Real-time Embedded Java Virtual Machine for Application Development in Wireless Sensor Network

Xing Liu¹,², Kun Mean Hou²*, Christophe de Vaulx², and Khalid El Gholami²

1. Laboratory of Internet and Information Technology, Wuhan University, Wuhan, China
2. LIMOS Laboratory UMR 6158 CNRS, Blaise Pascal University, Clermont-Ferrand, France
*Corresponding author, Email: [liu, kun-mean.hou, devaulx, elgholam]@isima.fr

Abstract—The application development in wireless sensor network (WSN) is complicated due to the diverse software and hardware platforms. One way to solve this problem is to apply the embedded Java virtual machine (EJVM) on the WSN nodes. With the EJVM, the users can program the applications by the popular and high-level abstract Java language. However, several challenges exist for applying the EJVM on the WSN nodes, such as the memory resources on the WSN nodes are constrained, the multitasking programming and real-time response are required by many WSN applications. Currently, these challenges cannot be addressed well by most EJVMs. And in order to address these challenges, a real-time memory-efficient Java operating system HEROS is developed. By means of HEROS, the multitasking real-time Java applications can be developed even on the memory-constrained WSN nodes. Nevertheless, the execution efficiency of the Java bytecode is not high, making the EJVM not suitable to be used on the tight energy-constrained WSN nodes. To solve this problem, a new mid-layer software REMID, which is designed to have some similar functionalities as the EJVM, is also developed. Different from the EJVM, REMID is designed to be both memory efficient and energy efficient, thus it can substitute the EJVM to be used even on the severe energy-constrained WSN nodes. The final evaluation works prove that a user-friendly WSN application development environment can be provided to the users by means of the EJVM, HEROS and REMID.

Index Terms—Operating System; Virtual Machine; Wireless Sensor Network; Real-Time

I. INTRODUCTION AND BACKGROUND

The recent advances in the microelectro-mechanical and communication technologies make the wireless sensor nodes been used in widespread application domains [1-5]. Since most WSN nodes are designed to be small in the physic size and low in the cost, they are constrained in the memory and energy resources [6, 7], e.g., the node MicaZ has only 4KB RAM resources and is equipped with only two AA batteries.

A key challenge for WSN research is that the user application development is difficult. This is because the hardware and software platforms in WSN are diverse. On one hand, different WSN nodes with different installations are implemented, such as the IMote, Mica, MicaZ, sunSPOT, TelosB, XYZ. On the other hand, different WSN OSs with different features are developed, such as the TinyOS [8], Contiki [9], SOS [10], mantisOS [11], LiteOS [12], RETOS [13], nano-RK [14]. Due to these platform diversities, the application development in WSN becomes difficult as the diverse low-level details are needed to be understood by the users.

In order to address the challenge above, the embedded Java virtual machines (EJVM) are motivated to be used on the WSN nodes. With the EJVMs, the WSN application can be programmed by the popular, robust and high-level abstract Java language; the application design, test and maintenance process can be simplified. Moreover, the application can be decoupled from the underlying systems, and this will improve the WSN reprogramming performance as only the Java application image other than the monolithic software image is needed to be updated.

Currently, many embedded JVMs have been developed, including the JamaicaVM [15], JamVM [16], TinyVM [17], Darjeeling VM [18], simpleRTJ [19], nanoVM [20], Jwik [21], Java Card VM [22] and so on. However, most of these JVMs are not developed dedicatedly for the WSN applications. Instead, they are designed to be used for some typical applications like the robotic control, the electronic toys, the smart card reader, etc. Due to this reason, several challenges exist for applying the EJVM on the WSN nodes: 1) the memory resources on the WSN nodes are limited, thus the EJVM should be memory-efficient. 2) many WSN applications require real-time (RT) support, e.g., the industrial engine control applications. Thus, the RT performance of the EJVMs should be well. 3) most WSN applications are multitasking ones. Thus, the multitasking Java programming should be supported by the EJVMs. Due to the challenges above, the selection of a suitable EJVM and the adaption or improvement of an existed EJVM becomes important.

In this paper, the main concepts of EJVM and the feasibility analysis of using the EJVM in the WSN are presented firstly in the section II. This section can help the researchers to understand the challenges of using the EJVM to support the WSN applications. And then, in the section III, a new memory-efficient real-time Java OS
HEROS is developed to address the challenges proposed in the section II. With the new OS HEROS, the real-time multitasking WSN applications can be achieved by using Java, and the memory consumption is also not high. However, the Java bytecode execution efficiency is low, making the energy consumption of the EJVM be high. Thus, the EJVMs are restricted to be used on the high energy constrained WSN nodes. To solve this problem, a new mid-layer software REMID is developed (in the section IV). REMID has the similar functionality to the EJVMs, but is designed to be both memory and energy efficient. Thus, it can substitute the EJVM to be used even on the severe energy-constrained WSN nodes. Finally in section V, VI and VII, the evaluation works, the conclusions and the ongoing works are done respectively.

II. CONCEPTS AND FEASIBILITY ANALYSIS OF THE EJVM

In this section, the key concepts of the EJVM will be presented firstly, and then the feasibility of using EJVM to execute the WSN applications will be analyzed. By means of this analysis, the way to improve or adapt an EJVM can be concluded.

A. Key Concepts of the EJVM

In this part, the simpleRTJ which is a clean room implementation of the JVM for the embedded devices is chosen as the example for the presentation of the key EJVM concepts:

JVM Architecture: The JVM architecture of simpleRTJ is shown in the Fig. 1. Two components constitute the simpleRTJ: the simpleRTJ VM and the multithreaded JavaOS. The simpleRTJ VM is in charge of interpreting the Java bytecode. However, only single-tasking Java application can be supported if just the simpleRTJ VM is used. In order to support multitasking Java applications, a multitasking JavaOS is needed to be embedded. In simpleRTJ, a multithreaded JavaOS is developed. With this JavaOS, the multitasking Java applications can be achieved through the Java threads.

![Figure 1. JVM Architecture of simpleRTJ](image)

With the EJVM, the Java application can be separated from the lower system. Thus, the software space is divided into two parts: the application space and the system space. The application space is programmed by Java while the system space is programmed by C. Two spaces are built independently, and two independent images are generated: the Java application image and the system image. The system image is pre-burned to the WSN nodes by the WSN experts. And then, the WSN users only need to focus on the application space without the necessity of understanding the low-level system details. After the Java application image is generated, it can be updated to the target WSN nodes to perform the WSN reprogramming. Since only the Java application image needs to be transmitted, the reprogramming performance of the JVM-based system is higher if compared with that in the monolithic software system (e.g., the TinyOS).

Bytecode interpretation process: The process of the Java bytecode interpretation in the simpleRTJ is shown in Fig. 2. All the bytecode will be interpreted one by one, and after a Java bytecode is interpreted, the related bytecode handler (programmed in C) will be called.

![Figure 2. Single-thread bytecode interpretation process in simpleRTJ](image)

Interaction between the Java code and underlying C code: Since the application is programmed by Java and the low-level system is programmed by C, the interaction between the application Java code and the system C code is essential. The access from the system C code to the application Java code is achieved through the function interface "vm_run(method_t *method)". Once this function is called, the related Java method will be interpreted. The access from the application Java code to the system C code is achieved by the Java native methods. The Java native methods are different from the non-native ones. The non-native ones are used for the call from one Java method to another while the native ones are used for the call from the Java method to the low-level C functions (seen in the previous Fig. 2). With the "vm_run" interface as well as the Java native method mechanism, the application Java space and the system C space can interact with each other. These two mechanisms are important to be used for porting an EJVM to a given C programmed JavaOS.

JavaOS support for developing multitasking Java applications: A multitasking JavaOS is required for the support of the multitasking Java applications. In simpleRTJ, a multithreaded Java OS is developed. With
this OS, the multitasking Java applications can be achieved by using Java threads, seen in the Fig. 3. By means of Java threads, several application tasks can be executed concurrently by the thread switch, and the Round Robin (RR) algorithm is used for the switch operation. In the Fig. 4, the multithreaded bytecode interpretation process of simpleRTJ is depicted. By the comparison between the Fig. 2 and 4, the way to extend a single-thread JVM to a multithreaded one can be concluded.

![Multitasking WSN application supported by Java threads](image-url)

**Real-time performance**: To support the multitasking Java applications, a multithreaded JavaOS is developed in simpleRTJ. However, the RAM usage of the simpleRTJ increases greatly by this way. Because the run-time context of the Java thread needs to be saved before the thread switch, thus every thread in the multithreaded OS should have an independent stack, and this stack will be used for the saving of the thread’s run-time context. In simpleRTJ, in order to reduce the RAM usage, the thread switch is performed in the bytecode granularity. This means that after a switch request is posted, the switch operation will not be processed immediately. Instead, it will be deferred until the current bytecode handler runs to completion, seen in the Fig. 4. By this way, the bytecode handler in each thread (programmed in C) will not be preempted during its midcourse execution. In result, this handler’s run-time context, including the local variables, the processor registers (e.g., r0-r31 in the AVR ATmega1281 micro-controller) and so on, are not needed to be saved in the thread stacks. Consequently, the thread stack size can be decreased greatly, and the total RAM consumption can become lower if compared with that in the general multithreaded systems such as the mantisOS [11], nano-RK [14].

Although the RAM usage of simpleRTJ can be reduced by using the bytecode preemption, the real-time performance of the simpleRTJ is also decreased. This is because the real-time events cannot be processed immediately after they are generated. Instead, they should be deferred until the completion of the bytecode execution. Due to this reason, the current simpleRTJ cannot be used to execute the real-time WSN applications.

**Miscellaneous**: Besides the above VM concepts, some other mechanisms such as the garbage collection, the exception handling, the run-time execution tracing, the remote debugging, etc., are also implemented in simpleRTJ [23].

![Multithreaded bytecode interpretation process in simpleRTJ](image-url)

**B. Feasibility Analysis of using EJVM in the WSN**

Different JVMs have different features, and they strike the tradeoff between the performance and the VM code size. To evaluate the feasibility of using an EJVM in the WSN, three key topics, which are the VM code size, the multitasking Java programming and the VM real-time performance, should be investigated.

**VM code size**: The memory resources on most WSN nodes are constrained, thus the VM code size will determine whether a JVM can be applied on a WN or not. The JamaicaVM [15] and JamVM [16] have powerful VM performance, e.g. in JamaicaVM the hard real-time response down to a few μs can be supported, but the code size of these VMs [15, 16] are too high to be used on most WSN nodes. The nanoVM [20], Jwik [21] and Java Card VM [22] have less code memory consumption (commonly less than 10 KB), but the supported VM features are limited, e.g. in Jwik [21] the exception handling and the garbage collection cannot be supported.

**Multitasking Java programming**: Multitasking programming is essential for the WSN applications. In some VMs like TinyVM [17], Darjeeling VM [18] and simpleRTJ [19], a lightweight multithreaded Java OS is embedded. By means of these Java OSs, the multitasking Java applications can be achieved by using the Java threads. In the other VMs such as nanoVM [20], Jwik [21] and Java Card VM [22], no Java OS is developed. In these VMs, only single-thread Java applications can be executed.

**VM real-time performance**: Real-time guarantee is required by many WSN applications, such as the industrial engine control and the WSN medical care. In nanoVM [20], Jwik [21] and Java Card VM [22], the real-time cannot be supported since no Java OS exist in these VMs. In JamaicaVM [15] and JamVM [16], the real-time performance is well, but these VMs should be built on the native RTOSs like Linux and VxWorks, thus the memory consumption of these VMs are too high for the WSN platforms. In some other VMs [17-19], a lightweight multithreaded Java OS is embedded inside the EJVM. In these VMs, the preemption among the different tasks can be supported. However, these VMs are still not real-time ones, and there are two reasons for this. Firstly, no real-time scheduling algorithm is used in some multithreaded Java OSs, e.g. in simpleRTJ [19], the...
The shared stack and thread stacks exist: three of them for the RT tasks ($T_1, T_2, ..., T_6$), and three of them are RT ones: $T_7, T_8, T_9$. If the event-driven dispatcher cannot support the preemption and has poor real-time capability, the thread switch is realized in the bytecode of these VMs. In result, the VMs [17-19] are not strict real-time systems.

From the analysis above, it can be seen that most of the current EJVMs [15-22] are not suitable to be used in the WSN. Some of them are too high in the memory consumption, some cannot guarantee the real-time deadlines, and some can only execute the single-tasking Java application. Therefore, the development of a lightweight multitasking Java OS, which is real-time and meanwhile keeps small memory footprint, is significant for the usage of EJVM in the WSN. To address this challenge, a new Java OS named HEROS (Hybrid Embedded Real-time Operating System) is implemented and presented in the next section.

III. DESIGN AND IMPLEMENTATION OF HEROS

In this section, the design and implementation of HEROS will be presented. Firstly, the HEROS scheduler and memory management mechanisms are investigated in the part A. And then, in the part B, the way to build an EJVM upon the HEROS is presented. Finally, a brief discussion is given in the part C.

A. Implementation of HEROS

In HEROS, the hybrid scheduling model and the dynamic memory allocator are implemented to achieve a real-time OS with small memory footprint.

Hybrid scheduler: The scheduling model of WSN OSs can be classified into two types: the event-driven scheduling model and the multitthreaded scheduling model. The tasks in the event-driven OSs are executed one by one, one task can be executed only after the previous task’s run-time context has been released, and the preemption is not enabled. Thus, only one stack is required in the event-driven OS, and the RAM usage of this OS is low [25]. Currently, this scheduling model has been used in many WSN OSs, including the Contiki [9], TinyOS [8], SOS [10] and so on. The drawback of this scheduling model is that the system RT performance is poor, this is because the time-critical tasks cannot be processed immediately after they are generated.

The tasks in the multitthreaded OSs can be executed concurrently by means of the thread switch, and the preemption among the threads is enabled. Therefore, the RT performance of the multitthreaded OS is better than that in the event-driven OS. However, the thread’s run-time context needs to be stored before being preempted, thus every thread in the multitthreaded OS needs to have a private stack. In result, the RAM usage of the multitthreaded OS becomes larger.

In order to reduce the RAM usage of the multitthreaded OS, the thread switch is realized in the bytecode granularity in simpleRTJ [19] and TinyVM [17]. This mechanism decreases the OS RAM usage, but the OS RT performance is decreased as well.

To achieve the objective of being an OS that is both real-time and low RAM consumption, the hybrid scheduling principle is used in HEROS. Firstly, the event-driven scheduling model is used in HEROS for the purpose of keeping low memory consumption. However, the event-driven scheduler cannot support the preemption and has poor real-time capability. Thus, the multitthreaded scheduler is also implemented in HEROS, and it is used dedicatedly to schedule the RT tasks. Consequently, a hybrid scheduling model (both event-driven and multitthreaded scheduling) is realized in HEROS. By means of this hybrid scheduler, the advantages of event-driven system’s low RAM usage and multitthreaded system’s good real-time performance can be both achieved.

Two schedulers are implemented in parallel in HEROS, seen in the Fig. 5. The system tasks are classified into two kinds: the RT ones and the non-RT ones. The RT tasks are scheduled by the multitthreaded scheduler, and each thread has an independent run-time stack. The non-RT tasks are scheduled by the event-driven scheduler, and all of them share one stack. At any time, only one scheduler can be active. The multitthreaded scheduler has the priority higher than the event-driven one, thus it can preempt the event-driven scheduler when required. If all the RT tasks are inactive, the event-driven scheduler will start the scheduling. However, if any RT task becomes active, the event-driven scheduler will be suspended, and then the OS will switch to the multitthreaded scheduling model.

Figure 5. Hybrid scheduling structure in HEROS

Two scheduling models need to switch to each other. In order to perform this switch efficiently, the event-driven scheduler in HEROS is also programmed as a thread (with a thread control block allocated), named common_thread. By this way, the switch from the multitthreaded scheduler to the event-driven scheduler can be operated as the switch from the RT thread to the common_thread.

The advantage of this hybrid scheduling is that a RT OS can be achieved with less RAM resources. Seen in the Fig. 5, assumed that there are nine system tasks ($T_1, T_2, ..., T_9$), and three of them are RT ones: $T_7, T_8, T_9$. If the pure multitthreaded OSs (such as mantisOS [11], uCOS [24],) are used for the scheduling of these tasks, nine thread stacks are required to be created. This is because in the pure multitthreaded OSs, no matter a task is RT or not, a thread should be created for its execution. However, if the HEROS hybrid scheduling model is applied, only four thread stacks exist: three of them for the RT tasks ($T_7, T_8, T_9$), and the left one for the non-RT tasks ($T_1$ to $T_6$). By
this way, the thread stack number decreases greatly. Since the thread stack is large in the memory size, the RAM usage of MIROS will decrease greatly if compared with the pure multithreaded WSN OSs.

The event queue is used for the implementation of the event-driven scheduler (Fig. 5). All the generated events enter this queue by FIFO (first input, first output). Then, they are withdrawn and dispatched one by one by the dispatcher. The reason to choose the FIFO principle is because the event-driven scheduler in HEROS is only used to dispatch the non-RT tasks. The non-RT tasks have loose restriction to the response time, thus there is no necessity to use a complicated scheduling algorithm.

The real-time scheduling algorithm RMS (rate-monotonic scheduling) [26] is used for the thread scheduling in HEROS. Compared with another popular real-time scheduling algorithm EDF (Earliest deadline first scheduling), the drawback of RMS is that the RT tasks’ CPU utilization is limited to a given level. Nevertheless, this algorithm is still chosen in HEROS due to two reasons: 1). The RMS uses the static-priority scheduling, it is easier to be implemented and the run-time overhead is lower. Thus, it is more suitable for the resource-constrained WSN nodes. 2). Although the CPU utilization of the RT tasks is not 100% in the RMS, it doesn’t matter since the left CPU resource can be used by the event-driven scheduler to schedule the non-RT tasks.

In terms of the article [26], the RT tasks scheduled by the RMS can be schedulable if the CPU utilization of these tasks is below a specific bound as follows:

\[
U = \sum_{i=1}^{k} \frac{C_i}{T_i} \leq k(2^{1/k} - 1)
\]

where \(C_i\) and \(T_i\) represent the computation time and the release period of Task \(i\) respectively, and \(k\) is the number of RT tasks. Assumed that two RT tasks exist in HEROS \((k=2)\), and their \((C_i, T_i)\) are \((1, 4), (3, 6)\) respectively, then the \(U\) will be 75% and \(k(2^{1/k} - 1)\) will be 82.8%. Since 75% is less than 82.8%, these two RT tasks can be schedulable. As for the left CPU resource (1-75% = 25%), it can be left for the non-RT tasks.

Although the implementation of two schedulers in HEROS increases the ROM memory consumption in a degree (several kilobytes), it is not a noticeable problem. This is because the WSN nodes are mostly constrained in the RAM resources rather than the ROM resources. More significantly, the hybrid scheduling model in HEROS can decrease the RAM usage greatly. Thus, it is suitable to be used on the WSN platforms.

Dynamic memory management: Currently, the static allocation mechanism is used in many OSs, e.g. in simpleRTJ the memory for the object references, the thread control block and so on are reserved statically. The implementation of the static allocation is simple and the execution overhead is low, but the memory resources cannot be utilized efficiently. To reduce the OS RAM usage, the dynamic memory allocation is implemented in HEROS.

In HEROS, the RAM resources are allocated dynamically by using the fixed-size blocks. Since the size of the objects to be allocated can be different, several segregated block sections are created, and each section is used for the allocation of a given kind of objects, seen in Fig. 6. In each block section, a free queue is used for the management of the free memory. By means of this queue, the allocation can be completed with a constant response time.

The advantage of this block allocation is that it is easy to be implemented. However, the pre-reserved size of each block section is difficult to be determined. This is because the required size of each block section can be different if used in the different application environments. To avoid the memory overflow, each block section can be reserved to the maximum value it may be needed, but the memory insufficiency problem can take place in this case.

To make the memory allocator be flexible to the varied application contexts and meanwhile avoid the memory insufficiency problem, the heap extending mechanism is used in HEROS. After the block sections are pre-reserved, the left free memory space will be used both for the system run-time stack and the heap extending space. By doing this, the size of each block section is not required to be set to the maximum value. Instead, a moderate value can be used. And in case that a block section is overflowed, the allocation can be continued inside the extending heap space. By this way, the memory insufficiency problem can be avoided. Moreover, the allocator can adapt well to the different applied contexts.

![Figure 6. Memory allocation mechanism in HEROS.](image)

**B. Building of an EJVM upon a Java OS**

After the JavaOS is developed, the way to build the EJVM on this OS is an essential topic. In this part, the example of building the nanoVM onto the HEROS is chosen to present this topic.

To build an EJVM on a JavaOS, it is essential to understand how the JVM interacts with the JavaOS. As discussed in previous section II, the access from Java application space to the underlying system space can be achieved through the Java native method, and the access from the system space to the Java application space can be achieved through the `vm_run` interface "vm_run("Java_method")". According to these principles, the nanoVM can be built upon the HEROS and the system architecture is shown in Fig. 7. In this example, not all tasks in HEROS are programmed by Java. Instead, only the user-defined application tasks are open to the users.
users and be developed by Java. By doing this, the advantages of both C programming and Java programming can be combined.

To address the challenge above, a new mid-layer software named REMID is developed. REMID is designed to absorb some advantages from the EJVM, including the decoupling of the application from the underlying system, the providing of the abstract interfaces to applications, etc. By doing this, a friendly development environment can also be provided to the WSN users. However, different from the EJVM, REMID is designed to be both memory and energy efficient, thus it can substitute the EJVM to be used on the high energy-constrained WSN nodes.

A. Development Process of REMID

The development process of REMID is depicted in the Fig. 9. The same as the EJVM, the software space will be divided into two parts by REMID: the application space and the system space. Two independent binaries will be generated for these two spaces. And REMID acts as the bridge between these two spaces.

In REMID, the pre-linked machine code is chosen for the application binary. There are two reasons for this choice: Firstly, the pre-linked machine code needs not to be interpreted or resolved by the lower OS, thus the underlying system architecture can be simplified. Secondly, the pre-link machine code can be executed by the processor directly, thus the application code execution efficiency can be higher and less energy resources will be consumed.

The functions in the REMID applications are classified into two kinds: the local-call functions (similar to Java non-native methods) and the remote-call functions (similar to Java native methods). The former kind is used for the call within the application space, while the latter type is used for the call from the application space to the system space (similar to the call from Java application to the underlying native C code in EJVM). After the application programs are built, the raw application binary will be linked for another time. During this process, the application remote-call functions will be linked to the underlying service provider functions. And in order to ease the flexible problem of pre-linked code, a function jump table is provided in the system space. With this table, the modification to the system space will not cause the application image to be invalidated in case that the jump table is put in a fixed address in the system space.

As the WSN applications are mostly multitasking ones, it is essential for the REMID to support the multitasking application programming. To achieve this objective, the task registration mechanism is used in REMID. With this
mechanism, several independent application tasks can be defined in one application program and then be registered. After registered, their entry addresses will be passed to the underlying system. And then, these application tasks can be scheduled by the underlying OS scheduler.

In Fig. 10, the architecture of building the REMID onto the HEROS is depicted. Since REMID is efficient in both the memory and energy consumption, it can be used even on the high energy-constrained WSN nodes for the purpose of providing a user-friendly development environment to the WSN users.

**Figure 10. Architecture of building REMID upon HEROS**

### B. Discussion

Currently, less attention is paid to the pre-linked mechanism for the software development in WSN, this is because the pre-linked method is considered to be inflexible, e.g. in the Contiki, the significance of decoupling the applications from the underlying systems is also known. And to address this challenge, a dynamic linker [27] is developed in Contiki. With this dynamic linker, the application program can be built independently into an ELF module and then be updated to the WSN nodes. On the WSN nodes, this module will be resolved and linked by the linker and then be executed.

Dynamic linking method is chosen in Contiki due to its high flexibility. However, the experimental experiences to the REMID prove that the flexibility of pre-linked code is not a critical problem in case that an intermediate jump table is provided. More significantly, both the memory consumption and code loading overhead of the pre-linked mechanism are less than those in the dynamic linking mechanism. Thus, it is more suitable to be used on the tight resources-constraint WSN nodes.

### V. EXPERIMENTS AND EVALUATIONS

In this section, the performance of HEROS and REMID is evaluated. The evaluation is done on the iLive platform which is equipped with an AVR ATMega1281 microcontroller (8KB RAM and 128 KB ROM).

#### A. Evaluation of HEROS

HEROS is evaluated by comparing with Contiki [9], mantisOS [11] and simpleRTJ OS [19] from the aspects of scheduling model, real-time performance, memory allocation mechanism and memory consumption. And the comparison results are listed in the Table I and II.

Contiki [9] uses the event-driven scheduling in the native scheduling layer, thus it is still an event-driven OS in the native scheduling layer (discussed in previous part C section III). The preemption can be achieved in Contiki, but only within one Contiki process. In result, only the soft RT applications can be supported in Contiki. MantisOS [11] uses the multithreaded scheduling, and the round-robin (RR) scheduling algorithm is used. Since RR is not a real-time scheduling algorithm, mantisOS is not a real-time OS although its RT performance is better than that in the event-driven OSs (e.g., the Contiki). SimpleRTJ OS [19] implements the multithreaded scheduling as well. It is also not a real-time OS not only because the RR scheduling algorithm is used, but also because the thread switch is done in the bytecode granularity. HEROS realizes the hybrid scheduling. By means of the thread preemption and the real-time scheduling algorithm RMS, HEROS becomes a RTOS.

In Contiki and HEROS, the segregated block allocation is used for the memory management. The main difference is that the heap extending mechanism is not implemented in the Contiki. Thus, the allocation will fail if the overflow problem occurs. In mantisOS, the dynamic sequential allocation is used, all the different sizes of objects are allocated sequentially in the heap. Compared with segregated block allocation, the sequential allocation method is more suitable to be used when the objects to be allocated are diverse. In simpleRTJ, the stacks and Java objects are allocated dynamically, but the other objects are reserved statically.

The comparison of the ROM consumption in different OSs is shown in the Table II. For the implementation of the event-driven scheduler, less ROM resources are consumed in the HEROS than that in the Contiki, this is because more event-driven scheduling features are achieved in Contiki, such as the protothread [28] and the high-priority polling scheduling mechanism. For the implementation of the multithreading scheduler, more ROM resources are consumed in the mantisOS than that in the Contiki and HEROS. This is because mantisOS is a pure multithreaded OS, all the tasks in mantisOS are executed by threads. Thus, more threads need to be created. And in order to manage these threads efficiently, six thread queues are used. In result, the scheduling architecture of mantisOS becomes complex, and more ROM resources are taken up. For the implementation of the memory allocator, more ROM resources are consumed in HEROS than that in the other OSs, this is because the heap extending mechanism is implemented in the HEROS allocator.

<table>
<thead>
<tr>
<th>WSN OSs</th>
<th>OS concerns</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scheduling model</td>
<td>RT support</td>
<td>Memory allocation</td>
<td></td>
</tr>
<tr>
<td>Contiki</td>
<td>Native event-driven</td>
<td>No</td>
<td>Dynamic, segregated blocks</td>
<td></td>
</tr>
<tr>
<td>mantisOS</td>
<td>Multithreaded</td>
<td>Not well</td>
<td>Dynamic, sequential allocation</td>
<td></td>
</tr>
<tr>
<td>simpleRTJ OS</td>
<td>Multithreaded</td>
<td>Not well</td>
<td>Mainly static</td>
<td></td>
</tr>
<tr>
<td>HEROS</td>
<td>Hybrid</td>
<td>Yes</td>
<td>Dynamic, segregated blocks</td>
<td></td>
</tr>
</tbody>
</table>
As for the RAM consumption of different OSs, Contiki consumes the minimum, this is because it uses the event-driven scheduling and only one stack is required. mantisOS consumes the maximum RAM resources, this is because mantisOS is a pure multithreaded OS, more threads need to be created in it. For HEROS, the RAM usage is moderate, and the exact required size is dependent on the number of the RT tasks. If no RT tasks are defined, HEROS runs in the pure event-driven model. In this case, the RAM consumption is in the same level as Contiki. However, if the RT tasks exist, HEROS will run in the hybrid scheduling model. For example, if 3 RT tasks and 8 non RT tasks are defined in HEROS, 4 threads will be created: three RT threads and one common_thread. If the thread stack size is set to 120 bytes, then 480 bytes will be consumed. For most current WSN nodes, this requirement of this RAM usage can be met. In the simpleRTJ OS, the RAM consumption is less than that in mantisOS. This is because the thread is switched in the bytecode granularity in mantisOS, thus the size of the thread stack is smaller (section II).

B. Evaluation of REMID.

REMID and EJVM can both decouple the applications from the underlying systems. Besides REMID and EJVM, the dynamic linker in Contiki (ContikiDL) [27] also has this functionality. In this part, these three mechanisms are compared from the aspects of application programming language, application binary format, application binary execution mechanism, multitasking application programming, supported application image size as well as the ROM consumption. The comparison result is shown in the Table III.

The advantage of the EJVM (simpleRTJ) mechanism is that the application can be developed by the Java language, thus the typical Java features such as the exception handling, the automatic GC can be supported. However, the ROM consumption of the EJVM is higher if compared with the other two mechanisms. Although some EJVMs have low ROM consumption, such as the nanoVM and Jwik, the JVM features are limited and the JVM performances are not good. REMID uses the pre-linked machine code and the pre-linking process is done on the PC. Thus, few ROM resources are needed on the WSN device. Moreover, the execution efficiency of the pre-linked machine code is high. ContikiDL uses the dynamic loading mechanism, the application module is high in the flexibility. However, each module needs to be resolved by the linker before being executed. Thus, the execution efficiency of the application code is lower if compared with that in the REMID. Moreover, more ROM resource will be taken up for the implementation of the dynamic linker.

To compare the application image size, an example in the Fig. 11 is implemented by different mechanisms, and the comparison result is shown in the Table III. REMID uses the pre-linked machine code, and the code size is only 120 bytes. ContikiDL uses the ELF application module, and the module size is 786 bytes. SimpleRTJ uses the pre-linked Java bytecode, and the code size is 2482 bytes. The application code size in ContikiDL is larger than that in REMID. This is because the function and variable references should be contained in the application ELF module. The application code of the simpleRTJ is the largest. This is because some extra information should be included in the application image for the Java bytecode interpretation. The application size of REMID is the smallest. This is because it uses the pre-linked machine code, no extra interpretation or resolving information is contained in the application image.

C. Comparison of Different Software Structures

With the development of HEROS and REMID, different software structures can be built to simplify the application development process, such as, the building of EJVM onto the HEROS, the building of REMID onto the HEROS. In the table IV, several software structures are investigated and evaluated from the aspects of multitasking application programming (MAP), the real-time scheduling (RTS), the automatic garbage collection (GC), the application exception handling (EH) and the total ROM consumption. These mechanisms keep a tradeoff between the code size and the performance, and they can be selected in terms of the different applied contexts.

D. Energy Consumption

The byte code is used in the application image of the EJVM while the machine code is used in the application image of the REMID. In the table V, the energy consumed between the calling of an application function and the executing of the corresponding system function in the machine code and the byte code is calculated. From this result, it can be concluded that less energy will be consumed by using the REMID. This result is significant for the WSN since most sensor nodes are deployed in the harsh environments where human cannot access, and the reducing of the energy consumption will prolong the lifetime of the WSN nodes and avoid the labor work of bringing the sensor nodes back to change the batteries.

<table>
<thead>
<tr>
<th>WSN OSs</th>
<th>ROM Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Event-driven Scheduler</td>
</tr>
<tr>
<td>Contiki</td>
<td>936</td>
</tr>
<tr>
<td>mantisOS</td>
<td>N/A</td>
</tr>
<tr>
<td>simpleRTJ OS</td>
<td>N/A</td>
</tr>
<tr>
<td>HEROS</td>
<td>602</td>
</tr>
</tbody>
</table>

TABLE II. MEMORY CONSUMPTION OF HEROS, CONTIKI AND MANTISOS

Figure 11. Application example for size comparison

/* A simple application example for configuring and staring the WSN */
int main(void)
{
      uint8_t Cset[3] = {11, 12, 13}; /* sensor device management */
      network_start();          /* start to establish WSN networking */
}
nodes are abundant, the combination of the full-time JVM is not an advanced technology, it is suitable to be used on an intermediate jump table is used to ease the flexibility problem. Although the pre-linked method is an intermediate jump table is used to ease the flexibility problem. Although the pre-linked method is not reasonable to find the best solution, but it is essential to find a suitable method for a given application context.

Mechanisms have different merits and drawbacks. It is not reasonable to find the best solution, but it is essential to find a suitable method for a given application context.

1). The decoupling of the user application from the underlying system is significant for the application development in WSN. Currently, several mechanisms can be implemented to achieve this objective, including the EJVM, the dynamic linking and the REMID. Different mechanisms have different merits and drawbacks. It is not reasonable to find the best solution, but it is essential to find a suitable method for a given application context.

2). HEROS is a hybrid OS. It can support the multitasking RT WSN applications, and the memory consumption is also not high. Therefore, it can be applied on many WSN nodes, including the BTnodes, Imote, Imote2, Mica, TelosB, SenseNode, SunSPOT, T-Mote Sky, XYZ, etc.

3). REMID uses the pre-linked machine code mechanism to separate the application from the system, and an intermediate jump table is used to ease the flexibility problem. Although the pre-linked method is not an advanced technology, it is suitable to be used on the resource constrained WSN nodes.

4). If the memory and energy resources on the WSN nodes are abundant, the combination of the full-time JVM (e.g., the simpleRTJ VM) and the HEROS can be a good choice. However, if the memory and energy resources are high constrained, the combination of the REMID and the HEROS can be a sound choice.

VI. CONCLUSION

By means of the presentations in the above sections, some conclusions can be reached:

1) The decoupling of the user application from the underlying system is significant for the application development in WSN. Currently, several mechanisms can be implemented to achieve this objective, including the EJVM, the dynamic linking and the REMID. Different mechanisms have different merits and drawbacks. It is not reasonable to find the best solution, but it is essential to find a suitable method for a given application context.

2) HEROS is a hybrid OS. It can support the multitasking RT WSN applications, and the memory consumption is also not high. Therefore, it can be applied on many WSN nodes, including the BTnodes, Imote, Imote2, Mica, TelosB, SenseNode, SunSPOT, T-Mote Sky, XYZ, etc.

3) REMID uses the pre-linked machine code mechanism to separate the application from the system, and an intermediate jump table is used to ease the flexibility problem. Although the pre-linked method is not an advanced technology, it is suitable to be used on the resource constrained WSN nodes.

4) If the memory and energy resources on the WSN nodes are abundant, the combination of the full-time JVM (e.g., the simpleRTJ VM) and the HEROS can be a good choice. However, if the memory and energy resources are high constrained, the combination of the REMID and the HEROS can be a sound choice.

VII. PERSPECTIVES

The ongoing works will focus on the following two topics:

Software reliability: In order to improve the reliability of the HEROS system, the state machine validation approach will be implemented in the next version of HEROS. By using this approach, the system services in the HEROS will be programmed by the state machine. Before the execution of a state code, the global variables related to this state will be stored, and after the state is completed, the execution result will be checked. If the result is not OK, the stored data will be recovered, and the code related to this state will be executed again. By this step-to-step validation, the software reliability can be improved.

Multi-core sensor node for energy conservation: REMID is efficient in the energy consumption, but it is still not sufficient for the energy-constrained nodes. To conserve the energy resources better, the multi-core WSN nodes are currently developing in our work. With the multi-core platform, a sensor node can be configured to work in different modes and adapt to the different application environments. By this way, the energy resources can be utilized more efficiently [29].

ACKNOWLEDGMENT

Our thanks also to the China Scholarship Council (No. 2009627016), the China National Natural Science (No. 60903195), and the support from the project "Space satellite research on collaborative ad hoc networks" by the Innovation Fund of Satellite Application Institute in China Aerospace Science and Technology Corporation.

REFERENCES

Xing Liu is currently a PhD student in Université Blaise Pascal. He received his bachelor engineering degree in 2007 from the Electronic Information School in Wuhan University. In 2004, he obtained the National First Classical Student Scholarship (top 0.3% in Chinese national key universities). In 2007, he was recommended to become a graduate student of Wuhan University without the necessity of participating in the National Entrance Examination for MS. Candidates (top 10% of Wuhan University). In 2009, he received the MS. Engineering degree from the department of Communication and Information System in Wuhan University. From 2009 to now, he was assigned by the Chinese government and the China Scholarship Council to study in the LIMOS (LIMOS, CNRS UMR 6158, FRANCE) to acquire the PhD degrees of both University Blaise Pascal and Wuhan University. His research interests focus on the development of wireless sensor network (WSN), including the development of the real-time WSN operating system, the embedded middlewares and the network protocols.