DYNAMIC JAVA COMPONENTS IN PERVASIVE SYSTEMS

A review of the feasibility of dynamic data processing on wireless platforms

Alexander Dannies[1], Javier Palafox-Albarrán[1], Walter Lang[1] and Reiner Jedermann[1]
[1]*Institute for Microsensors, -actuators and -systems, University of Bremen, Otto-Hahn-Allee NW1, Bremen, Germany*
{adannies, jpalafox, wlang, rjedermann}@imsas.uni-bremen.de

Proc. of PECCS 2012 - International Conference on Pervasive and Embedded Computing and Communication Systems, Rome, Italy, 2012, DOI:10.5220/0003802300580066

Keywords: java:data processing:wireless sensors:OSGi:dynamic components:pervasive computing

Abstract: A wireless sensor network (WSN), which is one type of pervasive system, has the goal of networking heterogeneous systems and communicating through a gateway. However, it is also necessary to provide dynamic features to wireless nodes for updating applications and services during runtime. Dynamic updates can be handled either by the intrinsic features of Java or by advanced frameworks such as MIDP or OSGi. This paper investigates the software background and the feasibility of these three options in the context of WSNs. Java Virtual Machines were tested on sensor nodes and gateways currently available on the market. Two synthetic benchmarks were utilized to compare their performance. In addition, we tested the performance of an exemplary algorithm for a real life application during transportation in food logistics. Our experimental results showed that the performance of the benchmarks varied by a factor of more than 50, depending on the platform. Nevertheless, our chosen example algorithm could be executed on all platforms within an acceptable amount of CPU time. Pre-processing of data can be applied on wireless devices to reduce communication volume and provide conclusions instead of raw data. However the use of advanced frameworks, enabling extended dynamisation, are so far very limited.

1 INTRODUCTION

The vision of omnipresent information processing in everyday environments, for example in logistic transport and production processes by a network of pervasive systems, becomes more and more feasible with the decreasing cost and increasing computational power of wireless devices.

Pervasive systems have the aim of networking everyday life by the use of intelligent objects. They can utilize different technologies and systems with different computational capabilities. Wireless sensor networks (WSN) are examples where different hardware types are involved. These devices communicate with the external world through a gateway which possesses extended communication and processing capabilities.

The nodes are placed in an environment in order to sense physical parameters, communicate with the network's neighbours and determine useful actions. One of the most important challenges is to deploy applications in which families of hardware devices interact with each other. As mentioned by Vazquez, Almeida, Doamo, Laiseca and Orduña (2009), it is desirable to implement monitoring solutions based on WSNs with distributed intelligence in which sensor populated scenarios may communicate with internet-based solutions.

Our vision of an intelligent network, in the project "Intelligent Container" (IMSAS, 2011), includes the capability to handle dynamic changes in the environment. The network nodes should be able to interpret and to react to environmental data and situations that were unknown at the time of their installation and initial programming as access to the container is not possible during transport. Therefore it is necessary to provide dynamic features in wireless network nodes to allow updating during runtime, not only of one application but also the services provided. For example, the sensor can be moved to a new environment, which requires a modified network protocol to make the best use of the available gateways, whereas the remainder of the software remains unchanged. Alternatively, the intelligent data processing may consist of multiple

services which are updated at different time intervals.

The concept of the Open Source Gateway initiative (OSGi) enables Java software modules to be updated or replaced on a system during runtime, for example remotely over an internet-connection.

Research has been carried out including OSGi in pervasive systems, but the main focus is telematics systems (Lee et al., 2006) and mobile agents (Lee et. al., 2010) as its merits make it suitable for these purposes.

Implementing dynamic features in WSNs is even more complicated. The difference between WSNs and other pervasive systems is that the main concern is the energy consumption of the application. In Rellermeyer et al. (2008) they go one step further, making the concept of OSGi compatible with resource-constrained devices. They propose OSGi-like services in devices with no JVM and with limited operating system support. The only full OSGi framework exists on the gateway.

The question arises whether it is possible to have a full OSGi heterogeneous system which is energy aware and intelligent, or whether pure Java without a middleware on top is sufficient at the lower levels.

The aim of this paper is to investigate the feasibility of OSGi and Java implementation in several WSN platforms. Verifying this would allow a shift in the paradigm from a centralised WSN – in which the majority of sensors are just collecting environmental data and transmitting these to one processing unit – to a WSN with sensors which are pre-processing data to reduce network traffic and save energy. The requirements for a specific application can differ from our selected application domain of interest, which is monitoring during transportation in food logistics. The main issues to be taken into account are the time needed for the platforms to perform complex computer operations and the limitations of each of them regarding resource requirements for enabling a dynamic platform.

First, in section 2 we will provide an overview of Java and dynamic components. Section 3 will explain the methods used to test the performance and the configuration, followed by section 4 which describes the hardware platforms used in the experiments. In section 5 we discuss the results and finally offer conclusions.

2 STATE OF THE ART

Java and OSGi are two promising technologies for solving the challenge of dynamic software updates. The former is well-known to be portable and independent of platform. Pervasive Java (also called wireless Java and mobile Java) attempts to bridge the gap between different devices and platforms. However, Java Standard Edition (JavaSE) was conceived for personal computers and is not suitable for resource constrained devices. Realising this need, Sun Microsystems introduced the Java 2 Platform, Micro Edition. The second system, OSGi, is based on Java with a focus on modular, service-oriented architecture (SOA) as well as dynamic installation and updates of bundles.

2.1 Java as dynamic language

Java is the most common language that meets the requirement of the ability to extend software with dynamic code segments, because dynamic class loading is one of its intrinsic features. Some of the additional benefits one obtains by choosing Java are object-oriented programming and automated memory and heap management (garbage collector). Furthermore, using Java as a programming language is programming platform-independent. This so called concept of “write once, run anywhere” (WORA) becomes possible through the use of a Java virtual machine (JVM) which is adapted to specific hardware architecture, e.g. x86 or ARM. The JVM does not execute a compiled program, but rather a Java bytecode which will be interpreted differently on each platform.

2.1.1 Limitations of JavaME

It has to be questioned whether JavaME is still sufficient for the increasing demands of data processing algorithms. The situation-dependent loading and installation of new software bundles requires flexible class loaders. Storage of recorded data and knowledge might require a file system. Most data analysis algorithms require floating or even double precision arithmetic.

JavaSE is the standard platform for programming in the Java language. It consists of a Java virtual machine for executing the compiled class-files (the Java program) and a set of libraries which makes it possible to access, e.g. the file system, graphical or network interfaces from a Java program.

The Java Micro Edition (JavaME) is aimed at embedded systems, e.g. mobile phones with limited resources. A developer programming for a platform

using JavaME has to keep in mind that he is restricted to the features of JRE 1.3. Three components belong to the JavaME-stack: the configuration which contains the Java virtual machine, the profile, which adds a certain set of API, and optional packages for additional functionality in regards to the profile scope. There are two configurations: The Connected Device Configuration (CDC) includes almost the entire scope of JavaSE except for GUI-related libraries. In contrast, the Connected Limited Device Configuration (CLDC) only contains the minimum amount of classes necessary to enable the operation of a JVM. In version CLDC 1.1, the previous version has been extended by the classes double and float, so that floating-point operations now are enabled. On the other hand, the Java-math-library is still not available.
On top of the configuration, a profile can be chosen which fits the desired target application. For example, the Mobile Information Device Profile (MIDP) for mobile devices such as mobile phones, or the Personal Profile for consumer electronics. Applications written based on the former profile are called MIDlets.

2.2 Java Virtual Machines

In this paper, four selected JVMs are installed in diverse sensor nodes in order to be tested, namely JamVM (Lougher, 2010), JamaicaVM (Siebert, 2002), Squawk (Oracle, 2011a) and SwissQM (Müller, Alonso and Kossmann, 2007). The first two JVMs are based on the JavaSE standard. The third is based on the JavaME standard, and the fourth was specifically developed for sensor nodes.

JamVM is an open source JVM which makes use of the GNU Classpath (Lougher, 2010). The developers claim that their implementation is extremely small and still able to support the full specification, including class-unloading and native support. It supports several operating systems such as Linux, Mac and Solaris as well as different hardware architectures like PowerPC, ARM and AMD64.

JamaicaVM is a commercial JVM of AICAS. It provides, according to the developer, Hard Realtime Execution, Realtime Garbage Collection, the best trade-off between runtime performance and code size, dynamic loading, multi-core support, and native support. It is available for diverse operating systems like Linux and Windows, and several architectures like x86 and ARM. In addition to the traditional method of building a JVM and executing a class-file, the JamaicaVM Builder offers another solution. All files relevant to the application (a set of class files) and the Jamaica VM are combined into a standalone application in a single executable file.

Squawk is a JavaME VM which targets small resource constrained devices. Its core is mostly written in Java. Classes are not transferred directly to the execution environment but combined in a suite and prelinked to each other, which results in a reduced size of around one third of the original size. The omission of dynamic class loading in these immutable suites significantly decreases the start up time of the applications. Squawk utilizes the concept of isolates, where an application can be represented as an object. This allows common suites to be shared between multiple applications that run in the single JVM which can lead to a significantly reduced memory footprint.

Finally, SwissQM is a combination of a JVM and a query machine (QM) that was developed specifically to be run on platforms with TinyOS as operating system, such as TelosB (Crossbow, 2011). It is based on 16-bit integer values, and floating-point types are not supported. It only supports 37 instructions of the instruction set of the JVM specification, and adds 22 specific instructions related to processing queries. Programs are executed at every sampling period or when data from other nodes arrive. It has a small footprint – 33kB of Flash and 3kB of SRAM memory – and is able to execute up to six QM programs concurrently in TelosB, but, in principle, it can run an arbitrary number of concurrent programs.

2.3 Management of software bundles

Furthermore, several Java based frameworks can be found that make the exchange of software bundles easier. Software agent platforms such as the Java Agent DEvelopment Framework (JADE) (Bellifemine, Caire, Poggi, and Rimassa, 2008), a Mobile Agent Platform for WSNs based on JavaSun Spots (MAPS) (Aiello, Fortino, Gravina, and Guerrieri, 2011), and the Agent Factory Micro Edition (AFME) (Müller, 2007) are mainly found in academic research.
There are two different application models (AM) for the Java language. On the one hand, the unmanaged approach exists, which is basically the start of the static method main() when executing a class file. On the other hand, there are several concepts for managed AMs like applets (for web browsers), Xlets (for advanced content on Blu-ray discs), MIDlets

(for mobile devices like mobile phones), OSGi Bundles and the OSGi R4 Mobile Expert Group (MEG) AM.
Here, we wish to compare the two concepts of MIDlets and OSGi Bundles.

2.3.1 MIDlets

A device running a Mobile Information Device Application (called a MIDlet) has an environment which enables the user to choose MIDlets for installing, starting and removing. This so called Application Management Software (AMS) to control the life-cycle is responsible for the interaction with the user as well as for error handling. MIDlets have three possible states: paused, active and destroyed.
To achieve modularity with MIDlets, a method needs to exist which enables the communication between different software modules which can be updated independently of each other. For this the following two approaches are possible. On the one hand, the Record Management Store (RMS) enables indirect communication between MIDlet-suites (introduced in MIDP2.0) and on the other hand, Inter-MIDlet-Communication (IMC) enables direct communication between MIDlets (introduced in MIDP3.0). However, at the moment there is no SDK available, which supports MIDP3.0. The question can be raised whether it will ever be available because the standard was approved in the end of 2009 but has not yet been introduced in SDKs. This is reasoned by the fact that the development of applications for mobile devices has shifted to android applications and the like. With the RMS-approach, every MIDlet can access record stores created of other MIDlets in the same suite. Starting from MIDP2.0 it is also possible that MIDlets of a second suite outside the first suite access the record store, but only when allowed by the MIDlet which created the record store. The data inside the record store stays as long as the suite stays on the device, independent of the status of the MIDlet (active or paused). If the device is switched off and then on later, the data is still available. When a MIDlet suite is removed from a device the record store is no longer available.

2.3.2 OSGi

Sometimes just running a Java-application on a device is not enough because the modularity is missing, for example when new implementations have to be integrated into the running program. To add this functionality, the Open Services Gateway initiative framework (OSGi) can be used. This concept enables the remote update or replacement of Java software modules on a system during runtime, for example over an internet-connection. So the concept of OSGi enables a high degree of dynamics because it is possible to update an existing piece of software running on a device without being on site or restarting the machine.
In order to be able to use the concept of OSGi an OSGi framework on top of a Java Virtual Machine (JVM) is necessary. This framework utilizes a so-called Service Registry to make it possible to modularise and manage applications and their services with the help of a component model by using bundles and services. On the one hand there are open source frameworks available for the current specification “OSGi Service Platform Release 4“ (R4), namely Apache Felix, Eclipse Equinox and Makewave Knoplerfish and next to it a R3 implementation named Concierge from the Institute for Pervasive Computing, ETH Zurich. On the other hand there are commercial OSGi frameworks available, e.g. ProSyst Software mBedded Server and Makewave Knopflerfish Pro to name two of them.
It is possible to run OSGi on mobile phones, e.g. with Linux as an OS and JavaME VM with the CDC configuration and FP 1.1. Even though OSGi was originally designed for environments with CDC or above, it is generally possible to implement OSGi in a CLDC environment. The “mBS Mobile for JavaME” from ProSyst proves that, but with the limitation to version R3 of the OSGi specification. The platform requirement is a J2ME CLDC/MIDP 2.0 compatible JVM. Typically OSGi R4 needs 8 MB volatile memory and a CPU clock speed of at least 150 MHz to operate (Prosyst, 2010).
The basic feature of OSGi is to enable access to certain Java classes which are published as services, while protecting the remaining code against external access, which largely depends on user defined class loaders.
The inclusion of user defined class loaders produces a lot of overhead for the JVM, which is the reason why they were excluded from the CLDC profile. This raises the challenge to find another mechanism to implement code sharing and isolation in an optimised OSGi framework.
The Request for Proposal (RFP) 126 (Bottaro and Rivard, 2009) suggests a new approach – the implementation of a new OSGi specification which aims to replace a profile, e.g. MIDP. So on top of the CLDC1.1 configuration, the OSGi ME Profile is used to add an OSGi environment to resource-constrained environments. In addition to keeping the

core features of the OSGi technology and the Java ME CLDC compliance, the proposition includes a strengthened robustness and a simplification which requires much less resources and thus enables mass deployment. Although the RFP sounds promising it remains to be seen whether the concept offers what it promises. In 2010 it was mentioned in Bottaro and Rivard (2010) that the specification would soon be public and that the company IS2T was about to sell the first OSGi ME. In autumn 2011 the specification was made public and IS2T has a reference implementation and sells early solutions. The question arises if and when an open-source solution will be accessible.

3 METHODS

In order to have a reference point for the abilities of the different platforms (see section 4) we are using two synthetic benchmark-algorithms: Dhrystone which uses integer values and LINPACK which uses floating point values. Due to the fact that the whetstone benchmark uses trigonometric functions it is not possible to execute it on platforms with CLDC1.1, at least without the use of external classes. So as a replacement we used the LINPACK benchmark.

In addition, we tested the performance of an exemplary algorithm for the analysis of sensor data as an example of a real life application.

3.1 Feedback-Hammerstein

The exemplary algorithm estimates the three parameters required for a model that represents the factors affecting the temperature inside a refrigerated container transporting perishable goods, and does not need any matrix inversion. The so-called Feedback-Hammerstein system takes into account the effect of organic heat using a static non-linear feedback system. In order to provide an accurate prediction, the model parameters have to be iterated over three days at a measurement interval of one hour, equivalent to 72 cycles. Details of the algorithm are described in Palafox-Albarrán, Jedermann and Lang (2011).

3.2 Test configurations

There are three options available to run Java code.
Option one is the compilation of the Java code into the Java VM. This is mainly used for standard functions and services. Whilst this approach results in fast execution speed, it does have some drawbacks. Firstly, it is no longer possible to update the code. And secondly, the code size increases because the compiled native code is less memory-efficient than byte code. The second option is that the code is interpreted at run-time by the VM. This can be a class-file or jar-file. The third option is that the code is compiled just-in-time (JIT) before execution. This is especially beneficial for classes, which are often called or contain a high number of nested loops. Which of these options are available depends on the implementation of the VM.
The open-source solution JamVM only enables the second option. In constrast to the open-source solution, the JamaicaVM implements the Realtime Specification for Java (RTSJ) and a deterministic garbage collection. It also supports all three options described above. The footprint and execution time of a built-in-application (option 1) can be influenced immensely by choosing the correct settings for the parameters in the building process, such as heapsize, timeslicing, stack size, percentage compiled or number of threads, just to name some of the many options available.

3.2.1 OSGi

A preliminary OSGi implementation on the sensor platform Imote2 (Memsic, 2011) was conducted by Wessels, Jedermann and Lang (2010). Back then, an executable was built with the JamaicaBuilder which contains the Jamaica VM in version 3.4 and the Equinox Framework in version 3.4
In addition to this we used the current version (6) of the JamaicaVM and the equinox framework in version 3.7. The newer JamaicaVM has the capability of creating a JVM with an integrated just-in-time (JIT)-compiler. Furthermore an open-source solution of a JVM was used, namely JamVM in version 1.5.4 (with classpath in version 0.98). These configurations were utilized for the tests on both ARM-platforms (see section 4 for details). On the x86- platform both JVM were tested.

3.2.2 MIDlets and Java archives

On the other sensor platforms OSGi implementation was not possible. On Preon32 and SunSPOT the benchmarks were written in Java in their corresponding development environments and then transferred to the devices as MIDlets or Proglets (Virtenio VM) respectively.
In SwissQM, the user query is run as a bundle in the SwissQM gateway and not in the sensor node. It sends bytecode to the sensor network for query processing purposes on the sensor nodes. The

limited VM on the sensor nodes is not capable of running complex operations; user defined functions are possible but they are written in a C-like language and only integer types and no arrays are supported. This makes it very difficult to program even simple algorithms like Feedback-Hammerstein as it requires at least simple matrix operations like addition and multiplication.

4 INTRODUCTION OF TESTED HARDWARE PLATFORMS

Tables 1 and 2 contain the different platforms we used with their properties.

Table 1: 802.15.4 Wireless sensor platforms

	TelosB	Imote2	Sun SPOT	Preon32
CPU (MHz)	MSP 430 (8)	PXA 271 (416)	SAM 9G20 (400)	Cortex-M3 (72)
RAM	10 kB	32 MB	1 MB	64 kB
OS	TinyOS	Linux	None	None
JVM	Swiss QM	any	Squawk	Custom
Java Edition	Reduced byte code	SE	ME CLDC 1.1	ME almost CLDC 1.1

Table 2: Telematics platforms

	DuraNAV	VTC6100
CPU (MHz)	PXA255 (400)	N270 (1600)
RAM	64 MB	1 GB
OS	Linux	Linux
JVM	Any	Any
Java Edition	SE	SE
OSGi	Any	Any

4.1 Wireless sensor platforms

All the tested wireless sensor platforms (Table 1) support the communication standard 802.15.4 (IEEE Computer Society, 2006).

The sensor node TelosB with the smallest memory footprint and the slowest CPU from our selected test field runs TinyOS, a small, open-source, energy-efficient software operating system. It is an event driven OS designed for sensor network nodes that have limited resources. TinyOS is an embedded operating system written to provide interfaces and components for common abstractions such as packet communication, routing, sensing, actuation and storage. Applications are built using nesC, an extension of the C programming language optimised for the memory limits of sensor networks. Support, libraries are already available, such as the nesC compiler.

The advantages of Imote2 are high processing capabilities in comparison with TelosB, a large memory, and power saving abilities by the scalable processor. The design is modular and stackable with interface connectors for expansion boards on both the top and bottom sides, it allows the interconnection of additional devices, such as temperature sensor cards.

Regarding the operating system, it can run Linux (e.g. built with OpenEmbedded) that allows exhausting the possibilities of Imote2 and the development of higher application software which enhances single motes.

The company Virtenio has developed a new concept of a sensor node (Preon32). By using a 32-Bit ARM-Cortex-M3 microcontroller in combination with a Java virtual machine for the device it becomes possible to execute Java applications on the sensor node. The limiting factor is the power of the CPU which has a maximum clock of 72 MHz and also RAM of 64 KB in size. Access to hardware components like the radio is written in C and can be used through the Java Native Interface (JNI). For wireless communication the ATRF231 radio chip is utilized. Support for IPv6 is currently under development. In addition, this sensor node has the possibility of being connected to the CAN-Bus – the support for this feature is also currently under development.

The Sun Small Programmable Object Technology (SunSPOT) is a mote developed by Sun Microsystems (Oracle, 2011b), currently available in an 8th revision. It features support for IPv6. The difference with respect to all the other platforms is the built-in lithium-ion-battery, which can be charged via the USB-interface.

4.2 Telematics

Beside the wireless sensor we also tested two telematics units for two reasons: Firstly, they can bridge the gap between the local sensor network and

the outside world. This becomes possible by the use of WLAN or UMTS for enabling access to the internet. In order to make it possible to connect the telematics unit to the WSN, one wireless sensor acts as a base station. The connection is established via a serial connection (RS232 or USB). Secondly, we include them as a reference for embedded platforms with extended computation capabilities. Telematics units are typically used to supervise goods inside trucks or containers; they collect data from local sensors and send compressed messages over the GPRS network; but the typical hardware units are not restricted to these applications.
We have not tested Java-enabled cell phones or smartphones based on the Android operating system, because we have only focused on embedded devices which can operate without human involvement.

5 PERFORMANCE MEASUREMENTS

Micro benchmarks or mini test applications can only give a rough estimation of the capabilities of a system. The performance of a real-world application is almost always different. It should also be noted that in cases where a JIT-compiler is used, some warm-up-time is required, and the result of such benchmarks is thereby affected. So the execution time in the long run – after the warm-up phase – can be much faster. Results of real-world tests at extended periods of time are more meaningful.
Table 3 presents the results for the different benchmarks executed on the reference platform VTC.

Table 3: Results of the chosen benchmarks on the reference platform (VTC)

Dhrystone	Linpack	Feedback-Hammerstein
523 ms	45,778 Mflops/s	7 ms

Combined with the resulting performances of the chosen benchmarks on the other platforms, ratios were calculated, which are displayed in the following figures.

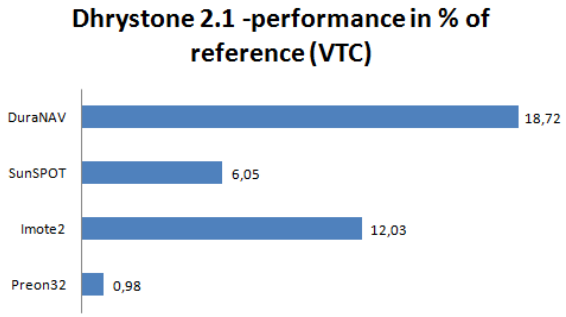

Figure 1: Results of the Dhrystone 2.1 benchmark

As can be seen in Figure 1, the processing power of the platforms is correlating with the CPU and RAM at hand. Although the CPU clock-rate of DuraNAV, Imote2 and SunSPOT are the same, the performance seems to be linked to the available RAM of the systems. So the order of the performances is the same as the devices sorted by memory in descending order.

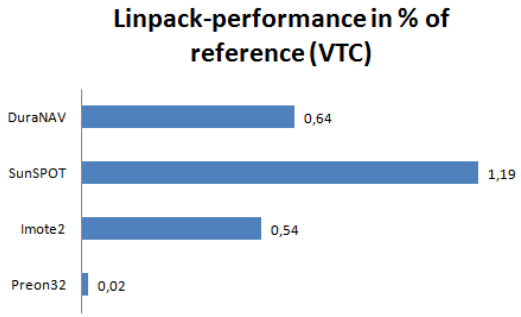

Figure 2: Results of the Linpack benchmark

In contrast to the previous benchmark, Figure 2 shows a different outcome. Here the requirement for RAM seems to be less significant. The SunSPOT performance is the best followed by DuraNAV and Imote2. This could be explained by the newer CPU architecture of SunSPOT which seems to have improved floating-point processing power.

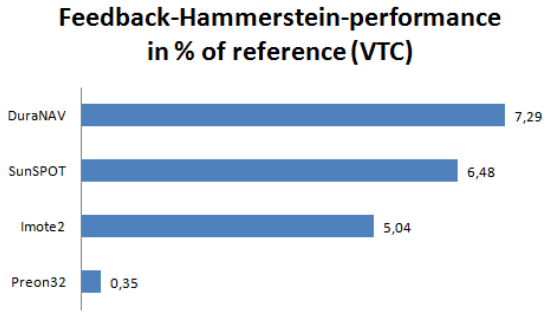

Figure 3: Results of the Feedback-Hammerstein algorithm

Figure 3 depicts the results for the exemplary tested real-world algorithm. Here the best performance is produced by the DuraNAV (96 ms) followed by SunSPOT (108 ms) and Imote2 (139 ms). A correlation between the need for memory or processing capabilities cannot be directly determined.

In all three benchmarks, Preon32 has the poorest performance which is obviously caused by the extremely low amount of available RAM in combination with a low clock-rate. It is possible to pre-process data with this node with chosen algorithms, whilst keeping in mind that the resources are extremely limited. Although the execution time for the Feedback-Hammerstein-algorithm was around 285 times slower than on the reference platform, the absolute value of 2027 ms is still fast enough to be applicable, because it is only executed every three days.

SunSPOT exhibits a good overall performance if the algorithms are not dependent on large amounts of memory.

Figure 4 compares the execution time on Imote2 and DuraNAV of the benchmarks as a class-file or an OSGi-bundle. The execution time of the Dhrystone benchmark on the Imote2 is around 8% slower when deployed as an OSGi-bundle instead of being executed from a class-file. In contrast, the execution time of the Feedback-Hammerstein-algorithm as an OSGi-bundle is around 10% faster on DuraNAV and around 6.8% faster on Imote2 than when executed from a class-file.

The dynamics is higly improved with the usage of an OSGi-framework, so that a trade-off in a reduction in execution time can be accepted.

It remains to be seen how the execution time varies on SunSPOT when an OSGi implementaion is possible.

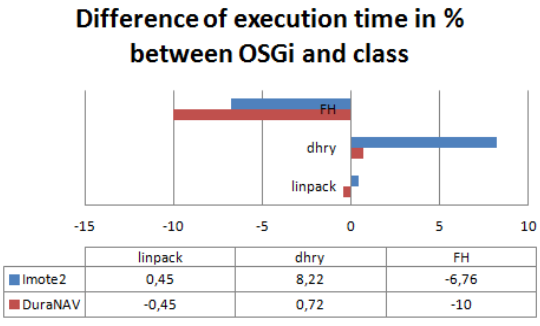

Figure 4: Difference in execution time in % between OSGi and class

6 CONCLUSIONS

The selection of the right framework and platform depends on the requirements of the application. They can be divided into two: those requiring computations of complex time-consuming operations and those requiring large amounts of memory.

At the moment, OSGi implementation is not applicable for pervasive networks in the way that is desired. Only Imote2 has enough resources for making an OSGi-framework available on a sensor-platform. It can manage complex mathematical operations and store large amounts of data. But due to discontinued development since it was introduced to the market, its power consumption remains relatively high. Therefore the platform can be used for scientific evaluation but is not applicable for remote WSNs where one of the main features should be long battery life-time. The mentioned request for proposal for a JavaME implementation of OSGi sounds promising, but it is not yet available.

Data pre-processing of sensor nodes reduces communications resulting in lower energy consumption. Although the execution times of Preon32 for the chosen algorithms were not the best, it can be used for calculations which do not rely on a lot of data because of the limited amount of RAM. SunSPOT can be used for data-processing with even more data because of the increased amount of available memory. Especially on floating-point algorithms, the newer CPU has its advantages. It will have to be assessed how the OSGiME profile changes the application of dynamic components in WSNs.

ACKNOWLEDGEMENTS

The research project “The Intelligent Container” is supported by the Federal Ministry of Education and Research, Germany, under reference number 01IA10001, and by the German Research Foundation (DFG) as part of the Collaborative Research Centre 637 “Autonomous Cooperating Logistic Processes”.

REFERENCES

Aiello, F., Fortino, G., Gravina, R. & Guerrieri, A. (2011). A Java-based agent platform for programming wireless sensor networks. *Computer Journal*, 54, 439-454.

Bellifemine, F., Caire, G., Poggi, A. & Rimassa, G. (2008). JADE: A software framework for developing

multi-agent applications. Lessons learned. *Information and Software Technology*, 50, 10-21.

Bottaro, A. & Rivard, F. (2009). *RFP 126 - OSGi ME: An OSGi Profile for Embedded Devices*. OSGi Alliance.

Bottaro, A. & Rivard, F. (2010). *OSGi ME An OSGi Profile for Embedded Devices.* OSGi Community Event, London, UK, September 2010.

Breymann, U., Mosemann, H. (2008). Java ME : Anwendungsentwicklung für Handys, PDA und Co. München; Wien: Hanser

Crossbow (2011). TelosB Datasheet [Online]. Retrieved September 5 2011, http://www.willow.co.uk/TelosB_Datasheet.pdf

IEEE Computer Society. (2006). Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (WPANs) [Online] Retrieved September 6, 2011, from http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf.

IMSAS . (2011). „The intelligent container: Networked intelligent objects in logistics". [Online]. Retrieved September 6, 2011, from http://www.intelligentcontainer.com

Jedermann, R., Palafox-Albarrán, J., Jabbari, A. & Lang, W. (2011). Embedded intelligent objects in food logistics - Technical limits of local decision making. In: Hülsmann, M., Scholz-Reiter, B. & Windt, K. (eds.) *Autonomous cooperation and control in logistics. Berlin: Springer*

Lee, S., Kim, I., Rim, K. & Lee, J. (2006). Service mobility manager for OSGi framework. In: Gavrilova, M., Gervasi, O., Kumar, V., Tan, C., Taniar, D., Laganá, A., Mun, Y. & Choo, H. (eds.). *Computational science and its applications - ICCSA 2006.* Berlin / Heidelberg: Springer.

Lee, J., Lee, S.-J., Chen, H.-M. & Lee, W.-T. (2010). Telematics services through mobile agents. In: Zeng, Z. & Wang, J. (eds.). *Advances in neural network research and applications.* Berlin / Heidelberg: Springer.

LOUGHER, R. (2010). *JamVM* [Online]. Retrieved September 6, 2011, from http://jamvm.sourceforge.net.

Memsic (2011). Imote2 High-performance Wireless Sensor Network Node [Online]. Retrieved September 5 2011, from http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=134%3Aimote2.

Müller, R., Alonso, G. & Kossmann, D. (2007). A virtual machine for sensor networks. *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. Lisbon, Portugal: ACM.

Oracle. (2011a). *Squawk Development Wiki* [Online]. Retrieved September 6, 2011, from http://java.net/projects/squawk/pages/SquawkDevelopment.

Oracle. (2011b). *SunSPOT* [Online]. Retrieved September 6, 2011, from http://www.sunspotworld.com.

Palafox-Albarrán, J., Jedermann, R. & Lang, W. (2011). Energy-efficient parameter adaptation and prediction algorithms for the estimation of temperature development inside a food container. In: Cetto, A. J., Ferrier, J.-L. & Filipe, J. (eds.) *Lecture Notes in Electrical Engineering - Informatics in Control, Automation and Robotics*. Berlin: Springer.

ProSyst. (2010). *The world's smallest OSGi solution.* [Online]. Retrieved September 6, 2011, from http://www.prosyst.com/index.php/de/html/news/details/18/smallest-OSGi.

Rellermeyer, J. S., Duller, M., Gilmer, K., Maragkos, D., Papageorgiou, D. & Alonso, G. (2008). The software fabric for the Internet of Things. In: Floerkemeier, C., Langheinrich, M., Fleisch, E., Mattern, F. & Sarma, S. E. (eds.) *Proceedings of the IEEE International Conference on the Internet of Things*. 26-28 March 2008, Zurich, Switzerland.

Schmatz, K.-D. (2007). Java Micro Edition Entwicklung mobile JavaME-Anwendungen mit CLDC und MIDP. Heidelberg: dpunkt.verlag GmbH

Siebert, F. (2002). *Hard Realtime Garbage Collection*, Karlsruhe: aicas GmbH.

Vazquez, J., Almeida, A., Doamo, I., Laiseca, X. & Orduña, P. (2009). Flexeo: An architecture for integrating wireless sensor networks into the Internet of Things. *3rd Symposium of Ubiquitous Computing and Ambient Intelligence*. 22-24 October 2008, Salamanca, Spain.

Virtenio. (2011). 2.4 GHz Funkmodul „Preon32" mit überlegener Technik [Online] Retrieved September 6, 2011, from http://www.virtenio.com/de/produkte/hardware/preon32.html.

Wessels, A., Jedermann, R. & Lang, W. (2010). Embedded context aware objects for the transport supervision of perishable goods. In: Zadeh, A. (ed.) *Recent advances in electronics, hardware, wireless and optical communications*. Cambridge, UK: Wseas Press.