8

Java in the Loop of Data Acquisition Systems

Pedro Mestre[1], Carlos Serodio[1], João Matias[2],
João Monteiro[3] and Carlos Couto[3]
[1]*Centre for the Research and Technology of Agro-Environment and Biological Sciences, University of Trás-os-Montes and Alto Douro*
[2]*Centre for the Mathematics, University of Trás-os-Montes and Alto Douro*
[3]*Industrial Electronics Department, University of Minho*
Portugal

1. Introduction

Modern Distributed Data Acquisition Systems consist of many different components. Those components can be made by different manufacturers, be based on different hardware platforms, running different software or firmware applications and implement by different hardware/software system developers.
This component variety makes distributed systems to be heterogeneous at two levels: at executable code level, due to the fact that the binary code of a platform might be incompatible with other platforms; at data level, because different platforms might have different ways to represent, store and deal with data.
To overcome issues raised by this platform heterogeneity it can be used a Virtual Machine technology, a Middleware technology or both technologies together. Virtual Machines present to the programmer a homogeneous development platform, therefore the programmer does not need to concern about the final platform where the application will be used. Middleware offers the tools for programmers to develop distributed applications without concerning about details of objects' communications.
Java is one of the most used Virtual Machine technologies and it has support for different platforms. Its support goes from desktop computers to mobile phones. A key point of this technology is its support for many different Operating Systems such as Microsoft Windows family, Solaris, Linux and Mac OS. Java also supports the most used Middleware technologies, allowing the implementation of distributed data acquisition systems compatible with components made with other development tools.
Based on the latest trends in consumer electronics and industrial applications using WSN (Wireless Sensor Network), smart sensors and embedded systems, we can conclude that future nodes must become smarter, cheaper and smaller.
So, the major challenge is to produce smart sensing devices which have very low resources. Besides that, also the reduction of both the development time and costs are needed. Considering this, two approaches can be used: in the first, hardware resources are made accessible, e.g. connecting them over TCP/IP, putting these devices at the distance of *a click*; the second is to give to bus-level devices some characteristics that until now are mainly found in the traditional computing systems. The second approach is based on the

application of "write once, run anywhere and any time" Java concept. However, it must be taken in consideration that bus-level devices, like sensor nodes, have several constraints regarding the lack of resources, particularly memory issues.
In this chapter the use of Java Technology is proposed. Not only in desktop applications where it is traditionally used, but also on the other system components. Java can be used not only at the higher layers, on data acquisition software applications where data is gathered, processed and stored, but also in device drivers that do the interface between applications and data communication systems used for data acquisition, such as wireless communications and fieldbuses. To implement the concepts presented in this chapter, IEEE802.15.4 for wireless communications and CAN (Controller Area Network) as fieldbus technology were used, for proof of concept purposes.
Java can even be used in networked distributed nodes where data are collected. Not only in state-of-the-art embedded systems but also in low-resource devices, such as 8-bit microcontrollers with a few kilobyte of memory and low clock speed. For such systems it is presented a solution based on a dedicated Java Virtual Machine, embedded in a 8-bit microcontroller, which acts as its Operating System.
Authors do not intend to present a replacement for JDDAC (Java Distributed Data Acquisition and Control), (Engel et al., 2006), which is a platform to build Java-based data acquisition systems and sensor networks. In fact, some of the aims of the present work are similar, even though it is not an API (Application Programming Interface) or a Framework, JDDAC can be fitted in some of the layers of the model presented in this chapter.

2. Java technology and platform heterogeneity

A completely homogeneous platform where all the system components have the same characteristics is not possible to achieve. Even if a newly deployed distributed system is made of homogeneous components, sooner or later, the elements that make part of it will start to be different from each other, as new devices are added. Those devices might not have all the same characteristics in what concerns to processor architecture, processing power, memory amount and communications technologies.
Being a technology which supports multiple computation platforms, Java is one of the solutions to take in account when the subject is the platform heterogeneity. It is based on a Virtual Machine, the Java Virtual Machine (JVM), which has support for the most used computing platforms. The use of Virtual Machines to overcome problems related with platform heterogeneity has more than 30 years (Gough, 2005), with the definition of a Virtual Machine able to run code generated from PASCAL, the P-Code.
When Virtual Machines are used the programmer works without the need to concern about the target processor or operating system. There is no need to know any detail about the real platforms used in the system. This gives the ability to develop applications targeted for currently available and future architectures and devices.
Besides offering a homogeneous execution environment, Java technology offers different kind of support according to the characteristics of the used devices and the type of service to be implemented.

2.1 Java solutions

From the most traditional computing platforms such as Personal Computers, to embedded systems, passing by interactive television applications, PDA (Personal Digital Assistant) and

mobile phones, Java supports a large number of different types of computing platforms. Java support for the traditional computing platforms is offered through two different solutions (Sun, 2003):

- Java EE (Java Platform, Enterprise Edition) for enterprise applications and servers;
- Java SE (Java Platform, Standard Edition) for desktop applications and servers.

For use in consumer electronics devices, and for embedded servers and applications as well, Java support for the existing technologies is offered by the following solutions:

- Java ME (Java Platform, Micro Edition), for use in PDAs, Mobile Phones and specific embedded applications;
- Java Card, for applications with Smart Cards;
- Java TV, to be used in Iterative Television applications;
- JES (Java Embedded Server), for embedded applications.

The use of Java is a step to achieve a system with homogeneous components. Although each type of node, in distributed data acquisition systems, has a different type of Java support, they all have a similar behaviour. Their applications are based on a similar API (Application Programming Interface), use the same programming language, share a ”virtual processor” that uses the same bytecodes, and represent and store data the same way.

It is possible to build distributed systems where the nodes are compatible at data and executable code levels. This means that nodes can share data without concerning about its representation and, executable code can be shared by the nodes.

Using Java in all system components of the distributed system, besides hiding from the programmer the ”real” platform and supporting data mobility, also code mobility is possible. This code mobility can be explicit by downloading code from a server and then execute it or, code can be embedded in method invocations. Java applications can call remote code in other JVMs and, when doing this remote code invocation, they can send as input parameters or receive as return values of the called methods, data or objects (which are made of code and data). Code can dynamically be sent between networked nodes. This dynamic code mobility is a key point in this technology.

2.2 Integrating Java with other technologies

Not all applications used in distributed system are developed using Java. This leads to the need of finding a solutions that enable the Java components to communicate with other applications. This can be done using Middleware Technologies.

Middleware technologies work as an abstraction layer to the programmer. Its objective is to hide system implementation details, offering a uniform view of the network and the Operating Systems (Sun & Blatecky, 2004). It supports communications between distributed software components and copes with the problems related with the platform heterogeneity (Mattern & Sturm, 2003). For each platform that makes part of the distributed system, an implementation of the middleware technology must exist.

From the programmer’s point of view, middleware offers the needed abstraction for the programmer not having to concern about low-level issues of the deployment platform and the communications protocols used in the distributed system. It offers transparency to the network. Independently of protocols and platforms used to deploy and develop the different components, the programmer sees a uniform system.

Programmers only need to concern about the coding of their applications and with the interfaces made available by the programming language to access the functions offered by

the middleware layer. All the issues related with the transport of requests and responses between objects in a distributed system, are dealt by the middleware layer. Also details about discovering remote services and their interfaces are dealt by it.
Java Technology, besides supporting Java-centric middleware technologies, also has support for other widely adopted middleware technologies, enabling applications developed with this platform to communicate with applications developed using other tools. From the list of middleware technologies supported by Java Web Services, which use the same technology that is used on the Web, is one of the most adopted.

2.2.1 Web services

Created to be applied in de development of distributed systems that operate over the Internet, the aim of Web Services is to have a technology that is independent of the programming paradigms, languages and technologies. They represent a return to the old client/server model (Coulouris et al., 2005) on which both peers are functionality specialized.
Message exchanges between client and server are made using SOAP (Simple Object Access Protocol) and the transport of these messages is typically done using the HTTP (Hypertext Transfer Protocol) protocol. However, other protocols can be used to transport it.
SOAP defines how information must be formatted in XML (Extensible Markup Language) for information exchange between the elements that are part of the distributed system (Mitra & Lafon, 2007).
The use of XML presents several advantages when compared with binary formats to transport data, for example it can be interpreted by humans making the debug process more easy. XML also helps in the cross platform compatibility, enabling the definition of structured documents that can be interpreted by most platforms. However, to transport the same information, XML has a higher overhead and it needs more processor and memory resources to be decoded, when compared with binary formats.
As for any other technology that allows a client to remotely use services, clients need to know which methods are available on the server, i.e., client needs to know the server remote interface, which describes the methods available on the server. This description is called WSDL (Web Service Definition Language).
Independently of their programming language (Java, C++, C#, etc) and running platform, applications can communicate with each other over the network. This is thus a middleware technology to be taken in account when communication between heterogeneous networked elements is needed.

3. Generic model for Java based data acquisition

A model for Networked Data Acquisition Systems, based on Java Technology and its remote code invocation features, is presented in Fig 1. Besides distributed data acquisition devices, e.g. smart sensors spread in the field, it also supports Distributed Device Drivers and remote Data Acquisition Applications.
This model has the following four layers:

- Device Layer – Where the data acquisition devices are located. These devices can be connected directly to a computer or through a data communications network (e.g. a filedbus, WSN technology);

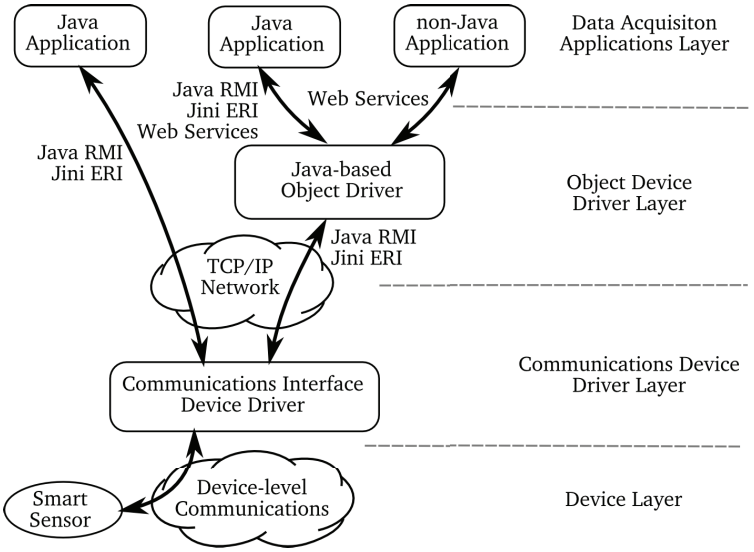


Fig. 1. Generic model for Java-based networked data acquisition systems.

- Communications Device Driver Layer – This layer is responsible for the interface with the data acquisition devices. It sends messages from the upper layers to devices and *vice versa*;
- Object Device Driver Layer – For higher level interfacing with applications. For each smart sensor, an object driver can exist in this layer;
- Data Acquisition Applications Layer – Where the final consumers for the data from the sensors are located.

A common philosophy in this model is the fact that all elements can be spread over the network. From the lower layers (device level) to the upper layer (applications level), nodes can be anywhere in the network, as long as they have network connectivity and there is a possible route between them.

3.1 Data acquisition applications layer

Applications can consume data from two sources, depending the degree of knowledge they have about the target device. If an application knows how to communicate with the hardware, i.e., it knows the low-level protocol used by the device, it can interface directly with the data communications device driver. When the application does not know such details or, a higher level of abstraction is needed, then applications can communicate with the corresponding Object Device Driver that routes information requests and responses between information source and consumer.

These two approaches for device interfacing give two different abstraction levels, to be used in two different situations. One for direct communications with the field devices and the other to gather data from the sensors. When accessing the Object Device Driver, independently of the network technology used by the field-level device, applications should be able to interface in the same way sensors that have the same function. For example, a temperature sensor should have the same software interface either when using CAN or IEEE802.15.4 wireless communications.

In the Applications Layer, two types of software applications might exist: Java-based applications and non-Java applications. These two types of applications communicate with the Java-based Object Driver, using Web Services. In the case of Java applications also Java RMI (Remote Method Invocation) and Jini ERI (Extensible Remote Invocation) are available. Java applications can also use Java RMI or Jini ERI to communicate directly with the Communication Device Driver, bypassing the Object Driver.

3.2 Object device driver layer

Applications do not have to know how to communicate with the real device, where data is actually being acquired. Remote sensors are modelled as objects according to their functions and, hardware devices can be modelled as several different objects. One object per acquisition function that it has.
Requests for data are made by applications to the Object Driver, which forwards the data requests to the real device, using the lower layer services. When the device answers, it sends the responses back to the applications. These communications between the Object Driver and the Communications Device Driver are made using Java RMI and Jini ERI, since these two elements are implemented in Java.

3.3 Communications device driver layer

For communications with the real device, the Object Device Driver Layer and some Java applications must use the lower level driver, the Communications Interface Device Driver. This driver is to be used only by components that understand both the Layer 2 and high level protocols used by devices.
While the Object Driver works mainly in a request/response way, using the traditional client/server model, the low-level device driver works in asynchronous mode. The driver is stateless and does not know the meaning of messages exchanged between applications and devices. Therefore it cannot track connections. So, the driver is unable to know if new data is expected or even if data will arrive from remote nodes. Data exchange sequence depends on the network technology, time needed by the remote node to process data and on the higher level protocols used by devices.

3.4 Device layer

In the lowest layer, devices capture data consumed by the applications in the upper layer. These acquisition devices can be directly connected to a computer or can be connected to a data acquisition network, such as a fieldbus or WSN technology. Devices communicate, through the Communications Interface Device Driver, with the Object Driver or the Applications. They do not communicate with the Communications Interface Device Driver, which only transports data and does not know its meaning.
Implementation of the high level communications protocol used to exchange information with the remote data acquisition devices must be done either in the Object Driver Layer or in the Application Layer.

4. Java device drivers

When new hardware is added to a computer or to a networked environment it is needed to install a device driver to handle communications between applications, Operating System

and the device. This leads to the need of multiple versions for the device driver, one for each supported platform. If the device driver is built using a Virtual Machine Technology supported by multiple Operating Systems, such as Java, one version of the driver can be used by several different platforms.
Based on the model above presented in section 3, two different concepts and philosophies of device drivers are presented:

- Low-level Communications Device Driver, used for direct interface with the communications hardware. Although it can be used by any application, this driver is intended to be used by the next driver type;
- High level Object Driver, used by applications to access data from objects available on the network. In the context of data acquisition, an object is any source of data, e.g. a smart sensor.

The first belongs to the Communications Device Driver Layer and the second to the Object Device Driver Interface Layer. These two drivers implement two different low-level communications protocols:

- Low-level communications between the Java device driver and the communications hardware, dealt by the Low-level device driver;
- Low-level communications between the Object Driver (and some applications) and the remote data acquisition devices.

4.1 Low-level communications device driver

Low-level interfacing with the device driver involves the use of multiple technologies: native code for basic hardware level communications; Java to implement the core of the device driver; Java-based middleware technology enabling communications between the device driver and the remote object drivers and applications. The layered model for this device driver is presented in Fig. 2.

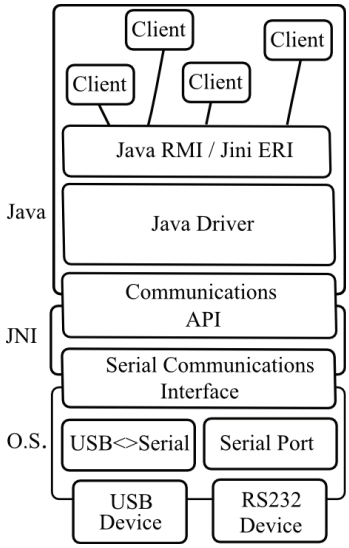


Fig. 2. Low-level Java-based device driver layered model

This device driver model has the following four layers:

- Hardware Layer, where the communication hardware interfaces can be found;
- Operating System Layer, where platform serial drivers are;
- Java Native Interface Layer, which makes the bridge between the Java world and the Operating System;
- Java Layer, where the core of the device driver for the communications protocol is implemented.

4.1.1 Hardware layer

This is a hardware only layer, responsible for the physical connection between the communications hardware (Network Interface Card) and the host computer that will share this hardware with the other components of the distributed data acquisition system.
For this layer two of the industry adopted solutions for communications with devices are used: serial port (RS-232 compatible) and serial port emulation over USB (Universal Serial Bus).
Although in the implementation presented in this chapter the device driver is serial port centric, it can be adapted to other type of hardware. Whenever it is possible to communicate with the hardware using JNI (Java Native Interface), then this device driver concepts can also be used.

4.1.2 Operating system layer

This is a platform dependent layer since it relies on the existence of a native device driver to deal with the low-level communications between the Operating System and the serial or USB port. This is the first software layer and it is implemented in native code.
The Operating System Layer does not need to know how to communicate with the Network Interface Card, it only needs to know how to send and receive data from the serial port. Independently of the type of interface used in the Hardware Layer (RS-232 or USB), it presents to the Java Native Interface Layer a generic serial port. Hardware isolation is then another of its features.

4.1.3 Java native interface layer

Interfacing between the Java core of the device driver (implemented in the upper layer) and the Operating System Layer is done by the Java Native Interface Layer. It is implemented in the platform native code and Java code, using JNI.
Although this layer is platform dependent, the used API for communications with the serial port, the Java Communications API, has implementations for many platforms: officially by Orcale for the Linux and Solaris platforms and by a third party for Linux, Solaris, MacOS and Microsoft Windows[1].

4.1.4 Java layer

This is the core of the low-level Java device driver, where the low-level communications protocol used to interface the network interface hardware is implemented. Completely implemented in Java, this is the first system independent layer of the device driver. For

[1] The gnu.io library, an implementation of the Java Communicatons API that can be found at http://rxtx.org/

communications between the driver and the remote client applications, this layer supports both Java RMI and Jini ERI.
Being the core of the device driver, this layer is network technology dependent. It must know the low-level protocol used in the acquisition network and provide to applications the correct remote interface. It is also network hardware dependent, since it must know how to communicate with the hardware.
Traditionally device drivers and applications must be executed in the same computing device to which the network interface card is connected. In the presented solution, by using Java RMI and Jini ERI, it is possible for applications to be located in a remote computer. Devices can then be remotely accessed by applications and the device can be shared by multiple distributed applications. Since the used technologies rely on the TCP/IP protocol suite, applications and drivers can be locate anywhere in the Internet. However it must be taken in account that connections over the Internet might have a high latency. Real time requirements of applications must then be considered.
In the present work two types of communications protocols were used. One wired (CAN) and another wireless (IEEE802.15.4). Nevertheless these two protocols were used in the tests, concepts presented here can be applied to other protocols.

4.2 Interface between applications and the low-level device drivers

Interfacing between applications and device drivers must be done using well defined Interfaces. The driver must allow applications to send messages to the devices and, when a message arrives the driver must deliver it to all applications that need that information. Driver and applications use the client/server model.
Since this part of the network interfacing operates at the Data Link Layer of the OSI (Open System Interconnection) model, decision about to which application send the information must be done based on this layer addressing scheme, or equivalent concept, of the used technology. For this to be possible, applications must be able to register the interest on receiving certain frames.
Applications must also be able to asynchronously receive the frames from the device driver. This is done by using a call back mechanism, on which the device driver calls remote methods implemented by the application. Client and server switch roles.
When an application sends frames and when it registers its interest on receiving some types of frames, the driver acts as the server and the applications as the client. When a message arrives to the device driver, the application acts as the server and the device driver acts as the client.
Besides the definition of object interfaces, used for communication between driver and applications, also the inter-process communication method(s) must be specified. The technology must be wide accepted and allow clients to be running remotely, not only in the same computer where the driver is installed. Since this part of the device driver and the applications that use it (object driver) are implemented in Java, two of the Java inter-process communication technologies are used: Java RMI and Jini ERI.
Java RMI is a very popular inter-process communication for Java-based distributed applications and it is part of the standard distribution of Java since the version 1.1 of JDK (Java Development Kit). It has multiple implementations, besides the original JRMP (Java Remote Method Protocol) (Newmarch, 2006), implemented based on TCP/IP. However those implementations have different programming interfaces, having thus a lack of technology abstraction.

A solution for this issue is presented by Jini Technology, which itself was built on top of Java RMI. This solution is called Jini ERI (Extensible Remote Invocation). One of its main characteristics is the possibility of the programmer to access each layer of an RMI call and allowing the service deployer to decide at runtime (not hard-coded in the applications) the most suitable implementation for those layers for a specific deployment scenario (Sommer, 2003).
Providing support for Jini ERI allows the use of modern Jini-enabled systems, which benefit with its Plug-and-Work features. On the other hand, support for Java RMI allows the integration of legacy systems, without support for Jini, or modern systems where Jini is not installed. For testing and proof of concept purposes, low-level device drivers for CAN fieldbus and IEEE802.15.4 were implemented.

4.2.1 Interfacing the CAN bus

Controller Area Network is a fieldbus designed for the automotive industry but with application in many other areas related with data acquisition and control. It does not use the traditional Layer 2 protocol addressing scheme. Instead of source and destination addresses, CAN uses the concept of message significance. Each message has its own meaning and is received and interpreted by all nodes to which it may concern.
For applications to be able to communicate with the device driver, a well defined interface must exist. Independently of the technology type used by the CAN interface card to connect to the host computer, its Java device driver must implement the `CANDriverInterface` shown in Fig. 3.

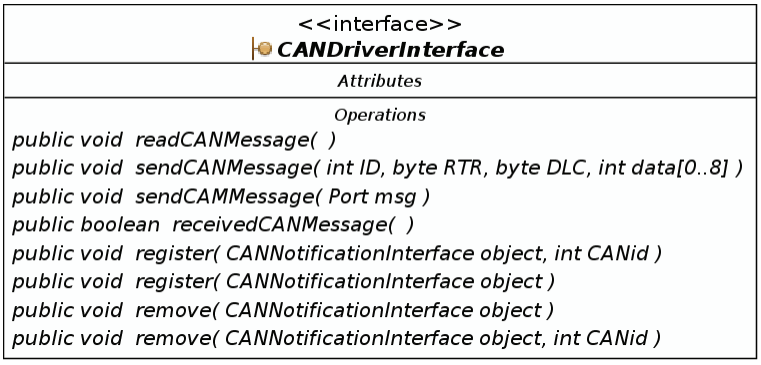
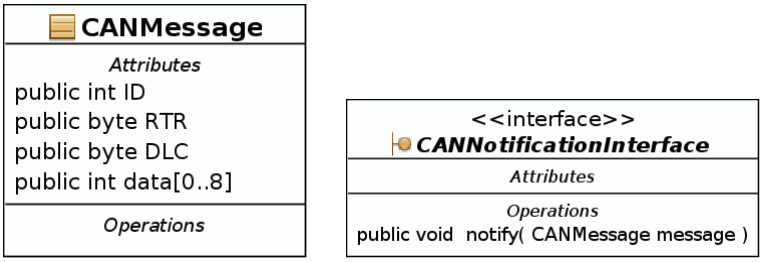


Fig. 3. CAN Driver Interface definition, which must be implemented by all drivers.

Applications can send CAN message using the `sendCANMessage` method, either specifying the CAN message parameters, according to the CAN specifications (Robert Bosch GmbH, 1991), or by sending an object of the type `CANMessage` which represents the CAN message (Fig 4). This class has four fields, representing each field of a CAN message: CAN Identifier (ID); Remote Transmission Request bit (RTR); Data Length Code (DLC); the message data (data), if any.
When a message, coming from the bus arrives to the driver, it will try to deliver it to all message consumers. For an application to register itself as a consumer for arriving CAN messages, it must call the `register` method. If an application wants to receive all CAN

messages, then it must call it without specifying the message ID. Otherwise it must call the other `register` method specifying the ID of the message it wants to receive. The other parameter of these methods is the *stub* of the client application.
If an applications wants to stop receiving some or all CAN messages, then they need to unregister themselves from the device driver using one if the `remove` methods.
For an application to be able to consume CAN messages, it must implement the `NotificationInterface` (Fig. 4). This Interface defines a single method (`notify`) used by the CAN driver to send a received message to its consumer.



(a) CAN Message (b) CAN Notification Interface

Fig. 4. Definition of the CAN Message Class (a) and the Notification Interface that applications must implement to be able to receive CAN messages from the driver (b).

Only applications that implement this Interface can be notified, by the device driver, when a new message is received by the communications hardware. These notifications are made using the call-back mechanism of Java RMI or Jini JERI.

4.2.2 Interfacing IEEE802.15.4

Similarly to CAN interfacing, the IEEE802.15.4 device driver needs to implement an interface known by the clients, presented in Fig. 5. This interface defines the `sendFrame` method that accepts an IEEE802.15.4 Frame (Fig. 6) as input parameter.

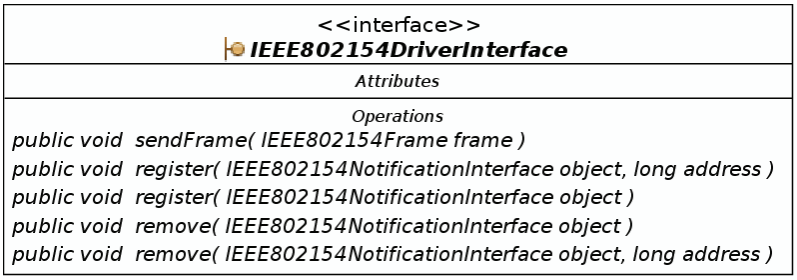
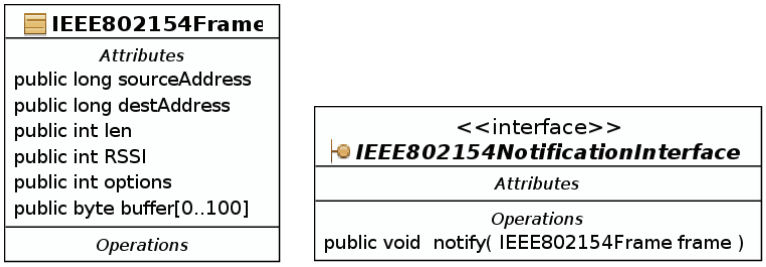


Fig. 5. IEEE802.15.4 Driver Interface which must be implemented by all drivers.

It also has methods for clients to register themselves as consumers of all frames or frames with a specific source MAC (Medium Access Control) address, received by the driver coming from the wireless interface, the `register` methods. With the opposite meaning, also two `remove` methods are defined.



(a) IEEE802.15.4 Frame (b) IEEE802.15.4Notification Interface

Fig. 6. Definition of the IEEE802.15.4 Message Class (a) and the Notification Interface that applications must implement to be able to receive frames from the driver (b).

The fields of the class which represents the IEEE 802.15.4 Frames were define according to the IEEE802.15.4 specifications (IEEE, 2006). It contains the destination MAC address field (`destAddress`), to be used only when sending frames since the source address of the frames is automatically set by the hardware. Therefore the field `srcAdrress` is to be used in the received frames to determine the sender, and the `RSSI` (Received Signal Strength Indicator) to indicate the RSSI value of the received frame.
To be used in both sending and receiving operations, there are the `data` and the `len` fields used for data buffering and to indicate the data length of the frame (received or to be sent) respectively.
As for the CAN bus, for applications to be notified by the driver when a new fame arrives, they must implement the `IEEE802154NotificationInterface`, presented in Fig. 6.

4.3 High level object interfacing

High level interfacing with the system is the preferable entry point for applications and, it should be in this layer that final consumers gather data from the sensors. It is also this layer that is responsible for communications with the non-Java applications, since this layer besides supporting Jini JERI and Java RMI also supports Web Services. Although only these middleware technologies were included in the current implementation, integration of other middleware technologies can be made, as long as it is supported by Java. For communications between this layer and the lower level layer, the Java based middleware technologies are used. Each data source in acquisition network (e.g. sensors) which are available for applications through this layer must have an object device driver. Applications using this layer do not know how to communicate with the device and therefore cannot use the lower level Communications Device Driver.
Fig. 7 shows an example of an interface defined for a sensor, in this case a temperature sensor. The object driver for the specific sensor must implement this interface. The implementation of Object Driver for the sensor must contain the code needed to contact the real sensor hardware, using the sensor low-level communications protocol (through the lower layer device driver), obtain the sensor reading and return the value to the caller application.
The interface presented in the Fig. 7 is implemented by one of the device drivers used in an example, shown in section 6, where a non-Java applications, implemented using LabView, gathers data from a wireless temperature sensor. In that example, the application does not know where the real device is nor how to communicate with the hardware. It uses a Web Service made available by the Object Driver.

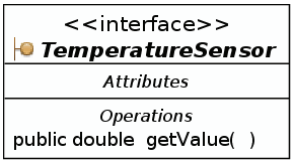


Fig. 7. Temperature Sensor Interface.

5. Java in embedded devices

Devices that belong to the Device Layer, of the above presented model, typically execute their applications using native code. One step that can help to achieve device homogeneity, even at the field level, is the ability of running Java on this type of devices. For this to be possible Java support must be provided by an embedded JVM.
Java in embedded systems has been subject of several projects. Its implementation has been based on FPGA (Field-Programmable Gate Array) and on microcontrollers (Hardin, 2001; Ito et al., 2001; Pfeffer & Ungerer, 2004). One of this embedded systems that runs Java was presented by Sun Microsystems, the Sun SPOT (Sun, 2005) which runs on a 32bit microcontroller with 256Kbyte of RAM and 2Mbyte of FLASH.
It is however possible to run embedded Java in low resources microcontrollers, such as the 80C592, a 8051 based microcontroller (Serodio et al., 2001) or PIC18F2680 (Serôdio et al., 2007), both 8-bit microcontrollers. Obviously that these implementations of the JVM are optimized for data acquisition tasks, so some details of a full JVM are not implemented on them.

5.1 Embedded Java implementations

Levis (Levis & Culler, 2002) and Rosenblum (Rosenblum & Garfinkel, 2005) proposed a definition for the JVM that includes the traditional approach, on which Virtual Machines are focused on hardware virtualization, intermediate program representation or bytecode interpretation. This means that the development of JVM is frequently specific or dedicate to the application target. It is easy understood that at the bus or sensor-level, the JVM must be re-designed, making it lighter, for it to be supported by devices with poor and constrained resources.
The Virtual Machine focused on hardware virtualization can run as an Operating System and its application, which means, works as a Monitor Program. The virtualization based on byte-code interpretation works like an application on top of a Operating System. Applications developed for that Virtual Machine abstraction can run independent of the platform Operating System. Besides this, Virtual Machines can also provide code safety, for example the JVM applications are executed in a sandbox.
From the literature related with JVM development and implementation we encounter essentially two major classification groups: JVM which works as an Operating System and JVM which acts as a Class of Middleware. The classification method chosen are based in the following issues: memory footprint (code and data (ROM and RAM)), execution method, execution model and specific application domain.
Although Maté (Levis & Culler, 2002) is considered the first JVM for sensors, authors in (Serodio et al., 1999) have proposed a thin and dedicated JVM solution for 8-bit microcontrollers. This JVM was developed essentially to perform Data Acquisition and PID

control tasks applied to agricultural environments. This JVM evolved to a new solution with some features of distributed applications with the incorporation of Jini proxies (Serodio et al., 2001). These solutions belong to the Operating System Level group.
From literature and considering only the WSN universe with poor resources, it could be incorporated in this group other solutions like: MagnetOS (Barr et al., 2002) a JVM solution for distributed WSN which have support for remote call based on RMI; Squawk (Simon et al., 2006) solution, which is a full JVM implementation with no needs of OS; TinyVM and LeJOS solution from SourgeForce which support threads and dynamic linking.
In the Middleware Level Virtual Machine it can be highlighted the following initiatives: Maté (Levis & Culler, 2002) expressly developed to be applied to sensor nodes with poor resources; Scylla (Stanley-Marbell & Iftode, 2000) assumed as the first Virtual Machine developed for mobile tiny devices; SensorWare (Boulis et al., 2003) middleware to develop Virtual Machine for Wireless ad-hoc sensor networks (WASNs); DAVIM (Michiels et al., 2006) based on Mate in which the instruction set is grouped on libraries; VM* (Koshy & Pandey, 2005) framework to develop JVM for WSNs.

5.2 An implementation for 8-bit microcontrollers

Fig. 8 shows the block diagram of a JVM implemented on a PIC18F2680, a 8-bit controller from Microchip, which has 64Kbytes of FLASH memory, used to store the JVM itself, 4Kbytes of RAM, used to save all dynamic data needed by the Java applications and by the JVM and, 1Kbyte of EEPROM, used to store the `.class` file of the Java application to execute. This JVM was implemented using C programming language.

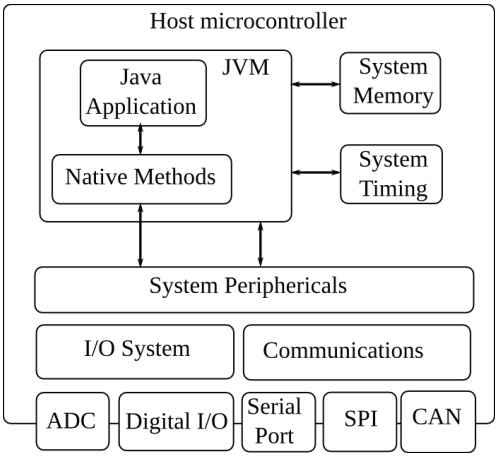
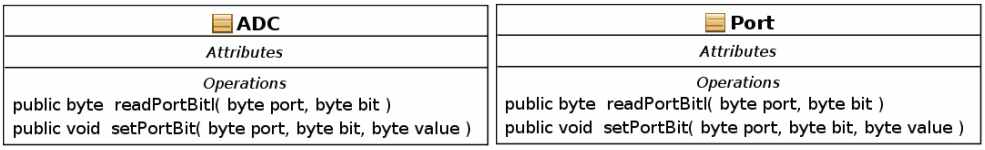


Fig. 8. Embedded Java Virtual Machine diagram block.

This system does not have an Operating System, instead the JVM running on the microcontroller acts as its Operating System. It can execute Java applications previously stored in the system FLASH memory or, it can download Java applications from the network and then execute them. Also the management of data memory used by the Java stacks and for application execution is done by the JVM.
Interaction with system peripherals is another responsibility of the JVM, either upon applications request through the implemented native methods, which includes data

acquisition from the ADC (Analogue to Digital Converter) and bit read/set of digital I/O (Input/Output), or for its own needs (timing, memory and communications).
On this prototype, communications with the outside world are made using the system console and via network interface. The system console sends its information through the microcontroller serial port and network communications are made using the microcontroller CAN interface. While console support exists in Java and therefore it was only needed to send data to the serial port in the implementation, CAN on the other hand is not supported in the standard Java distribution.
It was needed to add to the JVM and extra API to support this JVM additional functionalities of networking and data acquisition. Figure 9 shows the Java classes related with data acquisition added to the JVM API.



(a) ADC Class (b) Port Class

Fig. 9. ADC and Port classes added to API of the implemented JVM.

The ADC class has a single method used to read a channel of the microcontroller's ADC. To read the state of a single bit from a digital input pin, the Port class has the `readPortBit` and to set its value the `setPortBit` method is available. This last method is useful in actuation tasks or when it is needed to power on a sensor prior to data acquisition.
It is expected for a JVM running on a low-resource microcontroller to have worse performance than native code. To assess this performance difference in data acquisition tasks, some benchmarking tests were made to our JVM, running with a clock speed of 16MHz. Results of these test, which included reading values from the ADC and an digital input pin, are presented in Table 1. In the table are shown the results obtained using Java and native code to do the same tasks.

Operation	Java Code	Native Code	Speed
Read an input bit	368,00μs	1,70μs	210,29 ×
ADC reading,$v_{in} = V_{DD}$	4,54ms	4,22ms	1,08 ×
ADC reading ,$v_{in} = V_{SS}$	3,74ms	3,44ms	1,09 ×

Table 1. Time needed by the embedded JVM to do data acquisition tasks.

As expected the execution time of Java applications is higher than the time needed by native code to do the same task. These differences are highly noticed when reading a single bit from an input port of the microcontroller, however when reading the ADC this difference is only around 8 to 9%. It is then possible to use it on data acquisition task without much performance loss.
To be noticed that when a reading is made from the ADC, several samples are actually taken and then filtered. This procedure is made in native the code and, it is called by native code application and by the JVM, when a call to the `readADC` method is made.

6. Testing scenarios

Is this section a set of three data acquisition tests, made using the above presented concepts, are described and the achieved results are shown. These tests involved the acquisition of data from remote sensors, using Java and non-Java applications. As non-Java application used for testing, the well known LabView from National Instruments was used. While the Java applications use both the low-level and the high level device-drivers, communicating with them using Jini ERI or Java RMI, the LabView application used Web Services to interface the high-level object driver.
The set of three tests includes:

- Room temperature monitoring with a wireless sensor. To gather data from the sensor, using Web Services, a LabView application was used;
- Temperature control of a miniature greenhouse model, using two sensors and one actuator connected to a Java-based embedded system;
- Rotation speed control of a DC (Direct Current) shunt motor, using a Java based application.

In the last two tests, besides data acquisition, also actuation tasks are made. For this no changes needed to be made in the concepts, models and reference implementations presented in the previous sections.

6.1 Room temperaure monitorization with LabView

In this first test, the temperature acquisition of a room was made, over a period of 24 hours, using a LabView Applications. In Fig. 10 the scenario set-up used for this experiment is presented. The main objective of this experiment is to demonstrate the interoperability between the two different types of drivers and, the interoperability between Java and non-Java components.

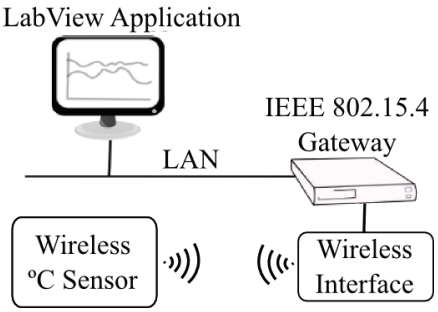


Fig. 10. Testing scenario used to monitor the air temperature using a wireless sensor.

A wireless sensor, based on a PIC18F2620 microcontroller from Microchip using, using IEEE 802.15.4 as wireless communications protocol, transmits its data to a wireless interface card. This card, based on an XBee IEEE802.15.4 transceiver, is connected to a computer where the two device drivers above presented are running: the IEEE 802.15.4 network interface device driver and the temperature sensor object driver.
On a remote computer, connected to the LAN (Local Area Network), there is a LabView application that collects data from the wireless sensor. This application communicates with

the object driver using Web Services. On Fig. 11 a screen-shot of this application is presented, showing the plot of data gathered over a period of 24 hours, with a sampling period of 5 minutes.

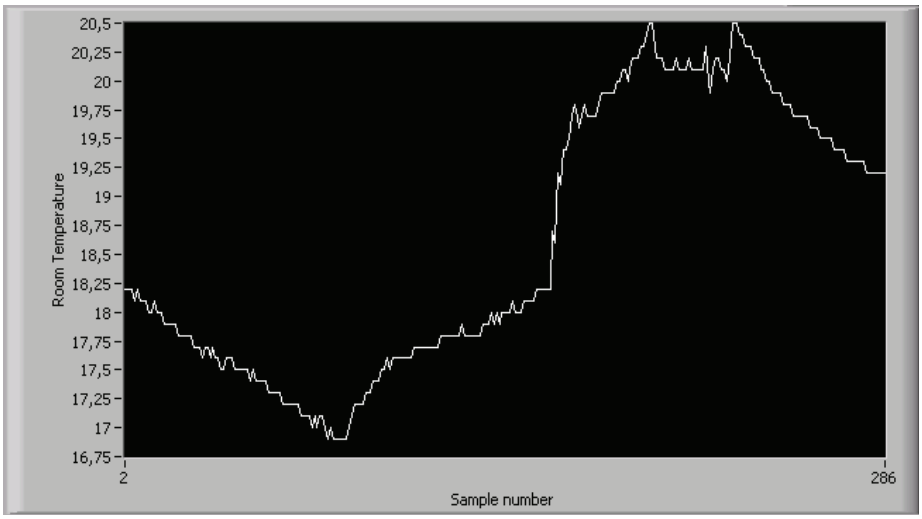


Fig. 11. Temperature acquisition from a wireless sensor using LabView.

6.2 Temperature control of a greenhouse model

This test consisted in the temperature control of a greenhouse model, using the scenario presented in Fig. 12. The objective it to control the temperature of the greenhouse, testing the proposed system in data acquisition and actuation tasks using embedded Java and the Java-based low-level device driver.

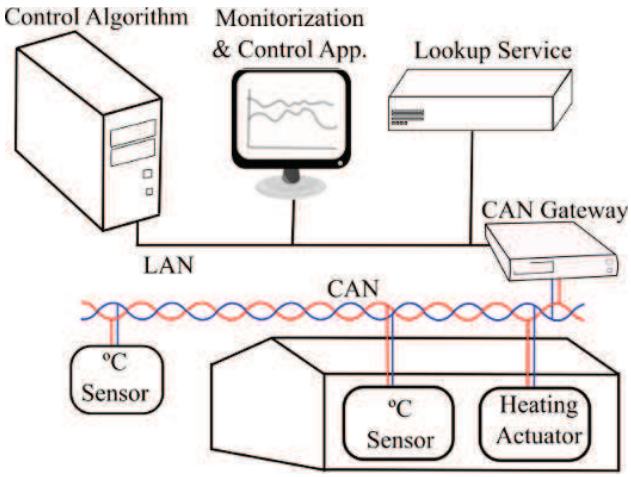


Fig. 12. Testing scenario used to control and monitor a greenhouse air temperature.

In this example two analogue temperature sensors with voltage output (LM35DZ) were plugged to two acquisition boards which were connected to a CAN bus. These boards are based on the 8-bit microcontroller PIC18F2680 and were running the embedded Java Virtual Machine. Another microcontroller-based board, also connected to the CAN bus, was used to control the heating system of the greenhouse.
A CAN gateway, running the low-level CAN device driver, is connected to the CAN bus and to the Local Area Network used in the tests. For this test, communications between applications and the CAN driver are made using Jini JERI, so a Lookup Locator Service also exists in the network. This server is used by the driver to register itself as a Jini service and by the data acquisition application to locate the services on the network.
Acquisition of temperature is made by a Java application, running on a computer connected to the Local Area Network. This application uses directly the low-level CAN driver, sending and receiving CAN message objects. The purpose of this application is to collect data from the temperature sensors (inside and outside the greenhouse) and decide if it is needed to turn on the heating inside the greenhouse.
To assess the need for heating, the Monitorization and Control Application uses an additional service connected to the network. This service, the Control Algorithm Service, is also a Jini Service that registers itself in the local Jini Lookup Locator Service. When the Monitorization and Control Application starts-up it searches, in the Lookup Locator, for the CAN device driver and Control Algorithm services.
After finding the needed services, the applications starts collecting data from the temperature sensors, sends the current temperature and set-point to the control algorithm. The control algorithm computes its output and sends it to the Monitorization and Control Application. Based on this value it regulates the energy amount to send to the heating system.
As control algorithm a P.I.D. (Proportional-Integrative-Derivative) was used to control the temperature inside the greenhouse model. In Fig 13 two plots corresponding to the temperature inside and outside the greenhouse are presented. In this test a set-point of 22°C was used and the sampling period was 1 minute.

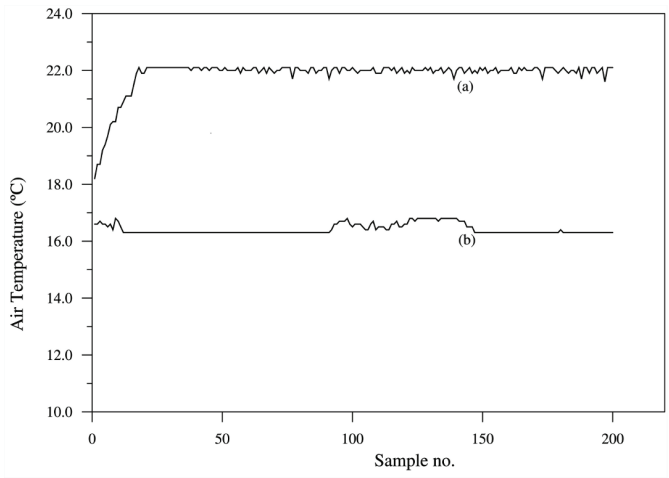


Fig. 13. P.I.D. temperature control of a miniature model of a greenhouse: a) temperature inside the greenhouse; b) temperature outside the greenhouse.

6.3 Speed control of a DC shunt motor

This last test consisted in controlling the rotation speed of a DC shunt motor, using a remote Java application. This application reads the motor speed from a sensor connected to the motor and controls its speed sending control messages to a power actuator. The scenario used to do the speed control is presented in Fig. 14.

The objective of this test is to show that with Java-based systems, it is possible to do tasks where a fast response time and low latency are needed. Speed control of a DC motor is only an example of such tasks.

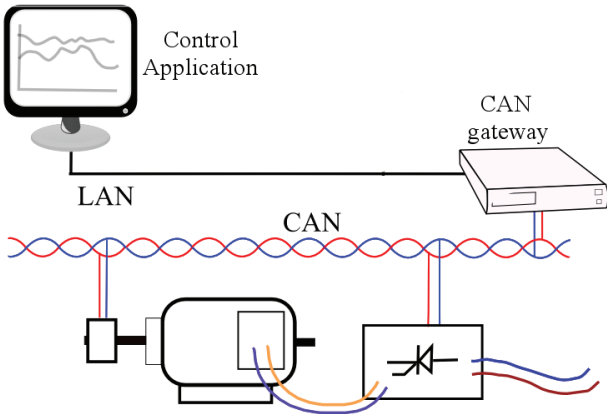


Fig. 14. Scenario used to control the speed of a DC motor.

A digital tachograph, coupled to the motor, is connected to the CAN bus, through which it sends readings of the motor rotation speed. It sends rotation speed values at a fixed rate of 5 samples per second. To be noticed that this sampling interval of 200*ms* was limited by the tachograph used in the tests, not by any constraint imposed by Java, the CAN bus speed or any other system component.

Connected to the same CAN bus there is also a controlled AC-DC converter. The output of this controller, which is remotely controlled, applies to the armature of the DC motor a variable voltage and therefore can be used to change its rotation speed.

In the scenario it also exists a computer to which the CAN interface card is connect and where the Java Device Driver is running and, in the same LAN segment, also a Java application to do the P.I.D. control of the DC motor rotation speed is running. Although this P.I.D. controller is on the same LAN segment, it is possible to connect it remotely, using the Internet, and despite the network latency it is possible to control the motor rotation speed (Silva et al., 2008).

Readings from the tachograph are sent asynchronously, i.e., when a new reading is available in the sensor it is broadcast to the CAN bus. Since the control application uses Java RMI to communicate with the device driver, it receives the speed value through the call-back mechanism. Based on the reading and the set-point, it computes the output of the P.I.D. controller and sends back to the actuator the value of the voltage to apply to the motor armature. The rotation speed of the motor is then corrected. Results from an experiment are presented in the plot of Fig. 15.

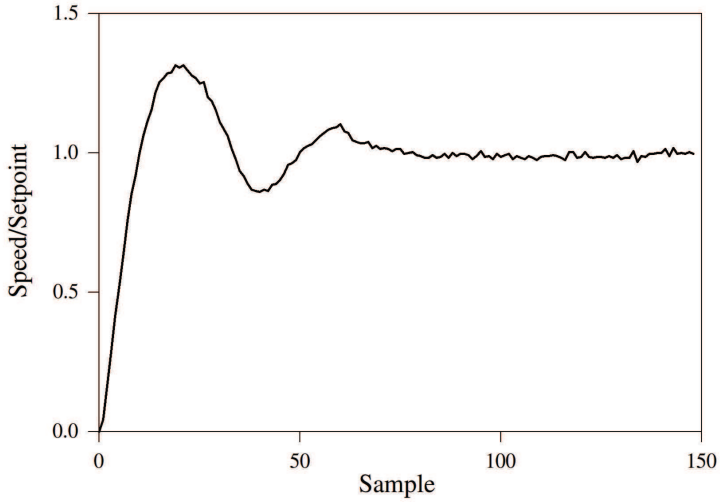


Fig. 15. P.I.D. Controll of the DC shunt motor rotation speed.

7. Conclusion

Example applications presented above demonstrate the feasibility and robustness of Java based systems for data acquisition in distributed environments, and, although Java is the core technology of the system, other execution environments can be integrated. This integration is made using a middleware technology. Being one of the most used middleware technologies, Web Services were adopted in the current implementation.

Besides these applications, authors have also been using the presented approach to Distributed Data Acquisition Systems in other work areas. Some examples of those applications include:

- DC-Motor speed control, using Matlab (Silva et al., 2008);
- Filling level gathering of recycle bins, using IEEE802.15.4;
- Wireless signal attenuation acquisition for wireless propagation studies, using IEEE802.15.4;
- Vegetation Growth detection.

The use of Java has proven to be reliable and very useful, allowing several different platforms to be used in the applications mentioned above. With this approach the examples presented in section 6 were implemented without having to concern about the target platforms. Applications and device drivers used in the testing scenarios were executed under Linux and Microsoft Windows family Operating Systems without any compatibility or performance issues.

Although Java is the preferable platform to develop the system components, and in fact it is presented as the core technology, applications developed using other tools can also be integrated in the proposed architecture. A non-Java application that interacted with the device driver using Web Services, implemented using LabView, was presented. Being a widely adopted middleware technology, Web Services allow to span the number of different platforms that can communicate with our system.

When using LabView a small wrapper application was used to convert communications between LabView and the Java-based Web Service. This wrapper was needed due to the fact that LabView does not communicate directly with Java Web Services, so a small application was developed in C# to cope with the problem. By doing this it is also shown that applications written in other programming languages, such as C#, can interact with the Object Device Driver.

At desktop and device driver levels applications have no noticeable constraints, while Java in microcontrollers can be limited by type of application (process time constants) and type of microcontroller used. In the tests made in silvopastoral and agricultural environments allowed to conclude that Java can be used in microcontrollers for this type of application without any performance constraint.

Although IEEE802.15.4 and CAN where used in our implementations, the concepts presented here can be used to implement driver to other communications technologies.

Integration of JDDAC in the layers of the proposed model and the use of IEEE 1451 are some the possibilities for future research in this project. This integration has as objective to span its compatibility, interoperability and openness levels, so desirable in a distributed data and control acquisition system.

Although tests made using a DC motor were successful, the response time of the system could not be warranted. So, another interesting future research field is the inclusion of Java Real-Time System (Sun, 2010) to deal with time critical situations. This will make the driver more reliable for systems with real time requirements.

8. References

Barr, R., Bicket, J. C., Dantas, D. S., Du, B., Kim, T. W. D., Zhou, B. & Sirer, E. G. (2002). On the need for system-level support for ad hoc and sensor networks, *ACM SIGOPS Operating Systems Review* 36(2): 1–5.

Boulis, A., Han, C.-C. & Srivastava, M. B. (2003). Design and implementation of a framework for efficient and programmable sensor networks, *MobiSys '03 - 1st International Conference on Mobile Systems, Applications and Services*, pp. 187–200.

Coulouris, G., Dollimore, J. & Kindberg, T. (2005). *Distributed Systems: Concepts and Design*, International Computer Science, 4 edn, Addison-Wesley.

Engel, G., Liu, J. & Purdy, G. (2006). *Java Distributed Data Acquisition and Control - User's Guide*, Agilent Technologies.

Gough, J. (2005). Virtual Machines, Managed Code and Component Technology, *Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05).*

Hardin, D. S. (2001). Crafting a Java virtual machine in silicon, *IEEE Instrumentation & Measurement Magazine* 4(1): 54–56.

IEEE (2006). *IEEE standard 802.15.4 – Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*.

Ito, S. A., Carri, L. & Jacobi, R. P. (2001). Making java work for microcontroller applications, *IEEE Design & Test of Computers* 18(5): 100–110.

Koshy, J. & Pandey, R. (2005). Vm*: Synthesizing scalable runtime environments for sensor networks, *Sensys - 3rd International Conference on Embedded Networked Sensor Systems*, pp. 243–254.

Levis, P. & Culler, D. (2002). Mate: A tiny virtual machine for sensor networks, *In International Conference on Architectural Support for Programming Languages and Operating Systems)*, pp. 85–95.

Mattern, F. & Sturm, P. (2003). From distributed systems to ubiquitous computing – the state of the art, trends, and prospects of future networked systems, *in* K. Irmscher & K.-P. Fähnrich (eds), *Proc. KIVS 2003*, Springer-Verlag, pp. 3–25.

Michiels, S., Horré, W., Joosen, W. & Verbaeten, P. (2006). Davim: a dynamically adaptable virtual machine for sensor networks, *MidSens '06 - Proceedings of the international workshop on Middleware for sensor networks*, pp. 7–12.

Mitra, N. & Lafon, E. Y. (2007). *SOAP Version 1.2 Part 0: Primer*, W3C Recommendation. URL: http://www.w3.org/TR/soap12-part0/

Newmarch, J. (2006). *Foundations of Jini 2 Programming,* APress.

Pfeffer, M. & Ungerer, T. (2004). Dynamic real-time reconfiguration on a multithreaded java-microcontroller, *Proceedings of Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 86–92.

Robert Bosch GmbH (ed.) (1991). *CAN Specification*.

Rosenblum, M. & Garfinkel, T. (2005). Virtual machine monitors: Current technology and future trends, *Computer* 38: 39–47.

Serôdio, C. M., Silva, P. M. M. A. & Monteiro, J. L. (2007). A Java Virtual Machine for Smart Sensors and Actuators, *Proceedings of 2007 IEEE International Symposium on Industrial Electronics (ISIE'07)*, Centro Cultural and Centro Social Caixanova - Vigo, Spain, pp. 1514–1519.

Serodio, C., Silva, P., Couto, C. & Monteiro, J. (1999). Embedded java in agricultural control systems, *IECON '99 - The 25th Annual Conference of the IEEE Industrial Electronics Society*, Vol. 2, pp. 716–721.

Serodio, C., Silva, P., Couto, C. & Monteiro, J. (2001). Embedded java to enable jini facilities in agricultural networked systems, *IECON '01 - The 27th Annual Conference of the IEEE Industrial Electronics Society*, Vol. 1, pp. 255–260.

Silva, P., Serôdio, C. & Monteiro, J. (2008). A java-based controller area network device driver for utilization in data acquisition and actuation systems, *ISIE 2008 - IEEE International Symposium on Industrial Electronics, 2008*, pp. 1855–1860.

Simon, D., Cifuentes, C., Cleal, D., Daniels, J. & White, D. (2006). JavaTM on the bare metal of wireless sensor devices: the squawk java virtual machine, *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pp. 78–88.

Sommer, F. (2003). *Call on extensible RMI: An introduction to* JERI. URL: JavaWorld, Online – http://www.javaworld.com/javaworld/jw-12-2003/jw-1219-jiniology.html

Stanley-Marbell, P. & Iftode, L. (2000). Scylla: A smart virtual machine for mobile embedded systems, *WMCSA2000 - 3rd IEEE Workshop on Mobile Computing Systems and Applications*, pp. 41–50.

Sun (2003). *Java Technology Concept Map.* URL: http://java.sun.com/developer/onlineTraining/ new2java/javamap/

Sun (2005). *Sun SPOT System: Turning Vision into Reality.* URL: http://labs.oracle.com/spotlight/SunSPOTSJune30.pdf

Sun (2010). *Sun Java Real-Time System.* URL: http://java.sun.com/javase/technologies/realtime/rts/

Sun, X.-H. & Blatecky, A. R. (2004). Middleware: the key to next generation computing, *Journal of Parallel and Distributed Computing* 64(6): 689 – 691. YJPDC Special Issue on Middleware.

Data Acquisition

Edited by Michele Vadursi

ISBN 978-953-307-193-0
Hard cover, 344 pages
Publisher Sciyo
Published online 28, September, 2010
Published in print edition September, 2010

The book is intended to be a collection of contributions providing a birdâ€™s eye view of some relevant multidisciplinary applications of data acquisition. While assuming that the reader is familiar with the basics of sampling theory and analog-to-digital conversion, the attention is focused on applied research and industrial applications of data acquisition. Even in the few cases when theoretical issues are investigated, the goal is making the theory comprehensible to a wide, application- oriented, audience.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Pedro Mestre, Carlos Serôdio, João Matias, João Monteiro and Carlos Couto (2010). Java in the Loop of Data Acquisition Systems, Data Acquisition, Michele Vadursi (Ed.), ISBN: 978-953-307-193-0, InTech, Available from: http://www.intechopen.com/books/data-acquisition/java-in-the-loop-of-data-acquisition-systems

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821