An In-Vehicle Infotainment Software Architecture Based on Google Android

Conference Paper · August 2009

DOI: 10.1109/SIES.2009.5196223 · Source: IEEE Xplore

3 authors:

Gianpaolo Macario

4 PUBLICATIONS **34** CITATIONS

SEE PROFILE

Marco Torchiano

Politecnico di Torino

158 PUBLICATIONS **2,046** CITATIONS

SEE PROFILE

Massimo Violante

Politecnico di Torino

248 PUBLICATIONS **3,651** CITATIONS

SEE PROFILE

An In-Vehicle Infotainment Software Architecture Based on Google Android

Gianpaolo Macario

Magneti Marelli Electronic Systems
Torino, Italy
gianpaolo.macario@mmarelli-se.com

Marco Torchiano, Massimo Violante
Computer and Control Department
Politecnico di Torino
Torino, Italy
{marco.torchiano, massimo.violante}@polito.it

Abstract— **The automotive infotainment industry is currently pressured with many challenges. Tier-one manufactures must accommodate disparate and quickly changing features for different carmakers. Moreover, the use of a dedicated platform for each brand and model is no more viable. The use of an open platform would permit sharing costs across the whole customer spectrum, and it will allow products to grow and adapt to the user preferences, by providing the possibility of executing third-party applications. Google Android is a recent operating system, designed for mobile devices that perfectly fits to embedded devices such as those used for automotive infotainment. In this paper we present a proof-of-concept architecture developed in cooperation between Magneti Marelli and Politecnico di Torino, whose main contribution is an automotive-oriented extension of Google Android that provides features for combining extendibility and safety requirements.**

Automotive; In-vehicle infotainment; Open-source; Software architecture

I. INTRODUCTION

The automotive infotainment industry is currently heavily pressured with a horde of requirements. The customers are used to the most breathtaking features on their mobile phones and they expect the same from their vehicles, which are, at any rate, bigger and more expensive.

Tier-one manufactures need to accommodate different requirements from different carmakers; moreover, frequently, different models of the same brand may need a different platform. Furthermore, users are today used to customize their personal devices both in terms of user interface and applications, and therefore they are expecting the same functionalities from infotainment systems that equip vehicles. In the case of mobile phones, which can be considered as best-effort systems, the possibility of running third-party applications, downloaded from the web and installed on the device, is already a reality. To migrate the same functionality to infotainment systems, designers have to face a difficult challenge. Third-party applications must be executed in a segregated environment, to prevent them from interfering with the vehicle, creating hazard to the safety of its occupants. At the same time, third-party applications must be allowed (through suitable authentication mechanisms) to access to vehicle functions to provide innovative services (e.g., augmented navigation facility by exploiting dead reckoning algorithms based on the actual vehicle speed as read from the vehicle CAN bus). Finally, the developers of third-party applications should be provided with a set of vehicle features common to different carmakers, so that the same application can be used seamlessly on different brands of vehicles. Vehicle functions should be provided at high abstraction levels, enabling their usage without disclosing proprietary information.

In this paper we introduce an innovative extension to Google Android that provides support for third-party application segregation: a safety related layer provides an abstraction of the vehicle functions that can be used by trusted third party applications only, while not trusted applications cannot access vehicle functions therefore preventing interference with vehicle safety.

This paper is organized as follows: section II presents an overview of the open issues in automotive infotainment, section III describes the main features of Android, section IV illustrates the proposed architecture, and section V draws the conclusions.

II. BACKGROUND

Requirements for In-Vehicle Infotainment (IVI) are more complex than other consumer electronics devices. While IVI units seem to offer similar features to those available on Personal Digital Assistants (PDAs), Smart Phones, Media Players, etc, they need to provide to the vehicle users and passengers a seamless user experience without interfering with the actual driving, which is the main and most important task when using a vehicle. Also, most IVI units come factory-fitted and must be working across the lifetime of a vehicle, which is

typically 5 to 10 times longer than a mobile phone or media player.

Those factors have a deep impact on the OEM request to shorten the design cycle and support post-factory upgrade of applications and features. Also, due to market pressure, the Research and Development costs must be kept under control.

The GENIVI Alliance [4] has recently been established between leading automotive manufacturers, tier-ones, silicon vendors and software providers with the purpose of collecting requirements, developing common architectures and reference implementations for the development of standardized and open platforms for IVI.

One of the objectives of the GENIVI Alliance is to develop a scalable architecture that may be deployed to future generation of IVI units with the increasing benefits of:

- Adding more contents and features through the seamless integration of software components adopted from the open source community or specifically developed by the Alliance or its partners, always keeping into account the system security/safety.
- Increasing the performances perceived by the end-user (i.e. vehicle owner or passengers) without incurring into a corresponding price penalty in the per-unit cost, or even better trying to reduce the final device cost.

The GENIVI software architecture leverages the Moblin framework [2] by adding or extending components that will address specific automotive requirements and use cases. The current target for GENIVI is the low-level middleware that is developed as native code, within the host operating system, to allow a close interaction with the underlying hardware platform. The software layer that supports user application is still under discussion, and a number of alternatives are under evaluation, being Google Android one of them.

It is important to stress the fact that the current strength of GENIVI and Moblin consists in the support for the automotive specific devices (e.g. CAN networks) that is missing in Android. On the other end, Android exhibits a strong support for managed code, allowing an easy deployment of end-user applications even by third parties, these features are not addressed in Moblin yet.

III. GOOGLE ANDROID

At the end of 2007 the Open Handset Alliance presented Google Android [3], a complete and free mobile platform, whose goal is to allow the development and diffusion of mobile applications that potentially will be delivered to a wide range of devices.

A fundamental feature of Android consists in its openness: a free SDK (Software Development Kit) is available for developers who wish to address this platform. This quickly prompted the formation of a large developers' community. Android is a software platform developed specifically for mobile devices, it includes an operating system, middleware and a few bundled applications. In the following sub-sections we briefly outline its main features. The full project is released under Apache version two license, which basically has no copyleft clause. Therefore mobile operators, software companies, and any developer can add or remove features. In agreement with the Web 2.0 paradigm, information sharing between different processes and applications is possible though content providers. The platform provides a few native content providers (e.g. media, contacts, etc.) and new ones can be added, so enabling the developers to build rich peer-to-peer applications.

A. Architecture overview

The high-level architecture of Android consists of five main components:

- Linux kernel,
- Libraries,
- Android runtime,
- Application framework,
- Applications.

At the bottom of the hierarchy there is the Linux Kernel. Basically, it is the 2.6.27 version of Linux Kernel, to which are applied Android specific patches. This element is responsible of managing the core system services and driver model. The root file system uses *rootfs*, whereas data and system are using *YAFFS*, which is a file system specifically designed for NAND and NOR Flash drives.

The application framework and Android runtime rely on a set of C/C++ libraries. The set of libraries include the standard C libraries, media libraries, graphical libraries, a browser engine (LibWebCore), font libraries (FreeType), and database libraries (SQLite).

The Android runtime consists of Core libraries and the Dalvik Java virtual machine. Dalvik is optimized to allow multiple instances of the virtual machine to run at the same time using a limited amount of memory. Each instance runs in a separate Linux process.

The application framework is a large set of classes, interfaces, and packages. Its goal is to provide an easy and consistent way to manage graphical user interfaces, access resources and content, receive notifications, or to handle incoming calls. The main components are: the view system, the activity manager, content providers, the resource manager, the notification manager, and the telephony manager.

B. Security

Android inter-process communication and security are designed to keep the system as stable as possible in presence of user-installed applications.

The low-level permission mechanisms are provided by Linux (i.e. kernel and file system) and they are functionally equivalent to any other Linux-based system. Since Android devices are intended to be inherently single user, the multiuser facilities are used to provide high inter-application security by assigning each application a unique user.

In addition Android provides a static permission system that is enforced at application install time.

C. Inter-application communication

Google Android has two inter-application communication modes: *intents* and *code binding*.

The *intents framework* provides a high level Inter Process Communication (IPC). This is the best way to implement dynamic functionality binding between applications developed using the SDK. The Intent class contains several fields describing what a caller would like to do. The caller sends its intent to Android's intent resolver, which looks through the intent filters of all applications to find the activity most suited to handle the issued intent. Intent fields include the desired action, category, data string, MIME type of the data, handling class, and security restrictions. Intents can be used to launch activities, to send data in broadcast, and to start services. The security restrictions are implemented using the *permissions framework* provided by Android.

Since each application runs in its own process, and programmers can write a service that runs in a different process than the application user interface, sometimes object passing between processes is required. In the Android platform one process normally cannot access the memory of another process. Therefore to communicate, two processes need to decompose their objects into primitives that the operating system can understand, and marshal the object across the process boundary. The Android Interface Definition Language (AIDL) tool provided with SDK creates the marshalling code automatically. AIDL is an Interface Description Language (IDL) used to generate code that enables two processes to interact using IPC. The AIDL IPC mechanism uses a proxy class to pass values between the client and the implementation.

IV. Proposed Software Architecture

The overall architecture of the automotive extensions to Android is presented in Figure 1. The custom Android platform applications sit aside the automotive-specific application and supporting components. The goal of the extension is to provide a safe mechanism for allowing trusted applications to access to vehicle's functions (e.g., reading/writing information on the vehicle CAN bus), while not trusted applications are segregated and cannot access to vehicle functions. The enforcement of safety policies among applications is motivated by the dependability requirement posed by IVI systems. Although not in charge of managing vital functions of the vehicle (like braking, steering, or torque distribution), being IVIs integrated in the vehicle, they have access to features (like the vehicle CAN bus) that, if handled improperly, can jeopardize the vehicle safety (e.g., saturating the bandwidth of the CAN bus by sending useless data frames continuously).

The main feature of the proposed extension of the Android platform is the decoupling of the high value-added logic, which processes and provides data to applications from the infrastructure used to access the low-level raw data coming or going from/to vehicle functions.

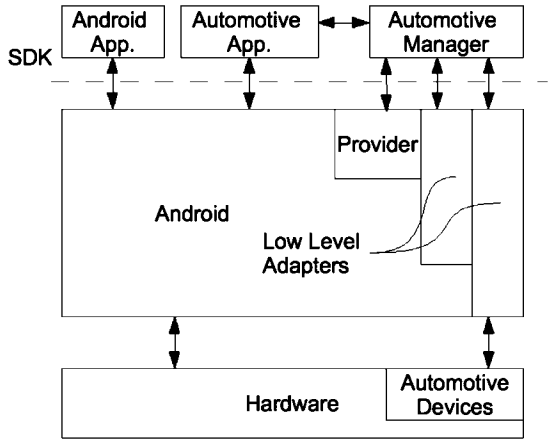

Figure 1. Architecture overview.

A. Automotive Manager

The Automotive Manager is an application that takes care of interacting with the automotive extension of the Android platform.

It has two interfaces: one to the applications and another to the components of the platform. It is an ordinary Android application developed upon the Automotive Android SDK and signed with platform certificate. Since the manager resides outside the platform, the user can update it without help from specialized personnel (even trough the internet).

B. Interaction with Automotive Applications

Android is based upon an *opaque* IPC model. Applications expose to the system their functionalities and, at runtime, other applications can request these functionalities. Essentially, the platform provides a managed and secured late code binding. This model is

used also in the interaction between third-party applications and automotive manager.

The automotive manager handles the car model by means of a set of *properties*.

For each property (e.g., the frame on the vehicle CAN bus delivering the actual speed), two Android permissions (for reading and writing) are created and assigned to the manager, e.g.:

```
android.permission.car.SPEED.read
android.permission.car.SPEED.write
```

Then, leveraging the predefined security levels, it is possible to assign different levels of security for each of these permissions:

1) All: access is allowed to applications from anyone.

2) Normal: access is controlled by permissions but such permissions are given to applications without explicit user intervention; it is possible to revoke manually such permissions after application installation.

3) Dangerous: access is controlled by permissions and the user is asked explicitly about the permission grant during application installation.

4) Signature: access is controlled by permissions which are granted automatically and if and only if the application is signed with the platform certificate.

The platform certificate is the one the manufacturer uses to sign platforms during installation. It is used also to sign the automotive manager. If a third-party application is signed with this certificate it has full control of automotive extension (in fact automotive manager is just an application signed with platform certificate).

Each vehicle function for which a property is defined can be accessed through a high level AIDL interface:

```
interface IAutomotiveManager {
  void write(String prop, Bundle data);
  Bundle read(String prop);
  int addListener(String prop,
                  IAutomotiveListener l);
  void removeListener(String prop,
                  IAutomotiveListener l);
}
```

The *read()* and *write()* methods allow accessing the properties' values in input and output. The *addListener()* and *removeListener()* methods allow registering and unregistering a callback for asynchronous notification of a property value changes.

The automotive manager implements the interface and provides those methods. Individual calls are checked against Android permissions of the related properties. If the caller is allowed to perform the requested action the automotive manager proceeds to handle it, otherwise a security exception is thrown.

The developer who writes a third-party application and want to interact with a property needs to know only the property name (also known as tree-location) and the data type. Moreover all the interactions take place through the previous AIDL interface. This implies, thanks to the architecture of Android, that developers does not need to known the entire Automotive SDK but only the AIDL file defining the calls shown previously and the AIDL file that describes the callbacks.

Moreover, if different IVIs coming from different manufactures offer the same properties (which are implemented in different ways on different hardware installed in different vehicles), a third-party application using such properties can run seamlessly on the IVIs.

C. Prototype

To prove the feasibility of the architecture outlined above, we developed a fully working prototype, which we do not present here for lack of space.

The prototype consists in a proof-of-concept customized release of Android. The prototype has been tested on both the Android emulator, based on ARM processor, which we customized to model the typical functionalities and user interface of an IVI, and on a netbook powered by an Intel Atom processor.

V. Conclusions

In the paper we outlined the main issues of automotive infotainment systems and presented the potential of using an open source solution. We designed a proof-of-concept architecture to extend the Google Android platform to accommodate automotive-specific needs. In addition we developed a fully working prototype that implements the architecture and runs on general-purpose hardware.

The next step in our work will be to port the system on custom hardware and access actual automotive data.

Acknowledgment

The authors wish to thank the students Filippo Pagin and Luca Belluccini for their precious work.

References

[1] Enck, W.; Ongtang, M.; McDaniel, P., "Understanding Android Security", IEEE Security & Privacy, 7(1), pp.50-57, 2009.

[2] Moblin home page, at http://moblin.org, last visited on April 22, 2009.

[3] Android home page, at http://www.android.com/, last visited on April 22, 2009.

[4] GENIVI home page, at http://www.genivi.org/, last visited on April 22, 2009.