Prototyping Low-Cost and Flexible Vehicle Diagnostic Systems

Marisol García-Valls
Universidad Carlos III de Madrid, Av. de la universidad s/n, Leganés, 28911
mvalls@it.uc3m.es

KEYWORD

Distribution middleware; Diagnostic systems; Software design

ABSTRACT

Diagnostic systems are software and hardware-based equipment that interoperate with an external monitored system. Traditionally, they have been expensive equipment running test algorithms to monitor physical properties of, e.g., vehicles, or civil infrastructure equipment, among others. As computer hardware is increasingly powerful (whereas its cost and size is decreasing) and communication software becomes easier to program and more run-time efficient, new scenarios are enabled that yield to lower cost monitoring solutions. This paper presents a low cost approach towards the development of a diagnostic system relying on a modular component-based approach and running on a resource limited embedded computer. Results on a prototype implementation are shown that validate the presented design, its flexibility, performance, and communication latency.

1. Introduction

Most systems that sorround us are truly complex systems, needing to elaborate solutions that not only provide the right answer and service to users, but also identify faults and anomalous operation. A clear example are *diagnosic systems* (e.g. vehicle diagnosis) that can be part of a cyber-physical system in the sense that they may interact with physical processes that sense and monitor in order to detect faulty operation. These systems typically provide comprehensive functionality, a self-explanatory operating concept, and the capacity of handling increasing data volumes that must fuse to extract meaningful results to users. On the one side, the software logic and programmed data fusion and analysis techniques are among the most important parts of such systems. As the amount of monitored and sampled data may yield huge volumes, these can be analysed in a server or in a cloud infrastructure. In time sensitive domains, the threats to the predictable cloud computing technologies have to be carefully considered as explained in (García-Valls et al., 2014a) to guarantee operation time bounds. On the other side, the hardware equipment that supports the execution is of paramount importance; it has to integrate the suitable communication interface to the monitored object (e.g., vehicle), the needed computation power to provide timely execution of the diagnosis functions, and suitable verification operations (García-Valls et al., 2014b).

From the early days of life of such systems where the cost of the hardware part was typically high, evolution towards more affordable systems has been favored by the fast progress of the computers technology. Nowadays, the market offers unexpensive solutions based on embedded processors such as RaspberriPi (Halfacree and Upton, 2012), ARM processors (ARM, 2016), Arduino systems (Arduino, 2014; Margolis, 2011), or Java bytecode processors (aJile Systems, 2016), among others. These range from a few dolars to a few hundred dolars; therefore, the actual cost of the hardware equipment is not a barrier to provide sophisticated and powerful diagnostic systems over plaftorms with a number of processing cores, capable of supporting parallel processing.

This paper presents a low cost alternative to designing a diagnosis system. A modular software design based on portable code technology that may execute a number of state machines with the diagnostic functions. The

©Ediciones Universidad de Salamanca / cc by-nc-nd


design is prototyped on a low cost hardware processor, to show the feasibility of the approach. Open source distribution middleware is used to provide the software bus, facilitating high interoperability and data exchange. This paper provides an improved contribution from that presented in (García-Valls, 2016) by providing a more detailed context and background, detailing the requirements of the system, and presenting the structure of the validation implementation.

The rest of the paper is structured as follows. Section 2 describes the related work. based on the technological contributions that support the software development of these platforms: middleware designs, operating systems enhanced with resource management logic, and current trends towards cyber-physical solutions. Section 3 presents the approach provided to the engineering of a diagnosis system; the modular design is presented, the state machine integration is provided, and the interoperability component is described. Section 4 validates the design with a prototype implementation. Section 5 concludes the work and draws future work.

2. Background

A number of diagnostic systems have appeared over the last decades that integrate intensive software usage in the monitoring and detection of operational faults (software and hardware) in all types of systems. The most related to the present contribution are those related to vehicle diagnosics, where a number of contributions have appeared mainly as patents.One example is a system for remote machine control that utilizes embedded or on-board computers or microprocessors for controlling various aspects of the machine's performance and activity (Brunemann et al., 2002); this system even supported the modification of the operation data and functions of the embedded computer by remote communications access. Another example is a system for performing diagnostics via a wireless link (Parrillo, 1995), that focuses on the hardware design providing a transceiver and additional memory that are connected to the microprocessor in a vehicle so that parts of the operating data is stored in memory and it is periodically and transmitted to a remote station. The data is later transmitted to the remote station for analysis, diagnostics, and, for minor repairs, a fix is transmitted back to the vehicle. An additional example is provided by (Lowrey et al., 2003). that is similar to the previous ones but it focuses on the Internet enabled connection for remote transmission of data and analysis. These, and other related inventions and works, do not expose cleanly the software design of the system. Despite the fact that some of them are intensive software systems, the necessary software platform and architecture is mostly hidden in these works. Fundamental elements such as the communication middleware architecture and its relation to the actual software pattern of the communication units is not well elaborated in the patent descriptions.

Distribution middleware is at the heart of enabling the interaction between the two main parts of a diagnostic system: the monitored object (vehicle) and the diagnostic computer. For further processing of the monitored data, the diagnostic computer can also be connected to other servers in the cloud, also through distribution middleware. In the last few decades, a number of middleware technologies have appeared for supporting remote operation and easing interoperability. Examples are traditional component based technologies such as Corba (OMG, 2012), its light-weight evolution Ice (Internet Communication Engine) (ZeroC, 2003); object oriented middleware such as RMI (Remote Method Invoction) (Sun, 2016) that is a language dependant solution but platform independant solution, its service enhancement named River (Apache, 2013), and other message based technologies such as JMS (Java Messaging Service), AMQP (Advanced Message Queuing Protocol) (IITF, 2014). In the last decade, publish-subscribe data centric middleware such as DDS (Data Distribution Systems for Real-Time applications) (OMG, 2015) have become de-facto standards for system interoperability in a number of domains from web applications, industrial automation, or remote real-time video surveillance (García-Valls et al., 2010).

Enhancements to these technologies have improved their benefits for specific contexts, such as to support dinamic execution (Romero and García-Valls, 2014); for real-time reconfiguration of service-oriented architectures the iLAND middleware (García-Valls et al., 2013) provides time-bounded operation. Improved resource

©Ediciones Universidad de Salamanca / cc by-nc-nd


management to the operating systems has also enabled the higher efficiency in these solutions. Recently, the *Oma-Cy* architecture (García-Valls and Baldoni, 2015) has provided a reference design to implement middleware for cyber-physical systems such as (García-Valls et al., 2017), where the on-line verification part is of paramount importance to support dynamic behavior.

To improve communication predictability, the current processor architectures for diagnostic computer can incorporate a real-time operating system that offers bounded-time primitives. This is the case of RaspberriPi (Halfacree and Upton, 2012) and FreeRTOS (Barry, 2010) or even any more powerful server architectures with real-time Linux. On these platforms, it is possible to set a software stack with a predictable schedulability model to achieve time bounded communication in both a bare machine (García-Valls, 2016) or in a virtualized setting (García-Valls and Basanta-Val, 2017).

In summary, approaches to diagnostic systems have not sufficiently exposed the software design part, nor have they been designed to favor the portability and flexibility diagnostic computer and its counter part in the monitored system. This paper contributes to filling this gap by providing a simple but clean software design that supports the development of diagnotic systems focusing at the components that act as bridges to different underlying middleware solutions. Previous contributions on middleware bridges were presented in (Rodríguez-López and García-Valls, 2011) for DDS based communications in remote surveillance systems, and (García-Valls and Ibáñez-Vázquez, 2012) for distributed Ada applications in critical domains. However, these earlier approaches focuse strictly on the bridging of a middleware technology towards other generic middleware implementations.

3. System design

This section presents the design of the diagnostic system, starting with the requirements to fulfill. Later, the system components are presented and, eventually, the actual software design is described. The system is governed by an operational logic that runs a state machine that controls the interface with the user and guides the execution of the diagnostic algorithms. The state machine is realized with a component based design; it is presented as class, component and deployment diagrams. Lastly, the interoperability elements are described that enable the communication with the user.

3.1 Requirements

The system comprises two main ends that are the *diagnostic system* (that is at the part of the system that is at the side of the operator that starts the diagnostic process) and the *monitored system* or *vehicle software* part (that contains an interface to the specific hardware elements to be checked). Both parts of the sytsem have to communicate using open source technology and must comply with basic requirements of portability, flexibility, and hardware independence. Following, the main requirements are listed:

- Flexibility; the design of the system must be modular, based on the decoupled aggregation of functional units to enable easy replacement of functions. It will also be ensured the functional flexibility of the system to that it is possible to replace tests and other logic such as the moding machine (the logic that dictates the state transitions of the diagnotic process). It will be possible to apply a variable number of tests to analyze the monitored system.

- Interoperability; both system parts, the diagnostic system and the vehicle software, must be designed and implemented in a fully interoperable way. They should follow the interoperability model of a well-known interaction paradigm of remote invocations supported by a distribution middleware.

©Ediciones Universidad de Salamanca / cc by-nc-nd


- Distribution middleware; a well-known, highly portable distribution middleware technology will be used such as Java RMI (Sun, 2016). Also, a service-based approach is easily derived by using a distributed Java service-based middleware such as River (Apache, 2013).
- Platform independence. The software design must adhere to the principles of plaform independence, i.e., it will be independent from the underlying hardware and operating system. If the software should run on a different platform, minimum changes should be performed to the actual design and implementation. The selected hardware for the diagnostic computer executes direct bycodes more easily support backward compatibility.
- Performance. The system has to support timely communication and low latency in the interoperability between both parts.
- Open source technology. The system has to favor the usage of open source distribution middleware that is a basic approach to obtain low-cost solutions. There are a number of open source technological choices that have been extensively used in a number of application domains, proving to be highly reliable and robust.

3.2 Components design

The overview of the system is shown in figure 1. It has an emulated part that replicates the vehicle logic for performing auto-tests upon request from the user or operator. The system contains two main blocks: the *vehicle* software and the *diagnosic computer* (DC) software. DC modules are described below:

- *Diagnosis computer core State Machine* (DSM) is the active entity that governs the execution of the diagnosic system. It determines the execution path of the different functions and units in the system, acting as a moding machine, i.e., it guides the mode transitions of the system.
- *Communications module* (Comms) implements the communication protocol between the monitored vehicle and DC. The protocol identifies the specific data that is exchanged and the communication sequence (containing acknowledgements and startup).
- *Graphical User/operator Interface* (GUI) is the component that displays the information to the user/operator and requests inputs from the operator that will guide the diagnostic process. Operator inputs are fed to the state machine component, parameterizing its execution.

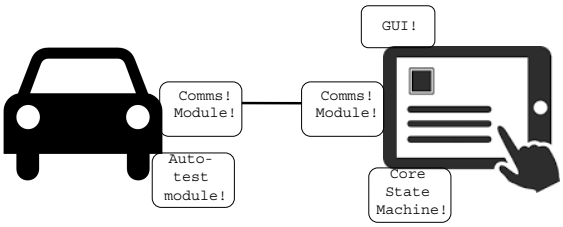

Figure 1: Diagnosis system design based on modules.

The vehicle components are listed as follows. *Auto-test module* (ATM) contains the algorithms that monitor parameters of the physical system. The ATM receives the operator input through the communication

©Ediciones Universidad de Salamanca / cc by-nc-nd

module and runs the requested test. A variable number of algorithms (i.e., the actual tests) can be executed, favoring maintanance and flexibility. *Communications Module* (VComms) implements the vehicle side of the communication protocol to exchange information with the DC and, in the end, with the operator.

3.3 Software design

The software is structured as components that implement the above described modules. Figure 2 shows the three main components from the DC side.

The *Core State Machine* component contains a set of classes that model the needed information about the vehicle and the tests to be run, i.e.:

- the vehicle model (`VehModel`) with all the vehicle paramaters that can be monitored through the tests;
- the list of all available tests for the specific vehicle (`TestList`);
- the parameter values that are fed to tests (`TestParameters`), coming from the user specification upon the launching of the test execution;
- the set of results that each test outputs (`TestResultsData`).

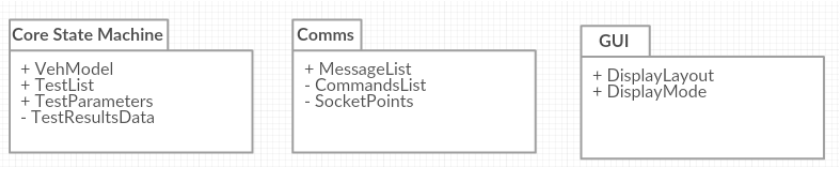

Figure 2: Software components of the diagnosic computer.

Comms module includes classes to design and implement the communications protocol among the vehicle and the diagnosic computer, i.e.:

- the set of messages that can be exchanged (`MessageList`) between the diagnosic computer and the vehicle communication module;
- the set of commands (`CommandsList`) that the diagnosis computer can issue to the vehicle. Examples are to run a specific test, to relaunch a test, or to provide further information on a test, among others.
- the set of connection resources used for the communication `SocketPoints` that uses sockets with a transport protocol that can be selected. Security can be added for data encription by using SSL.

The graphical user interface module *GUI* performs the friendly display of the diagnosic computer operation. It has information about the display characteristics (`DisplayLayout`); it allows to easily change to a different display computer (e.g., a touchpad display of a different resolution, size, etc.) as the specific characteristics of the hardware display are hidden in this class. Also, GUI is designed to support different display modes such as for technical users or for the accounting staff to gather statistics on the specific vehicle failures.

An abreviated class diagram of the system is shown in figure 3. It presents some selected data on two important classes. Class `VehModel` presents the data model of the vehicle. Its attributes are the set of parameters that define the vehicle characteristics refering to the ECUs or electronic control units. These units control all the operation of the car, carrying out functions at all levels from the engine control, driving assistance, or the less

©Ediciones Universidad de Salamanca / cc by-nc-nd


critical passenger comfort functions. Each car function is divided into subsystems that are reflected in each of the attributes `PhyParn` (*physical parameters of subsystem* `n`) that model a given subsystem (subsystem `n` in this case).

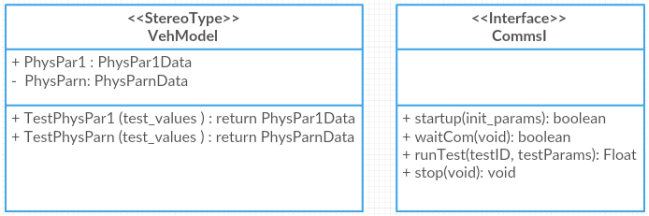

Figure 3: Simplified class diagram of the diagnosis computer side.

The communications module's interface is presented in figure 3, precisely in the interface `CommsI`. It shows the basic operations for the communication:

- Vehicle auto-test module start up (`startup`) and shut down (`stop`); by issueing this invocation, the vehicle module begins its operation and the values for the auto-test runs are initialized.
- The vehicle auto-test module has a mode to wait for communication from DC (`waitCom`). In this mode, the vehicle auto-test module is not running any test; it is idle in attention to receive connection requests from DC that are triggered by the operator.
- Specific tests can be requested by DC by invoking the method `runTest` and specifying as parameter a given test with its precise execution parameters.

The central module of the diagnosis computer, the *Core State Machine* module triggers the operation of the tests in the vehicle. Its operation is shown in figure 4. All indicated states are run in the diagnosic computer.

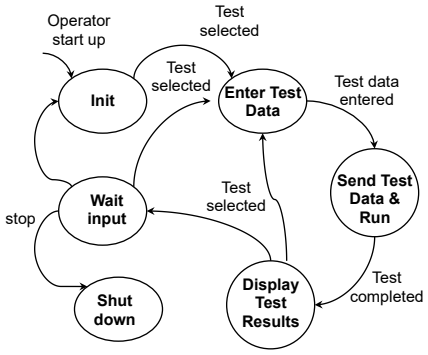

Figure 4: Operational sequence of the Core State Machine module.

The startup of the system (init button pressing) places the DC in the initial state (*Init*), indicating that no previous tests have been run since the system is active. From that state, an operator/user may select a specific test to be run. Then, the system enters the *Enter Test Data* state and indicates the threshold values to check for the given test. Upon completing the entering of the test data and pressing the *run test* option, the information is sent

©Ediciones Universidad de Salamanca / cc by-nc-nd

to the vehicle software and there, the test is run; the DC then enters state *Send Test Data & Run*. Once the test is run and the test results are sent back by the vehicle software, the DC enters state *Display Test Results* where the information is displayed to the operator. After this display, the DC enters state *Wait input* for further operation. If a new test is to be performed, the sequence resumes by entering state *Enter Test Data*.

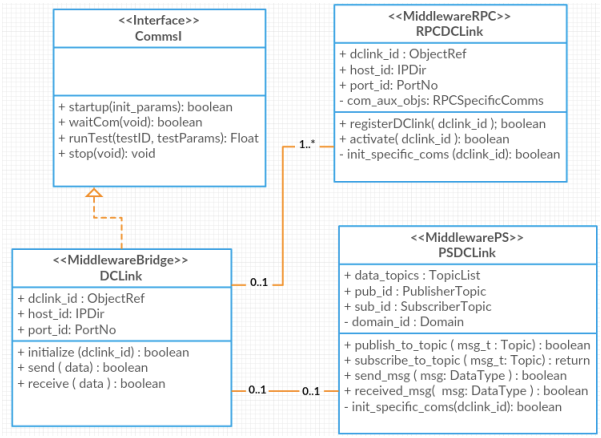

Figure 5: Flexible bridge to use multiple middleware backbones for communication

Figure 5 shows the modular design of the communication module. Multiple middleware technologies can be used as the actual communication means, as it is abstracted by the *DCLink* class that acts as a general communication description that can latter be mapped to different middleware technologies and even to specific implementations of the same middleware. In figure 5, two options are shown: an RPC (*remote procedure call* paradigm) such as Java Remote Method Invocation (Sun, 2016), CORBA (OMG, 2012) or the Internet Communication Engine (ZeroC, 2003); and a P/S (*publish-subscribe*) middleware such as DDS (OMG, 2015). Lower level mechanisms can be easily adapted to this structure, e.g., socket based communications. It is also possible to use this bridging for more complex communication schemes that could support communication with multiple vehicle units using paradigms that support dynamic reconfiguration such as iLand middleware (García-Valls et al., 2013; García-Valls et al., 2012).

The communication interface is `CommsI` that specifies the basic functionality of the DC interfacing module. This module is initiated via *startup* function that performs the initialization operations. These operations vary according to the given middleware implementation that is selected. For example, in the Internet Communication Engine, two environment objects must be created (i.e., *communicator* and *adapter*) that create a remote object that is exported to the public domain and is visible for the vehicle software part. In the case of other technologies such as DDS, a *domain* has to be created, among other entities such as the *domain participants*, *writers*, *readers*, *publishers*, and *subscribers*. This specific per-technology communication structure is abstracted in the `DCLink` class.

Most available middleware technologies support the usage of multi-language and multi-platform; it is possible that both ends of the communication are implemented in different programming languages over different operating systems that is often enabled by using an interface definition language (IDL). An important consideration in such a case is that some programming environments require the addtion of a virtual machine, e.g., Java or C#. For efficiency reasons, the proposed model considers that the same middleware technology is used at both communication end points (vehicle software and DC software).

©Ediciones Universidad de Salamanca / cc by-nc-nd

4. Validation

The diagnosis computer system prototype is implemented in two main parts: the monitored object is implemented and emulated in a desktop computer that simulates the execution of algorithms that monitor physical parameters in the engine subsystem; the diagnosis computer is realized in a bytecode processor that is an actual embedded computer with limited computation power such as aJile's aJ10x family of Java bytecode processor (aJile Systems, 2016). The desktop machines runs a Java based prototype on a Ubuntu 10.04 Linux distribution and a Java SE 7 for RMI-enabled connection. The *Comms* module uses a TCP/IP connection for message exchanges between DC and vehicle software. Figure 6 shows the overview of the prototype.

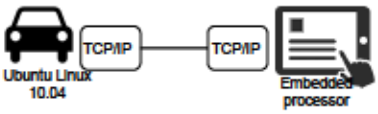

Figure 6: Prototype scheme.

Important aspects to be checked are the responsiveness from the DC side, the stability in the communication, and the flexibility of the design in order to modify the communication backbone by a different technology in short time. The first two paramaters have been measured and the results show that the system is quite responsive from the DC side, providing good interactivity to operators.

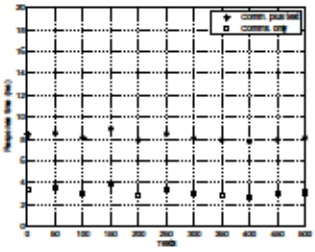

Figure 7: Communication time with and without including the vehicle side test times.

The results shown present the overhead incurred by the prototype in a sequence of 500 tests of an average duration of $5s$ each. The prototype uses a Java environment with an underlying TCP socket implementation. For the test duration, the communications module shows that the overhead is influenced by the characteristics of the aJ100 embedded processor that is a resource limited environment; it is a 32 bit processor with direct execution with support for multiple JVMs, 48KB RAM memory. By using a more powerful device such as RaspberriPi series with ARM processors, the increase in response time could be highly relevant. Figure 7 shows the communication time between the DC and the vehicle. On the one side, it shows only the time taken by the interaction (i.e., the network time plus the time taken by the TCP/IP software stack to process the communication). On the other side, it also shows the overall time that includes the temporal cost of the tests that are $5s$, therefore a consistent larger cost is demonstrated. The design is highly portable and has been adapted to use a full fledged C++ environment with Ice version 3.1 both over TCP and UDP transports. The portability of the *CommsI* module is easily achived. The results were tested over both destop patforms with Intel dual core processors running at 2.6GHz with 1GB of memory, and the experiments show that the overhead is reduced to less that 0.02%.

©Ediciones Universidad de Salamanca / cc by-nc-nd

5. Conclusion

The paper has presented the design of a flexible and modular diagnostic systems that supports platform portability and integration of different underlying middleware technologies. The proposed design has been prototyped in a resource constraint environment based on a Java embedded processor, showing a time overhead that is suitable for this type of domain and for the duration of the performed tests. The designed general framework provides a simple and clean structure that is easily adaptable to run on multiple underlying communication backbones. The design has also shown to be very flexible as it was ported to a different underlying middleware that uses a different programming language and an IDL for the specification of the *CommsI* interface in a reduced time. This shows that the proposed approach is suitable for achieving low-cost diagnosic systems, taking advantage of the robust and reliable middleware technologies that are open source and freely available.

Acknowledgement

This work has been partly funded by the project REM4VSS (TIN2011-28339) and M2C2 (TIN2014-56158-C4-3-P), funded by the Spanish Ministry of Economy and Competitiveness.

Vitae

Marisol García Valls is Associate Professor of Universidad Carlos III de Madrid, Spain. She holds the acreditation as Full Professor by ANECA-Spain since May 2014. Her interest research areas include reliable software technologies for distributed applications, distributed programming models, resource management and OS, performance of distributed systems, virtualization technology for predictable/reliable cloud computing, and distributed real-time systems. She is author of numerous articles in relevant scientific venues, around 40 of them in JCR indexed journals in the above areas. She has been enrolled in a number of European and national research projects, and has been the scientific and technical coordinator of iLAND project (EU Artemis-1-00026, 2009-2012) focused on the design and development of an enhaced communication middleware for supporting dynamic reconfiguration of real-time distributed systems based on services. She was awarded for the excellence in project coordination by the ARTEMIS JU (2012). She is part of the programme committee of a number of relevant international conferences and symposia. Also, she is associated editor of the Elsevier Journal of Software Architecture for the area of "Middleware" since 2013, and she is also associated editor of Future Generation Computer Systems of Elsevier since 2014. ACM member and IEEE Senior member. In 2014, she was awarded the Prize for Outstanding International Research funded by the Social Council of Universidad Carlos III de Madrid.

6. References

Apache, A. S. F., 2013. Jini network technologies specification. Apache River v2.2.0.

Arduino, 2014. Arduino uno.

ARM, 2016. ARM processor architecture.

Barry, R., 2010. *Using the FreeRTOS Real Time Kernel*. 1.3.0 edition.

Brunemann, G., Dollmeyer, T. A., and Mathew, J. C., 2002. System and method for transmission of application software to an embedded vehicle computer. US Patent 6,487,717.

García-Valls, M., 2016. *Low Cost Software Prototyping of a Diagnosis Computer*, pages 443–451. Springer International Publishing, Cham. ISBN 978-3-319-40162-1. doi:10.1007/978-3-319-40162-1_48.

©Ediciones Universidad de Salamanca / cc by-nc-nd

García-Valls, M., 2016. A Proposal for Cost-Effective Server Usage in CPS in the Presence of Dynamic Client Requests. In *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 19–26. doi:10.1109/ISORC.2016.13.

García-Valls, M., Alonso, A., and de la Puente, J. A., 2012. A dual-band priority assignment algorithm for dynamic QoS resource management. *Future Generation Computer Systems*, 28(6):902 – 912. ISSN 0167-739X. doi:http://dx.doi.org/10.1016/j.future.2011.10.005.

García-Valls, M. and Baldoni, R., 2015. Adaptive Middleware Design for CPS: Considerations on the OS, Resource Managers, and the Network Run-time. In *Proceedings of the 14th International Workshop on Adaptive and Reflective Middleware*, ARM 2015, pages 3:1–3:6. ACM, New York, NY, USA. ISBN 978-1-4503-3733-5. doi:10.1145/2834965.2834968.

García-Valls, M. and Basanta-Val, P., 2017. Analyzing point-to-point DDS communication over desktop virtualization software. *Computer Standards & Interfaces*, 49:11 – 21. ISSN 0920-5489. doi:http://dx.doi.org/10.1016/j.csi.2016.06.007.

García-Valls, M., Basanta-Val, P., and Estévez-Ayres, I., 2010. Adaptive real-time video transmission over DDS. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 130–135. IEEE.

García-Valls, M., Calva-Urrego, C., Alonso, A., and de la Puente, J. A., 2017. Adjusting middleware knobs to assess scalability limits of distributed cyber-physical systems. *Computer Standards & Interfaces*. ISSN 0920-5489. doi:http://dx.doi.org/10.1016/j.csi.2016.11.003.

García-Valls, M., Cucinotta, T., and Lu, C., 2014a. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726 – 740. ISSN 1383-7621. doi:http://dx.doi.org/10.1016/j.sysarc.2014.07.004.

García-Valls, M. and Ibáñez-Vázquez, F., 2012. *Integrating Middleware for Timely Reconfiguration of Distributed Soft Real-Time Systems with Ada DSA*, pages 35–48. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-30598-6. doi:10.1007/978-3-642-30598-6_3.

García-Valls, M., Perez-Palacin, D., and Mirandola, R., 2014b. Time-Sensitive Adaptation in CPS through Run-Time Configuration Generation and Verification. In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 332–337. doi:10.1109/COMPSAC.2014.55.

García-Valls, M., Rodríguez-López, I., and Fernández-Villar, L., 2013. iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 9(1):228–236. ISSN 1551-3203. doi:10.1109/TII.2012.2198662.

Halfacree, G. and Upton, E., 2012. *Raspberry Pi User Guide*. Wiley Publishing, 1st edition. ISBN 111846446X, 9781118464465.

IITF, I. I. T. T. F., 2014. OASIS AMQP1.0 – Advanced Message Queuing Protocol (AMQP), v1.0. ISO/IEC 19464.

Lowrey, L. H., Banet, M. J., Lightner, B., Borrego, D., Myers, C., and Williams, W., 2003. Internet-based vehicle-diagnostic system. US Patent 6,611,740.

Margolis, M., 2011. *Arduino Cookbook*. 2 edition.

OMG, O. M. G., 2012. The Common Object Request Broker. Architecture and Specification, Version 3.3.

OMG, O. M. G., 2015. A Data Distribution Service for Real-time Systems Version 1.4.

Parrillo, L., 1995. Wireless motor vehicle diagnostic and software upgrade system. US Patent 5,442,553.

Rodríguez-López, I. and García-Valls, M., 2011. *Architecting a Common Bridge Abstraction over Different Middleware Paradigms*, pages 132–146. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-21338-0. doi:10.1007/978-3-642-21338-0_10.

Romero, J. C. and García-Valls, M., 2014. Scheduling component replacement for timely execution in dynamic systems. *Software: Practice and Experience*, 44(8):889–910. ISSN 1097-024X. doi:10.1002/spe.2181.

Sun, S. M., 2016. Java Remote Method Invocation API.

©Ediciones Universidad de Salamanca / cc by-nc-nd


aJile Systems, I. a., 2016. Low power real-time network direct execution SOC for the Java ME platform.
ZeroC, Z. I., 2003. The Internet Communications Engine.

©Ediciones Universidad de Salamanca / cc by-nc-nd