose-coupled and lightweight services
- Easy API management
- Automatic service change response
- Remote device management
- Dynamic application update services

Using the proposed vehicular Android/OSGi platform, road-side centers can diagnose or manage the system status of vehicular platform remotely, and use visual intelligence to continually update their application services based on context-awareness without user intervention. This study also establishes a vehicular testbed to verify and analyze the functionalities of the Android/OSGi and original Android platform. Finally, this study proves that the proposed Android/OSGi platform has lighter applications and higher performance than a pure Android platform when performing complicated operations. This study will continue to correct system lack like combine other technology and architecture to achieve the best performance.

References

[1] Android Platform Official Site, <http://www.android.com/>.
[2] Open Mobile Alliance Official Site, <http://www.openmobilealliance.org/>.
[3] Google Android software stack in brief, <http://fuyichin.blogspot.com/2008/06/google-android-software-stack-in-brief.html>.
[4] OSGi Alliance, OSGi service platform, core specification release 4. Draft, July 2005.
[5] GST "Global System for Telematics" Official Site, <http://www.gstforum.org/>.
[6] 3GT Project Website, <http://www.ertico.com/en/activities/activities/3gt.htm>.
[7] European ITEA Project Website, <http://ralyx.inria.fr/2004/Raweb/trio/uid69.html>.
[8] Stadtinfokoeln Project Website, <http://www.koeln.de/koeln/die_domstadt/verkehr/>.
[9] Eureka-project TOP-IQ Website, <http:/www.eureka.be/inaction/>.
[10] About the OSGi Service Platform, OSGi Technical Whitepaper 4.1, June 2007.
[11] C. Lee, D. Nordstedt, S. Helal, Enabling smart spaces with OSGi, IEEE Pervasive Computing 2 (3) (2003) 89–94.
[12] K. Myoung, J. Heo, W.H. Kwon, D.S. Kim, Design and implementation of home network control protocol on OSGi for home automation, in: Proceedings of the IEEE International Conference on Advanced Communication Technology, vol. 2, July 2005, pp. 1163–1168.
[13] N. Gun, A. Held, J. Kaiser, Proactive services in a distributed traffic telematics application, in: Proceedings of the International Workshop on Mobile Communication Over Wireless LAN: Research and Applications, September 2001.
[14] Y. Li, F. Wang, F. He, Z. Li, OSGi-based service gateway architecture for intelligent automobiles, in: Proceedings of the IEEE International Conference on Vehicles, June 2005, pp. 861–865.
[15] D. Zhang, H.W. Xiao, K. Hackbarth, OSGi based service infrastructure for context aware automotive telematics, in: Proceedings of the IEEE International Conference on Vehicular Technology, vol. 5, May 2004, pp. 2957–2961.
[16] Y. Zhou, X. Wang, M. Zhou, The research and realization for passenger car CAN bus, in: Proceedings of the IEEE International Forum on Strategic Technology, October 2006, pp. 244–247.
[17] E. Weiss, G. Gehlen, S. Lukas, C. Rokitansky, MYCAREVENT – vehicular communication gateway for car maintenance and remote diagnosis, in: Proceedings of the IEEE International Conference on Computers and Communications, June 2006, pp. 318–323.
[18] R.C. Hsu, L.R. Chen, integrated embedded system architecture for in-vehicle telematics and infotainment system, in: Proceedings of the IEEE International Symposium on Industrial Electronics, vol. 4, June 2005, pp. 1409–1414.
[19] S.V. Alberto, M.D. Natale, Embedded system design for automotive applications, in: Proceedings of IEEE Computer Society on Computer Engineering, vol. 40, October 2007, pp. 42–51.
[20] Y. Ai, Y. Sun, W. Huang, X. Qiao, OSGi based integrated service platform for automotive telematics, in: Proceedings of the IEEE International Conference on Vehicular Electronic and Safety, December 2007, pp. 1–6.
[21] Y. Sun, W.L. Huang, S.M. Tang, X. Qiao, F.Y. Wang, Design of an OSEK/VDX and OSGi-based embedded software platform for vehicular applications, in: Proceedings of the IEEE International Conference on Vehicular Electronic and Safety, December 2007, pp. 1–6.
[22] P.H. Phung, D. Sands, Security policy enforcement in the OSGi framework using aspect-oriented programming, in: Proceedings of the IEEE International Conference on Computer Software and Applications, August 2008, pp. 1076–1082.
[23] R. Hyun, C. Sung, P. Shiquan, K. Sung, The design of remote vehicle management system based on OMA DM protocol and AUTOSAR S/W architecture, in: Proceedings of the IEEE International Conference on Advanced Language Processing and Web Information Technology, July 2008, pp. 393–397.