A Java Application Programming Interface for In-Vehicle Infotainment Devices

B. Kovacevic, *Graduate student member, IEEE*, M. Kovacevic, *Graduate student member, IEEE*, T. Maruna, *Member, IEEE*, and I. Papp, *Member, IEEE*

***Abstract*—Traveling long distances by vehicle in the modern world has become bearable with the availability of In-Vehicle Infotainment systems to entertain and inform both drivers and other occupants. One of the problems faced by application developers for In-Vehicle Infotainment systems is the lack of an open and standard platform across different vehicle manufacturers. The Application Programming Interface and the concepts proposed in this paper should reduce the closed, proprietary and non-extensible systems being released today and bolster the design and development of open, more complete, feature-full systems in vehicles that will not only assist but also entertain drivers and passengers.**

***Index Terms*— API specification, infotainment, multimedia, vehicle**

I. Introduction

IMPORTANT factors concerning an open system to be used in In-Vehicle Infotainment (IVI) systems is the ability to use the same software across multiple lines of products produced by the same manufacturer. With this ability, the firmware can easily be updated across all products, thereby removing the need to update each product separately. Currently, there is a need for the development of an infotainment system that runs on an open source platform and can be customized by users. For example, a taxi company using this system can design and develop their own custom made application that can be deployed in all their vehicles to monitor their locations in real-time. Some examples, like the one created by Tahat *et al.* [1], include Global Positioning System (GPS) tracking of vehicles in time critical jobs where tracking is necessary. Others include the development of applications that interact with the vehicle's internal system, such as analysis of the data provided by the vehicle's computer.

Manuscript received November 15, 2016; accepted February 28, 2017. Date of publication April 12, 2017. This work was partially supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia, under grant number III44009. *(Corresponding author: M. Kovacevic.)*

Branimir Kovacevic, Marko Kovacevic, and Istvan Papp are with the Computer Engineering and Computer Communications Department, Faculty of Technical Sciences, University of Novi Sad, Serbia (e-mail: branimir.kovacevic@rt-rk.com, marko.kovacevic@rtrk.uns.ac.rs, and istvan.papp@rt-rk.uns.ac.rs).

Tomislav Maruna is with the RT-RK Institute for Computer Based Systems, Novi Sad, Serbia (e-mail: tomislav.maruna@rt-rk.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCE.2017.014655

Each vehicle manufacturer provides its systems that are proprietary, closed, and offer no support for extensibility. Using the Java Application Programming Interfaces (API) proposed in this paper, developers can create custom applications that further expand the life cycle of the product. This will grant devices with newer applications and features that can be installed on demand by users, and thus satisfy each user's personal requirements.

Modern IVI devices are usually produced by vehicle manufacturers to be used in their own vehicles, or by aftermarket device manufacturers, usually within the in-vehicle entertainment industry. These infotainment systems are pre-installed before the purchase of the vehicle or can be offered as an "add-on" for consumers to purchase and install at any time. The software for these systems is developed by one of two methods: in-house, using proprietary custom built software, or built on top of an existing operating system and modified accordingly to match requirements within the limits of the adopted operating system. This paper proposes an integrated Java API that provides a centralized point of infotainment by incorporating entertainment and informative services to enhance driver and passenger experiences in the vehicle. Application APIs and an application base give device manufacturers the assurance that there will be a large application base with developers constantly providing applications for their platform. This leads to a shortening of the device manufacturing cycle, as manufacturers will have reliable applications that have been well tested. This will reduce development time, enabling manufacturers to focus only on the hardware components of the system (screen, input devices, etc.).

Users want to connect their mobile device and get content from it in the vehicle, allowing the latest technologies to be available in their vehicles, making this space as seamlessly connected as their living environments and containing the same connectivity features as consumer devices. However, current trends are such that developers have dedicated APIs that are used to develop applications on mobile devices so that the mobile device can interact with the vehicle and the IVI system. Helping the developers, by creating Integrated Development Environment (IDE) tools and uniform API, will also help Original Equipment Manufacturers (OEM) in the process of creating final products that are more desirable and within budget.

By using the proposed Java API, there is no need for developer to know exact hardware and software platform. Instead, it is sufficient to use the proposed Java API, which hides platform specific details from developers. The power of a full and complete abstracted API to enable the reading of all signals and sensor data from a vehicle should not be underestimated. The proposed Java API has an imperative that enables the open source IVI community rapid and easy prototyping, testing and production of innovative user experience concepts. Proposed Java API tends to become a widely used set of technologies to build applications across IVI platforms. Applications created with proposed Java API can quickly be adapted to different branded devices. Developers, by using the proposed API, are able to create new concepts for the vehicle rapidly in a way that is standardized across multiple brands.

Developers and OEMs can create experiences that let vehicle brands and third parties shine but keep costs in check, by reusing their already created applications. One example of this reusability is a differentiation on user interface application levels between family vehicles, personal vehicles, and rental vehicles, leaving the core platform the same for all.

As stated by Macario *et al.* [2], the automotive infotainment industry is currently pressured with many challenges. Tier-one manufacturers must accommodate the disparate and quickly changing features for different OEMs. Moreover, the use of a dedicated platform for each brand and model is no longer viable. The use of an open platform would permit sharing costs across the whole consumer spectrum, and also would allow products to grow and adapt to the user preferences, by providing the possibility of executing third-party applications.

The automotive infotainment industry is currently heavily pressured with a horde of expectations. Consumers 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. The automotive industry needs to be able to develop, acquire, and integrate software just as easily as the mobile phone industry. Tier-one manufacturers need to accommodate different requirements from different OEMs, moreover, frequently, different models of the same brand may need a different platform. Furthermore, today's consumers are used to customizing their personal devices both in terms of user interface and applications, and therefore they expect the same functionalities from IVI 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 IVI systems, designers have to face a difficult challenge. The developers of third-party applications should be provided with a set of vehicle features common to different OEMs 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.

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. These 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 (R&D) costs must be kept under control. One of the objectives defined by Java API proposed in this paper is to develop a scalable architecture that may be deployed to the future generation of IVI systems with the benefits of adding more content and features through the seamless integration of software components adopted from the open source community.

Flaws identified in today's IVI systems include:

- Development on different platforms
- Missing standardized application programming interface to develop applications for IVI devices
- Missing possibility of application distribution between different vehicle manufacturers
- Missing community for the development of IVI applications
- Missing option to upgrade IVI system (due to proprietary and closed source code)

In such an environment, IVI system development costs are high. There is a large gap between consumer electronics and vehicles (services and applications in vehicles are already out of date when they appear on the market).

Current IVI systems have a common feature base which includes: navigation, radio, multimedia, climate control, vehicle data monitoring and basic safety support. On-board diagnostics - is a reporting and diagnostics feature that is present in almost all vehicles on the road (with slightly different variants). The diagnostics collects and reports relevant data that can help identify problems or provide analysis of various factors. By using the standardized API, as the one proposed in this paper, and exploiting different API functionalities proposed in this paper, developers can use voice recognition to initiate air conditioning, activate window wipers and much more. This is why a standard framework needs to be designed, and followed by vehicle manufacturers. Vehicle manufacturers must expose API of the vehicle computer to allow developers to build applications that can control certain vehicle functionalities. Proposed API addresses all of the functionalities with a single, unified approach.

The rest of the paper is organized as follows. Section 2 considers a possibility of installing Multimedia services in IVI devices that support the Java programming language. Section 3 presents the details about proposed Java API. Section 4 presents proof of concept implementations that are based on the solution proposed in section 3. Section 5 quantifies the proposed API quality using some well-known metrics. Section 6 outlines the conclusion of the presented work and proposes future work potential.

II. Infotainment Implementation Overview

Infotainment, by definition, is the idea of broadcasting media that is intended to both entertain and inform. This originates from the idea of "blending" entertainment and information. An Infotainment System is a system that can

provide a blend of information and entertainment, usually using a screen to convey this information allowing users to input certain commands and retrieve information in different formats of media. The first example of infotainment was the television, as it provided informative media for its viewers. Today's sophisticated infotainment systems are much more interactive than the television itself. Interactivity allows users to input action commands and get the required information from the system. These devices can commonly be found in vehicles including motor vehicles and airplanes to keep passengers informed as well as entertained.

Features of IVI systems typically include a multimedia player, navigation system, the ability to read news, and the internet. There is no exact standard that can define an infotainment system, or what it must include. However, there are recurring features that appear throughout the current devices, labeled as in-vehicle entertainment or infotainment systems on the market. These include multimedia music, video, radio players, and GPS assisted navigation systems. As technology progresses new features are becoming normalized in IVI systems. One of these emerging features is Bluetooth connectivity, where a mobile phone is connected to the system to provide hands-free communication via Bluetooth interface. Another important feature is the ability to mount content from external devices via a USB interface (or other external storage slots). Currently, with more demand for natural interaction and the mass production of touch screens, the use of a touch capable screen is also becoming a more standard theme within the existing IVI industry. Current infotainment systems are being used in a wide range of areas, but this paper focuses on the ones used in an automotive environment. Infotainment systems are either fitted within the vehicle during manufacturing or fitted via an aftermarket supplier usually in the field of vehicle stereo systems.

There are certain problems associated with current IVI systems including:

- Lack of a standardized system
- Lack of an open source system
- Lack of ability to extend features of current systems
- Lack of community and third-party support

As stated by Hueger [3], compared to consumer devices, vehicles have a much longer lifecycle, with the result that a flexible solution for the integration of applications into IVI systems is needed. However, flexible solutions for the integration of new applications are not available in current vehicles. At the same time, the applications should be compatible with different systems so that development cost is minimized.

In order to fulfill the requirements regarding reusability and flexibility, Java API proposed in this paper defines both standard API and the ability to extend the functionalities with vehicle model specific API. This proposal includes a specification of a Java API that should be used to access and control in-vehicle information and entertainment related content.

Most modern IVI devices are based on proprietary systems. The integration of various services in infotainment devices is covered by many standards, but the support for the internet, games, and multimedia is not available in a systematic way. Instead of building all of the aforementioned functionalities on top of the proprietary systems, integration of IVI functionality into multimedia enabled devices could be used as an alternative approach.

Because of the fast evolution of electronic technologies, the automotive industry cannot adopt new technologies easily. IVI platform standards were developed by automotive OEM and suppliers, but as explained by Son *et al.* [4], the problem with faster adoption of new technologies still exists. Different limitations are identified in currently available IVI systems by Yamabe *et al.* [5]. The increasing activity in the automotive infotainment area faces a strong limitation: the slow pace at which the automotive industry is able to make vehicles "smarter". In contrast, Gandhewar and R. Sheikh [6] state that the smartphone industry is advancing quickly. There are already several Java-based smartphone environments that utilize the power of unified API and availability on different platforms, but there is no systematic approach in the utilization of vehicle data along with entertainment options inside the vehicle so that those options and data can be used by any application written for that particular platform.

The API proposed in this paper provides a standardized way of application development so development costs can be reduced, newly created applications can be easily introduced in already deployed devices, and the specific adaptions for certain systems can be done. The API is defined as modular as possible, in order to simplify porting of separate modules on different platforms. Even if one module is error prone, it will not jeopardize the functionality of other modules in the system. Application developers are able to provide the same applications for multiple devices, normally without modifications.

By using the proposed Java API, developers are able to create different applications that can utilize data from the vehicle. For example, Curguz *et al.* [7] created an application that sends a notification to the specified server that the driver has violated the speed limit. Another application, created by Puaca *et al.* [8], collects and processes different vehicle data in order to create a statistical output of the vehicle usage during the specified time period. Cikos *et al.* [9] created a 3D user interface application that displays relevant vehicle data, fetched by using the proposed Java API, in form of a vehicle dashboard.

Authors of the proposed API could not find any similar Java API unification for IVI systems, so there is not any good cross-comparison point available in open source community, and commercially available solutions all use proprietary APIs and do not tend to standardize their solutions.

III. IVI Integration Outline

The development of the software for IVI devices is not unified, so there is no compatibility between applications developed by different manufacturers. Some manufacturers offer applications that can be installed in the IVI, but those applications are customized for a particular device. For

example, it is possible to install a well-known navigation application into an IVI device, but the list of supported devices is limited. Furthermore, if a new model emerges, it will be necessary to adapt the application to work with that new device. Since all IVI manufacturers already have different services developed for native applications, this paper proposes additional software layers to enable support for Java applications. The proposal includes two parts: the Java API specification that is used to access various vehicle content (information & entertainment) from different applications, and the IVI native layer API. The latter is used to connect the native code to the Java API.

This paper extends the idea presented by Kovacevic *et al.* [10] in a way that it provides more details and proposes some improvements to the initial version. It proposes a Java API along with the native API that could be used to standardize IVI support in Java based applications. The Java API proposal is not bound to any operating system, or to any platform. It also presents an implementation of the IVI system, which is based on the API proposal given in this paper.

As explained by Tulach [11] designing an API is neither easy nor cheap. Trying to create an API is definitely more work than releasing a product without any API at all. Still, in the context of "cluelessness," the main message is that with a proper API, you can design better systems while minimizing your own understanding of them. Properly designing and using the APIs of individual components of the system can improve the system engineering methodologies used to design them. Improving system engineering skills is a way to maximize the benefits of cluelessness.

As explained by Bloch [12], a set of guidelines including:

- requirements gathering,
- use-cases definition,
- usage examples definition,
- and clear naming

were considered during the definition phase of the proposed Java API.

Due to the missing standardized support for Java in infotainment systems, during the creation of this Java API, a survey was conducted to define a common feature base and feature desirability for the end consumer. Different options were presented to the consumer using the Likert scale and they were asked to choose the desired option from levels of 1-5 (Must not contain this feature, Should not contain this feature, Neutral, Should contain this feature, Must contain this feature). The survey was taken by 15 participants. A visual representation of the results is given in Fig. 1.

By analyzing the results, it can be seen that the most important IVI features for the consumers are multimedia and navigation. Also, it can be observed that consumers would like to have browser and email functionalities, but these could increase the level of driver distraction. By using the Java API proposed in this paper, developers are able to combine voice interaction with email or browser functionality in order to reduce the driver distraction level, e.g. by implementing the feature of reading/writing emails with voice commands.

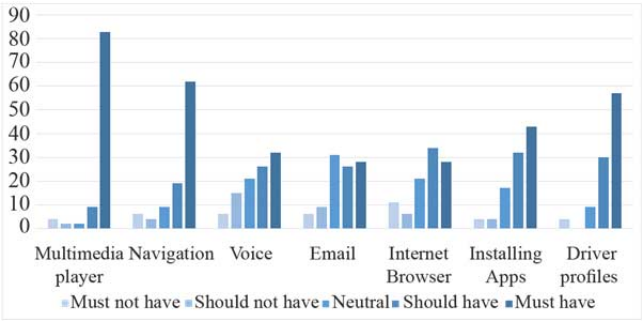

Fig. 1. IVI features desirability survey

The following key elements are identified for the implementation of Java-based IVI support in vehicles: the Java-based user interface application, the Java API, the IVI Java Native Interface (JNI), the IVI Middleware (Multimedia, Radio, Navigation, Connectivity, Climate, Vehicle Info, and Input Services - e.g. touch or voice input), and IVI Drivers, as depicted in Fig. 2.

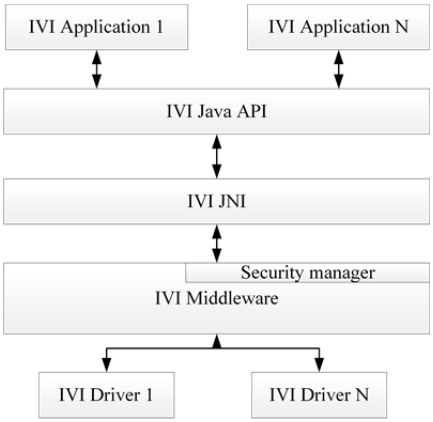

Fig. 2. Key IVI elements

The elements are defined as modular as possible, in order to simplify development of separate modules. Detailed explanations and examples of the Java API are provided here, whereas other software layers are out of the scope for this paper and will not be explained in detail.

Proposed Java API layer encapsulates IVI features into Java classes, interfaces, and enumerations, as it is depicted in Fig. 3.

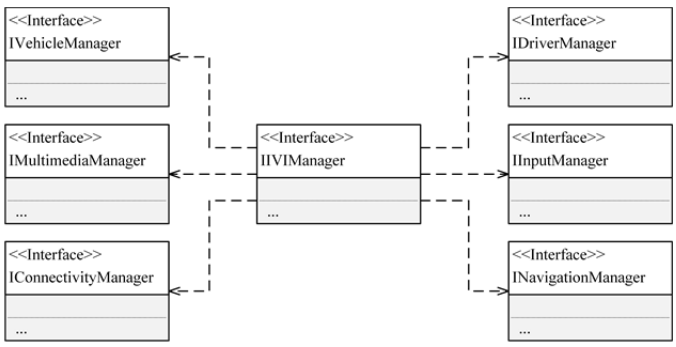

Fig. 3. Java API components

The main role of this layer is to provide the interface for creating applications with IVI features. Java API can use different native services (created by different vehicle manufacturers) to provide IVI and multimedia playback features. Native services utilize drivers, hardware abstraction layer, and different libraries. Most field-proven software stacks are already implemented in "C" programming language, because of migration from various embedded systems, or in order to achieve a significant performance gain. In order to connect already available software stacks written in "C" with Java API, the JNI layer is defined and used in this proposal. It was possible to do an automated Java-native-Java transition, like the one explained by Martin and H. A. Muller [13], but it was decided to do a manual conversion since the exact specification of source and target Java classes with interfaces was created. The platform dependent layer, so called IVI Middleware, is developed by Tier-one manufacturers.

In this proposal, access to IVI functionality is controlled by security manager (depicted in Fig. 2 as part of IVI Middleware). Access is granted by security manager only if the application that requires access is signed with the platform certificate. The platform certificate is issued by OEMs. It is used also to sign the proposed IVI Java API. If a third-party application is signed with this certificate it has access to IVI functionalities, otherwise, a security exception is thrown.

The complexity of the IVI Middleware and the need for timely responses are the reasons why it is not converted to pure Java. Instead, an adaptation layer was created, so that the IVI device drivers and IVI Middleware can be used by Java applications, as it is presented in this paper.

The JNI layer connects the IVI Java API to the native middleware code. The Java-to-native direction is implemented as an invocation of the native method in the JNI layer. This way the appropriate function of the middleware is invoked and the information is returned back to the Java layer. The other direction (native-to-Java) is a bit more complicated: when an IVI event occurs, the middleware invokes the callback method in the JNI layer (using pointers to functions), and this layer then invokes the appropriate Java method in the IVI Java API. The invocation of the Java method from the native layer is not straight-forward and is not placed in the same thread space, but it is feasible to perform such invocations by attaching the invocation thread to the Java Virtual Machine (JVM) thread.

IVI Java API encapsulates the functionality of IVI Middleware with classes, interfaces, and enumerations, all to be used by Java applications.

IVI Java API consists of several components (depicted in Fig. 3), the IVIManager component being the main one. This component holds all other components and is used to access them. It implements the IIVIManager interface, just as all other components implement their appropriate interfaces (e.g. the VehicleManager component implements the IVehicleManager interface). The IVI manager consists of the following interfaces (as depicted in Fig. 3): IIVIManager, IVehicleManager, IInputManager, IDriverManager, IMultimediaManager, IConnectivityManager, and INavigationManager.

Every component, besides the predefined set of functionalities, exposes a set of functions that are intended to be used by vehicle manufacturers for eventual API improvements.

The key API components are IVehicleManager and IMultimediaManager.

Vehicle Manager is used for gathering vehicle-related data, like fuel level, tire pressure level, speed, fault codes, etc.

As stated above, every manager provides a set of custom functions, in the case of IVehicleManager, it is ICustomDataManager. Every custom data manager is used for exposing generic APIs that can be used by vehicle manufacturers to gather custom data, only by implementing ICustomDataManager interface, without the need of defining additional APIs.

Multimedia Manager is used for manipulating audio, video, radio playback inside the vehicle.

Connectivity Manager is used for managing and browsing through different end user connection interfaces in the vehicle.

Driver Manager is used for customization of driver profiles, and storing driver habits, seat position, favorite radio stations, rearview mirror position, etc.

Input Manager is used for interaction with the end user, through touch, gesture, and button events.

Navigation Manager is used for navigation, additional customization of driver navigation, and navigation settings.

When a Java application needs to invoke a function from IVI Java API, it just gets the IVIManager component as a system service. If there is a need to get a component that is a part of the IVI service (for example Vehicle manager component that is depicted in Fig. 4), it can be fetched from the IVIManager.

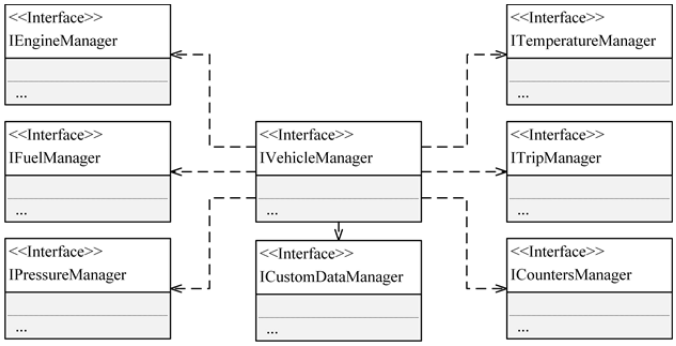

Fig. 4. Vehicle manager API interfaces

The example depicted in Fig. 5 explains how developers can access, subscribe and monitor tire pressure change. The notification contains information about the actual tire, its pressure, and a timestamp when a change occurred.

By using the same development principles, fetching appropriate manager, subscribing to an event change or by issuing a synchronous command, developers can send an action to the IVI system or get notified about the change in the IVI system.

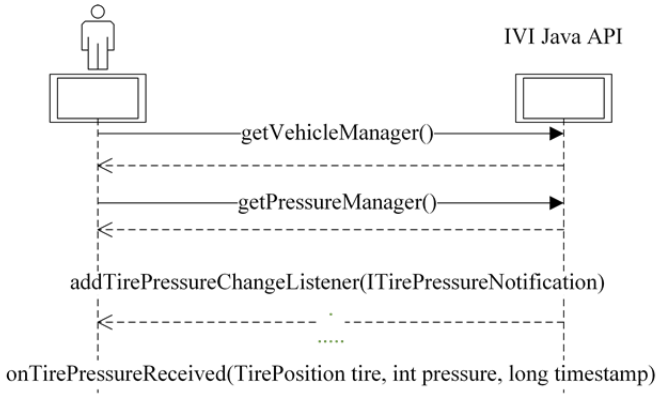

Fig. 5. Vehicle tire pressure monitoring example

The workflow depicted in Fig. 6 explains how the manufacturers can extend already available APIs.

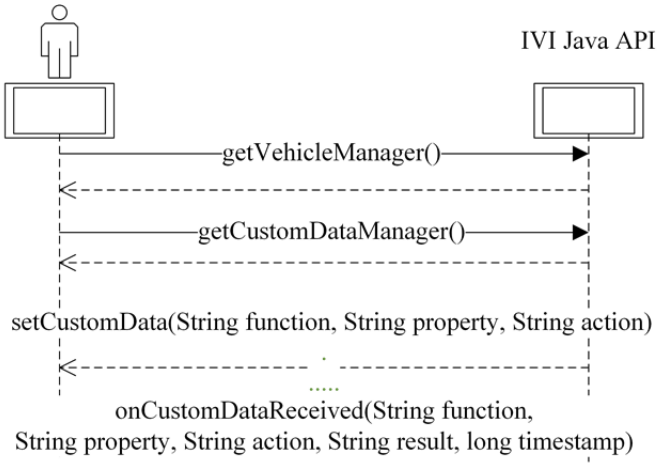

Fig. 6. Custom vehicle data monitoring workflow

One example of the workflow defined above presents monitoring of the information related to brake light status, available in Fig. 7.

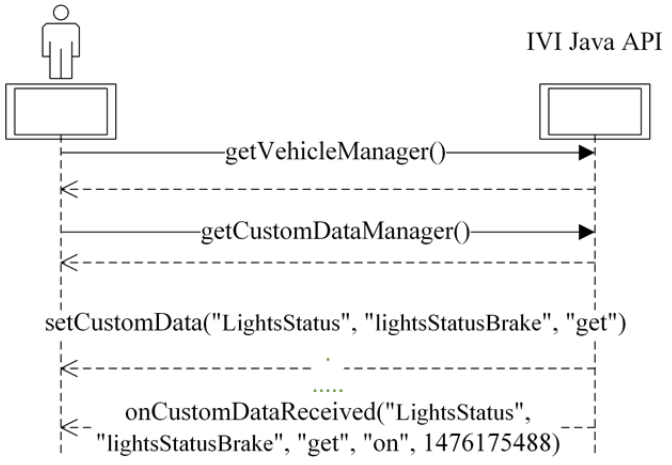

Fig. 7. Custom vehicle data monitoring brake light example

Another example includes a need for a multimedia feature, for example to tune to a specific radio station. This can also be done by manipulating the IVIManager, depicted in Fig. 8.

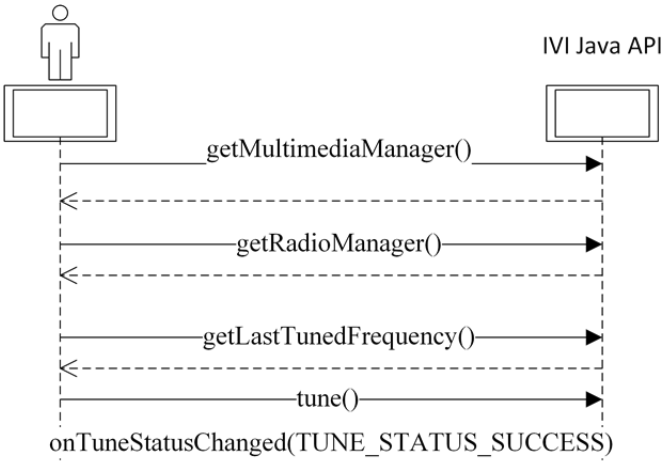

Fig. 8. Radio tuning example

The proposed API is designed not to duplicate the data in the system. Since the IVI Middleware holds all the data, the Java API does not need to hold the copy of it. It rather fetches only the necessary data from the middleware and uses it in the Java layer. In this way, only the restricted amount of data is duplicated.

Although the main focus of this paper is an integrated Java API that provides a centralized point of infotainment in vehicles, security aspects in commands and data exchange will also be considered here. As vehicles become more computerized, they are also facing a greater risk of being hacked. Today, vehicle manufacturers very often issue a recall notice for millions of vehicles in order to fix a software hole that allows hackers to wirelessly break into the vehicle and electronically control vital functions.

If there is a wireless connection available to the infotainment system, the hackers are able to use the connection to the vehicle's infotainment system, to gain access to other systems. The infotainment system is commonly connected to various Electronic Control Units (ECUs) located throughout a modern vehicle. There can be as many as 200 ECUs in a vehicle.

There are already different approaches for vehicle communication protection available, from vehicles that have a hardware based built-in firewall that separates the vehicle control network from the communications and entertainment network, to vehicles that use different inspection algorithms that scan all traffic in a vehicle's network, identifying abnormal transmissions and enabling real-time response to threats.

Garcia *et al.* [14] showed that by reverse-engineering vehicle firmware, and then eavesdropping on signals sent from a vehicle owner's key fob to the vehicle, they could remotely lock and unlock the doors. Vehicle manufacturer would need to roll out a costly firmware update to fix the problem, the researchers added. They conclude that for a "good" Remote Keyless (RKE) system, both secure cryptographic algorithms (e.g. Advanced Encryption Standard (AES)) and secure key distribution are necessary.

As explained by Rouf *et al.* [15], there are several steps that can improve the vehicle systems dependability and security.

Some of the problems arise from poor system design, whereas other problems are tied to the lack of cryptographic mechanisms. The system that they have evaluated was able to receive and process abnormal data as normal data within defined boundaries. A straightforward fix for this problem (and other similar problems) would be to update the software on the particular control unit to perform consistency checks between the values in the data fields and the warning flags. The control unit could have employed some detection mechanism to, at least, raise an alarm when detecting frequent conflicting information, or have enforced some majority logic operations to filter out suspicious transmissions of the data outside normal boundaries. Besides data boundary check, one fundamental reason that eavesdropping and spoofing attacks are feasible in vehicle systems is that packets are transmitted in plaintext. To prevent these attacks, the first line of defense is to encrypt data packets.

On the other hand, as explained by Mazloom *et al.* [16] no protection or protocols defined will help and prevent attacks or data breach if the OEMs or suppliers do not enforce them strictly.

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), IVIs integrated in the vehicle have access to features (like the vehicle Controller Area Network (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).

Above mentioned solutions including digital certificates with a handshake between different control units and hardware-based encryption with cyber-attack detection are the most promising for securing the future of the in-vehicles communication and the future of connected vehicles. They must be implemented in IVI Middleware or IVI driver layer (depicted in Fig. 2). Additionally, proposed digital signing solution with security manager provides an additional layer of protection designed to prevent hackers from using the platform resources through the proposed Java API, even in cases where the primary security and network in the vehicle is breached.

IV. Implementation

This section describes the implementation of test prototypes based on the proposal from the previous sections. The first prototype consists of an embedded device with an open source operating system, a PC device for simulation of vehicle events, an IP router for IP phone usage, a USB sound card for external microphone and speakers connection, a USB hub for interconnection between all components, a microcontroller-based device, which enables receiving of the user inputs from the jog shuttle and their propagation to the IVI platform, and a TV monitor with touch foil, connected to the video output connector of the host device, depicted in Fig. 9.

Based on the previously shown design implementation, the working prototype is created and depicted in Fig. 10.

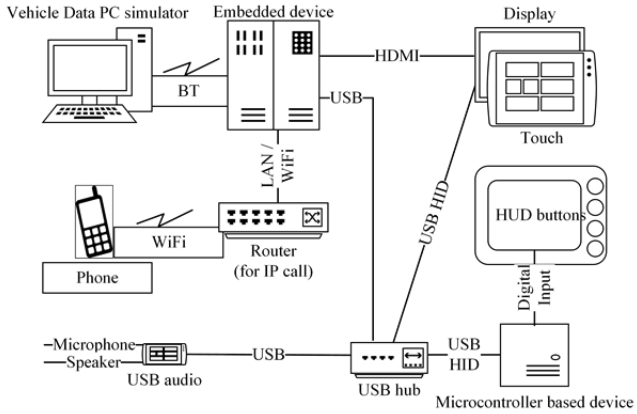

Fig. 9. Prototype 1 schematics

Fig. 10. Completed prototype 1

The prototype supports the following functionality: gathering and displaying of vehicle-related information (tire pressure, oil level, fuel consumption, vehicle speed, etc.); making/receiving phone calls; navigation; media playback and radio functionality; execution of the predefined set of applications.

The second prototype was made to verify the system portability to another platform that is based on the same operating system. The second prototype has the same components as the first, depicted in Fig. 11, except the host device, where embedded device with an open source operating system is replaced with the appropriate tablet device, containing the same open source operating system.

The idea of the second prototype was to change only the main component of the system and verify both prototypes support the same functionalities, without any drawbacks.

Use of standard open source platform should be important for manufacturers, application developers, and consumers for the following reasons:

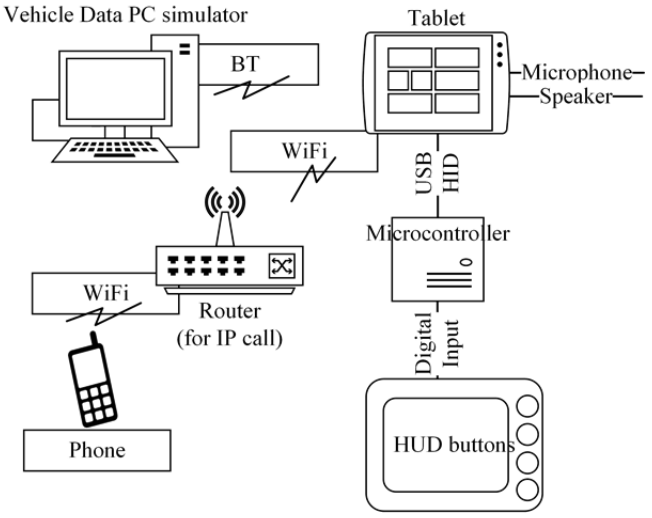

Fig. 11. Prototype 2 schematics

- An open source platform allows device manufacturers to take the platform and modify it appropriately to suit the hardware prerequisites of the specific device (this is more commonly known as "porting"). Porting an open source platform allows the platform to operate on multiple types of devices, but keep the same framework to allow applications to function accordingly across all types of platforms.
- It gives application developers the opportunity to create one application across multiple devices. This gives the newly developed IVI system an existing application base that can be further extended.
- Open source platform gives consumers a comfortable, easy to use system that they are familiar with. This minimizes time to familiarize with the system's graphical user interface and basic navigation and interaction with the system.

The operating system used in the proposed proof-of-concepts is an open-source operating system, ported to many different platforms, as explained by Gandhewar and R. Sheikh [6]. From PCs, mobile phones, and embedded development boards to digital television and set-top boxes, this OS proved to be both an agile and powerful operating system. It provides a hardware independent platform for Java application development. It also provides a Native Development Kit (NDK) for parts of the application that desire speed over portability (platform dependent code written in C/C++). This OS incorporates many open source libraries and programs into a program stack above the Linux kernel.

For IVI systems, there are special requirements for the user interface regarding driver distraction. IVI application that is placed on top of the API proposed in this paper was created considering user experience design paradigms defined by Udovicic *et al.* [17]. The common approach in vehicle applied user experience design is a touchscreen button hybrid. The problem is that choice of any option requires browsing through several menu layers, which can greatly add to driver's distraction. In order to avoid that, a simple design that provides instinctive access was created and the possibility for driver's distraction was reduced.

V. API Evaluation

Nowadays, usage of different software quality metrics is gaining importance and popularity with the goal to control, predict and improve the quality of the product. There are different metric sets that can be used during the product development life cycle, but taking into the consideration the scope of work defined in this paper, the emphasis is set to object-oriented metrics that are measuring the software quality of the proposed Java API.

TABLE I gives the main characteristics overview of the Java API proposed in this paper. Those characteristics include Lines of Code (LOC) and Weighted Methods metrics results.

TABLE I
API Characteristics with identified values

Characteristic	Value
No. of Java packets	15
No. of Java classes	32
No. of Java methods	450
No. of Java methods (by class)	14.06
LOC (total)	2887
LOC (by class)	90.21
LOC (by method)	6.42

By Lorenz and J. Kidd [18], the top threshold for LOC by method metric is set to 8 in object-oriented programming languages, so it is possible to confirm that the Java API proposed in this paper is within the defined boundaries. This metric is used as an indication of how complex it would be for the developer to reuse or maintain the defined programming code. General threshold values depend on the programming language that is analyzed and the method complexity. If the method contains a higher value of LOC, than the one defined by the threshold, it presents a good candidate for decomposition into smaller methods.

By Chidamber and C. F. Kemerer [19], Weighted Methods per Class are used to calculate the complexity of individual classes. If it is presumed that all methods in a class are equally complex, then the number of defined methods can be considered as an effort indication for the class creation and maintenance. Lorenz and J. Kidd [18] define a top threshold for the number of methods in individual classes. Threshold values depend on the fact that classes are used for the creation of the user interface or not, and varies between 20, for those that are not used for the user interface creation, up to 40, for those intended for the creation of user interface. Proposed Java API is not used for user interface creation, and as shown in Table I, the average number of methods inside individual classes is less than the top threshold previously defined.

VI. Conclusion

The average age of vehicles on United States (US) roads is about 11.4 years. The average age of a cell phone in the US is about 1.6 years. A technology gap grows over time. By 2018, 9 out of 10 phones will be smartphones. In 2008, 9 out of 10 phones were feature phones. By 2020 150 million vehicles will be connected, whereas in 2013, only 23 million vehicles were connected. The main contribution of this paper is to define a

unified API for design and development of open, more complete, feature-full systems in vehicles by different vehicle manufacturers. By exploiting the proposed Java API, remote platforms are able to harness the value of data generated by vehicles. Developers can build unique OEM apps, supporting many use cases, like "Car Health", "Roadside Assistance", "Maintenance Scheduling", and much more. One of the analyses that will be conducted in the future is the impact of the proposed Java API to the vehicle to cloud connectivity. Consumers could plug specific devices to the vehicle diagnostic port and the vehicle, and all its data, should be available, accessible, and manageable online, via a remote platform. One example of such implementation can be the creation of the connected fleet management system. Connectivity and the unified API is, therefore, an essential factor of success.

REFERENCES

[1] A. Tahat, A. Said, F. Jaouni, and W. Qadamani, "Android-based universal vehicle diagnostic and tracking system," in *Proc. IEEE International Symposium on Consumer Electronics*, Harrisburg, Pennsylvania, June 2012, pp. 813-819.

[2] G. Macario, M. Torchiano, and M. Violante, "An in-vehicle infotainment software architecture based on google android," in *Proc. IEEE International Symposium on Industrial Embedded Systems*, Lausanne, Switzerland, July 2009, pp. 257-260.

[3] F. Hueger, "Platform independent applications for in-vehicle infotainment systems via integration of CE devices," in *Proc. IEEE International Conference on Consumer Electronics*, Berlin, Germany, September 2012, pp. 221-222.

[4] I. Son, K. Han, D. Park, M. Di Yin, and J. Cho, "A study on implementation of IVI applications for connected vehicle using HTML5," in *Proc. International Conference on IT Convergence and Security*, Beijing, China, October 2014, pp. 1-4.

[5] T. Yamabe, S. Ikegami, A. Ishizaki, S. Kitagami, and R. Kiyohara, "Car navigation user interface based on a smartphone," in *Proc. International Conference on Mobile Computing and Ubiquitous Networking*, Singapore, January 2014, pp. 85-86.

[6] N. Gandhewar and R. Sheikh, "Google Android: An emerging software platform for mobile devices," *International Journal on Computer Science and Engineering*, pp. 12-17, February 2011.

[7] A. Curguz, T. Maruna, B. Kovacevic, and M. Bjelica, "Android application as parental control service in car," in *Proc. Telecommunications Forum Telfor*, Belgrade, Serbia, November 2015, pp. 934-937.

[8] N. Puaca, M. Kovacevic, B. Kovacevic, and T. Maruna, "One solution of Android service for communication with control unit in vehicle infotainment device," in *Proc. Telecommunications Forum Telfor*, Belgrade, Serbia, November 2015, pp. 942-945.

[9] V. Cikos, M. Kovacevic, B. Kovacevic, and G. Velikic, "One solution of 3D user interface for data display on a vehicle control panel," in *Proc. Telecommunications Forum Telfor*, Belgrade, Serbia, November 2015, pp. 958-961.

[10] B. Kovacevic, M. Kovacevic, T. Maruna, and D. Rapic, "Android4Auto: a Proposal for Integration of Android in Vehicle Infotainment Systems," in *Proc. IEEE International Conference on Consumer Electronics*, Las Vegas, NV, January 2016, pp. 109-110.

[11] J. Tulach, "Practical API Design: Confessions of a Java Framework Architect," Apress, Inc, 2008.

[12] J. Bloch, "How to design a good API and why it matters," in *Proc. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications,* Portland, OR, October 2006, pp. 506-507.

[13] J. Martin and H. A. Muller, "Strategies for migration from C to Java," in *Proc. 5th European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal, March 2001, pp. 200-209.

[14] F. D. Garcia, D. Oswald, T. Kasper, and P. Pavlidès, "Lock It and Still Lose It - On the (In)Security of Automotive Remote Keyless Entry Systems," in *Proc. USENIX Security Symposium*, Austin, TX, August 2016, pp. 929-944.

[15] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar, "Security and privacy vulnerabilities of in-car wireless networks: a tire pressure monitoring system case study," in *Proc. USENIX conference on Security*, Washington, DC, August 2010, pp. 1-16.

[16] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy, "A Security Analysis of an In-Vehicle Infotainment and App Platform," in *Proc. USENIX Workshop on Offensive Technologies*, Austin, TX, August 2016, pp. 1-12.

[17] K. Udovicic, N. Jovanovic, and M. Bjelica, "In-Vehicle Infotainment System for Android OS: User Experience Challenges and a Proposal," in *Proc. IEEE International Conference on Consumer Electronics*, Berlin, Germany, September 2015, pp. 150-152.

[18] M. Lorenz and J. Kidd, "Object-Oriented Software Metrics," Englewood, NJ, Prentice Hall, 1994.

[19] S. R. Chidamber and C. F. Kemerer, "A metric suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994.

Branimir Kovacevic (S'13) received the B.Sc. and M.Sc. degrees in computer science from the Faculty of Technical Sciences University of Novi Sad, Serbia in 2012 and 2013 respectively. He is currently working as a teaching assistant at the Computing and Control Department, University of Novi Sad, Serbia, where he also pursuits his Ph.D. degree. His research interests are design and implementation of highly integrated embedded systems in consumer devices.

Marko Kovacevic (S'13) received the B.Sc. and M.Sc. degrees in computer science from the Faculty of Technical Sciences University of Novi Sad, Serbia, in 2012 and 2013 respectively. He is currently working as a teaching assistant at the Computing and Control Department, University of Novi Sad, Serbia, where he also pursuits his Ph.D. degree. His research interest is software programming for embedded systems.

Tomislav Maruna (M'10) received the B.Sc. and M.Sc. degrees in electrical engineering from the Faculty of Technical Sciences, University of Novi Sad, Serbia, in 1994 and 2001 respectively. He is leading the Automotive Business unit in RT-RK Institute for Computer Based Systems, Novi Sad, Serbia. His main occupation is the system and multimedia software for embedded systems. His research interest is the deployment and usability of embedded operating systems within consumer devices.

Istvan Papp (M'08) received the B.Sc., M.Ss. and Ph.D. degrees in electrical engineering on the Faculty of Technical Sciences, University of Novi Sad, Serbia, in 1998, 2001 and 2009 respectively. He is an assistant professor at Computing and Control Department, University of Novi Sad, Serbia. His research interest is related to digital signal processing and toolchain developments for DSPs.