Virtual Machines Applied to WSN’s
The state-of-the-art and classification

Nuno Costa\textsuperscript{1,2}, António Pereira\textsuperscript{1}, Carlos Serôdio\textsuperscript{2}
\textsuperscript{1}Superior School of Technology and Management
Polytechnic Institute of Leiria
Morro do Lena – Alto do Vieiro, 2411-901 Leiria, Portugal
\{nuno.costa, apereira\}@estg.ipleiria.pt
\textsuperscript{2}CETAV-UTAD University
Quinta dos Prados
5000-911 Vila Real, Portugal
cserodio@utad.pt

Abstract

In the last few years, various middleware solutions have been proposed for bridging the gap between the complexity of the applications and the low hardware abstraction in order to program and manage the sensor nodes of the wireless sensor networks. The proposed middleware range from market and data-centric approaches to message oriented middleware. This survey discusses the representative state-of-the-art virtual machine (VM) class of middleware for WSNs highlighting the state of the research.

1. Introduction

Wireless Sensor Networks (WSNs) have been an interesting area which has received significant attention from the research community in the last years. The advances in micro electro-mechanical systems (MEMS), wireless communications and low cost of CMOS technologies have lead to the development of low cost, small sized and battery powered sensor nodes. These low cost wireless sensor nodes, equipped with sensors, processor, memory and wireless communication interface, had promoting the deployment of wireless sensor networks to support applications like large-scale environment monitoring, medical care, target tracking, intrusion detection, surveillance, etc. Today, the environmental monitoring is the prime application field of the WSNs [4]. If on the one hand the Moore’s law predicts that these micro sensor nodes will become smaller and cheap, on the other hand this does not means that in the future sensor nodes will have abundant resources. Thus, system software like operating systems (OSs), virtual machines (VMs) or other middleware have to address this (continuous) lack of resources, besides facing possible node failures, security issues, high level programming abstractions, heterogeneity, while promoting the transparency of the distributed essence of the WSNs. Apart from operating systems, various middleware have been proposed to fulfill the gap between the applications and the low hardware abstraction. This middleware were classified in [2] in to two broad classes: (1) middleware based on programming abstractions and (2) middleware for programming support. The class of middleware for programming abstractions includes both global and local behavior abstractions, which means that the WSN could be programmed as a whole (global) or by regions (local). The class of middleware for programming support was partitioned in five groups [2]: virtual machines, databases, agents (modular), application driven and message-oriented middleware. Virtual machines are an approach useful to virtualize real hardware, intermediate program representation or bytecode interpretation [5], while the database programming support virtualizes the WSNs as a distributed database, where user can use SQL like languages to interact with the network. The agent approach promotes the modular programming in order to facilitate the distribution of code through mobile agents. Application driven is another programming support approach that focuses on tuning the network in order to support the application requirements. At last, the message-oriented middleware constitutes a
This paper surveys the state-of-the-art of the current virtual machine class of middleware for WSNs, ranging from tiny application oriented virtual machines to distributed VMs, analyses them based on six key features (i) memory footprint (code and data), (ii) application execution method, (iii) execution model, (iv) application or domain specific, (v) remarkable characteristics and (vi) availability) and classifies them based on the software layer they seek on target system software stack.

This paper was written for two reasons: firstly, our research group believes that the virtual machine middleware is a valid approach to program and reprogram WSNs and secondly, it was felt the lack of articles concerning this specific subject. To the best of our knowledge, some researchers had surveyed the middleware for WSNs [1] [2] [3], but, no surveys have been presented focusing specifically in the virtual machine class of middleware, which have today a broad range of proposals. The rest of the paper is structured as follows. Section 2 describes the contributions a VM approach could bring to WSNs. Section 3 presents the virtual machines and virtual machine-based middleware already proposed for WSNs. Section 4 provides a classification scheme for representative virtual machine approaches and section 5 summarizes the contribution of this paper.

2. The virtual machine approach

According to [5], virtual machines have traditionally focused on virtualizing real hardware (where virtual machine monitors are included), intermediate program representation or bytecode interpretation. This paper is oriented for the bytecode interpretation class of virtual machines.

When we think in virtual machines we always remember the overhead of the interpretation. Apart from this thought, virtual machine constitutes a valid approach to be applied to WSNs. The prove is the broad range of software proposals for WSNs based on virtual machines.

VMs have been using to mitigate some problems related with WSNs. At first, the hardware abstraction they provide. As WSNs hardware evolves, more heterogeneous platforms have become available. Using virtual machines, applications run transparently in any platform and there is no need to rewrite applications for different architectures, this because VMs provide a common runtime environment. Also, as VM programming is platform independent, it promotes portability and thus mobility and facilitates the distributed computing [9], for example, by using the mobile agents approach. Virtual machines could also add capabilities to the underlying operating systems. Suppose an operating system, kernel or executive that is single thread. If a virtual machine, with support for multi-thread, is installed in these types of system software, thus virtual machines could extend the capabilities of the whole system. Virtual machines also provide code safety because they usually interpret programs in a sandbox, from where hardware is not directly accessible. Virtual machines also facilitate the network programming and reprogramming (an important requirement in WSNs) because applications became more compact. This compactness leads to flexibility and low energy consumption both during broadcast and memory write operations on target memory. Thus, virtual machines are a way to change from coarse-grained to fine-grained software updating. These advantages could also be improved if applications become themselves even more compact. This could be achieved using application specific virtual machines (ASVMs) [7]. This type of VMs also improves the interpretation of bytecode because there are less bytecode to interpret and the bytecode is high-level operation oriented. The big disadvantage of this approach is that virtual machine become oriented to special types of application (like environment monitoring) and if one need to execute other type of application (like target tracking) a new virtual machine must be built and deployed. Other solution to mitigate interpretation overhead, like in desktop systems, is the use of an on-the-fly compiler to native code. The proposal described in [18] addresses this method. Of course, this method is dependent of the existence of memory needed during the conversion from bytecode to native code, where both must be temporarily in memory. Finally, interpretation overhead could also be naturally minimized by applications that have a low duty cycle. On the hardware side, interpretation overhead could also be minimized using bytecode specific CPUs. But this is an approach that has two main drawbacks: first, according to [35], this is not a competitive approach and second, this method represents a big investment and thus it doesn’t contribute to the vision of the WSNs hardware where it must progressively become smaller and cheap.

3. Virtual machine-based proposals for WSN’s

Maté [5] is probably the first virtual machine for resource constrained sensor nodes proposed by Levis et al. This stack-based bytecode interpreter was introduced as an alternative way to program and
reprogram WSNs based on the TinyOS executive [6].
As Maté uses a specific bytecode (named tinyscript),
that is dense and concise, it means that complex
programs could be written in a few lines of tinyscript,
saving resources like memory and energy, especially
during transmission over the air. In order to reprogram
(infect) the network, Maté implements a viral program
mechanism. If Maté receives a more recent version of
the program, the code is installed and also broadcasted
to the network neighbors. Maté also separates the
kernel space from user space. This provides a safe
execution environment where Maté programs can’t
compromise the good functioning of the underlying
software and hardware. As execution model, Maté
inherits the event-based model of the TinyOS but it
hides the TinyOS asynchrony by providing a simpler
synchronous interface. To reduce Maté memory
footprint, programs are broken into capsules, of at most
24 byte long instructions, to take advantage of the
TinyOS packet length. The Agilla [19] is a mobile
agent middleware for TinyOS devices, which is based
on the Maté virtual machine, but instead of flood code
capsules throughout the network like Maté, it supports
applications deployment by injecting mobile agents.
More recently, the same authors of the Maté had
extended it turning the virtual machine into a
framework to build application specific virtual
machines (ASVMs) [7]. This topic is described later.

Scylla [18] also belongs to the first general purpose
virtual machines proposed for sensor networks. Scylla
does not follow the interpreter approach. Instead, it
includes an on-the-fly compiler to compile applications
to platform native code. Scylla is targeted to program
mobile tiny devices, providing them with a safe
environment and abstracting them from the possible
heterogeneity of the underlying hardware. It supports
inter-device communication at the level of the virtual
machine (useful to control communications), active
and passive power management, admission control and
error recovery. It also uses a simple instruction set that
maps directly to the instructions sets of the modern
microcontrollers [18] in order to enhance the on-the-fly
compilation of the application source code. According
to the authors, an on-the-fly compiler for the Hitachi
SH-3 microcontroller occupies 1.1 kB. The Scylla also
supports application migration by structuring
applications using modules. This modules are composed by the the application code, the memory
image and the fault handler sections. The memory
image contains the data to be loaded into memory
before execution and it is this section that preserves the
state of the application data during migration of
applications. The Scylla virtual machine was used in
the Smart Messages project [25]. This project aims
computation and communication on large networks of
networked embedded systems based on the sending of
Smart Messages through the network.

SensorWare [20] is a middleware that implements a
tcl [21] virtual machine, or more strictly, a script
interpreter, in order to provide a way to program
WSNs based on mobile scripts. This programming
model is especially useful for the localized distributed
computation application types, where only some nodes
of the network are involved in. The authors does not
explicitly describe the virtual machine but it is clear
that this virtual machine based middleware targets
sensor nodes with better resources than mica motes, as
it occupies about 180kB of memory code.

Application Specific Virtual Machines (ASVMs)
[7] is a method to safely and efficiently program sensor
networks. The initial Maté virtual machine was
improved and was generalized in a framework that
allows programmers to generate and tailor Maté VMs
to a specific application domain by selecting a
scripting language and a set of domain-specific
extensions (library functions and events that trigger
execution) [7]. Proceeding by this way, the framework
provides the necessary primitives for the needs of the
specified application domain making programs shorter
and simple, which is very important as far as code
execution and code propagation are concerned. Each
generated virtual machine automatically includes the
code propagation system to achieve the propagation of
programs through the network. Instead of maintaining
the simple program broadcast approach of the old Maté
with known disadvantages [7], they developed and
included the Trickle [8] algorithm to improve code
propagation and also a security system to avoid
propagation of malicious programs through the
network. A Maté ASVM is available with the TinyOS
distribution by the name of Bombilla.

In [34] the authors proposed an approach that goes a
step ahead than ASVMs. Instead of implement a virtual
machine for each application domain, the authors argue
to virtualize the virtual machine in order to develop a
virtual virtual machine (VVM). The applications to be
executed by the VVM must carry a so-called VMlet. It
is the VMlet that configures the VVM to become itself
oriented to that specific application. There are VM
proposals described in this paper that support great
dynamism at the VM level but, to the best of our
knowledge, no one reach the plenitude of a VVM as
specified in [34].

Dynamically extensible Virtual Machine (DVM)
[15] is a virtual machine developed to support multiple
application, software reprogramming of sensor
networks and it is an improvement of the
TinyOS/Maté. This VM sits on top of the SOS
operating system [16] and its design was inspired in
Maté as well as existing reprogramming approaches
like full binary upgrades, virtual machine, parameter tuning and modular upgrades, in order to present a moderate solution which minimizes the disadvantages of the methods of inspiration, but at the same time preserving the advantages of them. In TinyOS/Maté, the reprogramming of the system core requires the Deluge mechanism (updating the entire binary) while in SOS/DVM only the modified modules are updated. The DVM approach uses the SOS incremental binary module upgrade capability in conjunction with a virtual machine which is also extensive at run-time. The virtual machine is used to execute application scripts in response to events. This execution is based on calls to the dynamic binary modules of the lower system by the interfaces they provide to the DVM. The inverse interaction is also supported. The SOS modules can also invoke the scripts and pass data to them [15]. Another feature the DVM supports is the ability to add or modify VM instructions at runtime by the implementation of extension library modules programmed in native code [15]. This binary modules could be exposed to the virtual machine as a VM instruction. Thus, it is possible to orient the Dynamic Virtual Machine (DVM) to specific applications or domains of applications in run-time, turning it into a Dynamic Application Specific Virtual Machine (DASVM). The authors presented a robotic sensor network (mobile sensor nodes) [17] powered by SOS operating system and a Dynamic Application Specific Virtual Machine in order to control the mobile nodes using multiple user consoles.

Distrinet Adaptable Virtual Machine (DAViM) [22] is another virtual machine for Sensor Nodes that is inspired in Maté [5] and it is similar to DVM [15]. The DAViM main aims are: control sensor behavior by dynamically injecting application code, support system customization for specific application domains, allow system extendibility and support multiple running applications by running multiple isolated VMs. The virtual machine instruction sets are grouped in libraries (operating system modules) to be possible to load, unload, change or remove them dynamically. The VM instruction set that will take advantage of a new instruction set library is also distributed across the network. However, this virtual machine expects dynamic memory and dynamic modules services from the underling system software and it targets for recent sensor hardware, which means that, it is not suitable for the first generations of Mica motes.

Query Machine (QM) [23] is an integer, stack-based, dynamic, extensible and multiple program virtual machine targeted for data acquisition and data processing in sensor networks, that sits on top of the TinyOS executive [6]. It uses a small subset of the Java Virtual Machine [24] (instead of queries) with optimized bytecode to make the VM independent of the query language used to interact with the network. The QM programs are divided into three sections, each one executed at distinct times. According to the authors, the QM bytecode is the most compact and space efficient representation of programs for sensor networks. The QM virtual machine is oriented for data acquisition applications and not oriented to, for example, tracking applications due to its basic data acquisition orientation and network topology assumed (dynamic routing tree). However, it would not be too much difficult to implement a tracking application using this VM due to its flexibility and extensibility and also because its available source code (to implement, for example, a grid network topology).

ScatterVM [26] is also a virtual machine for sensor networks, more precisely, an academic project for the ScatterWeb platform [27], but without any published information.

VM* [9] is another approach to generate virtual machines for WSNs having applications as its basis. This approach goes further when compared to Maté because it generates optimized Java-based interpreters, having the application and the target hardware as input. This leads to the construction of optimized virtual machines where only the needed services (based on application and target device) are implemented and the way the available services are implemented depends from the resource availability on the target device. The VM* framework also implements some techniques in order to minimize the overhead induced by the interpretation like compacted classes, serialization, optimized interpreter, etc. Node updates are also a improved topic. Instead of being application update oriented like Maté, the VM* includes an incremental linker [10] for fine-grained updates of VM applications and virtual machine itself. Thus, if for instance a new application is installed, the virtual machine is easily extended to support it.

Doug Palmer in [11] proposes a virtual machine generator to generate stack-based virtual machines tailored to the capabilities of the underlying hardware layer as a way to control the complexity of heterogeneous smart spaces, where the WSNs are included. Thus, for resource rich devices a full virtual machine is implemented and for the others only a subset of the full virtual machine is available. The compatibility between a subset virtual machine and superset implementation is guaranteed [11], which means that there is a common runtime environment for both of them.

RealMachine [12] is a virtual machine that does not follow the classical virtual machine approach of present a virtual processor to the programmers [12].
Instead, the RealMachine was developed in such a way to achieve the best execution performance in order to reach the performance of a compiled C application to the target hardware. To achieve this aim, the firmware is exposed to the byte code level, through function pointers, instead of re-implementing them through basic commands of the virtual processor [12]. According to the authors, even an application that is written in C and compiled for the target platform would not perform faster as it used the same function.

Squawk [13] is a small java virtual machine that assures the services usually provided by an operating system and a virtual machine. The Squawk virtual machine was developed mostly in Java and constitutes the system software of the (rich) Sun SPOT [14] device. This VM supports multiple applications through isolation mechanism [13], application migration, provides radio and radiogram (point-to-point) connection types [13], suppresses the use of an operating system and uses optimized to size Java classes in order to reduce its memory footprint. According to [13], the Squawk virtual machine occupies 270 kB of flash and 80kB of RAM, which is too much to be used or ported to Mica2 platforms.

TinyVM and leJOS [28] [29] are two Java subsets virtual machines, from the same author, targeted for replacement of the system software (firmware) of the RCX brick of the Lego MindstormsTM platform. The TinyVM was the first of both virtual machines been developed. It implements a reduced version of the Java virtual machine and fits in 10kB. It includes preemptive threads, exceptions, synchronization, arrays LCD support, etc. However this virtual machine doesn’t includes a garbage collector, floating point and string data type. The leJOS virtual machine was developed based on TinyVM to address the leak of functionalities of the TinyVM with just more 5 kB of code. The leJOS VM was already ported to WSNs in order to evaluate a dynamic linking mechanism for reprogramming WSNs [31] related with the Contiki [30] operating system. Due to the different capacity of RAM in the RCX brick and Contiki devices, the Contiki version of the leJOS VM loads class files from ROM while the RCX version loads class files from RAM.

MagnetOS [32] is a distributed operating system for sensor and ad hoc networks that abstracts the whole network as a single, unified Java Virtual Machine, exposing a runtime environment for java programs. MagnetOS applications are specified as regular Java programs. An application partitioning tool takes the monolithic Java application and converts it into components. Then the components are automatically and transparently distributed through the network, based on components communication/interaction, to reduce energy consumption and increase network longevity. The components use Java Remote Method Invocation (JRMI) for inter-component communication. Besides energy conservation, the mechanism also has the benefit of free the programmer from writing programs tailored for the MagnetOS operating system architecture. As MagnetOS needs Java heavy mechanisms, like JRMI, it targetts devices with better resources than, for instance, the Mica devices.

In [33] it is presented a microkernel for hard real-time applications based on two interacting virtual machines: the reactive Embedded machine (E) and the proactive Scheduling machine (S) [33]. Due to this approach, the code that runs on this kernel is partitioned in the E and S code. The E part of the code is executed in response of external events while the S code handles the interaction between the system and the processor and is driven by internal events, like time-slice, for example. According to the authors, this code separations promotes the independent programming, verification, optimization, composition, dynamic adaption and reuse of both reaction and scheduling mechanisms. This kernel exebits a linear behavior in loading conditions (lots of tasks). This is a signal of determinism, which is the more important characteristic of a real-time kernel, ready to serve hard real-time tasks. An implementation of this kernel occupies about 8kB in a StrongARM SA-1110 processor. To the best of our knowledge, there is no port or similar implementation targeting nodes of WSNs, but considering its properties, it would be interesting to observe the behavior of an kernel like this in resource constrained sensor networks hardware.

4. Classification

The virtual machine approaches described in the last section could be classified based on the software layer they seek on target system software stack. Some approaches follow the classical model of virtual machine that target middleware by position between operating system and applications. We call these virtual machines «middleware level VMs». Other proposals address the substitution or replacement of the entire operating system. We call these virtual machines «system level VMs». Table 1 presents the grouping of the current virtual machine proposals based on this classification and also refers the main characteristics of each virtual machine based on the following properties:

- Memory footprint – Memory footprint defines the memory resources (for code and for data) needed to install and execute the virtual machine. This is an
extremely important attribute for each virtual machine that targets WSNs.

• Application execution – This property tells if the virtual machine effectively interprets the code or compiles it to the native platform code. Compilation to native code improves execution time but needs more resources either to store the on-the-fly compiler and for the translation operation.

• Execution model – This property shows the way instructions are interpreted by the virtual machine: register-based or stack-based. Typically, a stack-based virtual machine defines smaller bytecodes [23]. On the other hand, programs for register-based virtual machines need fewer instructions because there is no need for instructions to put operands in the top of the stack [23]. Thus, for WSNs, small and compact bytecode could be more attractive because it consumes less memory and is more suitable for dissemination over the network [23].

• Application or domain specific virtual machine – This property tells if virtual machine supports the customization for specific application or domain. This capability is useful because programs become very tiny. But there are some drawbacks on this approach, for instance, the virtual machine could not support a broad range of application types, and its replacement is needed if the WSN application domain changes. Also, a domain specific virtual machine could support multiple applications that belong to different domains.

• Remarkable characteristics – This property shows the main capabilities or remarkable enhancements included in the virtual machine. Characteristics like bytecode compactness, on-the-fly compilers, support for multiple applications, virtual machine updating during runtime (dynamic), debugging, etc, are some examples.

• Available – The available property tells if the virtual machine is available for download. If so, it also tells if it is open-source or not.

For better understanding of the Table 1 consider the following legend: «N/A» means not applicable, «---» means empty and «unknown» means that the property is not known due to lack of information.

Table 1. Virtual machines classification

<table>
<thead>
<tr>
<th>Name</th>
<th>Memory Requirements</th>
<th>Application Execution Model</th>
<th>Execution Model</th>
<th>Application or Domain Specific</th>
<th>Remarkable Characteristics</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>2MB</td>
<td>Stack-based</td>
<td>Stack-based</td>
<td>No</td>
<td>None</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Stack</td>
<td>Unknown</td>
<td>Native code</td>
<td>Unknown</td>
<td>No</td>
<td>None</td>
<td>Yes/No</td>
</tr>
<tr>
<td>SenseVM</td>
<td>10MB</td>
<td>Interpreted</td>
<td>Unknown</td>
<td>No</td>
<td>None</td>
<td>Yes/No</td>
</tr>
<tr>
<td>DVM</td>
<td>15MB</td>
<td>Interpretive</td>
<td>Stack-based</td>
<td>Yes</td>
<td>None</td>
<td>Yes/No</td>
</tr>
<tr>
<td>DIAM</td>
<td>Unknown</td>
<td>Interpreted</td>
<td>Stack-based</td>
<td>Yes</td>
<td>None</td>
<td>Yes/No</td>
</tr>
<tr>
<td>GVM</td>
<td>35MB</td>
<td>Interpreted</td>
<td>Stack-based</td>
<td>Yes</td>
<td>None</td>
<td>Yes/No</td>
</tr>
<tr>
<td>WVM</td>
<td>35MB</td>
<td>Interpreted</td>
<td>Stack-based</td>
<td>Yes</td>
<td>None</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Tropos</td>
<td>1MB</td>
<td>Interpreted</td>
<td>Unknown</td>
<td>No</td>
<td>Native updates</td>
<td>No</td>
</tr>
<tr>
<td>Symbi</td>
<td>1MB</td>
<td>Interpreted</td>
<td>Stack-based</td>
<td>No</td>
<td>Native updates</td>
<td>No</td>
</tr>
<tr>
<td>WixT</td>
<td>1MB</td>
<td>Interpreted</td>
<td>Unknown</td>
<td>No</td>
<td>Native updates</td>
<td>Yes/No</td>
</tr>
<tr>
<td>WixT</td>
<td>1MB</td>
<td>Interpreted</td>
<td>Stack-based</td>
<td>No</td>
<td>Native updates</td>
<td>Yes/No</td>
</tr>
</tbody>
</table>

5. Conclusion

This paper has presented the current state-of-the-art in virtual machine middleware for sensor networks. We also classified the proposed approaches based on the software layer they seek in target system software stack. We believe that virtual machines are a valid approach to be used in sensor networks and the “phantom” of the interpretation with respect to energy consumption and performance will be solved either by enhanced dynamic updating solutions, by enhanced on-the-fly compilers or other incoming solutions.

6. References


