Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2015, Article ID 478064, 13 pages
http://dx.doi.org/10.1155/2015/478064

Research Article

GRCBox: Extending Smartphone Connectivity in Vehicular Networks

Sergio M. Tornell, Subhadeep Patra, Carlos T. Calafate, Juan-Carlos Cano, and Pietro Manzoni

Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, Camino de Vera S/N, 46022 Valencia, Spain

Correspondence should be addressed to Sergio M. Tornell; sermarto@upv.es

Received 7 August 2014; Accepted 24 October 2014

Academic Editor: Dongkyun Kim

The low penetration of connectivity-enabled OBUs is delaying the deployment of vehicular networks (VNs) and therefore the development of vehicular delay tolerant network (VDTN) applications, among others. In this paper we present GRCBox, an architecture based on RaspberryPi that allows integrating smartphones in VNs. GRCBox is based on a low-cost device that combines several pieces of software to provide ad hoc and multi-interface connectivity to smartphones. Using GRCBox each application can choose the interface for its data flows, which increases flexibility and will allow developers to easily implement applications based on ad hoc connectivity, such as VDTN applications.

1. Introduction

Intelligent transportation systems (ITS) include a set of different technologies that aims at improving the efficiency and security of transportation by combining vehicular networks (VNs) and advanced logistics. VNs combine different communication technologies such as cellular networks, vehicular ad hoc networks (VANETs) [1], or 802.11 infrastructure networks to provide communication between vehicles (vehicle-to-vehicle (V2V)) and between vehicle and road infrastructure (vehicle-to-infrastructure (V2I)). The core of VNs is the 802.11p standard [2], which modifies the 802.11a standard to meet low delay requirements for safety applications.

Although this technology is ready for deployment, it is expected that car manufacturers will introduce it gradually, starting at high-level models, which, coupled with the low renovation rate of the vehicle fleet, will slow down the deployment of VNs. In addition, while dashboard-integrated on board units (OBUs) become technologically obsolete after a couple of years, they are usually not designed to be updated or replaced during the whole life of the vehicle, which leads to unsatisfied users.

Recently, part of the industry has realized these OBU problems and proposed alternatives to integrated OBUs based on smartphones. Thus, the car connectivity consortium (CCC), which is formed by several automotive and communication companies, released Mirrorlink [3], which is a standard technology that connects the user's smartphone with the OBU, moving the computing tasks from the OBU to the smartphone, displaying the information on the OBU display, and allowing the driver to interact with the smartphone through the dashboard elements. Besides, this technology enables users to follow the pace of technology throughout the vehicle lifetime by updating their phone. Google and Apple, two of the most important technology companies, have also released their own proposals for smartphone-in-vehicle integration, named Android Auto [4] and CarPlay [5], respectively. While their solutions are technically similar to MirrorLink, they have focused on the user interface, improving its quality and design.

Although these smartphone-oriented proposals solve some of the problems related to user interaction with dashboard-integrated OBU, they are limited by the smartphone's OS design. Smartphone OSs were not designed for peer-to-peer (P2P) ad hoc communication, focusing solely on Internet communication. Therefore, they lack of an Ad-hoc communications interface, and applications always use the default Internet-connected interface without the possibility of selecting another interface. Moreover, current smartphones cannot be extended by adding new communication interfaces. This hardware and software issues limit the full integration of smartphones in ITS.

In this paper we present the GRCBox architecture, which allows users to enjoy sophisticated ITS solutions by using their smartphones in current vehicles, avoiding the investment on expensive OBUs. In the GRCBox architecture, the user interacts with the smartphone interface while the communication is delegated to the GRCBox hardware module, which is placed in the vehicle and has multiple network interfaces, including ad hoc communication capabilities. The GRCBox architecture includes both the GRCBox hardware module and a set of libraries to allow application developers to easily create compatible smartphone applications. With the introduction of GRCBox we expect boosting the adoption of ITS and that users realize their benefits, starting a virtuous circle which leads to more applications and more services.

Ad hoc and multi-interface communication are a must for vehicular delay tolerant network (VDTN) protocols, since they rely onopportunistic contacts between nodes. Without ad hoc communication, these contacts are limited to infrastructure networks which are expected to be rare in VNs. Our GRCBox will also enable smartphones to take part in the VANETs created, thus empowering applications running on smartphones.

The rest of this paper is organized as follows. In Section 2 we survey other proposals in the literature previously presented. In Section 3, we will present the GRCBox architecture, by describing its modules, while in Section 4 we will detail its implementation. In Section 5, we evaluate the performance of the GRCBox hardware module, detailing how it impacts the performance of applications. Finally, Section 6 presents some features that will be added to the GRCBox in the near future and concludes this paper summarizing our contributions.

2. State of the Art

The works pertaining to testbed developments and implementation of vehicular networks that we have found can be classified according to their chronology, the proposed architecture, and the type of communication used or on the basis of the characteristics being studied. In this section we review the previous works chronologically.

The earliest works in real implementation and testing of VANETs started in the early 2000 when Singh et al. [6] presented one of the first works in this field. They experimented with link quality throughput and connectivity range. Their work involved just two vehicles with laptops and WLAN communications; thus it was a single hop network. Later, Wu et al. [7] and Hui and Mohapatra [8] used car mounted laptops with 802.11b compatible wireless cards and GPS receivers to study multihop communications in VANET and TCP/UDP performance, respectively.

Thereafter, VANETs began to draw more attention from the research community. In 2007 Jerbi et al. [9] studied the communication characteristics like SNR, jitter, and packet loss to evaluate the effect of the wireless channel and the environment on delivering multimedia applications in highly mobile networks. de la Fortelle et al. [10], on the other hand, developed a local cooperative traffic management system and Lee et al. [11], during the same period, developed a testbed for VANETs that was designed for testing an implementation of the CarTorrent protocol [11].

A year later, Pinart et al., in their work named DRIVE [12], presented a testbed that mainly consists of infrastructure element in the cars that would enable both V2V and V2I communications. It also includes human machine interfaces (HMIs) like a touch screen monitor for the GUI, a microphone, several speakers for the voice HMI, and a car area network- (CAN-) based HMI. Also, an open research platform for VANETs called C-VeT [13, 14] came into light in 2008 and was further improved in 2010. The C-Vet network was formed of cars equipped with industrial strength Cappucino PC, GPS receiver, 802.11b/g MIMO wireless card, and a radio modem. It was tested by monitoring the network, file sharing using P2P, and video streaming applications.

Another example of a VANET testbed constructed using real cars is [15]. The hardware used in [15] is an embedded computer equipped with a mini-PCI 802.11 wireless transceiver and a compact flash hard disk, in each of the three cars used in the testbed. For sending and receiving of data, a source or sink in the form of a laptop had to be connected to these embedded computers used as mobile routers within the cars. A couple of years later, in 2011, Paula et al. [16] developed a real world VDTN testbed that was designed and tested using warning messages, and the test site was the Brazilian Fiat Automobile manufacturing plant.

Recently with the rise in smartphone popularity, many researchers have tried to investigate the integration of smartphones and vehicular networks [17]. Examples are works like [18–20]. Vandenberghe et al. [18] studied the feasibility of integrating smartphones with vehicular networks. In [19], the authors concentrated on the rapid dissemination of information among vehicles within the designated area. The application described in [19] is to form an ad hoc network using the WiFi available in smartphones and was tested on iPhones. In a similar way, an Android application that creates an ad hoc network to warn drivers of approaching emergency vehicles using an intelligent message dissemination protocol was proposed in [20].

Even though efforts have been made to integrate smartphones with vehicular networks, this integration will not be possible unless smartphones are equipped with 802.11p compatible hardware. Also, nowadays smartphones have different operating systems and, therefore, the difficulty of configuring them to create an ad hoc network depends on the running operating system. To overcome these problems, we have developed an architecture called GRCBox that consists of a low-cost device placed in the vehicle which provides V2X connectivity to associated smartphones. The smartphones are provided with the flexibility to connect to any of the networks supported by the GRCBox, including ad hoc networks, and 802.11p when available. As far as we know, the GRCBox architecture is the only proposal which aims to completely integrate smartphones into VNs, while also allowing deploying VDTN applications based on *peer-to-peer* neighbor discovery mechanisms.

3. GRCBox Architecture

The GRCBox architecture provides both a low-cost hardware module with multiple network interfaces placed in the vehicle and a software library based on a remote application programming interface (API) to allow applications to communicate with the different outer networks through it. The GRCBox hardware module allows applications running on the users' smartphones to select which network interface to use in a per connection basis, providing flexibility and enabling developers to effectively implement new communication paradigms such as VDTN. The GRCBox also allow developers to use the more suitable technology for their applications; for example, they can use 3G or LTE when a very stable connection is needed, the WiFi connection when a high throughput is required, or ad hoc communications when direct communication between vehicles, that is, without infrastructure, is required.

In this section we present the GRCBox architecture. First, we detail the different software components that run in the GRCBox hardware module. Second, we introduce the information exchanged between the GRCBox hardware module and the applications. Finally, we enumerate the features of the client library that allow developers to use the services offered by the GRCBox hardware module.

3.1. GRCBox Hardware Module Components

3.1. GRCBox Hardware Module Components. The GRCBox hardware module, which has to be placed into the vehicle, must have at least one *inner interface* to which smartphones are connected to, and at least two *outer interfaces* connected to several networks such as a cellular network, a WiFi infrastructure network, or a WiFi ad hoc network. Figure 1 shows a GRCBox with one inner interface and two outer interfaces placed in a car. The GRCBox hardware module is composed of several services which communicate with each other and with the user. These services are the discovery service, the routing and headers modification service, the multicast and broadcast proxy service, the interface monitoring service, and the core service. A scheme of their connections and the paths traversed by data flows is presented in Figure 2. The functions of each service are as follows.

3.1.1. Discovery Service. This service provides clients with a mechanism to detect and connect to the GRCBox core service without any previous knowledge.

3.1.2. Routing and Headers Modification Service. This service forwards packets between inner and outer networks according to connection rules defined by the applications. It also modifies the packet header when necessary.

3.1.3. Multicast and Broadcast Proxy Service. Since the routing module is limited to unicast packets, a new service is defined for broadcast and multicast packets. The multicast and broadcast proxy service listens for multicast and broadcast packets on all the interfaces and forwards them between

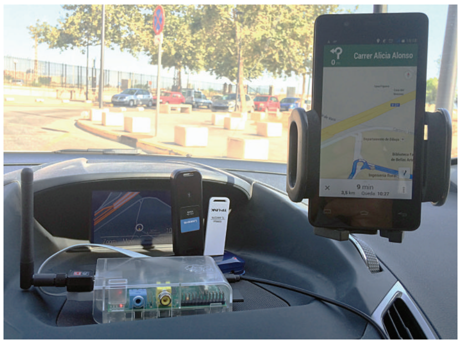

Figure 1: A GRCBox hardware module formed by a RaspberryPi and 3 WiFi interfaces: one in access point mode, one in station mode, and one in ad hoc mode.

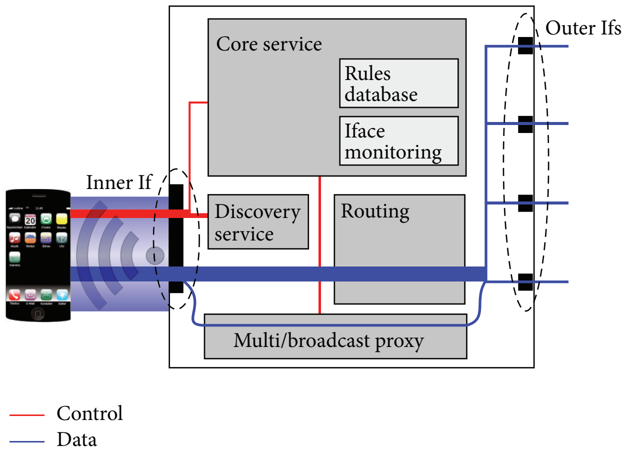

Figure 2: GRCBox hardware module architecture.

the inner interface and the outer interfaces according to the defined rules. It extends the broadcast domain of the outer interfaces to the inner interface and vice versa.

3.1.4. Interface Monitoring Service. This service provides the updated information about the status of the outer interfaces. This information is needed to configure the routing service and the multicast proxy service, as well as to inform applications about the characteristics of the different interfaces.

3.1.5. Core Service. The core service is in charge of communicating with applications, validating connection rules, maintaining the rules' database, and configuring the routing service and the multicast proxy services according to the interface configuration.

3.2. Interaction between Applications and the GRCBox Hardware Module. The GRCBox hardware module creates a WiFi access point to which smartphones, tablets, or any other user device in the vehicle will associate. Once the user devices connect to the "GRCBox" wireless network, they can communicate with each other to share contents or communicate with any of the outer networks to which the GRCBox hardware module is connected. Figure 3 shows a vehicle with

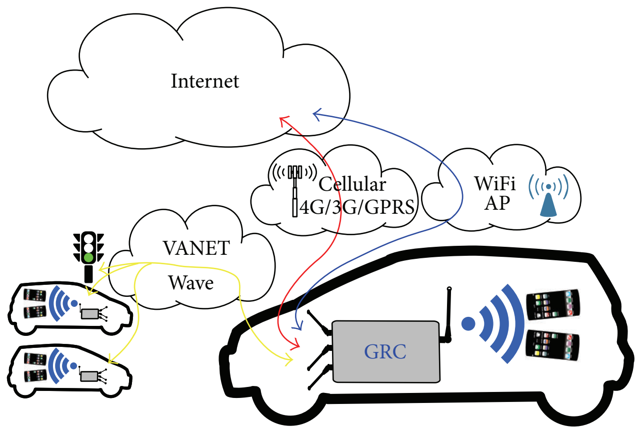

Figure 3: GRCBox hardware module connected to three different networks.

a GRCBox hardware module and several smartphones that are able to exchange information with the outer networks. In this example, an application in the smartphone may choose to reach the Internet using either the cellular network or the WiFi network. In case the application intends to use ad hoc communications, it would choose the ad hoc wireless interface instead.

The GRCBox architecture is based on rules which enable applications to choose the outgoing interface for a certain connection or to register as listeners for a defined incoming connection. A rule is a packet filter defined by the following elements.

(i) Rule type: the GRCBox platform defines three different kinds of rules, *incoming*, *outgoing*, and *multicast*.

(ii) Interface name: it is the name of the interface to which the rule applies.

(iii) Transport protocol: it is the transport protocol of the connection. At this moment GRCBox supports UDP and TCP, though we expect implementing more protocols in the future.

(iv) Source port: it is the source port of the connection.

(v) Source address: it is the source IP address of the connection.

(vi) Destination port: it is the destination port of the connection.

(vii) Destination address: it is the destination IP address of the connection.

According to these elements, the GRCBox routing service either waits for incoming connections on outer interfaces, routes outgoing connections to the specific outgoing interface, or forwards multicast packets in both directions between inner and outer interfaces. For incoming connections there are two extra elements stored in the rules database: the forwarding destination address and the forwarding destination port. The GRCBox routing service uses this information to modify the packet header and to forward the connection to the target user device.

In GRCBox, before being able to define a new connection rule, an application must register itself on the GRCBox server to get a private secret key. This key is then used for later client/server interactions to guarantee that only the "owner" of a rule renews, modifies, or removes it. After obtaining its secret, an application is able to register new rules. To register a new rule the application must send a request to the GRCBox core service, which checks that the rule definition does not collide with a previously defined rule, and configures the rule on the routing service. Once the rule has been configured, the GRCBox core service confirms it to the application. The application can then initiate the connection that will be routed according to the defined rule. Once the application has finished using this rule, the application must notify to the GRCBox core service that this rule will not be used again, so the GRCBox core service can remove it from the system. Figure 4 illustrates both how a GRCBox application interacts with the GRCBox hardware module and how a non-GRCBox application is transparently routed. More details about how this communication is done and how the GRCBox hardware module is configured are presented in Section 4.

3.3. Client Features. To allow developers to easily use the services offered by the GRCBox hardware module, we provide a programing-language-independent remote API, a set of software libraries, and a management application that allows the smartphone users to define rules for third party applications that do not support GRCBox. In particular, we provide the following software modules.

3.3.1. Remote API. The remote API allows defining rules remotely.

3.3.2. Client Library. The client library provides an almost-transparent way to use the GRCBox capabilities through the remote API.

3.3.3. Management Application. The management application allows users of a smartphone connected to a GRCBox hardware module to define rules for third party applications. Therefore, the user may request, for example, all the VoIP traffic to be routed through the cellular network or delay tolerant network (DTN) messages to be routed through the WiFi ad hoc network, and this configuration will affect all the applications running on the specific smartphone.

4. GRCBox Implementation Details

In this section we present the implementation details of the GRCBox architecture. First, we present the details of the different components of the GRCBox hardware module and how they communicate. Second, we detail the set of libraries we have implemented to enable developers to easily create GRCBox-aware applications. Finally, we describe in detail three different cases to illustrate the use of GRCBox in typical scenarios: when an application connects to an external server and when the application sends and receives UDP messages.

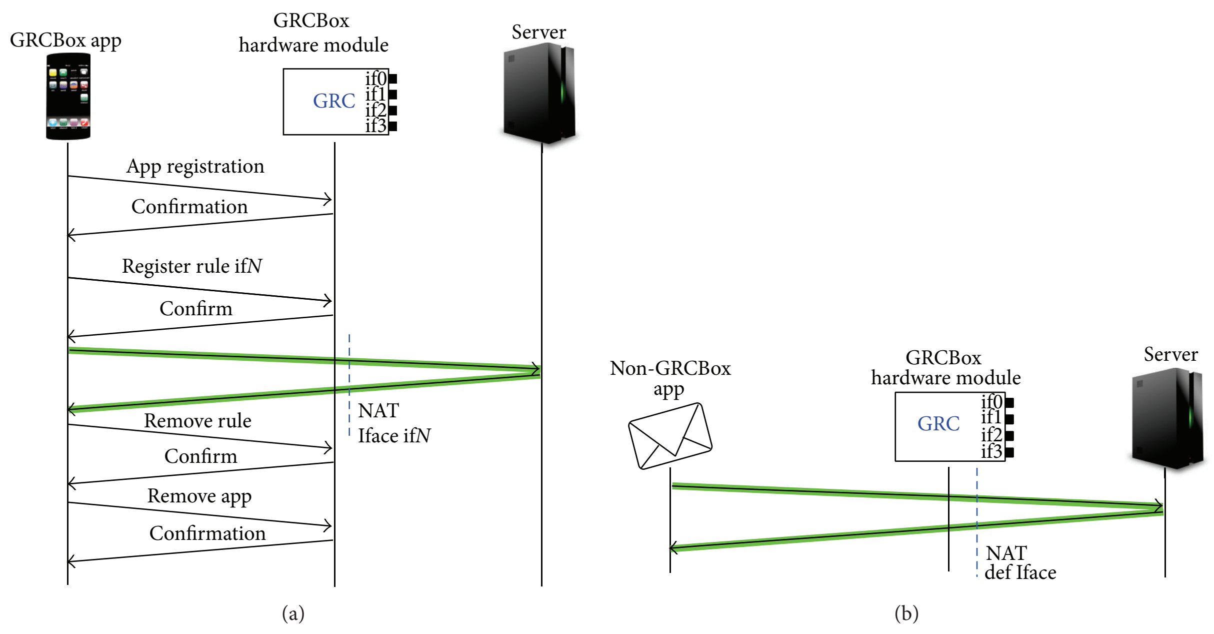

Figure 4: Connection phases of a GRCBox app (a) and a non-GRCBox app (b).

4.1. GRCBox Hardware Module Details. To implement the different services that run in the GRCBox hardware module, we took advantage of some well-known open source software solutions, creating a core service that coordinates them to configure a system that meets our requirements.

4.1.1. Hardware. To implement the first version of our GRCBox hardware module we have chosen an embedded computer called RaspberryPi [21]. The RaspberryPi is a credit-card size computer whose cost is only 35$ but that has enough power to perform low-scale network routing. In this computer we have installed a Raspbian [22] distribution, which is a general-purpose Linux distribution based on Debian and optimized for the RaspberryPi. Raspbian supports most current networking hardware, avoiding common problems of other embedded operating systems.

4.1.2. Discovery Service. To allow the GRCBox applications to discover which node in the network is the GRCBox hardware module, we decided to use both the domain name system (DNS) and the dynamic host configuration protocol (DHCP). The *dnsmasq* [23] service provides DNS and DHCP services to the inner network. When a new device connects to the GRCBox network, its connectivity is configured using DHCP; these connectivity settings include the default DNS server IP, which points to the GRCBox hardware module. Therefore, every DNS query will be resolved by the GRCBox hardware module. When a GRCBox application is started it tries to connect to the "grcbox" node, since the DNS server is configured to always resolve the name "grcbox" to the IP address of the GRCBox hardware module, thus allowing the application to connect to it without knowing its actual IP address.

4.1.3. Routing and NAT. The main reason to choose a Linux-based operating system is the flexibility this OS offers for packet routing configuration. The GRCBox hardware module is configured using *iptables* [24] to perform source network address translation (SNAT) on every public interface. The GRCBox core uses the routing policy data base (RPDB) feature from the Linux Kernel [25] to redirect the connections defined by the applications to the selected interface.

For every public interface, the GRCBox core defines a specific routing table applicable to all the packets *marked* with a certain label. This routing table contains only one default entry that redirect the packets to the specific interface independently of the routing entries defined in the default routing table. Specifically, using the command *"iptables-t mangle,"* the GRCBox core *marks* the packets to match the outgoing interface defined by the application. Moreover the *forwarding destination port* and the *forwarding destination address* are used to perform destination address network translation (DNAT) for incoming connections.

In Section 4.3 we introduce some examples that clarify the operation of the routing system.

4.1.4. Interface Monitoring. To maintain the routing and marking rules updated, the GRCBox core needs to monitor the status of the interfaces. To do so, we have created a set of classes that interact with the *NetworkManager* [26] system service through the command line tool *nmcli*. These classes allow obtaining the IP address and the gateway of each interface, as well as the type of the interface. In addition, they check whether an interface is connected to the Internet in order to provide clients with meaningful information for interface selection.

4.1.5. GRCBox Core Server. We implemented the GRCBox core server following a representational state transfer (REST)

[27] architecture. REST is a way to create, read, update, or delete information from a server using simple HTTP calls. To implement it, we have used the *RESTlet* [28] Java framework that provides a set of classes to quickly implement a REST architecture. The REST architecture allows us to abstract from communication protocol issues and focus solely on functionality. To implement our service we have defined several resources accessible from the inner network through "http://grcbox:8080/." This API allows clients to register a new application, to check the status of the server and its interfaces, and to register new rules. Appendix A describes it in detail.

4.1.6. Multicast and Broadcast Proxy. To provide multicast and broadcast forwarding between the outer networks and the inner network we have designed a multicast proxy that listens on specific outer interfaces, forwarding to the inner network only those packets with multicast addresses registered by a client inside the network.

We have implemented the proposed proxy using Rock-Saw [29], a Java library that allow us to completely define the low-level content of IP packets, including their headers. This multicast proxy also acts as a broadcast UDP proxy for registered directed-broadcast forwarding rules.

A new instance of this proxy is started by the core service for every new multicast or broadcast forwarding rule.

4.2. Client Library. Although we have designed the GRCBox architecture to be client/OS independent, in this first version we have focused on Android [30] based smartphones. We have created a Java library that allows the developer to easily implement GRCBox compatible applications. The implemented library integrates with Java networking classes, and its details are covered in Appendix B.

In the next subsection we present an example of several cases where these libraries are used by applications to define routing rules transparently.

4.3. Examples. In this subsection we present three different cases of GRCBox-aware applications.

In both cases we consider the same configuration: the GRCBox hardware module has 3 interfaces, a WiFi interface configured as a WiFi station, another WiFi interface operating in the ad hoc mode, and a 4 G interface. In the first case, the application wants to connect to a remote server located on the Internet using a specific interface. In the second case, the application wants to send and receive UDP packets through a specific interface. In the last example, the application performs multicast communication for neighbor discovering and relies on TCP peer-to-peer communications for information exchanging. Notice that most DTN applications can become integrated into the GRCBox framework in a similar way.

4.3.1. TCP Client Example. In this example, an application running on the smartphone wants to connect to host *google.com* through the station-mode WiFi interface, while a different interface is configured as the default one. In order to do so, the application needs to perform the following steps.

(i) Create a new instance of our GrcBoxClient class.

(ii) Call the GrcBoxClient.isServerAvailable() function to check GRCBox availability. In case the GRCBox hardware module is not available, the application can chose to use the standard networking libraries.

(iii) Register itself as a new application by calling GrcBox-Client.register("Name"). After the new application is registered, a new thread is started for periodic keep-alive notifications to the server.

(iv) Get a list of the available interfaces by calling GrcBox-Client.getInterfaces(). Then the client will iterate over the interfaces list and choose the desired one.

(v) Create a new socket by calling GrcBoxClient. create-Socket(dstAddr, dstPort, iface). At this moment, the library will notify the GRCBox core service about the new connection, and the GRCBox core service will create a new routing rule. The library will return a new GrcBoxSocket already connected to the remote host.

(vi) At this step the application can already send and receive data to/from the server. The packets will be routed through the specific interface.

(vii) Once the application has finished exchanging data with the remote host, it should close the GrcBox-Socket instance. This call will remove the rule, and special-purpose routing will be stopped.

(viii) If the application has finished communicating with the GRCBox core service, it should deregister itself from the GRCBox hardware module, which will stop the keep-alive thread.

Figure 5 represents the process described above.

4.3.2. Hybrid UDP Example. In this example, a VoIP application wants to use the 3G interface because it is usually more stable than WiFi, and its throughput is enough for VoIP communication. In order to do so, the application needs to perform the following steps:

(i) Create a new instance of the GrcBoxClient class.

(ii) Call the GrcBoxClient.isServerAvailable() to check GRCBox availability. In case the the GRCBox hardware module is not available, the application can choose to use the normal networking libraries.

(iii) Register itself as a new application by calling GrcBox-Client.register("Name"). After the new application is registered, a new thread is started for periodic keep-alive notifications to the server.

(iv) Get a list of the available interfaces by calling GrcBox-Client.getInterfaces(). Then the client will iterate over the interface list and choose the desired one.

(v) Since the application uses an external library for SIP communication, it must use the low-level method registerNewRule() to define a new rule according to the characteristics of the SIP protocol. When it is called, a new routing rule is created at the GRCBox hardware module.

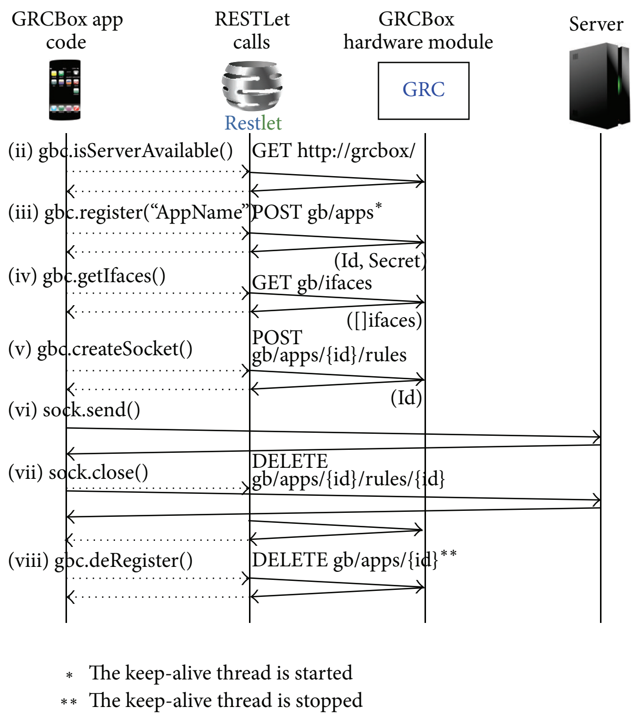

Figure 5: Example of a client application using GRCBox.

(vi) At this step the application can run normally, listening to data coming from the remote host, or sending data.

(vii) After negotiating using the SIP protocol, if the application wants to establish an RTP connection it must register a new rule.

(viii) All the rules must be removed from the GRCBox hardware module after the application finishes using them.

(ix) If the application no longer needs to communicate using GRCBox, it should deregister itself from the GRCBox hardware module.

Figure 6 represents the different steps involved in the process described above.

4.3.3. Multicast and TCP Example. In this example, a DTN application wants to use the WiFi ad hoc interface to send multicast packets to discover new neighbors and to establish a TCP connection with them in order to exchange information. The steps involved are the following.

(i) Create a new instance of the GrcBoxClient class.

(ii) Call the *GrcBoxClient.isServerAvailable()* to check GRCBox availability. In case the the GRCBox hardware module is not available, the application can choose to use the normal networking libraries.

(iii) Register itself as a new application by calling *GrcBoxClient.register("Name")*. After the new application is registered, a new thread is started for periodic keep-alive notifications to the server.

(iv) Get a list of the available interfaces by calling *GrcBoxClient.getInterfaces()*. Then the client will iterate over the interface list and choose the desired one.

(v) Create a new GrcBoxServerSocket by calling the method *GrcBoxClient.createServerSocket(port, outIface)*. This will register a new incoming rule for connections made by discovered neighbors.

(vi) Create a new GrcBoxMulticastSocket by calling the method *GrcBoxClient.createMulticastSocket(port, iface)*, and then join the desired multicast group. This will start a multicast proxy in the GrcBox hardware module. At this time the application becomes able to receive new connections from neighbors, as well as to discover neighbors from outside networks.

(vii) If the local IP address is included in the payload of the discovery packets (beacons), it should be replaced with the corresponding IP address of the public GRCBox interface; in such case, this information should be adapted to reference the public address of the chosen GrcBox interface instead.

(viii) If a neighbor is discovered a new GrcBoxSocket must be created by calling GrcBoxClient.createSocket(addr/host, port, iface, localPort). This will register a new outgoing rule through the ad hoc interface.

(ix) Once the application has finished using the GrcBoxSocket it can be closed. This will remove the outgoing rule from the GrcBox.

(x) The multicast rule and the incoming rule are expected to be useful as long as the application is running.

(xi) If the application no longer needs to communicate using GRCBox, it should deregister itself from the GRCBox hardware module. This will also remove all the rules associated with this application.

Figure 7 represents the different steps involved in the process described above.

5. Performance Evaluation

As we stated in the previous section, a GRCBox-aware application must cover several steps before being able to communicate through the desired interface with the outside networks. In this section we evaluate the overhead introduced by the GRCBox architecture by measuring the average time required to perform each of these steps and the delay introduced by the GRCBox architecture itself.

5.1. Test Configuration. To measure the times associated with the different steps, we have configured a GRCBox hardware module with three wireless interfaces: a TP-Link TL-WN727N usb adapter (chipset rt5370 from Ralink); a Linksys WUSB600N usb adapter (chipset rt2870 from Ralink); and a generic WiFi usb adapter with an rt5370 chipset from Ralink. The first interface is configured as an access point, while the other two are connected to an infrastructure access point

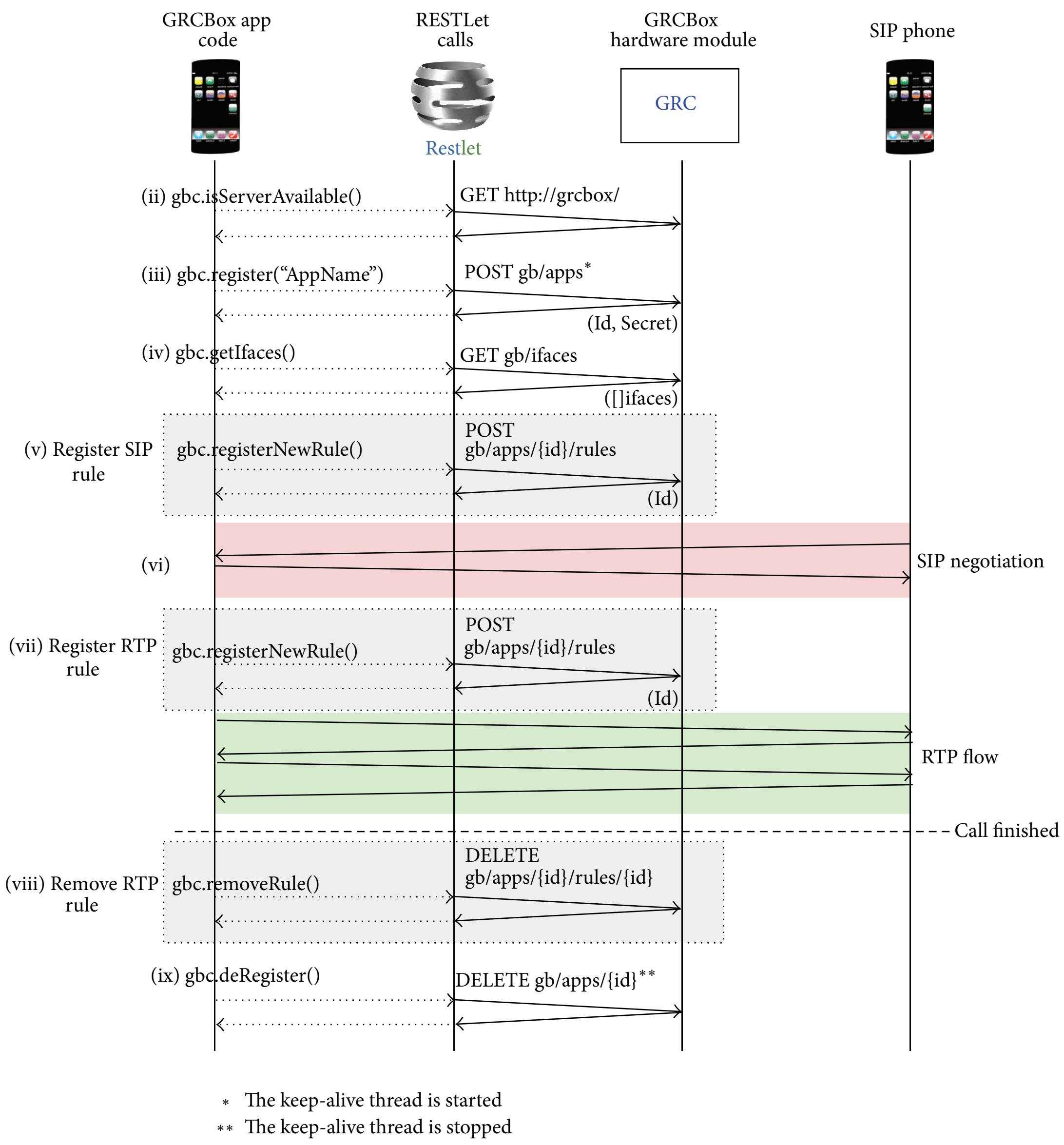

Figure 6: Example of a client application using GRCBox.

connected to the Internet using the university infrastructure. We have performed two different experiments, the first one using a laptop as a user terminal and the second one using a smartphone. The user terminal will connect to a host located in our university network through the GRCBox hardware module to download a file. All the devices (including the Internet-connected access point) are placed in the same room. Figure 8 shows how the different devices are connected for this test, while Figure 9 shows the real devices we have used to run this test.

5.2. Time Required to Control the GRCBox Hardware Module. To get conclusive results, we have repeated each experiment 100 times. To check if the computation power of the user terminal affects the average delay added by the GRCBox architecture, we performed the same set of experiments using both the notebook and the smartphone. The steps measured are the following:

(i) time to check the status of the GRCBox core (check time),

(ii) time to get the information about the interfaces available on the server (Ifaces time),

(iii) time to register a new application (Reg. app time),

(iv) time to register a new rule with the desired output interface (Reg. rule time),

(v) time to download the file from the Internet (download time),

(vi) time to remove a rule (Rm rule time),

(vii) time to remove an application (Rm app time).

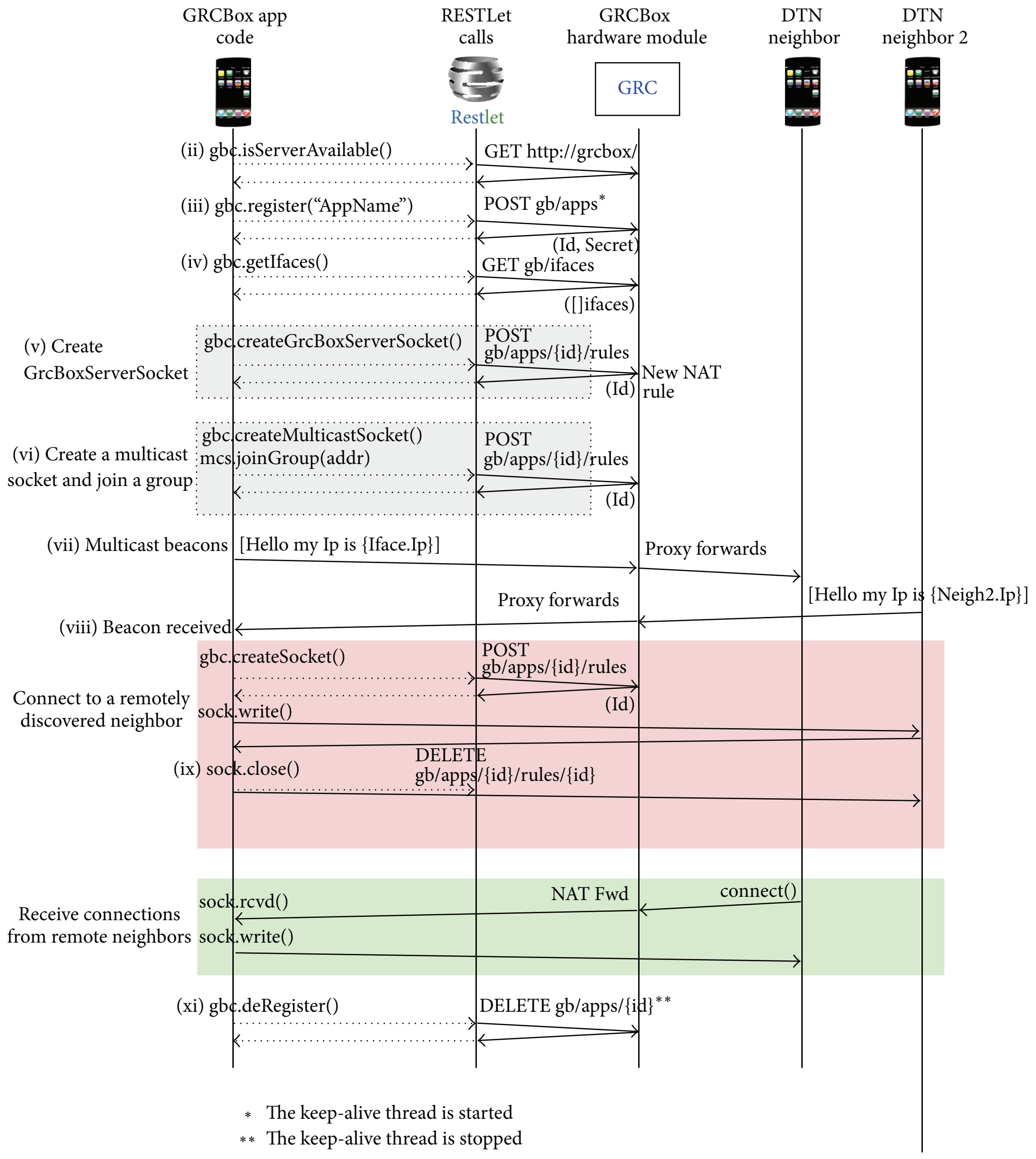

FIGURE 7: Example of a DTN application using multicast in a GRCBox client.

As a reference, we have also downloaded the same files through the GRCBox hardware module without using the GRCBox features. Table 1 summarizes the obtained results, where all the values presented represent time in seconds. The average time required for every step ranges from 0.2 s to 2.4 s. These values are tolerable when the user wants to perform a long lasting action, such asbrowsingor a VoIP call. According to the values we obtained, an application should be able to check if the GRCBox core service is available, get the list of interfaces, register itself in the GRCBox core service, and register a new rule to route the expected traffic through the desired out interface, transparently to the user, while it starts and configures the application (write a website address or dial a phone number).

Concerning *noninteractive autonomous applications*, such as a DTN services, or ad hoc warning notification systems, which usually run continuously for hours or even days, they only need to communicate with the GRCBox at the moment they start running. Therefore, the impact of the delay introduced by the GRCBox architecture is insignificant.

By examining the confidence interval of the first step (check the status of the server), we realized that its variability was very high. To find an explanation to this fact, we inspected the individual values, whose histogram is represented in Figure 10, and realized that the server took up to 8 s to resolve the request. By revising our code, we detected a problem in the *interface monitoring* module IV-A that blocks the processing of requests involving interface information

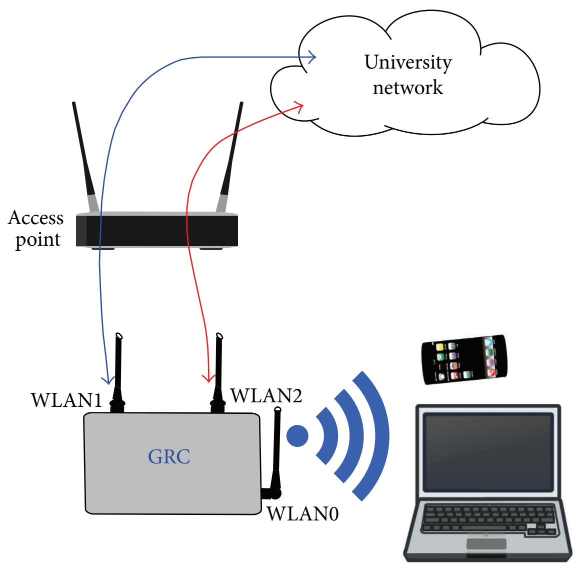

FIGURE 8: Test configuration.

FIGURE 9: A picture of the devices used for the test.

during the interface updating process, which takes about 10 s. In the next section we will shortly present how we propose solving this problem.

5.3. Delay Introduced by the GRCBox. To measure the delay introduced by the GRCBox architecture, we have used the well-known *ping* tool to measure the round trip time (RTT) between the user device and a server on the Internet in two different conditions: connected through the GRCBox and connected directly to the access point in our lab. To get a wide set of results we *pinged* 100 times 3 different hosts under different domains. Table 2 contains the results we obtained. The first important thing we notice is the high value of the *standard deviation,* which means that the network, in all the cases, was very unstable. Given this condition, it is hard, or even impossible, to conclude that there is a difference between the RTT experienced when using GRCBox and when connected directly to the access point acting as Internet gateway. In addition and despite the fact that it is obvious that adding an extra hop to the path between the user device and

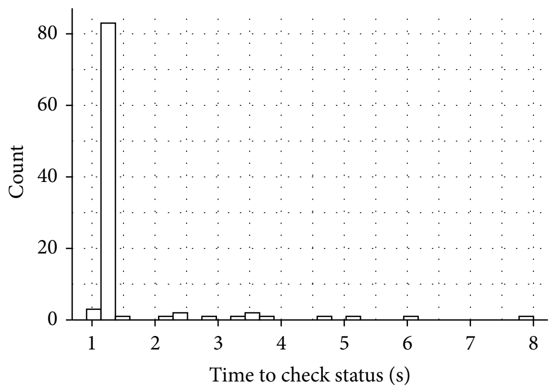

FIGURE 10: Time to check status of the GRCBox core service histogram when using a smartphone as user device.

the Internet server increases the RTT, the obtained data shows that this increment is negligible if we compare it to the effects of network instability.

6. Conclusions and Future Work

In this paper we presented the GRCBox architecture, which allows smartphones to be completely integrated in vehicular network (VN) environments. The GRCBox architecture is composed of a low-cost hardware module that is installed in the vehicle and a set of libraries that allow developers to use it. The GRCBox architecture will be released under an open-source license and will be available for downloading from our GitHub page [31].

We should highlight that our GRCBox architecture still has some issues which we expect to solve in the near future. The main issues we are currently focusing on are the slow interface monitoring service and the remote configuration application. Below we detail how we intend to solve these problems.

6.1. Interface Monitoring. As we described in Section 4.1, the GRCBox core service interacts with the *NetworkManager* daemon through a command line tool. Moreover, our service needs to repeatedly poll the *NetworkManager* daemon to check if there were changes in the interfaces' configuration. These two issues are associated not only with long blocking times when waiting for values to be returned but also with a waste of resources due to polling. In the future, we will migrate from the *NetworkManager* command line interface to its DBUS interface [32]. The DBUS interface allows accessing all the information needed by the GRCBox core service through shared memory objects and also provides a *subscriber-publisher* interface that will make polling unnecessary, thus saving resources.

6.2. Remote Configuration. We also perceived that it would be interesting to allow smartphone users to configure the outer network interfaces through an application on their mobile phones. This application may also be used to perform certain actions on the GRCBox, such as defining the default outgoing

Table 1: Time overhead for the different tasks involved when downloading a 5 MB file.

	GRCBox notebook		GRCBox smartphone		No-GRCBox notebook		No-GRCBox smartphone	
	Average	Conf. int. 95%	Average	Conf. int. 95%	Average	Conf. int. 95%	Average	Conf. int. 95%
Check	1.46	±0.24	1.55	±0.22	—	—	—	—
Ifaces	1.12	±0.13	1.25	±0.03	—	—	—	—
Reg. app	1.08	±0.05	0.96	±0.03	—	—	—	—
Reg. rule	1.21	±0.09	1.29	±0.03	—	—	—	—
Download	6.28	±0.39	6.34	±0.3	6.82	±0.89	7	±0.4
Rm rule	0.19	±0.02	0.21	±0.02	—	—	—	—
Rm app	1.11	±0.02	1.22	±0.02	—	—	—	—
Total	12.5	±0.54	12.83	±0.41	—	—	—	—

Table 2: Round trip times measured using *Ping*.

	http://www.upv.es/		http://google.com/		http://www.yahoo.com/	
	Average	Std dev.	Average	Std dev.	Average	Std dev.
GRCBox	16.96	23.97	40.222	54.584	108.899	89.240
Directly conn.	14.210	35.9	24.402	34.101	113.774	111.254

interface, establishing global rules for non-GRCBox enabled applications, or rebooting the GRCBox.

As an open source development, we want to invite the research community to download, use, and improve our GRCBox architecture.

Appendices

A. REST API

(i) Root resource "/":

 (a) Method GET: information about the status of the server and the number of rules already registered in the database.

(ii) Ifaces resource "/ifaces":

 (a) Method GET: a simplified list of all the available outgoing interfaces.

(iii) Iface resource "/ifaces/{ifId}":

 (a) Method GET: information of a specific interface.
 (b) Method POST: at this moment this is not implemented, but we expect allowing authorized applications to remotely configure certain interface parameters such as the SSID or the password for wireless interfaces.

(iv) Applications resource "/apps":

 (a) Method GET: a list of the currently registered applications in the system.
 (b) Method POST: register a new application and return a secret password for later authentication.

(v) Application resource "/apps/{appId}": when a new application is registered a new specific resource is created. Access to the POST and DELETE methods is restricted to the original application.

 (a) Method GET: information about the specific application, its name, and its last-seen value.
 (b) Method POST: a call to this method is interpreted as a keep-alive signal by the server. If an application does not post to its ID for a certain amount of time, the application is deregistered and its defined rules are deleted from the database and from the system.
 (c) Method DELETE: remove an application and all its rules from the database and from the system.

(vi) Rules resource "/apps/{appId}/rules": each registered application can access to its list of rules. Access to the POST method is restricted to the *owner* of the resource.

 (a) Method GET: a list of the rules defined by this application.
 (b) Method POST: create a new rule.

(vii) Rule resource "/apps/{appId}/rules/{ruleId}": this resource is accessible when a new rule is created.

 (a) Method GET: details of the rule.
 (b) Method DELETE: remove a rule from database and system.

B. Client Library Classes and Methods

The client library includes the following classes.

(i) GrcBoxSocket: this class extends the Java Socket class. It removes the associated routing rule from the GRCBox routing service after the socket is closed.

(ii) GrcBoxServerSocket: this class extends the Java ServerSocket class. It removes the associated routing rule from the GRCBox routing service after the socket is closed.

(iii) GrcBoxDatagramSocket: this class extends the Java DatagramSocket class. It adds a new close() method to remove the associated routing rule from the GRCBox routing service.

(iv) GrcBoxMulticastSocket: this class extends the Java MulticastSocket class. It adds or removes the specific routing rules to the GRCBox routing service when the joinGroup() and leaveGroup() methods are used.

(v) GrcBoxClient: this class manages the registration process with the GRCBox core service, and it is in charge of socket creation.

The GrcBoxClient implements the following methods.

(i) GrcBoxClient(): initialize a GrcBoxClient object.

(ii) isServerAvailable(): check if the GRCBox core service is accessible by obtaining the root resource described in the previous appendix.

(iii) register(name): register a new application with name "*name*." Receive and store the secret password for future interaction with the server. When this method is called, a new thread is started to perform keep-alive signaling to the server.

(iv) deregister(): remove the registered application and all its defined routing rules from the GRCBox routing service. When this method is called, the keep-alive thread is stopped.

(v) getInterfaces(): contact the server and get a list of the available outer interfaces and their characteristics.

(vi) createServerSocket(port, outIface): register a new incoming routing rule on interface *outIface* and local port "*port*." It returns a GrcBoxServerSocket object ready to be used.

(vii) createSocket(addr/host, port, iface): register an outgoing routing rule to the destination address or host *addr/host* and destination port *port* through the interface *iface*. It returns a GrcBoxSocket object already connected to the remote host.

(viii) createSocket(addr/host, port, iface, localPort): register an outgoing routing rule to the destination address or host *addr/host* and destination port *port* from the local port *localPort* through the interface *iface*. It returns a GrcBoxSocket object already connected to the remote host.

(ix) createDatagramSocket(iface): register a new incoming routing rule and a new outgoing routing rule using any available local port on the specified interface in the GRCBox hardware module. It returns a new GrcBoxDatagramSocket object ready to be used.

(x) createDatagramSocket(port, iface): register a new incoming routing rule and a new outgoing routing rule using the local port *port* on the specified interface in the GRCBox hardware module. It returns a new GrcBoxDatagramSocket object ready to be used.

(xi) createDatagramSocket(port, remotePort, iface): register a new incoming routing rule and a new outgoing routing rule using the local port *port* and the remote port *remotePort* on the specified interface in the GRCBox hardware module. It returns a new GrcBoxDatagramSocket object ready to be used.

(xii) createDatagramSocket(port, remoteAddr, remotePort, iface): register a new incoming routing rule and a new outgoing routing rule using the local port *port,* the remote address *remoteAddr,* and the remote port *remotePort* on the specified interface in the GRCBox hardware module. It returns a new GrcBoxDatagramSocket object already "connected" to the remote address ready to be used.

(xiii) createMulticastSocket(port, iface): prepare a multicast rule to be register when the method *joinGroup(address)* of the returned GrcBoxMulticastSocket is called. It returns a GrcBoxMulticastSocket associated with the specific local port *port*.

(xiv) registerNewRule(⋯): register a new routing rule with its properties defined in the argument list. This low-level method is useful when an external library is used to manage high-level protocols, such as JSIP for session initiation protocol (SIP) communication. The user must remove the rule after using it.

(xv) removeRule(id): remove a routing rule from the database. It should only be used after calling the registerNewRule() method. If the removed rule does not exist, it does not have any effect on the status of the GRCBox hardware module.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was partially supported by the *Ministerio de Economía y Competitividad*, Spain, under Grants TIN2011-27543-C03-01 and BES-2012-052673, and by the European Commission under Svagata.eu, the Erasmus Mundus Programme, Action 2 (EMA2).

References

[1] H. Hartenstein and K. P. Laberteaux, "A tutorial survey on vehicular ad hoc networks," *IEEE Communications Magazine*, vol. 46, no. 6, pp. 164–171, 2008.

[2] IEEE Standards, "IEEE Standard for Information Technology—telecommunications and information exchange between systems—local and metropolitan area networks—specific

requirements part 11: wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications amendment 10: mesh networking," Tech. Rep., 2011.

[3] Car Connectivity Consortium (CCC), "MirrorLink," http://www.mirrorlink.com/, June 2014.

[4] Google Inc, "Android Auto," 2014, http://www.android.com/auto/.

[5] Apple, "CarPlay," 2014, https://www.apple.com/ios/ carplay/.

[6] J. P. Singh, N. Bambos, B. Srinivasan, and D. Clawin, "Wireless LAN performance under varied stress conditions in vehicular traffic scenarios," in *Proceedings of the 56th Vehicular Technology Conference (VTC '02)*, pp. 743–747, IEEE, September 2002.

[7] H. Wu, M. Palekar, R. Fujimoto et al., "An empirical stud y of short range communications for vehicles," in *Proceedings of the 2nd ACM International Workshop on Vehicular Ad Hoc Networks (VANET '05)*, pp. 83–84, ACM, 2005.

[8] F. Hui and P. Mohapatra, "Experimental characterization of multi-hop communications in vehicular ad hoc network," in *Proceedings of the 2nd ACM International Workshop on Vehicular Ad Hoc Networks (VANET '05)*, pp. 85–86, ACM, September 2005.

[9] M. Jerbi, S. M. Senouci, and M. Al Haj, "Extensive experimental characterization of communications in vehicular ad hoc networks within different environments," in *Proceedings of the IEEE 65th Vehicular Technology Conference (VTC '07)*, pp. 2590–2594, IEEE, April 2007.

[10] A. de la Fortelle, C. Laurgeau, P. Muhlethaler et al., "Com2react: v2v communication for cooperative local traffic management," in *Proceedings of the ITS World Congress*, 2007.

[11] K. C. Lee, S.-H. Lee, R. Cheung, U. Lee, and M. Gerla, "First experience with CarTorrent in a real vehicular ad hoc network testbed," in *Proceedings of the Mobile Networking for Vehicular Environments (MOVE '07)*, pp. 109–114, Anchorage, Alaska, USA, May 2007.

[12] C. Pinart, P. Sanz, I. Lequerica, D. García, I. Barona, and D. Sánchez-Aparisi, "Drive: a reco nfigurable testbed for advanced vehicular services and communications," in *Proceedings of the 4th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, p. 16, Institute for Computer Sciences, Social-Inf ormatics and Telecommunications Engineering, 2008.

[13] E. Giordano, A. Tomatis, A. Ghosh, G. Pau, and M. Gerla, "C-VeT an open research platform for VANETs: evaluation of peer to peer applications in vehicular networks," in *Proceedings of the 68th Vehicular Technology Conference (VTC '08)*, pp. 1–2, IEEE, Calgary, Canada, September 2008.

[14] M. Cesana, L. Fratta, M. Gerla, E. Giordano, and G. Pau, "C-VET the UCLA campus vehicular testbed: integration of vanet and mesh networks," in *Proceedings of the European Wireless Conference (EW '10)*, pp. 689–695, IEEE, April 2010.

[15] J. Santa, M. Tsukadat, T. Emstt, and A. F. Gómez-Skarmeta, "Experimental analysis of multi-hop routing in vehicular ad-hoc networks," in *Proceedings of the 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities and Workshops (TridentCom '09)*, pp. 1–8, April 2009.

[16] M. C. G. Paula, J. N. Isento, J. A. Dias, and J. J. P. C. Rodrigues, "A real-world VDTN testbed for advanced vehicular services and applications," in *Proceedings of the IEEE 16th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD '11)*, pp. 16–20, IEEE, June 2011.

[17] A. Campbell and T. Choudhury, "From smart to cognitive phones," *IEEE Pervasive Computing*, vol. 11, no. 3, pp. 7–11, 2012.

[18] W. Vandenberghe, I. Moerman, and P. Demeester, "On the feasibility of utilizing smartphones for vehicular ad hoc networking," in *Proceedings of the 11th International Conference on ITS Telecommunications (ITST '11)*, pp. 246–251, St. Petersburg, Russia, August 2011.

[19] D. Sawada, M. Sato, K. Uehara, and J. Murai, "IDANS: a platform for disseminating information on a VANET consisting of smartphone nodes," in *Proceedings of the 11th International Conference on ITS Telecommunications (ITST '11)*, pp. 252–257, St. Petersburg, Russia, August 2011.

[20] S. M. Tornell, C. T. Calafate, J. C. Cano, P. Manzoni, M. Fogue, and F. J. Martinez, "Evaluating the feasibility of using smartphones for ITS safety applications," in *Proceedings of the IEEE 77th Vehicular Technology Conference (VTC '13)*, pp. 1–5, IEEE, June 2013.

[21] G. Mitchell, "The Raspberry Pi single-board computer will revolutionise computer science teaching [For Against]," *Engineering Technology*, vol. 7, p. 26, 2012.

[22] M. Thompson and P. Green, Raspbian, http://www.raspbian.org/, June 2014.

[23] S. Kelley, "Dnsmasq, network services for small networks," 2014, http://www.thekelleys.org.uk/dnsmasq/doc.html.

[24] Netfilter Core Team, "The netfilter.org iptables project," 2014, http://www.netfilter.org/projects/iptables/.

[25] M. G. Marsh, *Policy Routing Using Linux*, Sams Professional Series, Sams, 2001.

[26] The GNOME Project, "Networkmanager homepage at gnome wiki," 2014, https://wiki.gnome.org/action/show/Projects/NetworkManager.

[27] R. T. Fielding, *Architectural styles and the design of network-based software architectures [Ph.D. thesis]*, University of California, 2000.

[28] J. Louvel, T. Templier, and T. Boileau, *Restlet in Action: Developing RESTful Web APIs in Java*, Manning Publications, Greenwich, Conn, USA, 2012.

[29] Savarese Software Research Corporation, "RockSaw Home Page," 2014, https://www.savarese.com/software/rock- saw/.

[30] "Android website," 2014, http://www.android.com.

[31] GRC, "GRC GitHub Account," https://github.com/GRCDEV, August 2014.

[32] The GNOME Project, "Network Manager DBUS Interface Specification," 2014, https://developer.gnome.org/NetworkManager/unstable/spec.html.