A Middleware to Develop and Test Vehicular Sensor Network Applications

Article

Accepted version

S. Gaglio, G. Lo Re, G. Martorella, D. Peri

In Proceedings of 2019 AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE), Torino, Italy, 2019, pp. 1-6.

A Middleware to Develop and Test Vehicular Sensor Network Applications

Salvatore Gaglio
Department of Engineering
University of Palermo
Palermo, Italy
salvatore.gaglio@unipa.it

Giuseppe Lo Re
Department of Engineering
University of Palermo
Palermo, Italy
giuseppe.lore@unipa.it

Gloria Martorella
Department of Engineering
University of Palermo
Palermo, Italy
gloria.martorella@unipa.it

Daniele Peri
Department of Engineering
University of Palermo
Palermo, Italy
daniele.peri@unipa.it

***Abstract*—The Smart city ecosystem is composed of several networked devices that provide services to citizens and improve their quality of life. Basic services, which must be exposed by the underlying software infrastructure, require efficient networking and communication protocols to coordinate and manage all the system components. In particular, Vehicular Sensor Networks (VSNs) are envisioned as key components of smart cities. Verification is crucial in such a highly dynamic scenario to ensure operation correctness and to reduce the development cost of smart applications. However, the rigidity of existing middlewares makes development, reconfiguration, and testing rather difficult.**

In this work, we propose a middleware that supports development and testing of distributed applications in VSNs. The middleware is based on symbolic processing. Interactive testing as well as incremental development are enabled by the exchange of executable symbolic code among vehicles. The symbolic approach also fosters rapid prototyping and construction of testbeds.

I. Introduction

Providing advanced services to citizens is the main objective of smart cities. To this purpose, enriching vehicles with advanced technologies is nowadays a focus of paramount importance for the automotive industry [1]. Vehicular Sensor Networks (VSNs) enable a great variety of applications including traffic management [2], fire alarm [3], disaster detection [4], air pollution control [5], and urban planning [6]. Vehicles are thus destined to become smart entities cooperating with other vehicles or the physical environment. To achieve this purpose, both vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication schemes require basic functionalities providing information sharing, a data representation and exchange [7].

Embracing all of these domains is not possible at all without an underlying software infrastructure exposing basic services to the application level. As the number of services increase, so do the basic taskse.g. neighbor vehicle discovery and data aggregation are required to work as expected to ensure high quality of service. Vehicle mobility, heterogeneity of devices and sensors involved, and the great number of networked nodes are some of the key issues that make application development and, even more, testing particularly difficult.

978-8-8872-3743-6

A middleware software platform including verification at development and deployment stages is thus highly desirable to construct complex Smart Cities applications [8].

Existing middlewares have been proposed to ease application development. However, testing is not supported at all due to their rigid infrastructures providing scarce interoperability. This also reflects in the lack of opportune testbeds to experiment smart cities solutions [9]. Although simulators represent a low-cost tool [10] they eschew testing on real hardware. Moreover, simulators cannot reproduce the high dynamics of the physical environment at all. As a consequence, malfunctioning and bugs can be only identified at a late stage, after development and deployment.

For these reasons, we introduce a middleware supporting interactive testing. The proposed platform adopts a symbolic programming paradigm to ensure interoperability among several vehicles and sensors possibly from different vendors. High-level symbols, which are directly executable in our system, represent common knowledge, rules, data and code in a single way on heterogeneous nodes. The middleware also permits vehicles to exchange symbols that are executed on receipt. Such a feature enables incremental programming and interactive testing as the code for a certain task is delivered to devices and tested simultaneously.

II. Related Work

Middleware requirements, which are demanded by high-level applications, have been considered by several research works. In particular, a reference middleware architecture has to support dynamic behavior and on-line verification [1]. In fact, middleware platforms have to incorporate a fast verifier module to ensure that protocols and services work as expected, thus ensuring resource-saving and cost-saving operations [11]. Service-oriented approaches were proposed in the literature for networks of vehicles. This is the case of middleware solutions especially devoted to service discovery [12]. The middleware is based on the OSGi (VsdOSGi) technology. Testing is supported through simulation. The OneM2M IoT standardization proposals include devices cooperation through semantic technologies and a basic test environment to check device interoperability and compliance to the proposed paradigm [13]. Recent works proposed the combination of service-oriented

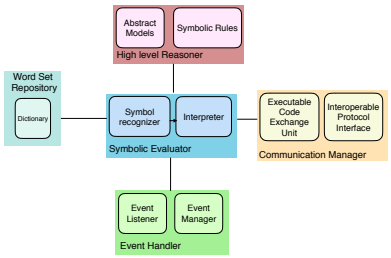

Fig. 1. Middleware structure overview and interaction among components

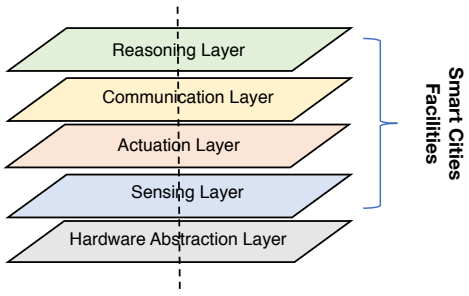

Fig. 2. Layered architecture of the middleware API word set

architectures with Cloud-based technologies [1]. Application development requires resorting to web services and wrapper implementations [14]. VSN operation does not rely on in-network processing but is delegated to the Cloude.g. services, data mining, simulation, data filtering, and fusion [15]. Although, this could permit lightweight implementations on devices, a hard decoupling exists between software and the underlying platforms in vehicles. Although in some cases this is a desirable feature, performing verification on real hardware during development is quite arduous. Rather, validation tests are carried on in simulation. Some other works present middleware solutions for Smart Cities environments.A proposed solution consists in software objects as a high-level abstraction for service development [16]. The middleware structure includes reasoning facilities, service interface, and publish-subscribe communication patterns. Different languages, which are rigidly integrated, are adopted to describe interfaces and running applications. However, testing is not mentioned at all.

Real testbeds exacerbate the demand of verification and testing facilities for moving vehicles. For instance, in the automotive context, the use of ontologies in middleware has been proposed to support multi-vendor and cross-industry interoperability among platforms, either sensors or vehicles [17]. The abstract semantics of ontologies need to be transformed into a model API. However, this process is transparent to developers, who cannot access underlying layers impacting device and protocol verification. Another real testbed in South Corea [18] is based on a middleware platform that provides isolated programming of VSN applications, while service addition require ad-hoc customization by vendors. Several architectural strategies have been thus proposed to expose homogeneous interfaces and verification tools, including an operation management layer which monitors system components and devices through periodic real-time analysis. However, this module is fixed and any detail concerning its real implementation was not reported by the authors.

III. Middleware Key Features

In this section, design principles of the proposed middleware platform are described.

- **Symbolic processing** is the adopted programming abstraction, which is rooted in the concept of *symbol*. A symbol can be a meaningful word, e.g. taken from common speaking, or even a word without a semantic correspondence. Due to this tie, from this point on the terms word and symbol are used interchangeably. Each word is associated to a particular execution task. As a consequence, running a word implies performing the task it is mapped to. Word executions can affect the hardware internal state of devices. Smart application code is presented as a succession of words that are processed one after another in a typical concatenative way. For instance, acquiring and storing a sample of carbon monoxide (CO) can be codified as follows:

```
CO sample store
```

With some more detail, a possible interpretation of the code above sees the execution of the word `CO` drive the hardware to read the sensor value, while the word `sample` defines the memory location the sample has to be stored to. Finally, the task associated to the word `store` stores the sensed carbon monoxide reading to the specified memory address. Other concatenative implementations are though possible for each symbol that lead to the same effect for the stated code phrase.

- **Incremental development** avoids cross-compilation and rebooting of the entire application when software changes are needed. The middleware API is composed of words, which are already implemented and stored in a dictionary. The system word set is not fixed and new symbols are easily added. While built-in words generally bind to assembly code, user-defined ones are implemented in terms of words known to the system, i.e. already found in the system word set. For instance, provided that the dictionary includes the words used above, the new word `CO!` that samples and stores the CO value is added as follows:

```
: CO! CO sample store ;
```

The word `:` enters the word definition while the word `;` ends it. What is enclosed between these two words is the word name followed by the succession of words that compose its execution task. In this case, the word `CO!` runs the sequence of words performing the sensing and storage of a CO sample. This definition ends up in a real compilation of the new code that is effective on resource-constrained devices [19]. As new words are built above the others, incremental development of applications

is thus a viable option.

- **Support to distributed computing and on-the-fly (re)programming** is a powerful mechanism that is pivotal in the implementation of our middleware. The simple but effective mechanism we devised is the exchange of executable code among networked entities [19]. An entity can send symbolic code to another one or broadcast it to the network through the following construct:

  `tell:` <*List of word to be sent*> `:tell`

  which has to be preceded by the destination node address. Words are sent as textual strings in the payload of IEEE 802.15.4 MAC frames. The code exchange is entirely based on high-level symbols and is performed without any encoding or compression techniques, as well as without any further translation steps. Nested usage of this symbolic construct allows to deliver symbolic code from a source to a destination node. Intermediate nodes are just forwarders to other nodes to reach destination. If an entity wants a neighbor to read and store another CO sample, it executes:

  ```
  tell: CO! :tell
  ```

  or equally:

  ```
  tell: CO sample store :tell
  ```

  The injected code can also modify the device word set and application, e.g. for system recovery or update. Configuration options and parameters, such as pollution thresholds or service priority, can be sent on-the-fly either to sensors, fixed stations, or vehicles.
- **Low footprint** is a key design principle to make software platform run on heterogeneous hardware platforms. The middleware stack we implemented is particularly lightweight. This allows the proposed middleware to be installed even on resource-constrained devices, e.g. Wireless Sensor Network nodes.
- **Support to interoperability** is high desirable as networked vehicles and sensors are certainly integrated within IoT applications through a set of service protocols that enable interaction schemes between different technologies and components. The proposed middleware provides symbolic TCP and MQTT implementations as basic protocols for interoperability among heterogeneous devices [20].
- **High-level reasoning** permits to implement smart behaviors that goes beyond simple data acquisition. To incorporate new abstractions for intelligent applications, our middleware includes a Fuzzy Logic extension [21]. Symbolic rules can also be exchanged among nodes.
- **High-level knowledge representation** is fundamental to inject common sense to the platform. While ontologies can be particularly large and need to be aligned on different devices, abstract models often require several translation steps to obtain the source code. Instead, in our middleware, semantics is natively supported as high-level domain concepts, models, and even informal specifications can be effectively mapped to executable code [22].

IV. Middleware Architecture

The main components of the system and its architecture derive from the design principles described previously. Considering the middleware structure, the core element is the *Symbolic Evaluator*. The evaluator processes a sequence of symbols looking for each of them in the dictionary and executing the respective definition, all this interactively. This environment is implemented as a text interpreter running on the bare hardware. The Evaluator uses a stack for parameter passing among words and consequently expressions are usually evaluated according to the postfix notation. The *Word Set Repository* is a memory area implemented as a linked list that holds the word dictionary. An *Event Handler* listens and manages hardware and data events through a sequence of high-level symbols. The *Communication Manager* implements the executable code exchange mechanism through the device communication interface–e.g. IEEE 802.15.4, Bluetooth, WiFi– with the chosen communication protocol. Finally, a *High-level Reasoner* handles on-board smart behaviors based on reasoning and possibly learning. The development of VSN protocols and services usually involves all the components. An overview of the middleware architecture is provided in Figure 1. The middleware API exposes services that range from hardware abstraction, sensing, and actuation to networking and high-level reasoning tasks. Due to the incremental development feature, the middleware symbolic API presents a layered structure, as depicted in Figure 2. The system infrastructure can be implemented either on resource-constrained devices, e.g. Wireless Sensor Network nodes, in less than 20 KB [19] and on Wi-fi enabled objects, such as ESP8266 chips, in less than 40 KB [20].

V. Middleware Support to VSN Testing

Verification of large-scale VSNs at deployment time is a challenging experience and testing using simulations cannot be exhaustive. However, in a real deployment, testing may imply collecting hardware state information, environmental changes, and, possibly, faults affecting both hardware and software.

The proposed system natively supports interactive testing. Due to its symbolic nature, two testing modes are supported: local and remote testing. Local testing operation is carried on during development and execution of a word. Indeed, the inclusion of a new symbol in the system word set allows to execute and test it simultaneously. If the symbol works as expected, no change is made to its definition. Otherwise, it is possible to redefine an existing symbol overwriting the task to which the symbol is bound. Recovery point words are also covered, therefore the middleware API can be reported to the previous configuration without rebooting the entire system. Most frequently, the execution of a single word is not exhaustive, as the result of verification cannot be evident. For instance, this is the case of hardware state changes or distributed protocols. In these situations, the runtime verification strategy is to make symbolic verification code follow the operational code, so that test code can explicitly expose the result of the verification process, as desired by the

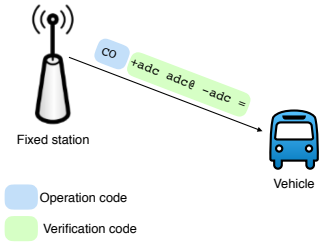

Fig. 3. Scenario I: a fixed station injects operational and verification code for the carbon monoxide sample acquisition task

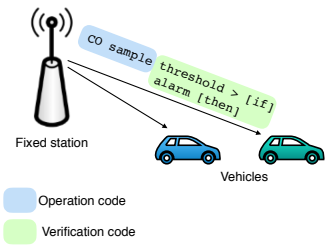

Fig. 4. Scenario I: a fixed station injects the operational and verification code for acquiring a carbon monoxide sample. Testing consists in sending back an alarm when the sensed value exceeds a threshold

developers. The middleware also provides remote testing to verify deployed VSN devices. Indeed, the approach described above naturally extends to real scenarios by sending both operational and symbolic verification code to VSN devices through executable code exchange. Such a testing tool is available without adding further abstraction layers or components to the middleware system. Runtime on-board verification is performed at hardware, API, and application levels, equally. However, the proposed approach does not prevent simulation from being carried out when desired. For instance, the task associated to the symbol `CO` can be such that either (i) enables the opportune sensor device and performing real sample acquisition or (ii) computes a synthetic test sample according to a given trend function. Different verification code must then be crafted for the two cases. In the former the hardware state changes must be compared to the expected ones. In the latter the computed value must be checked for correctness with respect to the values synthesized by the given trend function. In the following we describe how local and remote testing is applicable to different VSN scenarios.

A. Scenario I: Testing on-board vehicle sensors

The first scenario considers local testing of sensing equipment on-board each vehicle. In this use case scenario we consider testing the acquisition of a carbon monoxide sample. Let us suppose that the carbon monoxide sensor is connected through the ADC interface and that the word performing the sensor reading is `CO`. Testing that the word `CO` works properly requires executing the symbols: (i) `+adc` to enable the ADC, (ii) `adc@` to read the ADC register value, and (iii) `-adc` to disable the ADC. Finally, the comparison to the value stored in the ADC register to those acquired is done by executing `CO` as follows:

```
CO +adc adc@ -adc =
```

The result of verification is thus exposed to the programmer on the real device in terms of a boolean value computed by the comparison operator (=). Test code for checking on-board sensor operation can also be sent by fixed nodes that connect vehicles to the VSN infrastructure, as shown in the Figure 3.

B. Scenario II: Testing Reactive behaviors

A monitoring station can send symbolic verification code to mobile entities. A possible testing code can be the sequence of symbols that make a device send back an alarm when the level of a certain pollutant overcomes a specified threshold. As an example, a station tells a vehicle to acquire a sample of carbon monoxide. The testing code can process the acquired value for instance to trigger the alarm if the CO sample exceeds a predefined threshold as depicted in Figure 4. A simple rule implementing this behavior would be:

```
CO sample threshold > [if] alarm [then]
```

Therefore, reactive rules can be easily implemented and tested.

C. Scenario III: Collaborative testing of on-board sensors

Another possible scenario includes vehicles stopped at traffic lights (see Figure 5). Due to the fact that vehicles are spatially close to each other, it is plausible that they can measure the same values up to a small tolerance. Therefore, the symbolic code for testing on-board sensors can request neighbor vehicles to provide their sensor value. Then the requesting node compares them to the on-board sensor reading. For instance, the following protocol code:

```
bcst tell:
random ms
CO
reply tell:
~ CO - tolerance < if
    correct
else
    wrong
then
1 + !
:tell :tell
```

hlbroadcasts the code to wait a `random` millisecond (`ms`) time, make a receiving node sample `CO`, `reply` and `tell` the reading as well as the code to perform the measurement on the requesting node (Figure 6). Then the difference is computed and compared to a `tolerance` value. A few counters are updated to consider statistics eventually, i.e. `correct` and `wrong` results. The symbol is a placeholder that is replaced by the computed CO value at runtime.

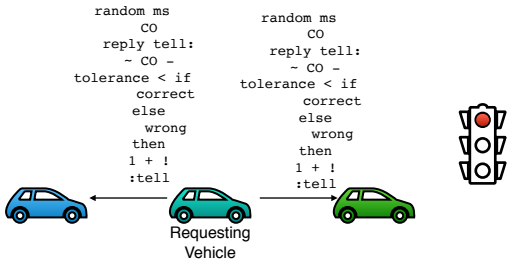

Fig. 5. Scenario III: executable code broadcast by a requesting vehicle to check the CO sensor

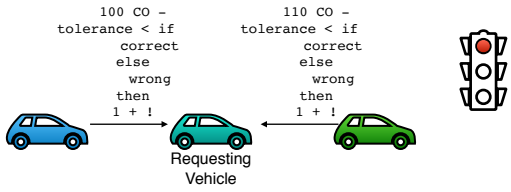

Fig. 6. Scenario III: example of executable code sent by neighbor vehicles to a requesting node to check its CO sensor

D. Scenario IV: Average protocol verification

Similarly to the previous scenario, neighbor entities can perform collaborative tasks, such as aggregation and data fusion. In this case, a fixed station, which can be placed close to the traffic lights, broadcasts the aggregation protocol code when the traffic light is red. For instance, consider a protocol for aggregation of temperature data [23]. Each node waits a time proportional to its ID–e.g. vehicle plate–performs sensing, updates the current aggregate value and number of nodes and exchange them with its neighbors along with the symbolic code for their update. At the end of the protocol execution, each node should have the same aggregate and number of node, that is, the same average value. As reported in our previous work [23] and in Figure 7, the code to start the protocol execution is:

```
bcst tell: 0 0 update :tell
```

where the first value is the current temperature aggregate–a simple sum–, while the second is the current count of nodes. The fixed station is itself an active node that contributes to the calculation. Therefore, at the end of protocol execution, it holds the two values needed to compute the average, the final sum and count of nodes. The final step of protocol code exchange is reported in Figure 8

A possible verification thus implies that the fixed station waits a given time and broadcast again the code to make each node signal an error if its own reading deviates from the shared average value, as reported in Figure 9. This way, each vehicle performs on-board verification of the protocol.

VI. Discussion

A review of the literature highlighted that existing middleware solutions do not provide adequate support for on-board testing in real VSN scenarios. Instead, code is tested using

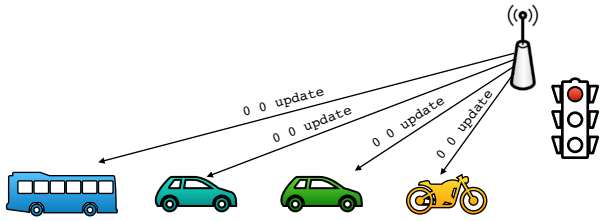

Fig. 7. Scenario IV: executable code sent by a fixed station to start distributed temperature aggregation protocol

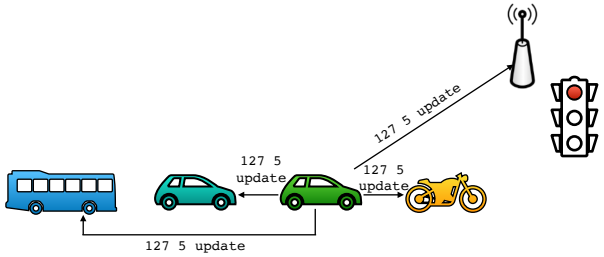

Fig. 8. Scenario IV: Final exchange of executable code sent by the last vehicle to compute temperature average

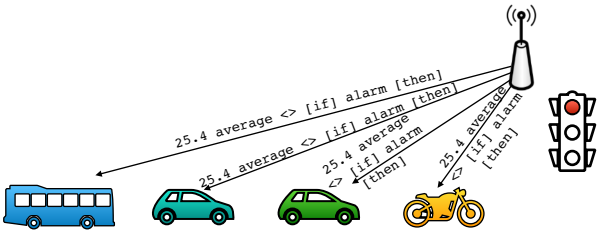

Fig. 9. Scenario IV: Symbolic verification code sent by the fixed station. Code is delivered at the end of protocol execution and consists in the sequence of symbols to report an alarm if the average differs from the value computed by the fixed station.

simulation, sometimes rather complicated, and then released. If a successful test done in simulation does not succeed also in the real scenario, the whole development process must be repeated. Moreover, realistic simulation would require a broad range of scenarios to be considered. The proposed middleware environment, which is based on interactive development, avoids cross-compilation and proves time-saving since code development and test collapse into a single phase. The strict coupling of the middleware to the hardware allows testing to be performed at all levels, from low-level hardware-bound software layers to application code, seamlessly. We showed how verification of both hardware driving and protocol symbolic code can be written and tested one definition at a time on the target hardware. On-line verification of distributed protocols can be done through executable code exchange, a feature that the other systems lack, in the absence of thick software layers, which often require code translation stages. For instance, the behavior of the alarm application described in previous section can be changed dynamically on VSN entities simply by sending them a new symbolic rule and test code. Our

results support the idea that interactive testing is particularly advantageous for creating VSN testbeds rapidly, as code can be tested on the real hardware and environment. However, the proposed middleware does not exclude the possibility of running simulations. Indeed, the same high level test code can run on simulated hardware. The same considerations apply to the Cloud, on which our symbolic middleware could run and periodically send symbolic test code to both vehicles and fixed stations. Finally, the flexibility of the middleware we developed can be considered inconvenient when compared to architectures offering ready-to-use functionalities tough difficult to modify. On the other hand, easy extension and upgrades of the middleware itself provides a generic solution which does not target any specific application while allowing software reuse.

VII. Conclusions

In this paper we presented a symbolic middleware providing entities in Vehicular Sensor Networks with advanced skills including on-board verification of code. On-board verification is a giant challenge in such a dynamic context and previously proposed middlewares only partially support testing VSN applications. The main feature of the proposed approach is the possibility exposed by the middleware to undertake verification of both hardware driving and protocol code on real scenarios and platforms. Application development and testing take place in terms of exchanged symbols among entities, e.g. vehicles, fixed stations, and other networked devices. Symbolic test programs are executed once received. An interpreter-based environment running on VSN entities makes the middleware able to be incrementally extended by defining new words based on previously defined words. The middleware infrastructure is not rigid and new implementations of protocols and communication functionalities can be easily added. Furthermore, its symbolic nature makes the middleware able to run on networks composed of heterogeneous devices.

References

[1] A. Boukerche and R. E. D. Grande, "Vehicular cloud computing: Architectures, applications, and mobility," *Computer Networks*, vol. 135, pp. 171 – 189, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128618300057

[2] Z. Ning, J. Huang, and X. Wang, "Vehicular fog computing: Enabling real-time traffic management for smart cities," *IEEE Wireless Communications*, vol. 26, no. 1, pp. 87–93, February 2019.

[3] R. Sowah, K. O. Ampadu, A. Ofoli, K. Koumadi, G. A. Mills, and J. Nortey, "Design and implementation of a fire detection and control system for automobiles using fuzzy logic," in *2016 IEEE Industry Applications Society Annual Meeting*, Oct 2016, pp. 1–8.

[4] P. Li, T. Miyazaki, K. Wang, S. Guo, and W. Zhuang, "Vehicle-assist resilient information and network system for disaster management," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 3, pp. 438–448, July 2017.

[5] G. Lo Re, D. Peri, and S. D. Vassallo, *Urban Air Quality Monitoring Using Vehicular Sensor Networks*. Cham: Springer International Publishing, 2014, pp. 311–323.

[6] M. M. Rathore, A. Ahmad, A. Paul, and S. Rho, "Urban planning and building smart cities based on the internet of things using big data analytics," *Computer Networks*, vol. 101, pp. 63 – 80, 2016, industrial Technologies and Applications for the Internet of Things. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128616000086

[7] C. T. Barba, M. A. Mateos, P. R. Soto, A. M. Mezher, and M. A. Igartua, "Smart city for vanets using warning messages, traffic statistics and intelligent traffic lights," in *2012 IEEE Intelligent Vehicles Symposium*, June 2012, pp. 902–907.

[8] E. F. Z. Santana, A. P. Chaves, M. A. Gerosa, F. Kon, and D. S. Milojicic, "software platforms for smart cities: Concepts, requirements, challenges, and a unified reference architecture."

[9] A. Elmangoush, H. Coskun, S. Wahle, and T. Magedanz, "Design aspects for a reference m2m communication platform for smart cities," in *2013 9th International Conference on Innovations in Information Technology (IIT)*, March 2013, pp. 204–209.

[10] D. Macedo Bastista, D. S Milojicic, E. F. Zambom Santana, and F. Kon, "SCSimulator: An OpenSource, Scalable Smart City Simulator," in *Tools Session of the Brazilian Symposium on Computer Networks*, 06 2016.

[11] M. Garcia-Valls and R. Baldoni, "Adaptive middleware design for cps: Considerations on the os, resource managers, and the network run-time," 12 2015.

[12] J. Luo, T. Zhong, and X. Jin, "Service discovery middleware based on qos in vanet," in *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, Aug 2016, pp. 2075–2080.

[13] H. Park, H. Kim, H. Joo, and J. Song, "Recent advancements in the Internet-of-Things related standards: A oneM2M perspective," *ICT Express*, vol. 2, no. 3, pp. 126 – 129, 2016, special Issue on ICT Convergence in the Internet of Things (IoT). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405959516300911

[14] N. Mohamed, J. Al-Jaroodi, S. Lazarova-Molnar, I. Jawhar, and S. Mahmoud, "A service-oriented middleware for cloud of things and fog computing supporting smart city applications," in *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, Aug 2017, pp. 1–7.

[15] M. Saini, K. M. Alam, H. Guo, A. Alelaiwi, and A. El Saddik, "Incloud: a cloud-based middleware for vehicular infotainment systems," *Multimedia Tools and Applications*, vol. 76, no. 9, pp. 11 621–11 649, May 2017. [Online]. Available: https://doi.org/10.1007/s11042-015-3158-4

[16] F. J. Villanueva, M. J. Santofimia, D. Villa, J. Barba, and J. C. López, "Civitas: The smart city middleware, from sensors to big data," in *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, July 2013, pp. 445–450.

[17] J. F. Gomez-Pimpollo and R. Otaolea, "Smart objects for intelligent applications - adk," in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, Sep. 2010, pp. 267–268.

[18] J. Lee, S. Baik, and C. Choonhwa Lee, "Building an Integrated Service Management Platform for Ubiquitous Cities," *Computer*, vol. 44, no. 6, pp. 56–63, June 2011.

[19] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "DC4CD: A Platform for Distributed Computing on Constrained Devices," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 1, pp. 27:1–27:25, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3105923

[20] S. Gaglio, L. Giuliana, G. Lo Re, G. Martorella, A. Montalto, and D. Peri, "Interoperable Real-Time Symbolic Programming for Smart Environments," in *Accepted at 2019 IEEE International Conference on Smart Computing, SMARTCOMP 2019, Washington DC, USA, June 12-14, 2019*, 2019.

[21] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "High-level programming and symbolic reasoning on iot resource constrained devices," in *Internet of Things. User-Centric IoT*, R. Giaffreda, R.-L. Vieriu, E. Pasher, G. Bendersky, A. J. Jara, J. J. Rodrigues, E. Dekel, and B. Mandler, Eds. Cham: Springer International Publishing, 2015, pp. 58–63.

[22] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "WSN Design and Verification Using On-Board Executable Specifications," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 710–718, Feb 2019.

[23] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "A lightweight network discovery algorithm for resource-constrained iot devices," in *2019 International Conference on Computing, Networking and Communications (ICNC)*, Feb 2019, pp. 355–359.