Simplifying the in-vehicle connectivity for ITS applications

Sergio M. Tornell, Carlos T. Calafate,
Juan-Carlos Cano, Pietro Manzoni
sermarto@upv.es,{calafate,jucano,
pmanzoni}@disca.upv.es
Department of Computer Engineering
Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain

Teemu Kärkkäinen, Jörg Ott
{teemuk, jo}@netlab.tkk.fi
Äalto University
Espoo, Finland

ABSTRACT

In-vehicle connectivity has experienced a big expansion in recent years; car manufacturers are very active in this sense, and are proposing OBU oriented solutions. This effort is justified by the user demands for always-on connectivity. However, currently available OBUs do not provide the desired flexibility and simplicity of use that would be desirable for ITS applications. For example, none of them considers the possibility for inter-vehicle device-to-device communications.

In this paper we present GRCBox, an architecture that extends the in-vehicle connectivity by providing inter and in-vehicular communication support. By creating private vehicular networks, GRCBox allows user devices' applications to perform direct peer-to-peer communication. In this paper we describe the GRCBox design along with four case studies. We also include the experimental results obtained from a test-bed to show that our solution does not have a negative impact on the performance when compared to a centralized solution.

Categories and Subject Descriptors

C2.1 [**Network Architecture and Design.**]: Distributed networks.

Keywords

Vehicular Networks, V2V, Smartphone, GRCBox, VANET, ITS

1. INTRODUCTION

Vehicular Networks (VNs) combine several technologies to provide Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) communications [1]. Opportunistic V2V networks [2] allow implementing applications such as road-status notification [3], vehicle platoon coordination, or collaborative content downloading [4]. Although the technology is ready for deployment, it is expected that car manufacturers will introduce it gradually, starting at high-cost models, which, coupled with the low renovation rate of the vehicle fleet, will slow down the deployment of VNs. In addition, dashboard-integrated On Board Units (OBUs) typically become technologically obsolete after a couple of years and they are usually not designed to be updated or replaced during the whole vehicle lifetime, which leads to unsatisfied users.

Meanwhile, the popularization of smartphones has brought devices with multiple network interfaces to almost everyone's pocket. Smartphones are continuously carried by users and have multiple network interfaces, which makes them a suitable platform for implementing applications based on opportunistic contacts. However, not only is the smartphone's connectivity restricted to infrastructure networks, such as WiFi or 3G/4G networks, but also the number of simultaneous active network interfaces is limited to one. These restrictions are limiting the adoption of smartphones for applications based on opportunistic connectivity in vehicular scenarios.

To extend in-vehicular connectivity to external networks such as Vehicular Ad-Hoc Networks (VANETs), we have designed the GRCBox Architecture. The GRCBox Architecture is based on the GRCBox Connectivity Manager (GCM), which is responsible for creating an intra-vehicle WiFi network. User devices inside the vehicle can connect to this network to share contents and to reach any of the external networks, as depicted in Figure 1. GRCBox allows implementing Internet-independent solutions that focus on applications that exploit local connectivity to provide new services, such as platoon-oriented applications where friends or workers share information while traveling together in different vehicles. Opportunistic applications

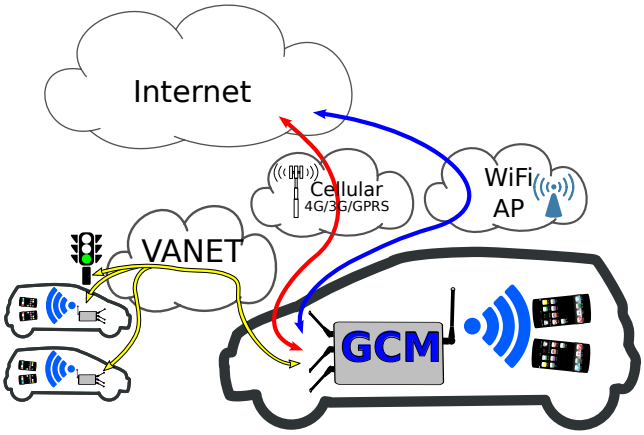


Figure 1: An example GCM connected to several networks.

are especially suitable for remote areas where infrastructure is expensive to deploy. Moreover, the short life and local propagation of the information favors privacy. The GRCBox Architecture provides a Representational State Transfer (REST) interface [5] and it is based on basic IP networking, thereby minimizing the modifications required to create GRCBox-aware applications. GRCBox also removes the dependency on car manufacturers when implementing V2V communications; by using GRCBox, users can now implement their own VANETs.

Part of the industry has proposed vendor-specific alternatives that integrate smartphones in VNs. The Car Connectivity Consortium (CCC), which integrates companies from the automotive and the telecommunications sector, released Mirrorlink [6], a standard technology that moves the computing tasks from the OBU to the smartphone, and present the information on the OBU's display. Users can also interact with the smartphone through the dashboard elements. Google and Apple, two of the biggest technology companies, have also proposed their own solutions, Android Auto [7], and CarPlay [8], respectively. However, all these proposals rely on the Internet infrastructure to provide in-vehicle connectivity, ignoring the advantages of V2V communication and opportunistic contacts. Moreover, these proposals are heavily dependent on companies and centralized service providers.

An example of new applications based on opportunistic peer-to-peer communications is the Scampi project [9]. Its authors developed a framework to provide opportunistic communication for smartphones. They proposed to deploy autonomous routers (called LibeRouters) that, by creating WiFi connectivity islands, provide a network for opportunistic contacts between smartphones. By using a Delay Tolerant Network (DTN) [10] architecture and taking advantage of node mobility, Scampi distributes messages to nodes connected to other Scampi routers. Since the Scampi platform requires nodes to be associated to the same router in order to exchange information, it is not suitable for VNs, where nodes move quickly and contacts based on the infrastructure last only for a short period. Our GRCBox can complement the Scampi platform by increasing the smartphones' connectivity beyond the local Scampi router, thereby increasing the number and duration of opportunistic contacts.

To the best of our knowledge, GRCBox is the first effort aimed at increasing the user device in-vehicle connectivity in order to allow users to create their own autonomous VN and test innovative VN applications.

The rest of this paper is organized as follows: Section 2 details the GRCBox Architecture. Section 3 presents four case studies, that illustrate the use and performance of the GRCBox. Later, section 4 provides some insights on ongoing developments and future plans. Finally, section 5 concludes the paper by summarizing our contributions.

2. THE GRCBOX ARCHITECTURE

The GRCBox Architecture defines both the GRCBox Connectivity Manager (GCM) placed in the vehicle and a client-server REST API that allows applications to interact with the GCM to reach external networks. To implement the REST API we used the RESTlet framework [11], which simplifies the implementation. Figure 2 represents the architecture including both parts the GCM and the client API. An example of a GCM placed in a vehicle and connected to three different external networks, is shown in Figure 1. In this example an application running in the user device may choose to connect to the VANET for local communication, or connect to either the cellular network or the WiFi network to reach the Internet. In this section, we first detail the different software modules running in the GCM. Then, we offer a general overview of the interaction between the GCM and the User Application.

2.1 The GCM

The GRCBox Connectivity Manager (GCM), which is placed inside vehicles, must have at least one WiFi interface to which user devices are connected to (called *inner interface*), and one or more *external interfaces* used to provide connectivity to external networks. The GCM is composed of several modules that work together. A scheme of the different components, their connections, and the paths traversed by data flows is presented in Figure 2. The GCM software is based on a Linux operating system, and it takes advantage of several well-known Linux services to provide the desired functionality. The different components running in the GCM are the following:

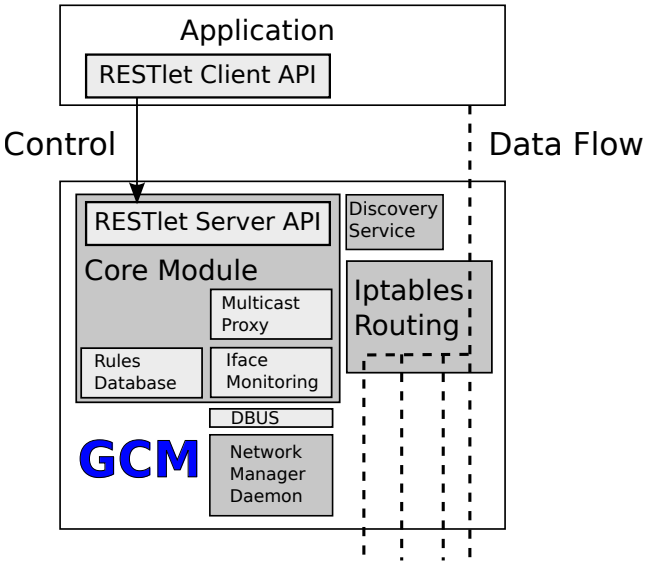


Figure 2: GRCBox Architecture with GCM modules in detail.

Discovery Service: The Linux daemon *dnsmasq* is used to answer DHCP and DNS requests. It is configured to resolve the "grcbox" domain name to the GCM inner interface. This way clients on the inner network can connect to the GCM without information about its IP address by attempting a connection to "http://grcbox/".

Packet Forwarding: To define fine grained, per connection routing, GCM uses Iptables for connection filtering and labeling, and the Linux kernel support for "Policy Routing".

Ifaces Monitoring: To monitor the status of the network interfaces, GCM connects to the NetworkManager daemon using the DBUS interface to perform event subscribing tasks.

Core Module: The most important part of the GCM is its core module. The core module performs several activities: it listens to clients' requests through the REST API, maintains a database of all registered rules, starts and stops multicast proxies when needed, and performs actions when events on the interfaces are notified.

Concerning the hardware required to run the GCM, our plans are to find a cheap and small embedded computer that can be easily plugged into vehicles. The computing power required by the GCM is minimal, and it should run flawlessly in embedded computers such as Raspberry Pi[1] or BeagleBone Black[2]. However, in our first experiments using Raspberry Pi, we found that the board presented power management and instability issues that prevent connecting more than one wireless interface. Currently we are testing several other embedded boards to find one suitable to our requirements. Meanwhile, we have used an Asus EeePC netbook with low computational power.

[1]http://www.raspberrypi.org/
[2]http://beagleboard.org/black

2.2 User Device-GCM Interaction

The GCM creates a WiFi access point to which smartphones, tablets, and other user devices in the vehicle will associate. Once the user devices connect to the GRCBox's wireless network, they can share contents between them, as well as access the external networks. By default, every new connection is forwarded from the GCM through the default Internet connection. In case an application requires the use of any other available interface, it must notify it to the GCM. In this section we enumerate the steps a GRCBox application must follow to communicate with a non-default network. First of all, we need to introduce the concept of "rules": A rule enables 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:

- Rule Type: The GRCBox Architecture defines three different kind of rules, *Incoming*, *Outgoing* and *Multicast*. Multicast rules define bi-directional multicast packet flows between the internal interface and one of the external interfaces.
- Interface Name: The name of the outer interface to which the rule applies.
- Protocol: The protocol of the connection. Currently, GRCBox supports UDP and TCP, though we expect to implement more protocols, such as SCMP or ICMP, in the future.
- Source Port: The source port of the connection.
- Source Address: The source address of the connection.
- Destination Port: The destination port of the connection.
- Destination Address: The destination IP address of the connection.

The steps that GRCBox applications must perform are the following:

1. **Check GRCBox availability:** Once the device is associated to the GRCBox wireless network, the application must check if a GCM is available. To do so, the application will try to connect to the "http://grcbox/" url to check the status of the GCM.

2. **Application Registration:** After checking the availability of the GCM, an application must register itself to get a key. This key will be used for later application-server interactions to ensure no other application but the owner of a rule can renew, remove, or modify it.

3. **Check the Status of the Interfaces:** The next step is to check the status of the different network interfaces to identify if the desired interface is available. At this point the application can also check other previously registered rules to avoid conflicts.

4. **Register the desired rule[s]:** Now the application can register as many rules as required to configure the GCM to forward specific incoming and outgoing connections, or to forward multicast packets to external interfaces.

5. **Transmit Data:** At this point the application can effectively use the registered connections which will be forwarded according to the defined rules.

6. **Close the Connection:** When a rule is no longer required, it must be removed from the GCM. This step is optional since rules are always removed from the GCM database if the application is disconnected.

7. **Application Disconnection:** Once the application ends its interaction with the GCM, it should notify it to allow removing its registered rules.

The interactions between GRCBox applications and the GCM rely on the RESTlet API exposed by the GCM. The details of this API are described in [12]. Section 3 includes examples of multiple case studies to clarify this communication.

GRCBox also supports the integration of third party applications by providing a management application that enables the interactive definition of new rules for non-GRCBox applications. Thereby, the user can define rules for well-known application protocols such as HTTP, POP3, etc.

3. CASE STUDIES

In this section we detail the interaction between a GRCBox-enabled application and the GCM through four different case studies. In the first case study, we analyze the communication between an application and the GCM in the case of a typical client-server connection between two user devices connected to different GCMs. In the second case study, we illustrate how the Scampi middleware can be adapted to support the GRCBox Architecture. The first and the second case studies focus on the most novel feature of the GRCBox Architecture;

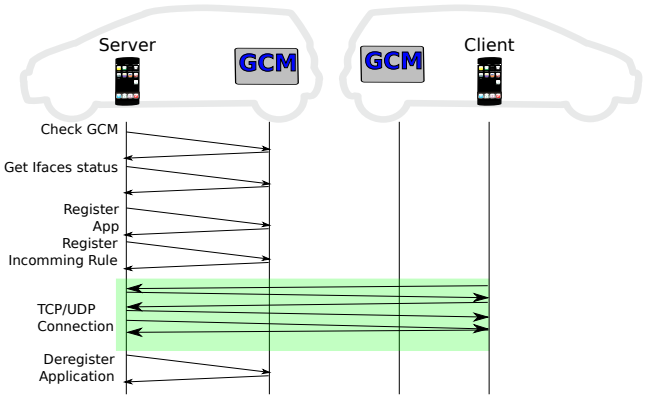


Figure 3: Example of a client GRCBox application connecting to a server.

i.e. VANET connectivity. Therefore, we do not limit our analysis to describing them, but also analyze the performance of the GRCBox Architecture. In the third case study, we describe the case of a TCP/IP application that wants to establish a connection with a server connected to the Internet using a specific external interface. The fourth case study presents the case of a VoIP application that requests to route all the VoIP connections through a specific interface. We believe that, although more specific use cases might occur, these case studies illustrate the flexibility offered by the GRCBox Architecture.

3.1 Direct VANET Communication

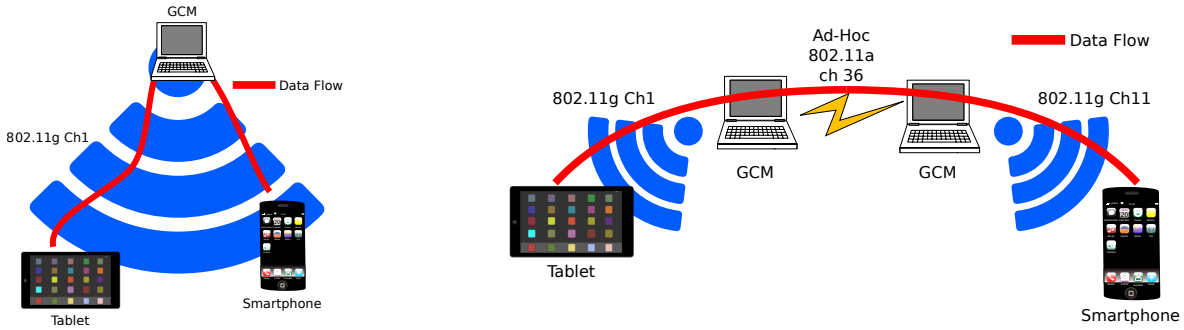
This is a basic case of direct client-server communication between devices. It assumes that the *client* and *server* connectivity has been configured using some auto-configuration system like the one presented in [13]. Therefore, the *client* knows the public IP of the vehicle connected to the VANET where the device acting as a *server* is connected. Neighbor discovery is outside the scope of this case study, and its impact is explored in the next case study. For both UDP flows and TCP connections, only one rule on the server side must be registered. The rule must specify which port is the *server* running in the user device listening for connections at, so the GCM can forward connections attempts on the external interface by performing Network Address Translation (NAT). As depicted in Figure 3, the steps performed by the *server* device for establishing such communication are the following:

1. The device must check the availability of the GCM.

2. Connect to the GCM and register itself.

3. Check the status of the interfaces and select the VANET interface.

4. Register a new rule to forward incoming connections.



(a) Scenario 1

(b) Scenario 2

Figure 4: Scenarios used in our experiments.

5. Wait for incoming connections from *clients* and process them.

6. When the server is stopped, the application should deregister itself from the GCM, which will also remove all its rules.

The *client* does not need to perform any interaction with its GCM, since the GCM will forward the connection through the VANET interface based on the destination IP.

3.1.1 Performance Analysis

We have run experiments to evaluate the performance of the GRCBox Architecture in 2 different scenarios: the first experiment is an analysis of the maximum throughput, and the second analyses the UDP Round Trip Time (RTT) between client and server. In the first scenario, we used an Android Nexus 7 tablet and a BQ4.5E smartphone connected to the same GCM, which acted as a standard WiFi Access Point, since connections can be established without interaction with the GCM, this is the baseline scenario. In the second scenario we connected each device to different GCMs, which were then connected to the same ad-hoc network, in this case the connection must be established through the GCMs. Table 1 and Figure 4 summarize the configuration of both scenarios. To implement the GCMs we used an Asus 1000h EeePC netbook running a Debian Linux distribution with an 802.11a USB wireless network interface configured in the Ad-Hoc mode.

The GRCBox management application was used to configure the required rule on the GCMs.

Maximum Throughput.

To evaluate the impact of the GRCBox Architecture on the maximum throughput experienced by a client-server connection when using the GRCBox, we tested the network performance using the *iperf* [14] tool. We have collected measurements for both UDP and TCP protocols. Each experiment was repeated 60 times to discard random effects, and the role of the user devices was interchanged after half of the experiments to discard the effects associated to the device's performance. Results are shown in Figure 5a in a boxplot chart.

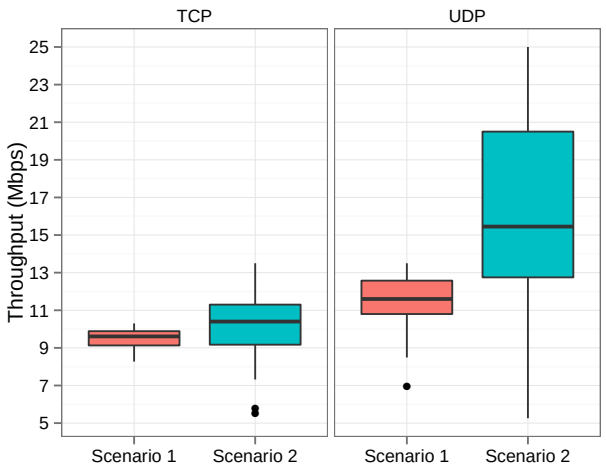
Table 1: Devices Configuration

Element	Characteristics
Tablet	Google Nexus 7 (2012)
Smartphone	BQ Aquaris 4.5E
GCMs	Asus EeePc 1000h, Intel Atom N270
Ad-Hoc Network	802.11a, Frequency:5.18 GHz
WiFi 1	802.11g, Frequency:2.462 GHz
WiFi 2	802.11g, Frequency:2.412 GHz

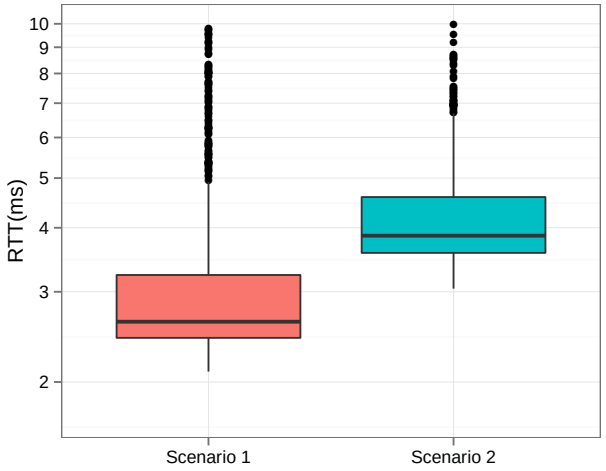
Notice that, no matter whether TCP or UDP is used, the maximum throughput achieved when using the GRCBox Architecture (Scenario 2) is slightly better than the one achieved when both devices are connected to the same WiFi Network (Scenario 1). The main cause behind this difference is the use of a different channel for each wireless network when using the GRCBox. This setup avoids collisions between nodes, when transmitting requests and responses. The high variability experienced in all the experiments is due to the presence of interference, which heavily affects throughput in wireless networks.

UDP Round Trip Time (RTT).

To test the delay introduced by the GRCBox Architecture when comparing against an infrastructure network we have developed a small application that sends an UDP message to a server running on another device. The server will then send a new UDP message as a response, so the RTT can be measured at the first sender. We performed the test on both scenarios presented before, collecting more than 500 measurements per scenario. Figure 5b shows a boxplot that summarizes the results we obtained. We have used a logarithmic scale to be able to clearly represent infrequent values in both the low and high ranges. It can be observed that, on



(a) Throughput obtained for UDP and TCP tests.



(b) UDP RTT results.

Figure 5: Direct VANET communication results.

average, the RTT is about 2 ms higher when using GRCBox. This effect is due to the multi-hop nature of the communication: adding an extra hop between sender and receiver increases the RTT. During this experiment we discovered that the main source of delay in Android devices is the WiFi interface power management performed by the Android operating system, which in some cases increased the RTT by up to 508 ms. If we compare the delay introduced by the GRCBox against the delay introduce by the OS, we can conclude that the GRCBox impact on the RTT is negligible.

3.2 Scampi: Neighbor Discovery and Connection Establishment

To illustrate the use of a more elaborated VANET application we describe the modifications required to adapt the Scampi router [9] to the GRCBox Architecture. Although Scampi was not designed for VANETs, it is a good example of a Device to Device (D2D) application. Scampi neighbors are discovered by multicasting beacon messages periodically. Once a new neighbor has been discovered, Scampi establishes a TCP/IP connection to exchange information. As shown in Figure 6, the modifications required on the the Scampi router to adapt it to the GRCBox are the following:

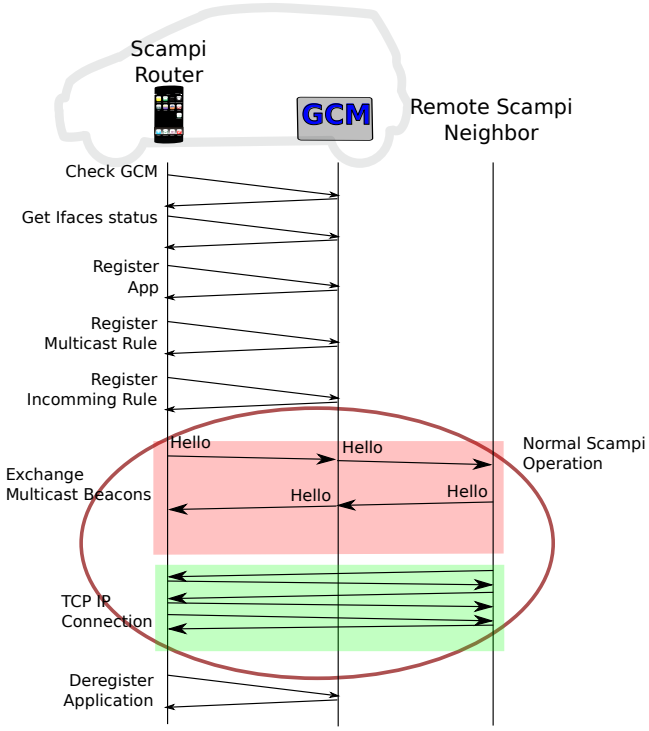


Figure 6: Example of integration between Scampi and GRCBox.

1. Check the GCM availability.

2. Get the list of available interfaces and their status.

3. Create a new multicast proxy associated to the Ad-Hoc interface to forward multicast beacons to the external networks.

4. Register a rule for incoming connections from the neighbors reached by the multicast messages. Now the application can receive connections from remote neighbors.

5. Perform standard Scampi activity.

6. Once the application is closed, it must remove itself from the GCM's applications database. This will remove both rules.

In this case, both rules are long-lasting rules that must be active as long as the application is running.

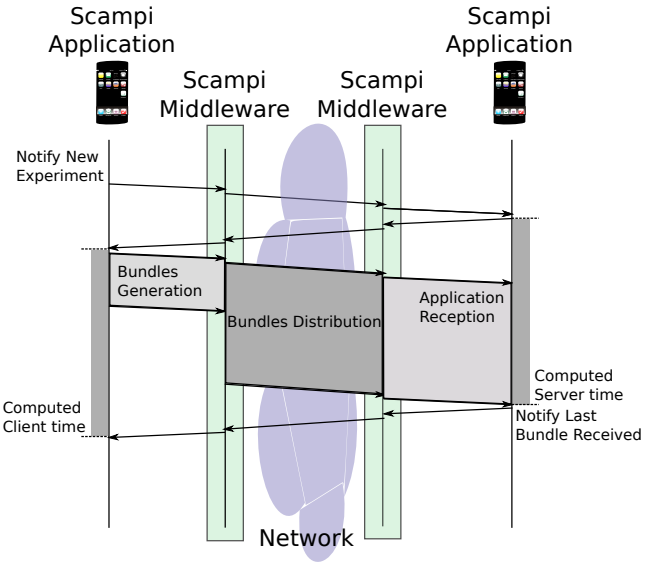


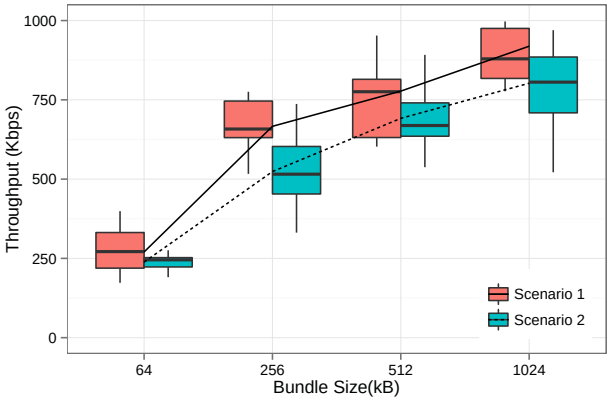
Figure 7: Bundles sent to measure the Scampi throughput.

3.2.1 *Performance Analysis*

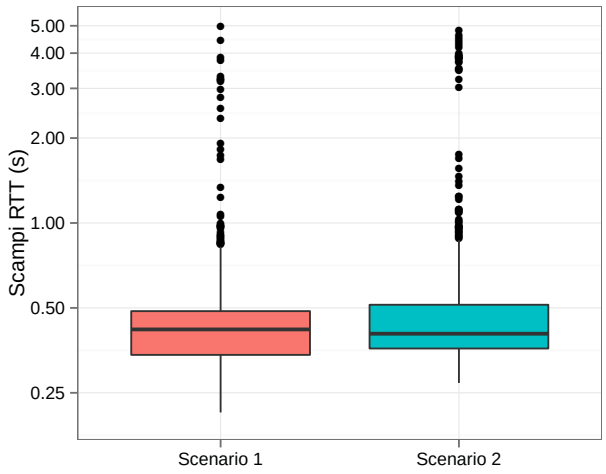
As in the previous case, we have run two different experiments to evaluate the performance of the GRCBox Architecture combined with Scampi: the first experiment measures the maximum bundle throughput, and the second experiment measures the Scampi RTT of bundles between two devices. To avoid modifying the Scampi's source code we have created a *multicast plugin* that can be activated through the management application to forward the multicast beacon messages sent by the Scampi router from the inner network to the desired external network. Both experiments were run once the Scampi nodes had discovered each other and the Scampi topology was stable.

Scampi Throughput.

To measure the performance of the Scampi platform when combined with the GRCBox Architecture we have developed a *client* application that, by using the Scampi API, generates a burst of bundles that are passed to the Scampi router in order to be distributed to scampi neighbors. Once the *server* application, which is running in the other device, receives all the bundles, it confirms their reception by creating a new bundle. The *server* application measures the time and the amount of data received to calculate the throughput in terms of Mbps. When running the scampi middleware under heavy load it presents some issues that complicate computing the time required to exchange a certain number of bundles. Therefore, we needed to signal the beginning of each experiment through a notification bundle, that has to be confirmed by the *server*. In the same way, the reception of the last bundle of the burst has



(a) Scampi throughput in Mbps.



(b) Scampi RTT test results.

Figure 8: Scampi results.

to be notified to the *client*, so it can compute the total required time. This behavior is summarized in Figure 7. It is important to notice that, when the scampi middleware receives a new bundle, it is not immediately delivered to the user application, and thereby the Scampi RTT does not depend on the network resources alone. We repeated the experiment varying the size of the bundles. Each experiment was repeated 10 times. Figure 8a shows the obtained results.

According to the results, the Scampi throughput is slightly better in scenario 1. However, the range of the boxes overlaps in most cases, which means that the difference between both scenarios is not statistically significant. Besides, the figure also shows that the performance of Scampi is really poor when compared to the results presented in the section 3.1, and that it becomes worse as the bundle size becomes smaller. The reason is that the Scampi platform has a big overhead, not only because of the bundle protocol, but also due to computation overhead at the middleware layer: every bundle

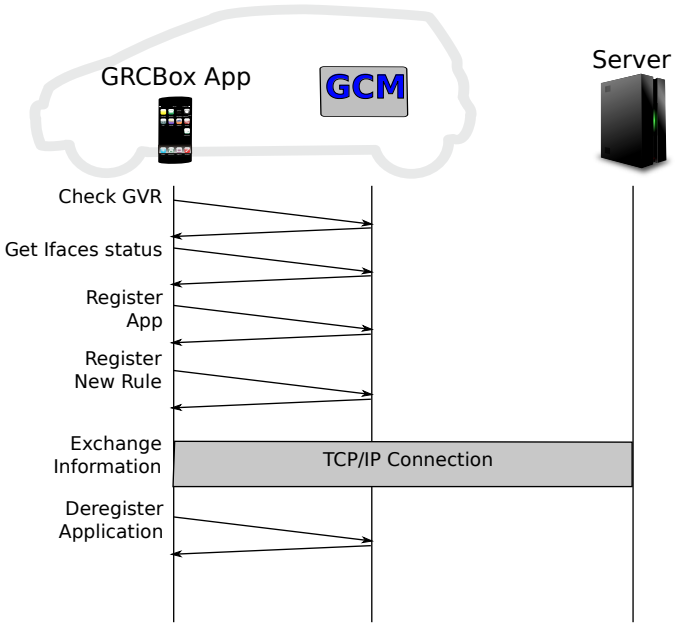


Figure 9: Example of a client GRCBox application connecting to a server.

must be processed by several threads, including copying it to permanent storage before notifying it to the application.

Scampi Round Trip Time (RTT).

The RTT test consisted on generating a minimum-size bundle on the *source node*, that is then distributed by the Scampi router. When the application running on the *destination device* receives the bundle, it generates a *response bundle* that confirms the reception of the first bundle. Finally, when the *source node* application receives the *response bundle* it computes the RTT. Figure 8b shows the obtained results.

As can be seen in the results, there is almost no difference between the distributions of the RTT in both scenarios. When both devices are connected to the same AP, the Scampi RTT is slightly better than when using the GRCBox. However, when focusing on the value of the quartiles, this small difference becomes insignificant. It is worth noticing the high number of measurements that experienced a high RTT (represented as outliers). These high values are due to some instability issues found in the Scampi middleware under heavy load in Android.

3.3 Device to Internet Connectivity over WiFi

Figure 9 illustrates the case where an application wants to use the WiFi interface to connect to a remote Internet server and avoid using the low-bandwidth 3G connection. To do so, the application must perform the following steps:

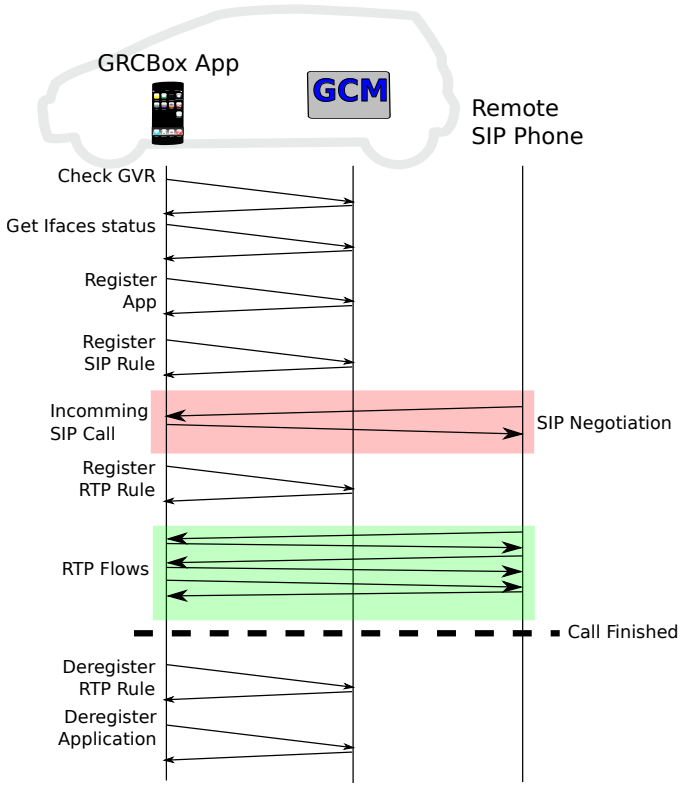


Figure 10: Example of a VoIP GRCBox application.

1. Check the GCM availability.
2. Get the list of available interfaces and their status.
3. Register an outgoing rule which includes the remote address, the remote port, and the desired interface.
4. Initiate the connection and exchange information as usually, using sockets or any other network API.
5. Once the connection has finished, the rule must be removed.
6. When closed, the application must remove itself from the GCM applications database.

In this case, the registered rule is only used during the connection to the server, and it is expected to be removed by the application as soon as the connection ends.

3.4 VoIP Application over 3G

Figure 10 illustrates the case of a GRCBox VoIP application attempting to use the highly-stable 3G connection to receive VoIP calls ignoring more unstable interfaces. Due to the nature of VoIP connections, the application must follow these steps, which include creating 2 different rules:

1. Check the GCM availability.

2. Get the list of available interfaces and their status.

3. Register an incoming rule for SIP connections that includes only the local port, the local address, and the desired listening external interface.

4. When a SIP negotiation occurs, the application must create as many rules as required according to the parameters of the negotiated RTP flows.

5. At this time, the RTP packets can flow between the GRCBox application and its remote peer.

6. Once the call is concluded, the RTP rule can be removed from the GCM.

7. When the application is closed it must remove itself from the GCM applications database. This will also remove any other rule registered by the application.

In this case, the SIP rule is a permanent rule that must be active as long as the application is running. On the other hand, the RTP rule is expected to be removed as soon as the VoIP call finishes.

4. WORK IN PROGRESS

This section includes some ongoing development. The first subsection presents two real applications we are currently developing and testing. The second subsection is a list of issues that remain open for the GRCBox.

4.1 Other GRCBox Applications

We have designed GRCBox with the goal of creating a platform to easily test collaborative vehicular communication solutions for user devices. We are currently working on a variety of GRCBox applications that use the ad-hoc communication capabilities.

One of the applications is an *overtake assistant* application that streams video captured from the camera of a smartphone placed on the dashboard of the vehicle in front to the rear vehicle. The application sends periodic beacons through the GCM to announce the location of the vehicle. Using this information, rear vehicles can detect vehicles in front and ask for an overtake-assisting video stream.

We have also adapted our previously presented *Warning Ambulance Application* [15] which shows a warning on the driver's smartphone when a warning message from a nearby ambulance is received. The warning message contains information about the ambulance's location and future route, so that the driver can take decisions in advance, increasing security and reducing ambulance delay. Moreover, the application also forwards the warning message to other nodes. Figure 11 shows a screen capture of our application showing the ambulance location and future route (in orange), as well as

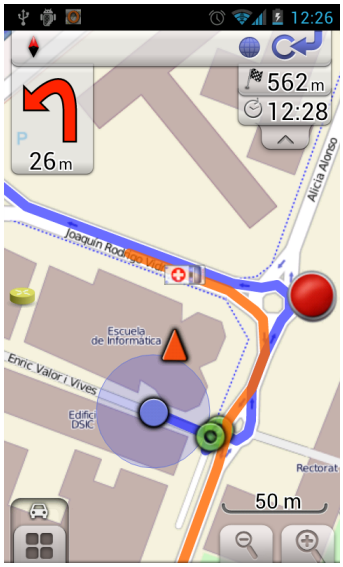


Figure 11: Ambulance warning application

the vehicle's current location and route (in blue)[3]. To run the original application, the user was forced to root its smartphone; in addition, no matter whether the device is rooted, not all the smartphones can be configured to use ad-hoc communications. By using the GRCBox architecture, the application is easier to deploy and test.

Beside testing services that would require a high number of GCMs to be deployed for these services to be useful, such as the ones introduced before, we envision GRCBox applications focused on vehicle platoons. The GRCBox architecture can be used to provide inter and intra-vehicle communication to friends or coworkers that move together using different vehicles. They can share multimedia content, directions, etc. without requiring a mobile broadband Internet connection which is usually hard to find on remote roads.

4.2 Open Issues

Although GRCBox is fully operational, some issues remain. We plan to improve GRCBox by implementing high-level rules, thereby creating a new semantic able to support rule definitions such as "use the most stable interface for VoIP calls" or "forward multicast packets to every ad-hoc or infrastructure network".

Currently, when using the management application, the user can only define which interface it wants to use, being unable to configure the interface properties. We plan to support remote interface configuration. The management application can also be improved by including some predefined rules for some common third-party applications.

We have designed the GRCBox Architecture as a

[3]A video of this application can be found on our channel in youtube `https://www.youtube.com/watch?v=Wh4cwmdvecM`

quick deployment platform for research, allowing to easily test vehicular applications for smartphones; therefore, we have considered that all the devices in the network were trustworthy. Currently, it would be simple to create a malicious application that compromises the system. In case of commercial deployment plans, further research in this direction must be carried out.

5. CONCLUSIONS

In this paper we have presented the GRCBox Architecture, which allows implementing vehicular applications by extending in-vehicle connectivity to external networks. The GRCBox Architecture is composed of a low-cost hardware module called GRCBox Connectivity Manager (GCM), and a set of libraries that allow developers to use it. We have presented four study cases that illustrate the use of the GRCBox Architecture: *i)* simple VANET communication, *ii)* neighbor discovering and connection establishing using Scampi, *iii)* access to the Internet, and *iv)* VoIP Internet application. We have evaluated the performance of the GRCBox on the two first cases. Our results shown that the GRCBox Architecture is fully operational and that, in terms of throughput and delay, the penalty to pay when comparing the GRCBox against an infrastructure network is minimal and clearly compensated by the extra connectivity offered by the GRCBox.

The GRCBox Architecture have been released under an open source license and can be found in our GitHub page [16]. As an open source development, we want to invite the research community to download, use, and improve our GRCBox Architecture.

Acknowledgments

This work was partially supported by the *Ministerio de Economía y Competitividad, Programa Estatal de Investigación, Desarrollo e Innovación Orientada a los Retos de la Sociedad, Proyectos I+D+I 2014*, Spain, under Grant TEC2014-52690-R, BES-2012-052673 and EEBB-I-14-07890.

References

[1] M. Gerla and L. Kleinrock. "Vehicular networks and the future of the mobile internet". In: *Computer Networks* 55.2 (Feb. 2011), pp. 457–469. ISSN: 13891286. DOI: 10.1016/j.comnet.2010.10.015.

[2] K. C. Lee and M. Gerla. "Opportunistic vehicular routing". In: *Wireless Conference (EW), 2010 European*. IEEE, Apr. 2010, pp. 873–880. ISBN: 978-1-4244-5999-5. DOI: 10.1109/EW.2010.5483530.

[3] J. Santa and A. Gomez-Skarmeta. "Sharing context-aware road and safety information". In: *Pervasive Computing, IEEE* (2009), pp. 58–65.

[4] O. Trullols-Cruces et al. "Cooperative Download in Vehicular Environments". In: *IEEE Transactions on Mobile Computing* 11.4 (Apr. 2012), pp. 663–678. ISSN: 1536-1233. DOI: 10.1109/TMC.2011.100.

[5] R. T. Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, 2000.

[6] Car Connectivity Consortium (CCC). *MirrorLink*. http://www.mirrorlink.com/. Feb. 2015.

[7] Google Inc. *Android Auto*. http://www.android.com/auto/. Feb. 2015.

[8] Apple Inc. *CarPlay*. https://www.apple.com/ios/carplay/. June 2014.

[9] T. Kärkkäinen et al. "SCAMPI Application Platform". In: *Proceedings of the Seventh ACM International Workshop on Challenged Networks*. CHANTS '12. New York, NY, USA: ACM, 2012, pp. 83–86. ISBN: 978-1-4503-1284-4. DOI: 10.1145/2348616.2348636.

[10] M. Khabbaz, C. Assi, and W. Fawaz. "Disruption-tolerant networking: A comprehensive survey on recent developments and persisting challenges". In: *IEEE Communications Surveys & Tutorials* 14.2 (2012), pp. 607–640.

[11] J. Louvel, T. Templier, and T. Boileau. *Restlet in Action: Developing RESTful Web APIs in Java*. Greenwich, CT, USA: Manning Publications Co., 2012. ISBN: 193518234X, 9781935182344.

[12] S. M. Tornell et al. "GRCBox : Extending Smartphone Connectivity in Vehicular Networks". In: *International Journal of Distributed Sensor Networks* (2014), Article ID 478064.

[13] J. Cano et al. "EasyMANET: An extensible and configurable platform for service provisioning in MANET environments". In: *IEEE Communications Magazine* 48 (2010), pp. 159–167. ISSN: 01636804. DOI: 10.1109/MCOM.2010.5673087.

[14] *Iperf homepage*. https://iperf.fr/. Feb. 2015.

[15] S. M. Tornell et al. "Evaluating the feasibility of using smartphones for ITS safety applications". In: *VTC Spring 2013 IEEE Vehicular Technology Conference*. 2013. ISBN: 9781467363372.

[16] GRC. *GRC GitHub Account*. https://github.com/GRCDEV. Aug. 2014.