Reprogramming Embedded Systems at Run-Time

Richard Oliver, Adriana Wilde IEEE(S) and Ed Zaluska SMIEEE
Electronics and Computer Science
University of Southampton, United Kingdom
{rjo2g10,agw106,ejz}@ecs.soton.ac.uk

Abstract—**The dynamic re-programming of embedded systems is a long-standing problem in the field. With the advent of wireless sensor networks and the 'Internet of Things' it has now become necessary to be able to reprogram at run-time due to the difficulty of gaining access to such systems once deployed. The issues of power consumption, flexibility, and operating system protections are examined for a range of approaches, and a critical comparison is given. A combination of approaches is recommended for the implementation of real-world systems and areas where further work is required are highlighted.**

Keywords-Wireless sensor networks; Programming; Runtime, Energy consumption

I. INTRODUCTION

Embedded systems are pervasive in the modern world, being found in systems as widely ranging as fridges, cars, traffic lights, and industrial automation. In the past, embedded systems have relied on purpose-built chips or large and expensive circuits in order to implement their functionality. Due to the availability of inexpensive microcontrollers designed for use in embedded systems, much of the implementation details of such systems has moved from the hardware to the software domain. A long standing problem in the field of embedded systems has been the reprogramming of such systems once deployed. Reprogramming is often required during the initial development of these systems, or to add new functionality and fix bugs once deployed [1], [2]. The relevance of this problem has been highlighted with the advent of the 'Internet of Things' and wireless sensor networks (WSNs). Many of these systems will be difficult (or impossible) to gain physical access to once deployed as well as costly and labour-intensive if nodes are distributed over a wide area, so methods must be devised if we wish to reprogram them [3], [4].

When we consider this problem in the context of WSNs, it is important to realise that the programmers wishing to use the network may not be embedded systems experts, so they may lack an understanding of the underlying operating system (OS). Also, the research areas of the end-users can be as wide-ranging as biology and environmental engineering [4], [5], [6]. Though this factor of accessibility for non-experts is arguably holding back the wider adoption of WSNs [3], [6], this paper focuses more closely on the lower-level problems of embedded systems reprogramming such as power, speed, down-time, and reliability [7]. There have been many different approaches proposed that solve this problem including scripting languages [8], [9], virtual machines (VMs), and various run-time link and load mechanisms. This paper aims to address the problem of dynamic reprogramming of such systems, and will offer comparison and a critical appraisal of the methods currently being used in the field.

The remainder of this paper is structured as follows. Section II explains code execution on embedded devices, section III discusses various existing methods, focusing on the issues of power consumption, flexibility of approach, and OS protection. Section IV follows with conclusions and suggests future work in the area.

II. BACKGROUND

This section will cover the execution of code on microcontrollers. This is typically of no concern when flashing a static image to a microcontroller but a greater understanding is required to implement dynamic code loading.

A factor of considerable importance is the addressing modes available for particular architectures. Two common addressing modes are **absolute** and **relative** addressing. With absolute addressing, jump instructions are provided with the actual address in memory of the destination. With relative addressing, jump instructions are provided with an offset that is added to the current value of the program counter. This is an important distinction to make as, with an embedded system, the program may be loaded at any arbitrary position in program memory. If relative addressing is used then this is not a problem and the program is able to execute from any location in memory without any further changes. If absolute addressing is used then the addresses must be changed to the correct values for the programs placement in memory before execution occurs. Relative addressing can be used as a basis for Position-independent code (PIC), whilst the use of absolute addressing would be an example of relocatable code.

Another important factor is linking. The process of linking is required if the address of a symbol (variable or function) is not known when the code is compiled (e.g., a call to a function provided by the operating system). If the application is compiled separately from the OS and the location of the desired function in memory is unknown, then the device must correctly insert the address of the function before execution. Some embedded operating systems avoid the need for linking by providing a kernel 'jump table'. The jump table is always at the same address in memory and it contains the addresses of any kernel-provided functions. In this way, linking becomes unnecessary. An example of the relocating, linking, and loading process is shown in Figure 1.

TABLE 1: FEATURE COMPARISON OF STUDIED METHODS

	Mechanism						
	VM	**PIC**	**Reloc.**	**OS Protection**	**Kernel Modification**	**Kernel Replacement**	**Loose Coupling**
Maté [11]	●			●			●1
TOSBoot [12]						●	
SOS [13]		●			●		●
Contiki [1]			●			●	●
RETOS [14]			●	●	●		●
Darjeeling [15]	●			●			●
SenSpire [10]			●		●		●
Enix [16]		●2					

1 Only if user-specified instructions are omitted.

2 Kernel-supported PIC

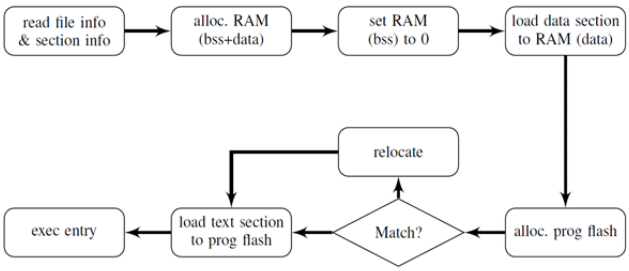

Figure 1. Dynamic linking and loading, adapted from [10]

III. ISSUES

To offer a critical comparison of the mechanisms available we will examine three issues in detail. These issues are power consumption, flexibility of approach, and operating system protection. Table I lists the approaches that will be studied and gives an overview of the features of each approach. The *mechanism* by which code is executed is listed for each approach as either virtual machine, position independent code, or relocatable code. The *flexibility* of each approach is shown by the kernel modification, kernel replacement, and loose coupling columns. An indication of whether the OS is *protected* in any way from remotely programmed code is given in the OS Protection column. These issues will be discussed in detail in what follows.

A. Power Consumption

In the area of embedded systems (wireless sensor networks in particular), power consumption is often of critical importance. In the case of WSNs, where many of the devices may be inaccessible, nodes may be expected to run for weeks (if not months) entirely unattended. All efforts must therefore be made to minimise power consumption when implementing a remote reprogramming mechanism. Considerations must be made when making use of the packet radio and when writing to flash, as these are typically considered some of the most energy intensive activities [7]. Also, when we consider reprogramming at run-time, we must consider if any CPU overhead exists (compared to native code) and whether rebooting the node is required (another power intensive operation).

Levis et al. [11] emphasise the importance of power consumption as a "precious resource" in their development of Maté. Maté is an application-specific virtual machine (or bytecode interpreter) with a high-level interface allowing complex applications to be written in a small amount of code. Programs implemented with Maté are made up of one or more *capsules*. Each capsule can contain 24 instructions and is the size of a single TinyOS packet[1]. This approach greatly reduces application upload energy requirements for certain applications with the most extreme example being *gdi-comm*, a program of 7130 bytes, being expressed in a single Maté capsule. Maté, however, exhibits a significant CPU overhead per instruction issued because it is a virtual machine. This overhead varies depending on the complexity of the instruction issued with the two extremes being 33:1 for a logical **and**, and 1.03:1 for a packet **send**. Besides making programs take longer to execute, this also increases power consumption due to the CPU having to remain powered for longer than it otherwise would before returning to sleep mode. Testing by Levis et al. [11] suggests that this CPU overhead outweighs the energy savings of the concise capsules after running for six days. This figure will vary for different applications but it suggests that Maté is only suitable for non-permanent changes to application code.

Another approach, TOSBoot [12], is a full binary replacement mechanism. The core of the paper concerns the Deluge data dissemination protocol; however, it is the TOSBoot bootloader that is used to re-image the device. This mechanism, although flexible, is exceptionally power intensive. The image transmitted over the network must contain the full operating system (including all libraries, applications, and deluge itself), and is written to external flash,

[1] TinyOS is the operating system that Maté is implemented on top of.

a power intensive operation. The node is then rebooted (also power intensive), and TOSBoot copies the image from external flash into program flash before executing the code, also energy intensive. This is certainly the most power intensive solution discussed in this paper.

Another approach is the SOS operating system by Han et al. [13]. SOS supports dynamically linked modules, enabled with Position Independent Code (PIC). The use of PIC mitigates the need for run-time relocation of code as it is possible to run PIC code anywhere in memory because of the use of relative jumps rather than absolute addresses. It is also possible that PIC code introduces a CPU overhead in the form of an indirection cost [10]; the address that must first be loaded before a jump can be made. In some applications, non-PIC code has been reported to run about 13% faster than PIC code [10]. Further CPU overheads are incurred if a call is made to another module or the kernel. These overheads are 21 and 12 clock cycles respectively, compared to only 4 clock cycles for a direct function call. This mechanism for communication with the kernel is known as a 'jump table'. Overheads are also incurred for function registration and de-registration. Despite these overheads, the execution of the majority of code will be as efficient as code flashed directly to the node. SOS never requires the node to be rebooted which also saves power.

Dunkels et al. present a run-time dynamic linking, relocating, and loading mechanism built on top of the Contiki operating system [1]. This is similar to the loadable modules mechanism provided by SOS. The use of dynamic linking incurs a large overhead as any addresses that were not resolved at compile time must be resolved by the OS. This also incurs a cost when the application is transferred over the network as any unresolved addresses must instead be represented by a much larger symbolic name to enable the dynamic linking. As absolute addresses are used for jumps rather than PIC code, these must be adjusted appropriately by the OS depending on the module's placement in memory; this also incurs overhead. This detailed information of unresolved symbols is typically held (with program code and data) in an executable file format such as the Executable and Linkable Format (ELF). Although the ELF file-format is supported for loading modules, a significant overhead for transferring programs is incurred by its use due to it being designed for 32 and 64-bit systems. Contiki provides the Compact-ELF (CELF) file format designed for embedded systems; this file format is typically half the size of ELF, greatly reducing the transmission overhead.

Cha et al. present RETOS, a Resilient, Expandable, and Threaded Operating System [14]. RETOS implements multithreading unlike the previous event-driven kernels studied such as Contiki and SOS. This is known to cause very large CPU overheads (and hence power use) in embedded systems as during a context switch the stack of one application must be saved whilst that of another is copied into its place. RETOS makes use of dynamic relocation and linking (the mechanism used in Contiki), but makes use of a RETOS-specific file format instead of ELF/CELF. This appears to contain an unnecessary overhead as the file format must contain a hardware-specific section in order to aid the kernel with the relocation process.

Brouwers et al. [15] present Darjeeling, a byte code interpretation approach, similar to that of Maté. Whereas Maté has an application-specific instruction set, Darjeeling is modelled after the Java VM and is a generic VM. As a result, Darjeeling programs will have a larger code size than Maté's thereby causing more power to be used during program upload to the nodes. However, the code size is still significantly smaller than that of native code with the authors demonstrating size reductions of 260 and 215 for the AVR microcontroller and MSP430 respectively for the implementation of the MD5 hashing algorithm. As with Maté, there is also a significant CPU overhead for the interpretation of instructions. CPU overhead was estimated with three real-world examples: bubble sort, vector convolution, and MD5 hashing. CPU overhead was estimated to be between 30–113 times worse than that of native code, depending on application.

Dong et al. have implemented *Dynamic Linking and Loading in Networked Embedded Systems* [10] on the SenSpire OS. This approach is similar to that of both SOS and Contiki. This approach uses relocatable rather than PIC code in the style of Contiki, but also uses the kernel jump table mechanism of SOS. Further improvements have been made over Contiki's CELF executable file format with Slim-ELF (SELF) tested as being 38%–83% the size of CELF. This is achieved by merging the relocation entries for identical symbols causing the relocation to grow linearly with the number of unique symbols rather than the number of unresolved references. The symbol and string tables are also tailored to reduce the total size of the file. The overheads incurred by run-time relocating and linking have also been greatly reduced due to a novel mechanism, shown in Figure 2, which has the host perform the CPU-intensive relocation once program and data memory have been allocated on the node. Pre-linking to kernel functions may also be performed due to the existence of the jump table which is alwasys in the same memory location. Dong et al. estimate that their improvements to loading speed over standard mechanisms are 40%–50%. This represents significant power saving as radio communication is the most expensive activity in a WSN.

Chen et al. present *Enix: A Lightweight Dynamic Operating System for Tightly Constrained Wireless Sensor Platforms* [16]. The size of uploaded programs is expected to be vastly reduced as Enix contains a large library called ELIB (contained on a Micro-SD card) which implements a wide range of common high-level activities. Both ELIB and user programs make use of kernel-supported PIC via code-modification. All calls to kernel functions or the ELIB library are linked fully at compile time removing any need for run-time linking for these calls. However, calls to ELIB could potentially be expensive as it exists in virtual memory and may have to be loaded itself when called. To avoid relocation costs, any code that makes use of an absolute jump is replaced at compile-time. The code that replaces it causes the program to store the program counter at run-time, and add the relative offset to it before making the jump. In this way relocation does not have to be handled at run-time, however, the single *ljmp* instruction is replaced instead by 18 instructions and includes a function call which will increase the transmitted program size and power requirements.

B. *Flexibility of Approach*

This section will discuss the flexibility of the various different approaches. In particular, the granularity of the code update mechanism will be examined. That is whether variables, programs, or the entire system image must be changed. The nature of applications for these different solutions will be examined, in particular, whether applications may be shipped as different modules that can communicate with each other. An account will be given as to whether or not the update mechanism allows for kernel, driver, or other low-level functionality to be updated. The coupling for applications and kernels will also be discussed, i.e. whether the core and application must be compiled by the same developer and whether or not applications rely on a specific kernel version.

Maté [11] is a relatively inflexible solution. The constructs/ instructions that Maté provides makes it possible to easily write and rapidly prototype common sensor network applications with a minimal amount of code. As Maté is Turing complete, it is technically possible to represent any algorithm with the instructions provided. This is ill-advised, however, if the algorithm is mathematically complex as the overhead on simple operations may be too great for Maté to compete with the other approaches. To solve this problem, the authors of Maté reserve eight instructions that can be user defined in the initial image placed on the device. Although this appears to address the above problem, these custom-defined instructions cannot be modified once the device has been deployed meaning their use is limited if vastly different behaviour is required. Maté also cannot be used to modify any other native code, meaning that it is not possible to modify the kernel or fix bugs in drivers. Many Maté applications may run simultaneously, and they can communicate with each other through a single shared variable, or more permanent storage. Maté applications are entirely independent of kernel versions and hardware platforms, and are entirely portable as long as the custom-defined instructions are not used. Maté's 'capsules' are also version numbered, meaning that they can be replaced without difficulty as the latest version of any program will always be automatically acquired.

TOSBoot [12] is an exceptionally flexible, albeit low complxity, approach. As TOSBoot is a complete binary-image replacement mechanism then any system functionality may be changed including bug-fixes, driver updates, or even an entirely new operating system if so desired. TOSBoot provides additional flexibility by having the ability to handle multiple binary images. This allows the node to be imaged with any of the stored images at boot-time meaning an older version of an image does not have to be re-uploaded over the network if it should be required again at a later date. TOSBoot is inflexible however in the fact that large binary images must be sent over the network, this mechanism may be too slow for a large amount of nodes. Also, any image that is used must have support for the Deluge protocol included if any other updates are to be carried out in the future; this also increases the size of the binary images that must be sent. TOSBoot also requires that nodes have access to external storage, such as flash, to store uploaded images, something that may not be present on all systems.

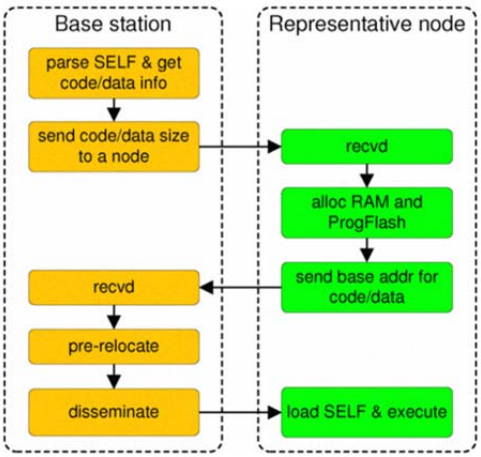

Figure 2. The pre-relocating process in SenSpire, reproduced from [10]

SOS [13] could be considered an intermediate solution between Maté and TOSBoot. As SOS programs are written in native code, it offers more flexibility than Maté because the programs are not limited by the expressibility of the underlying VM. SOS can handle multiple applications running simultaneously, and applications are able to communicate with each other by registering function entry points with the kernel. This, however, requires a message handler function to be placed at a specific place in the binary image. This could be viewed as an inflexible solution because it requires non-standard actions to be taken during the compilation process. Applications are also able to make use of dynamic memory and may transfer ownership to other applications as a means of inter-process communication (IPC). The mechanism provided by SOS can be used to update low-level functionality such as drivers, but cannot be used to update the SOS kernel. As in Maté, modules in SOS are version numbered and SOS will automatically acquire new versions of modules and insert them if it is able to do so. SOS programs are limited to 4KB on the AVR platform due to the use of PIC code. They also cannot make use of global variables as no address information for them is available at compile time. SOS is an example of a loosely coupled OS because of the use of a jump table. This means that SOS programs are independent of different kernel versions, applications may be distributed in binary form and recompilation is not required if the underlying OS is updated or upgraded.

Contiki [1] provides similar flexibility to SOS being a 'loadable module' solution. Contiki makes use of relocatable rather than PIC code, avoiding the disadvantages discussed previously. As Contiki provides dynamic linking and relocation, a standard compilation procedure can be used to create a binary in the ELF file format. This increases flexibility and simplifies the compilation process. As ELF is a standard format, various development tools are available that are able to operate on it. In a similar fashion to SOS, Contiki modules are able to communicate with each other by registering functionality with the kernel. Module versioning is not explicitly supported as in SOS; however, a program may replace a currently running program acquiring its state resulting in no downtime whatsoever. Mottola et al. have attempted to

implement versioning of modules on top of Contiki with FiGaRo [17]. The mechanism provided by Contiki cannot replace any of the core functionality at run-time including any low-level device drivers. Contiki does, however, contain a mechanism similar in functionality to TOSBoot that will allow the kernel and device drivers to be updated with the aid of a reboot. Contiki also achieves loose coupling due to the core's ability to perform dynamic linking with the use of a full symbol table. Although this symbol table is larger than SOS's jump table, Contiki's method avoids the indirection cost of using such a system.

RETOS [14] is another loadable module approach which uses relocatable code in a similar fashion to Contiki. Compiled code is stored in a RETOS-specific custom file format which potentially could reduce the flexibility of the system as standard analysis tools can no longer be used. The core system is able to dynamically link and relocate code, similar to Contiki. The RETOS kernel is split into a static image and loadable kernel modules. These modules are dynamically loaded/unloaded depending on the current requirements of user applications meaning that a minimal system is maintained at all times. RETOS does not provide any mechanism to update the static kernel image, but loadable kernel modules may be replaced. RETOS makes use of a jump table to access kernel and kernel module functionality. Modules may also make their functionality available and enable IPC by registering their functionality in this table. Due to the use of a jump-table, RETOS is another example of loose coupling.

Darjeeling [15] (the Java inspired VM) is similar to Maté in its abilities. As Darjeeling enables programs to be written in Java, it is straightforward for a programmer to write large and complex programs that take full advantage of the Java language features (e.g. object orientation, dynamic memory management, threading, exception handling, and garbage collection). As Darjeeling is a VM, it cannot be used to update any native code such as device drivers or the core kernel. Darjeeling provides support for calling native code but this must be compiled into the static image originally flashed to the node. This means that support for all hardware and any node-tonode communication must be written in advance. Although multiple Java applications are able to run simultaneously, they are unable to communicate with each other because of the lack of support for reflection.

SenSpire [10] is very similar to Contiki in concept with its dynamic link and load mechanism. However, unlike Contiki, SenSpire makes use of a kernel jump table which enables lowlevel kernel functions to be incrementally upgraded. The inclusion of a jump table also enables loose coupling. SenSpire, like Contiki, also makes use of a variation of the ELF file format which may be judged as being advantageous. It is unclear how IPC is handled in SenSpire, or indeed if it is possible at all.

Enix [16] makes use of ELIB, a dynamic loading library. Due to Enix's fast loading times due to kernel-supported PIC, components from ELIB may be loaded from virtual memory as applications require them in a similar style to RETOS. Whilst this mechanism can be used to access library functionality existing on the node when it was deployed, Enix currently does not have the ability to add additional programs to virtual memory at run-time. This seems to suggest that loose coupling is not available on Enix as the position of the required library in virtual memory must be known at compile time. No mention is made of IPC in Enix, although it is assumed that this is not possible due to the one-way nature of communication between remote programmed applications and the kernel/ELIB.

C. OS Protection

This section will give an account of OS protections for the approaches that implement them. In particular, whether or not an application is able to write to arbitrary memory locations and interrupt execution of other applications (or even the kernel itself) will be examined.

Both Maté [11] and Darjeeling [15] as virtual machines offer protection of the underlying OS. The code for both of these approaches is interpreted meaning that they have no way of executing code on hardware or otherwise disrupting the operating system. This is typically an issue for embedded operating systems as the lack of Memory Management Unit (MMU) means that applications may arbitrarily write to any memory location, potentially crashing the node. Because an embedded OS usually does not offer pre-emptive threading it is often possible for an errant application to crash the node (as it is never forced to pass execution back to the scheduler/kernel).

RETOS [14] makes use of a software technique known as 'application code checking' to work around the shortcomings of typical embedded system hardware, principally the lack of hardware memory protection such as a MMU. This technique operates by checking the source/destination field of hardware instructions to ensure they are in fact reading/writing to permitted memory locations. This is split up into both static and dynamic checks. Static checks can be carried out at compile-time for direct or immediate addressing instructions. Some checks, however, must be carried out at run-time if they use indirect addressing; these checks form the dynamic category of checks. These dynamic checks are inserted into the code at compile-time. Any illegal memory accesses not detected by the static checks will be caught when the application is run and errors will be reported to the kernel. RETOS also has dual mode operation (i.e. user/kernel modes) which is enabled by stack switching. Because of this context switching, threads are not forced to share the same stack which makes it easier for the end user to write code and offers additional protections. As user-mode threads are preempted in RTOS, an errant thread is unable to impede the proper execution of other threads or indeed the kernel.

Kajtazovic et al. have examined the use of dynamic loading mechanisms in safety-critical domains and suggest an approach where the results of linking can be predicted at compile-time rather than run-time [18].

IV. Conclusion and Future Work

This paper has examined several approaches for the run-time reprogramming of embedded systems. These methods included virtual machines, full binary image replacement mechanisms, kernel-supported PIC code, dynamic linking/loading and variations thereupon. The issue of power consumption was identified as being the one of most

significance, particularly in the area of wireless sensor networks.

Virtual machines such as in Maté [11] and Darjeeling [15] allow complex programs to be expressed in a very small amount of code. Due to the associated smaller data dissemination costs and the expressibility of their instruction sets, it would appear that VMs offer the best approach for the remote reprogramming of embedded systems. However, the CPU overhead of using such systems quickly mitigates any power savings from lower network communication costs. This, coupled with the lack of flexibility for fixing low-level bugs, makes VMs only suitable for temporarily re-tasking an embedded system or for rapid prototyping. The use of VMs is reliable however, and they offer the most protection to the underlying OS out of all the methods studied.

TOSBoot [12], conversely, allows any part of the system to be updated but is too power inefficient to be used for frequent updates such as those that might be generated during rapid prototyping of a network. The approach implemented by SOS [13] is also limited in its utility due to its reliance on PIC code, a mode of addressing only supported by certain hardware architectures and compilers. Although Enix [16] attempts a fix by the use of kernel-supported PIC, this mechanism incurs significant runtime overhead for jump instructions. Coupled with the lack of loose coupling, the inability to fix kernel/driver bugs and the lack of OS protection, Enix will not be suitable for many practical applications.

The approaches used by Contiki [1], RETOS [14], and SenSpire [10] all make use of relocatable code. Although this mechanism is typically associated with the large CPU overheads required for dynamic relocating and linking, SenSpire avoids this cost by pre-relocation of code. This approach, although innovative, is only of use if a single device is to be updated. If a bug-fix is to be distributed to an entire network, the same overheads present in Contiki and RETOS will be incurred. Contiki is clearly the most flexibile solution, as it is possible to replace the core kernel. However, RETOS is the only native code mechanism that implements any kind of OS protection.

In conclusion, native code update mechanisms using relocatable code are clearly the best solution for long-term updates to embedded systems because of their lack of run-time overhead once loaded. However, virtual machines should be used in conjunction with such mechanisms to allow temporary repurposing of the system and for rapid prototyping. Future work should focus on operating system protection. Currently remote reprogramming mechanisms are undermined as it is possible for incorrectly written programs to crash the nodes they are executing on. If the reliability of these mechanisms is improved, their use will become more widespread, and can be expected to include many practical, real-world applications.

REFERENCES

[1] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, “Run-time dynamic linking for reprogramming wireless sensor networks,” in Proceedings of the 4th international conference on Embedded networked sensor systems. ACM, 2006, pp. 15–28.

[2] R. K. Panta, S. Bagchi, and S. P. Midkiff, “Efficient incremental code update for sensor networks,” ACM Transactions on Sensor Networks (TOSN), vol. 7, no. 4, p. 30, 2011.

[3] Q. Wang, Y. Zhu, and L. Cheng, “Reprogramming wireless sensor networks: challenges and approaches,” Network, IEEE, vol. 20, no. 3, pp. 48–55, 2006.

[4] R. Sugihara and R. K. Gupta, “Programming models for sensor networks: A survey,” ACM Transactions on Sensor Networks (TOSN), vol. 4, no. 2, pp. 8:1–8:29, 2008.

[5] L. S. Bai, R. P. Dick, and P. A. Dinda, “Archetype-based design: Sensor network programming for application experts, not just programming experts,” in Proceedings of the 2009 International Conference on Information Processing in Sensor Networks, ser. IPSN ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 85–96. [Online]. Available: http://dl.acm.org/citation.cfm?id=1602165.1602175

[6] L. Mottola and G. P. Picco, “Programming wireless sensor networks: Fundamental concepts and state of the art,” ACM Computing Surveys (CSUR), vol. 43, no. 3, p. 19, 2011.

[7] N. Bin Shafi, K. Ali, and H. S. Hassanein, “No-reboot and zero-flash over-the-air programming for wireless sensor networks,” in Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2012 9th Annual IEEE Communications Society Conference on, 2012, pp. 371–379.

[8] A. Dunkels, “A low-overhead script language for tiny nenetwork embedded systems,” SICS Research Report, 2006.

[9] M. Kovatsch, M. Lanter, and S. Duquennoy, “Actinium: A restful runtime container for scriptable internet of things applications,” in Internet of Things (IOT), 2012 3rd International Conference on the. IEEE, 2012, pp. 135–142.

[10] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Liu, “Dynamic linking and loading in network embedded systems,” in Mobile Adhoc and Sensor Systems, 2009. MASS’09. IEEE 6th International Conference on. IEEE, 2009, pp. 554–562.

[11] P. Levis and D. Culler, “Mat´e: A tiny virtual machine for sensor networks,” in ACM Sigplan Notices, vol. 37, no. 10, 2002, pp. 85–95.

[12] J. W. Hui and D. Culler, “The dynamic behavior of a data dissemination protocol for network programming at scale,” in Proceedings of the 2nd international conference on Embedded networked sensor systems. ACM, 2004, pp. 81–94.

[13] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, “A dynamic operating system for sensor nodes,” in Proceedings of the 3rd international conference on Mobile systems, applications, and services. ACM, 2005, pp. 163–176.

[14] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon, “Retos: Resilient, expandable, and threaded operating system for wireless sensor networks,” in Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on, April 2007, pp. 148–157.

[15] N. Brouwers, K. Langendoen, and P. Corke, “Darjeeling, a feature-rich vm for the resource poor,” in Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems. ACM, 2009, pp. 169–182.

[16] Y.-T. Chen, T.-C. Chien, and P. H. Chou, “Enix: a lightweight dynamic operating system for tightly constrained wireless sensor platforms,” in Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems. ACM, 2010, pp. 183–196.

[17] L. Mottola, G. P. Picco, and A. A. Sheikh, “Figaro: Fine-grained software reconfiguration for wireless sensor networks,” in Wireless Sensor Networks. Springer, 2008, pp. 286–304.

[18] N. Kajtazovic, C. Preschern, and C. Kreiner, “A component-based dynamic link support for safety-critical embedded systems,” in Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the. IEEE, 2013.