Lightweight Virtualization as Enabling Technology for Future Smart Cars

Roberto Morabito[†], Riccardo Petrolo[§], Valeria Loscrì[*], Nathalie Mitton[*], Giuseppe Ruggeri[‡], and Antonella Molinaro[‡]
[†]Ericsson Research, Jorvas, Finland
[§]Rice University, Texas, USA
[*]Inria Lille - Nord Europe, France
[‡]DIIES, University Mediterranea of Reggio Calabria, Italy

***Abstract*—Modern vehicles are equipped with several interconnected sensors on board for monitoring and diagnosis purposes; their availability is a main driver for the development of novel applications in the smart vehicle domain. In this paper, we propose a Docker container-based platform as solution for implementing customized smart car applications. Through a proof-of-concept prototype—developed on a Raspberry Pi3 board—we show that a container-based virtualization approach is not only viable but also effective and flexible in the management of several parallel processes running on On Board Unit. More specifically, the platform can take priority-based decisions by handling multiple inputs, e.g., data from the CANbus based on the OBD II codes, video from the on-board webcam, and so on. Results are promising for the development of future in-vehicle virtualized platforms.**

Keywords*—*Internet of Things, Smart Vehicles, Internet of Vehicles, Virtualization Technologies, Container.

I. Introduction

Nowadays, modern vehicles are equipped with very sophisticated electronic systems and associated in-vehicle communication infrastructures, aimed to provide different functionalities which vary from engine control, predictive diagnostics, driving assistance, and infotainment. Implementing new functions requires more and more computing and communication resources to be installed on board. As an example, the increasing demand of safety boosted the trend to equip vehicles with single and multiple cameras that transfer recorded videos on the vehicle bus, to be processed by the On Board Unit (OBU) [1]. Furthermore, the recent development of Social Vehicular Networks [2] is another driver for the increase in the amount of generated data. By envisaging innovative real-time applications in next generation vehicles, we realize the great impact that an efficient and effective use of the bandwidth can play for the always-increasing bandwidth demanding applications.

The expectation for the near future is the above described trend to grow, fueled by the recent sales boom in the automotive market that has also revamped research funding on this topic [3]. In spite of such favorable perspectives, there are still many open issues and challenges to be addressed. Nowadays, OBUs are embedded systems with limited resources and hardly modifiable. Software updating of an OBU requires a long and cumbersome procedure to be carried out in a properly equipped workshop. This hardness combined with the typical software lifetime, which is much shorter than the lifetime of mechanical components, may result in a rapid obsolescence of modern vehicles. It is therefore crucial to devise a suitable mechanism to make OBU programming (and re-programming) easier. Also, the limited resources of current OBUs may be the cause of long latency in carrying out time-critical tasks. A clever way to manage resources and allocate them to more critical tasks is necessary. In this context—which is continually evolving—an efficient OBU design has to meet a number of requirements that makes the OBU updating process easier, enable the deployment of new software, and efficiently manage parallel processes with real-time constraints.

In this work, we investigate the potentialities of lightweight virtualization as enabling technology for the fulfillment of the aforementioned requirements. Docker[1] is an open-source container-based virtualization technique, easy to use and deploy, which represents a mandatory feature for applications development in modern smart vehicles. This feature also brings high flexibility, allowing flexible "activation" and "deactivation" of processes with a minimal impact in terms of overhead.

To achieve our objectives we follow a pragmatic approach and address two main points. First, we design a feasible architecture for a virtualized OBU and we illustrate how common vehicular applications can be implemented on it, while outlining the guidelines to extend our approach to a generic implementation. Second, we evaluate the achievable performance to assess if virtualization has caused some detrimental effect and to which extent this natural decay in performance remains within tolerable margins.

The main contributions of this paper can be summarized as follows:

- we propose the use of Docker containers in order to customize a smart car platform [4] and make it compliant with different end user requirements;
- we show how efficiently using containers to instantiate and schedule dedicated applications according to the vehicle status and/or on-demand services;
- by means of an empirical evaluation, we demonstrate that the proposed lightweight virtualization technique developed on top of devices with limited computational capability—such as a Raspberry Pi3—has an

[1]http://www.docker.com/

978-3-901882-89-0 @2017 IFIP

almost negligible adverse impact in terms of performance, also under heavy and heterogeneous workload conditions, and it allows the end user to dynamically add/eliminate running processes according to his/her needs.

To the best of our knowledge, this is the very first contribution that shows the real implementation of a container-based virtualization technique in the context of smart cars.

The rest of the paper is organized as follows. In Section II, we overview the main contributions related to our work and highlight the rationale behind our proposal. In Section III, we describe the architecture of the proposed container-based platform. Section IV describes the performance of our platform. Finally, Section V concludes the paper by also identifying potential extensions of our architecture to other interesting scenarios.

II. Related Work and Motivations

Docker open-source "containerization" is gaining consensus in several research and industrial domains as a lightweight virtualization technique alternative to more traditional Virtual Machine (VM) based approaches. In [5], the Docker performance is shown and interesting recommendations are given on the use of different disk-intensive workloads on container-based clouds. In [6], the authors draw similar conclusions claiming that both VMs and containers are mature technologies and introduce negligible overhead for memory and CPU performances, with Docker generally achieving equal or better performance than Kernel VMs [7]. Analogous conclusions can be found in [8] and [9], where traditional hypervisors and container-based solutions are compared. Since VMs share the hardware resources while containers share the operating system, this makes solutions like Docker easier to use and faster to deploy. These features make Docker a suitable "containerization" solution for the creation of an embedded versatile device able to manage smart car applications.

Container-based virtualization has been recently considered in the vehicular domain. In [10], a container-based approach is considered for self-driving vehicle applications with hard safety and timing requirements. Of course, given the real-time constraints of this application domain, any additional overhead could affect the system performance and introduce more latency, that could result in software malfunctioning. A different perspective has been considered in [11], where the authors present a network simulation method in the vehicular context. Their platform, exploiting both simulated and real network data traffics with the use of lightweight containers, has been tested by considering two Ethernet topology—Double Start and Daisy Chain—in order to conclude that also in the case of high workloads (i.e., 6 in-vehicle cameras broadcasting simultaneously, along with control infotainment and other types of traffic) the delay requirements for safety-critical video information may be violated.

Our approach differs from the previous ones under several aspects. We design and implement an embedded all-in-one Cyber-Physical-Device for smart car applications and contribute with a real prototype based on a Raspberry Pi 3 that is connected with the On Board Diagnostic (OBD) system of a car. We show how we can make it coexisting with other processes that span from infotainment applications to video data delivery (e.g., from a webcam mounted on top of our device), etc. To the best of our knowledge, this is the first real implementation of a Docker container-based prototype for vehicular applications. The interest behind this type of implementation is demonstrated by projects like Carberry[2], where a Raspberry Pi includes CAN bus, Gmlan, IR Led input, etc. The main and important difference of our approach with regard to these projects consists in the exploitation of Docker-based virtualization that, as we will show, gives very high flexibility to the final system that can be easily expanded with novel processes and enriched with new features.

III. System Design and Implementation

In this section, we first introduce the technologies used in the implementation of our prototype. Then, the entire platform architecture will be described in detail. Finally, through the analysis of different examples, we show how the main benefits introduced by container technologies can boost the development of a versatile and flexible platform.

A. Enabling Technologies

Raspberry Pi (RPi) is a single-board computer that leverages the low-power low-cost ARM processor architecture [12] and became popular thanks to its flexible use in several contexts. Raspberry Pi has been already used in in the vehicular context in projects like CarBerry and in OBD-Pi[3]. For the development of our platform, we selected the last generation of the Raspberry Pi, which is the Raspberry Pi 3 (RPi3) model B3. The main hardware characteristics of the board are summarized in Table I.

TABLE I. Raspberry Pi 3 hardware features.

Parameter	Description
Chipset	Broadcom BCM2837
CPU	Quad Core @900MHz ARMv7 Cortex-A7
Memory	1GB LP-DDR2 900 MHz
GPU	Broadcom VideoCore IV
Ethernet	10/100 Mb/s
Flash Storage	MicroSD
Connectivity USB	4 USB 2.0 Host
OS	Linux, Windows 10
Price	$35

Data coming directly from the vehicle can be read through the OBD-II standard [13] interface that provides two types of data: real-time vehicle data and several Diagnostic Trouble Codes (DTCs). The entire subset of such data can be efficiently used in order to monitor the current operating status of a vehicle, and to identify malfunctioning in the vehicle itself.

B. Functional Modules

The architecture of our prototype is depicted in Fig. 1. The platform has been designed to meet specific requirements: *(i)* fast allocation and flexibility in managing different services; *(ii)* isolation; *(iii)* backup capabilities. In the following, each key component is described in detail.

[2]http://www.carberry.it

[3]http://www.instructables.com/id/OBD-Pi

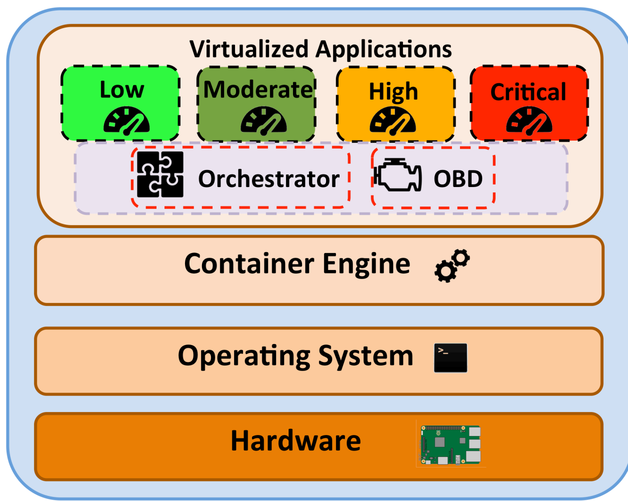

Fig. 1. Platform architecture.

The Hardware is a Raspberry Pi 3 board used to develop our platform. A key aspect that led us to the choice of the Raspberry Pi platform is its capability to efficiently run virtualized applications through the use of container technologies, in particular Docker. Furthermore, the ease with which different applications can be managed through containers well match our platform requirements. Finally, as demonstrated in [14], the introduction of the virtualization layer in devices with low computational resources does not adversely impact the performance.

As base Operating System we use the image provided by Hypriot running Raspbian Jessie with Linux kernel 4.4.10, with Docker version 1.12.0. As storage device, a 16 GB (Transcend Premium 400x Class 10 UHS-I microSDHC) memory card has been used.

At the application level, our architecture entails three main components: *(i)* a set of singularly virtualize applications tagged with different priority levels; *(ii)* an *OBD* Container that is in charge of receiving and handling the data from the vehicle; and *(iii)* the Orchestrator that has the task of monitoring the resources used by the entire system and by each virtualized application. Orchestrator and OBD are in the same functional block since they frequently interact. The central role played by the Orchestrator component will become clearer in the following sections.

We defined four application/service types (Fig. 2):

- **Critical** priority applications are characterized by the highest level of priority (e.g., applications dedicated to firmware update/restore, or demanding control data).
- **High** priority applications, which include e.g., driver assistance, camera data.
- **Moderate** priority applications consist of applications provided by auto insurance companies, which offer reduced premiums if OBD-II vehicle data loggers[4] are installed.
- **Low** priority applications include Entertainment/Multimedia contents streamed by an internal and/or external device.

In defining the priorities, we refer to the application requirements in [1]. Since the Raspberry Pi is a low-resource device with limited computation capabilities, specific allocation policies have to be defined and executed by the Orchestrator in order to give priority to the execution of a virtualized application over another. For each level of priority, an application that provides backup functionality is planned. This implementative choice is done by considering that possible malfunctions recorded by the OBD-II can have different priorities.

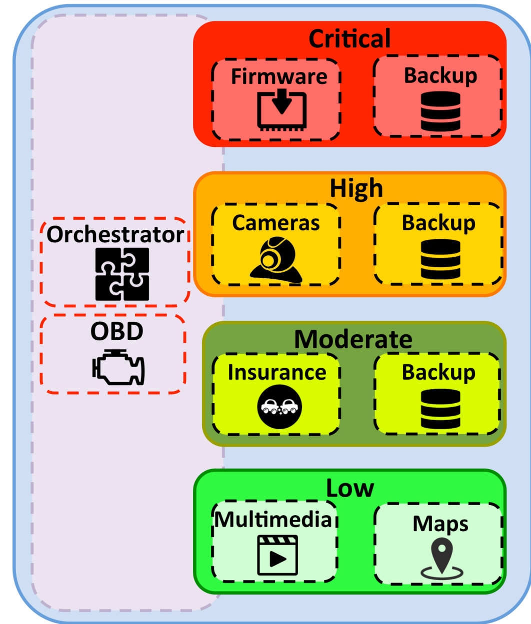

Fig. 2. Functional modules.

One of the key roles played by the orchestrator is to determine how many resources are employed by the running applications, and verify that the whole system is compliant with the priority of all the active instances. Furthermore, by acting as application manager, the orchestrator is responsible for the allocation of new instances.

In order to answer the needs of all hosted instances, the scheduling of new applications on the OBU has to consider three different aspects:

- priority of the instances already running;
- priority of the incoming instances;
- hardware resources already employed;

For estimating the used resources on the physical node, we refer to the *Volume* metric introduced by Wood et al. in [15]. This metric considers that a physical node can be loaded along one or more of three *dimensions*—CPU, network, and memory. The volume expresses how much the system is (over)loaded along multiple dimensions in combined way, and can be used to fairly estimate all resources used by each component. Equation 1 defines this metric; *cpu* stands for the normalised CPU usage, *mem* for memory, and *net* for network.

[4]https://www.progressive.com/auto/snapshot/

The higher the utilization of a resource, the greater the volume. As a consequence, if multiple resources are heavily utilized, the above product results in a correspondingly higher volume.

$$Vol = \frac{1}{1-cpu} \times \frac{1}{1-mem} \times \frac{1}{1-net} \qquad (1)$$

Moreover, the Docker remote API allows us to quantify the instant use of the three dimensions for each running container. Therefore, we are able to quantify the *Volume* generated by each individual container (Eq. 2).

$$Vol_{cont_n} = \frac{1}{1-cpu_{cont_n}} \times \frac{1}{1-mem_{cont_n}} \times \times \frac{1}{1-net_{cont_n}} \qquad (2)$$

The overall *Volume* can be furthermore characterized by Equation. 3. Resources employed by each virtualized application are quantified, with the addition of the *volume* due to the basic processes.

$$Vol = \sum_{i=1}^{n} Vol_{cont_i} + Vol_{base} \qquad (3)$$

Once outlined how to estimate the resources employed by the whole system and by each virtual component, hereafter is described how a new application is allocated in the system.

The orchestrator keeps memory of all the running applications and their associated priority. As soon as the request for a new instance allocation arrives, the orchestrator has to consider the priority associated to the incoming request, together with available resources on the physical node. If the available volume is greater than a certain threshold—which can be set according to the number of applications that are expected to be handled in total by the device—the new application is instantiated on top of the device. Otherwise, the discriminating factor for the instantiation of the new application is given by its own priority. Indeed, if the priority of the incoming request is set to *critical*, the orchestrator acts so as to guarantee the necessary resources for the execution of the application. If there are no available resources and the priority is not critical, the application request is entirely rejected. In the next subsection, the dynamic service allocation is further explained.

C. Application Scenarios

The Orchestrator dynamically allocates applications/services in the platform. This feature brings several benefits in terms of resource usage. Indeed, the container instances can be instantiated only when required or when specific events like a car operating anomaly occur.

For example, we analyze the case of an anomalous gas emission detected by the OBD (corresponding to the OBD code P0442). According to our requirements, the system must store in memory what happens at the time when the anomaly has been detected. The dynamic allocation of a container dedicated to this purpose will provide a detailed snapshot

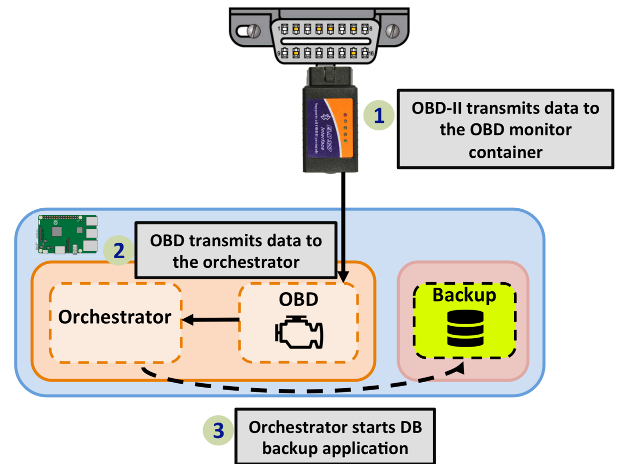

Fig. 3. Data Base Container activation.

of the evolution of this anomaly. In practice, as soon as the orchestrator—which constantly receives data from the OBD—detects the anomaly, it executes the activation of another virtualized application dedicated to the execution of a particular task. Referring to this example, a database is initialized in order to record all the parameters associated to the evaporative (EVAP) emission system (Fig. 3).

In the second example (Fig. 4), we consider the case in which the OBD detects a sudden vehicle speed decrease. This particular situation may correspond to an imminent danger. Also in this scenario, the Orchestrator can activate a container, which in turn starts the video recording of the surroundings through the camera connected to the Raspberry.

The aforementioned examples show how the flexibility related to the use of containers in this context enlarges the potentialities of our platform in terms of backup functionality and capacity of running-dedicated applications at given times. Flexibility can also be used to optimize the performance of the platform itself. Considering that the number of applications that can simultaneously run on top of the RPi3 is limited, the definition of different priorities, together with the dynamic management of containers, is also useful to schedule which application takes priority when the hardware resources are kept busy.

To better explain how containers can help on pursuing this goal, we suppose that two applications characterized by different priority levels, e.g., *Moderate* and *Low*, are simultaneously running. Similarly to the previous examples, the Orchestrator recognizes a possible critical status for the vehicle by means of data received by the OBD. The platform has to guarantee the execution of a specific application that can somehow help on solving and/or tracing the vehicle behavior during the critical status. At the same time, it has to be fast on reacting to this potential dangerous. Considering that the execution of this task has the highest priority, all the concurrent instances that are running on that given time have to be paused to make available all the resources to the application marked by critical priority. This can be easily achieved thanks to the Docker API, which allows every running container to pause (and un-pause). When the containers are in pause mode, no hardware resources are employed.

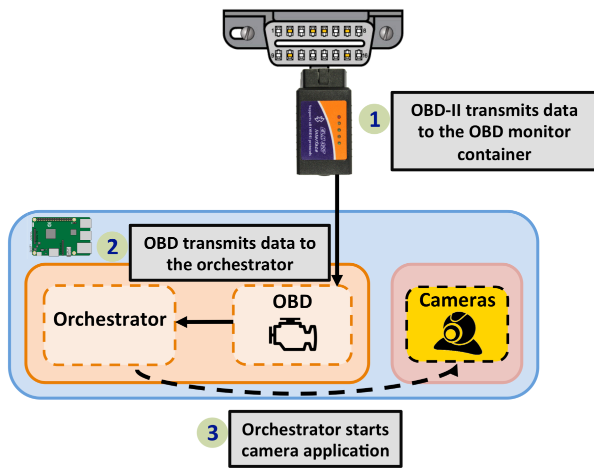

Fig. 4. Dash Cam container activation.

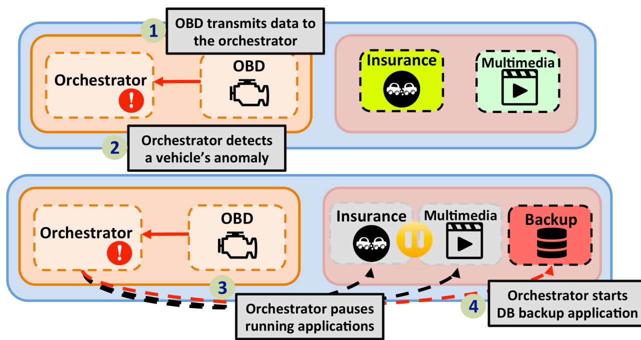

Fig. 5. Anomaly detected case.

IV. Performance Evaluation

The validation of our proposal covers two different aspects. First, we evaluate how a Raspberry Pi reacts, in terms of performances, to specific workloads generated by applications running within Docker containers. This operation lets us to quantify potential overhead introduced by the virtualization layer. Then, we present a first performance evaluation of our platform while performing specific tasks.

A. General Performance

The impact of virtualization of Docker on ARM devices has been already discussed in [14]. However, in this work the performance evaluation is carried out by means of benchmark tools that stress specific hardware segments of the device. Although this is a reasonable approach—which enable to assess an upper-bound to the performance of each system hardware portion—it can not be neglected that real-world applications challenge the hardware in a more distributed way. For this reason, we decide to evaluate the impact of virtualization on Raspberry 3 by means of the stress[5] benchmark tool, which is a workload generator that allows to allocate a configurable amount of CPU, memory, I/O, and disk stress on the system. This analysis also aims to evaluate the behavior of the Raspberry when it is subjected to an increasing workload. For this purpose, we have defined four different workload levels: *(i)* Base load; *(ii)* Low Load; *(iii)* Average Load; *(iv)* High Load. Table II shows more details about the four generated workloads.

TABLE II. Workload Characterization For General Performance Analysis.

Workload	Description
Base	A load average of one is imposed on the system by specifying one CPU-bound processes.
Low	A load average of two is imposed on the system by specifying one CPU-bound processes, and one memory allocator process.
Average	A load average of three is imposed on the system by specifying one CPU-bound processes, one memory allocator process, and one disk-bound process (50MB).
High	A load average of four is imposed on the system by specifying one CPU-bound processes, one memory allocator process, and one disk-bound process (100MB).

The performance metric that we are interested in monitoring is the *system load*, which indicates the overall amount of computational work that a system performs. The average load represents the average system load over a period of time. In our evaluation, such time interval is set to 400 seconds. We decided to use the *system load average* metric, as it includes all the processes or threads waiting on I/O, networking, database, etc. [16]. This process can help us to well characterize the platform performance, by taking into account the heterogeneous features of the applications that run on top of it. The aforementioned metric can be monitored by means of Unix tools like *dstat*[6]. This tool calculates the average load and provides three values for it referring to the past one, five, and fifteen minutes of system operation. In our experiments, we only consider the 1-minute average system load. Each test is started when the RPi3 shows a load number of 0.

To better understand what this metric expresses, we can consider the example of a single-CPU system that shows a 1-minute load average of 1.46. This means that during the last minute, the system was overloaded by 46% on average—1.46 runnable processes, so that 0.46 processes had to wait for a turn for a single CPU system on average. In other words, this also means that a system load average equal to one represents an upper-bound for this metric in a system with one CPU, after which the Operating System could behave unstably. Similarly, a system average load equal to four represents the upper-bound in a system with four CPUs.

In our evaluation, the *native performance*—i.e., running the *stress* tool without including any virtualization layer running on top of the underlying hardware—is used as a reference for comparison. This will be useful to quantify a possible overhead introduced by container technologies on the Raspberry Pi.

Fig. 6 shows the result of the *General Performance* test. Several insights about the performance impact introduced by Docker can be drawn. First, it can be noticed how for three of the four workloads (base, low, and average), native and Docker performance curves basically overlap. This represents

[5]http://manpages.ubuntu.com/manpages/wily/man1/stress.1.html

[6]http://dag.wiee.rs/home-made/dstat/

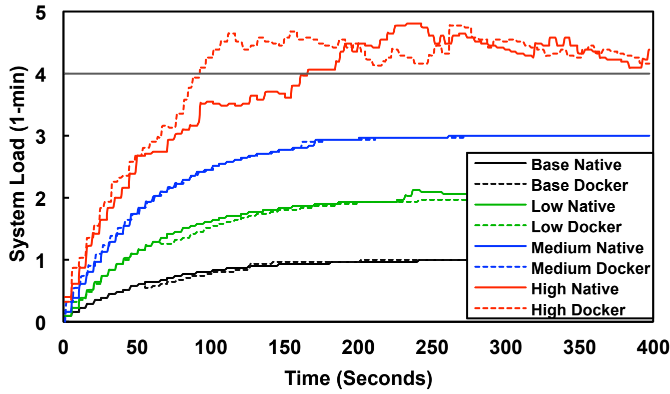

Fig. 6. General performance.

an important outcome since it confirms the lightweight characteristics of container technologies, even when the RPi3 has to handle mixed workload.

The only case in which a tangible overhead between native and Docker performance can be observed is when a *High* workload is assigned to the Raspberry. By analyzing the *High workload* curve, we can notice that Docker reaches earlier the system load upper-bound. The performance difference, between native and Docker performance, can be quantified in the order of the 15%. We can also observe the unstable behavior of the system -both for the native and virtualized cases- when the upper-bound is exceeded. As previously explained, this behavior is attributable to the system overloading. However, it is worthy clarify that the High workload has been defined in such a way to heavily challenge the system. Indeed, when memory and disk-bound processes are imposed to the system, the CPU has extra tasks to be executed.

Furthermore, in the next subsection it can be observed how in correspondence of complex workloads, the system load does not reach the value for which there is a not negligible performance overhead between native and virtualized applications.

B. Platform Performance

After showing the efficiency of the RPi3 in handling mixed workload running within Docker containers, we want to test the performance of our platform when it has to manage dedicated virtualized applications. To this end, we have defined a set of workloads that are closely related to the *application scenarios* described in Section III-C. A detailed explanation of five workloads is provided in Table III. It can be observed how those scenarios are characterized to deliver increasing load to the system. To accomplish this evaluation, we assume that the functional block *OBD+Orchestrator* is receiving data from the OBD-II interface—which is directly connected through serial port at the Raspberry Pi—every 10 ms.

Fig. 7 shows the results for the platform performance evaluation. The main outcome in the platform evaluation lies in the fact that, even when several virtualized instances are simultaneously running, the average system load is lower than the upper bound. The result is the same also for the case of a workload in which there are five containers running different services/applications, by enforcing the deployment feasibility of an efficient platform also in terms of scalability.

TABLE III. WORKLOAD CHARACTERIZATION FOR PLATFORM PERFORMANCE ANALYSIS.

Workload	Description
Workload 1	This workload refers to the simple scenario in which the OBD/Orchestrator container receives data from the OBD-II interface through the CAN bus, and forward the received data to a database in which the OBD logs are stored.
Workload 2	The second workload refers to the case in which the orchestrator has to activate and manage a single virtualized application. The application handles a video content from a camera directly connected to the Raspberry Pi. The video content is encoded in MPEG-4 format, and made available through to HTTP connection.
Workload 3	The third workload is a combination of Workload 1 and Workload 2.
Workload 4	In the fourth case, we add the streaming of a multimedia content to the Workload 3.
Workload 5	The Workload 5 combines the previous case with the activation of another container that interacts with the connected web-cam for recording and storing (on the MicroSD card) the transmitted video content.

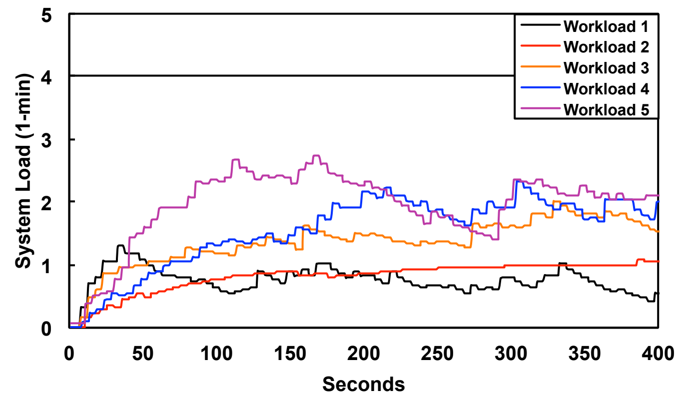

Fig. 7. Platform performance characterization for different workloads.

The only tangible difference between the behaviour introduced with the different workloads is that, when disk operation—such as database writing—are included, the system load average grows introducing a sort of oscillating characteristic. On the contrary, if disk operations are excluded—like in the Workload 2 case—the aforementioned behavior is not observable. Fig. 8 shows the Average and the Maximum values for the System Load. We can observe that the System Load increases with the complexity of the workloads. Taking into account the *average* value, we are interested on quantify how the increasing workload complexity affects the system load. As an example, there is an increase of approximately 13% between Workload 3 and Workload 4. The percentage increase between Workload 4 and Workload 5 is instead 25%.

As explained above, the system load is a metric that takes into account also the *average* of the system in a given period of

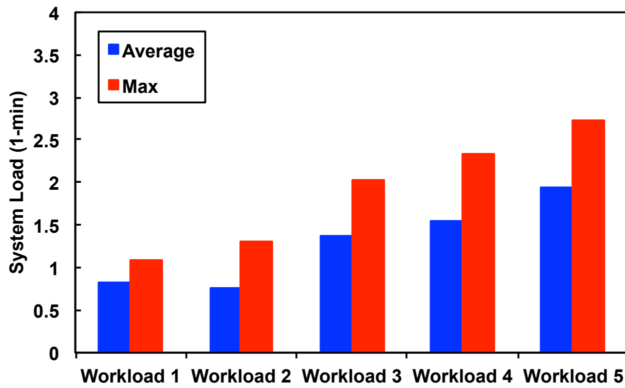

Fig. 8. Average and Max values for System Load evaluated in 1-min.

time. However, in Section III-B, we have described the way in which the orchestrator estimates the available resources. We have seen that the *volume* expresses how much the system is load, by considering the combination of the resources employed by CPU, memory, and networking.

Therefore, in order to have hints about the impact of each single dimension in all the different workloads, we have also estimated CPU and RAM usage. Fig. 9 and Fig. 10 show respectively CPU and Memory RAM usage for the different workloads. Similarly to what observed for the system load, an increase linked with the higher workload complexity can be observed. It is interesting to note also that for very complex workloads, e.g. 4 and 5, memory resources are far from saturation. However, this analysis shows how important is to monitor the contribute of each component of the volume, since that, from the knowledge of these, the orchestrator has a precise overview about how the resources are employed.

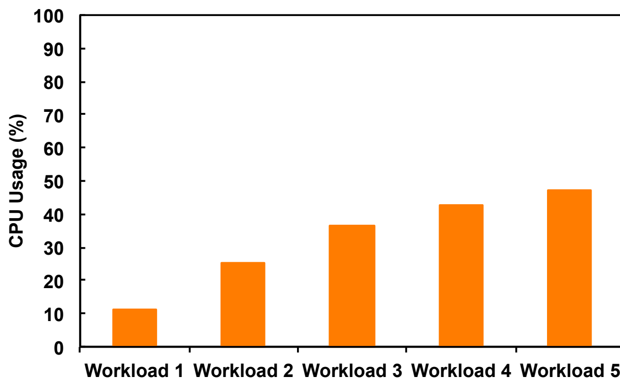

Fig. 9. CPU usage for each Workload.

Fig. 11 shows the overall volume characterization, and how each single component affects in it. For our experiments, networking has not been considered—and its contribute in the volume definition set to 1—as there are no relevant network interactions, such as to generate a non-negligible contribution. It must be kept in mind that volume is defined as the product of its CPU, network and memory load.

Fig. 12 shows the comparison between volume and system load. It can be observed how the two metrics follow the same

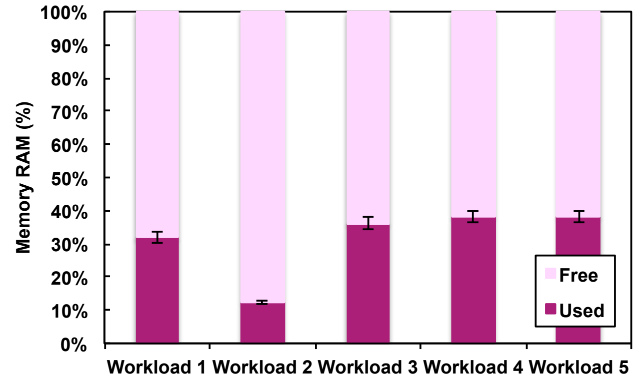

Fig. 10. Memory RAM footprint for each Workload.

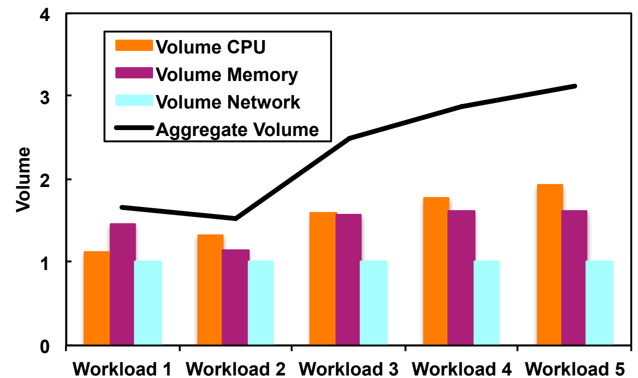

Fig. 11. Volume characterization.

trend, although they grow in different way. Knowledge of both metrics for different workloads can give a more complete description on the platform performance. Indeed, the system load although expresses the performance system without identify the various dimensions, it provides a description about the past state of the system.

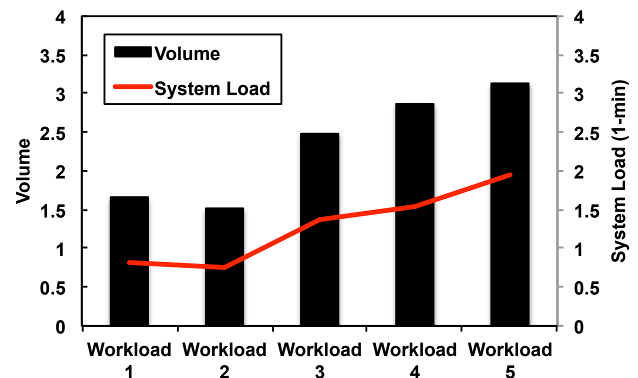

Fig. 12. Volume vs System Load in 1-min comparison.

V. Conclusions and Future Directions

In this paper, we have proposed the design of a platform representing a viable and efficient solution for smart car ap-

plications. In particular, the prototype that we have developed is based on Raspberry Pi3 and implements a container-based virtualization solution to manage different parallel processes, including the treatment of information generated by the CAN-bus. The different processes can be opportunistically scheduled based on specific requirements (e.g., the management of alert processes). Through a proof-of-concept testbed we have demonstrated both the feasibility of a smart car design based on Docker virtualization containers and its effectiveness in terms of responsiveness and reactivity of the system. The derived system is customized and compliant with the final user requirements.

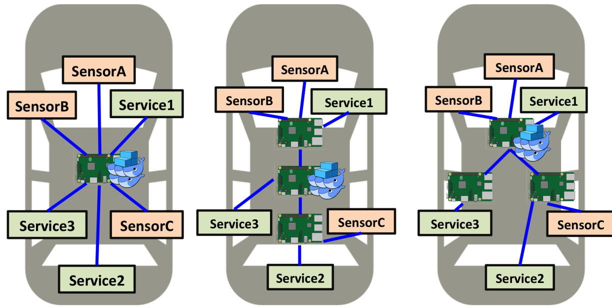

Fig. 13. Future work.

Nowadays vehicles include a growing number of distributed control units, which provide different services within the car. Such services have different requirements and generate extremely different traffic types. Such a distributed approach has already led to the deployment of in-vehicle networks that include several entities dedicated to the management of heterogeneous applications. Our idea for future work is that the flexibility introduced by containers can help in improving also the resource allocation management of in-vehicle networks. For example, the use of container orchestrator engine such as Docker Swarm[7] can be the enabling tool for allocating each container to the most suitable OBU (Fig. 13).

Acknowledgment

This work is partially supported by CPER DATA, the FP7 VITAL project, and by the FP7 Marie Curie METRICS project.

[7] https://www.docker.com/products/docker-swarm

References

[1] H.-T. Lim, L. Völker, and D. Herrscher, "Challenges in a Future IP/Ethernet-based In-car Network for Real-time Applications," in *Proceedings of DAC — 48th Design Automation Conference*, San Diego, California, USA, 2011.

[2] A. M. Vegni and V. Loscri, "A Survey on Vehicular Social Networks," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2397–2419, 2015.

[3] L. Ulrich, "2016's Top Ten Tech Cars," *IEEE Spectrum*.

[4] R. Morabito, R. Petrolo, V. Loscri, and N. Mitton, "Demo: Design of a Virtualized Smart Car Platform," in *Proceedings of EWSN — International Conference on Embedded Wireless Systems and Networks*, Uppsala, Sweden, 2017.

[5] M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos, K. J. Matteussi, and C. A. F. D. Rose, "A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds," in *Proceedings of PDP — 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Turku, Finland, 2015.

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," Philadelphia, Pennsylvania, USA, 2015.

[7] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux Virtual Machine Monitor," in *Proceedings of OLS — the Ottawa Linux Symposium*, Ottawa, Canada, 2007.

[8] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing," in *Proceedings of AIEEE — IEEE 3rd Workshop on Advances Information, Electronic and Electrical Engineering*, Riga, Latvia, 2015.

[9] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *Proceedings of IC2E — IEEE International Conference on Cloud Engineering*, Tempe, Arizona, USA, 2015.

[10] P. Masek, "Container Based Virtualisation for Software Deployment in Self-Driving Vehicles," Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2006.

[11] S. Tuohy, M. Glavin, E. Jones, and C. Hughes, "Hybrid testbed for simulating in-vehicle automotive networks," *Elsevier Simulation Modelling Practice and Theory*, vol. 66, pp. 193–211, 2016.

[12] B. Smith, "ARM and Intel battle over the mobile chip's future," *Computer*, vol. 41, no. 5, pp. 15–18, 2008.

[13] ISO, "15765-2004 Road Vehicles — Diagnostics on Controller Area Networks (CAN)."

[14] R. Morabito, "A performance evaluation of container technologies on Internet of Things devices," in *Proceedings of IEEE Infocom Workshop on Computer Communications*, San Francisco, California, USA, 2016.

[15] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Computer Networks*, vol. 53, no. 17, pp. 2923–2938, 2009.

[16] R. Walker, "Examining load average," *Linux Journal*, vol. 2006, no. 152, p. 5, 2006.