An Operating System Infrastructure for Remote Code Update in Deeply Embedded Systems

Giovani Gracioli and Antônio A. Fröhlich

Federal University of Santa Catarina (UFSC)
Laboratory for Software and Hardware Integration (LISHA)
PO Box 476 - 88049-900 - Florianópolis, SC, Brazil
{giovani,guto}@lisha.ufsc.br

Abstract

Deeply Embedded Systems are designed to perform a determined set of specific tasks, usually on low-cost, high-reliability platforms. In order to support on-site firmware updates, such systems are subject to severe resource limitation, like processing power, memory and energy, since the update mechanism itself must share the sparse resources with running applications. This work presents a low-overhead operating system infrastructure for remote code update. By using sophisticate C++ static metaprogramming techniques, the infrastructure and the code update become fully transparent to applications. Moreover, the infrastructure allows the system components to be marked as updatable or not at compilation time, and for those components that are not marked as updatable, no overhead is added. The infrastructure was implemented in EPOS, a multi-platform, component-based, embedded operating system [7], with positive preliminary results that show that the proposed strategy is a feasible solution for software update in deeply embedded systems.

Categories and Subject Descriptors C.3 [*Real-time systems and embedded systems*]; D.4.7 [*Organization and Design*]: Real-time systems and embedded systems; D.4.9 [*Systems Programs and Utilities*]: Loaders

General Terms Design, Experimentation

Keywords Embedded Systems, Operating Systems, Remote Update

1. Introduction

Deeply Embedded Systems are designed to perform a determined set of specific tasks, usually on low-cost, high-reliability platforms with severe resource limitation, including processing power, memory and energy. Yet, it is highly desirable that such systems be able to undergo online software updates, be it to correct bugs, add new features, or adapt the system to varying execution environments.

In this scenario, whichever update mechanism devised, it will have to operate under even more severe resource limitations, for it will compete for resources with the target embedded system and yet must not disrupt the embedded system operation [6]. Modern automotive systems, for instance, feature hundreds of dedicated microcontroller units that interact to perform specific functions, such as ABS (Anti-lock Brake System) and PCM (Powertrain Control Module). A study by Daimler-Chrysler has pointed out that updating such distributed embedded systems on-wheels in an authorized garage takes over 8 hours, with extremely high costs [5]. Another example are Wireless Sensor Networks, which comprise a large number of small sensors scattered throughout a given environment. Very often, collecting such sensors back for in-lab reprogramming is unpractical. In both cases, remote update mechanisms would be of great value, yet they would have to operate along with the target embedded systems on platforms that seldom exceed an 8-bit processor, some few kilobytes of memory and an interconnect (e.g., RS-485, CAN, ZigBee).

In this context, this paper presents a low-overhead infrastructure for remote code update developed around EPOS, a multi-platform, component-based, embedded operating system [7]. By using sophisticate C++ static metaprogramming techniques, the infrastructure and the code update become fully transparent to applications. The proposed code update mechanism is build within EPOS component framework, around the *remote invocation* aspect program, thus also giving the possibility to mark a system component as updatable or not at compile-time, and for those components that are not marked as updatable, no overhead is added. The infrastructure makes the component memory position independent, eliminating the need for a bootloader and/or linker to be installed in each node in order to perform update.

HotSWUp'08, October 20, 2008, Nashville, Tennessee, USA.
Copyright © 2008 ACM ISBN 978-1-60558-304-4/08/10…$5.00

2. Related Work

Reijers and Langendoen created an infrastructure that reduces the amount of data transferred over the network and also the energy consumed by each node by receiving only the difference between the new and the old system images [16]. After receiving the new image and checking for transmission errors, the system is restarted and the new image is loaded. Besides requiring a system restart, this technique is limited to binary updates and requires the whole setup of nodes to be stored at a central station.

FLEXCUP is an update system for TINYCUBUS. It relies on meta-data collected during the compilation of components in order to support updates [15]. Meta-data include symbol and relocation tables that are subsequently used by a linker installed on each node in order to merge updates into the final image. The system must be restarted after updating. One major disadvantage of this approach is the dependency established between the compiler and the update infrastructure, since both must work together for an update to succeed.

Felser also uses information collected by compiler to identify situations in which a “safe” update is possible [6]. When an unsafe update is found (e.g. an update involving a function currently being executed), the system first asks for human intervention before proceeding with the update and thus tries to preserve the system state. Besides being dependent on the compiler, this solution assumes that the person caring out the update has deep knowledge about the software architecture while deciding whether an upgrade is safe or not. Koshy and Pandey try to reduce upgrade overhead by partitioning the update among nodes and base stations with higher processing capacities [13]. It relies on an incremental linker that is able to control where updates (i.e., modified functions) must be placed on the node’s memory.

2.1 Virtual Machines

MATÉ [14] is a virtual machine that runs on top of TINYOS [11]. It provides a few high-level instructions (e.g. sense, halt) that enables applications to be easily coded. The reduced instruction set imply in small bytecode images, which in turn reduces transmission time and energy consumed during updates. Nevertheless, it also considerably limits the number of applications that can be built [3]. Moreover, for long-running applications, the energy spent to interpret the bytecode can outweigh the benefits [14]. MATÉ’s `forw` instruction is used to broadcast code to neighbor nodes.

SENSORWARE implements a VM that supports node programming through a sensor script language [3]. The language features commands to replicate and migrate code and data across nodes. Its deployment in the scenario of deeply embedded systems, however, is compromised by excessive resource utilization (according to authors, SENSORWARE is too large to run on a traditional AVR-based mote).

DVM is a VM built on top of SOS [9]. The VM interprets high-level scripts written in a language that is compile to a portable bytecode format [2]. It relies on SOS modules to load and unload VM extensions at run-time. Similarly to other VMs, DVM is usually too resource consuming to support a deeply embedded system. Any VM-based solution for this scenario is limited by the challenge of having to implement a bytecode interpreter in such an extremely constrained environment and therefore is seldom used [13].

2.2 Data Dissemination Protocols

Although not directly addressed by this paper, data dissemination protocols are fundamental for any update infrastructure that aims at addressing mass updates (i.e. updates of several nodes at the same time). Partitioning, routing, propagation, and reliability are some of the main guidelines of such protocols. MOAP [17] and DELUGE [12] are data dissemination protocols implemented in the realm of TINYOS. These protocols implement packet retransmission, segment management and sender selection to reduce energy and memory consumption during update operations. MNP uses a sender-centered scheduling mechanism to avoid collisions and message losses [19]. In order to reduce overall energy consumption, nodes not involved in an update are put in sleep mode whenever a segment is transmitted.

The infrastructure proposed in this paper could benefit from any of these dissemination protocols.

2.3 Operating Systems

CONTIKI is an operating system for wireless sensor networks that implements special processes, called *services*, that provide functionality to other processes [4]. Services can be replaced at run-time through a *stub interface* that is responsible for redirecting function calls to a *service interface*, which holds the actual pointers to the functions implementing the corresponding services. The *service interface* also keeps track of versions during updates.

SOS is an operating system for sensor networks that allows nodes to be updated on-the-fly [9]. The operating system is built around modules that can be inserted, removed or replaced at run-time. By using relative calls, the code in each module becomes position independent. References to functions and data outside the scope of a module are implemented through an indirection table and, in some cases, are simply not allowed. This approach is conceptually similar to the one proposed in this paper, but the update infrastructure presented in the next section takes advantage of static metaprogramming techniques to eliminate part of the overhead associated to indirection tables, thus yielding a slimmer mechanism and also achieving application transparency.

3. Assumptions

The following assumptions are made for our code update mechanism: *(i)* it supports online updates and the update unit is a component. The system components can be marked as updatable or not at compile-time and only those marked as

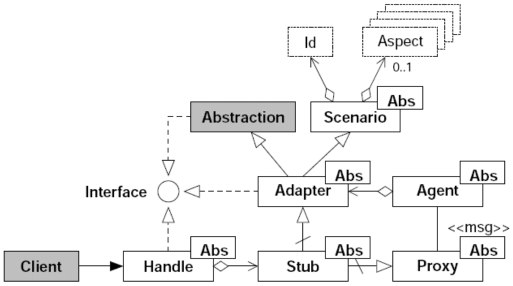

Figure 1. EPOS metaprogrammed framework overview [7].

updatable can be updated. For those that are not marked as updatable, no overhead is added. It also guarantees the upgrading of a component that is not being used at the moment of an updating; *(ii)* the method signatures of the updatable component must be the same in the two versions (no API changes). Since a component is marked as updatable at compile-time, all dependencies between this component and the rest of the system are already resolved, allowing a good performance at run-time. On the other hand, this limits the update scope. Then, we decided to have a good performance at run-time instead of a big update scope; *(iii)* the hardware is not modified, and both hardware and software have the same endianess; *(iv)* the system does not have virtual memory; and *(v)* it is up to the developer to certify that the new component upgrade does not remove any method that is being used by other components.

4. Infrastructure for Remote Code Update

Embedded Parallel Operating System (EPOS) [7] is a multi-platform, component-based operating system for embedded systems. Although no infrastructure for remote update exists in EPOS, the operating system has a C++ static metaprogrammed framework for remote invocation, where characteristics like confinement and isolation are found. Both characteristics are important to encapsulate the system components by creating an indirection level between every component method call configured as remote. This makes the components memory position independent and allows the components' replacement without restarting the system.

4.1 EPOS Remote Method Invocation

An overview of the EPOS metaprogrammed framework is presented in Figure 1. In this Figure, the Client wants to invoke a method of an Abstraction (e.g. a system component that can a Thread, UART, etc). The parameterized class *Handle* receives a system abstraction as parameter. It acts as a handle for the supplied abstractions, forwarding the method invocations to *Stub* element.

The *Stub* element is a parameterized class responsible for verifying whether the aspect of remote invocation is active for that abstraction (the remote invocation aspect is selected by the abstraction through its *Traits Class* [18]). If it is not, the *Stub* will inherit the abstraction's scenario adapter [8]. Otherwise, a *Stub's* specialization, namely *Stub<Abstraction, true>*, will inherit the abstraction's *Proxy*. Therefore, when *Traits<Abstraction>::remote = false*, makes *Handle* to take the scenario adapter as the *Stub*, while making it true makes *Handle* to take *Proxy*.

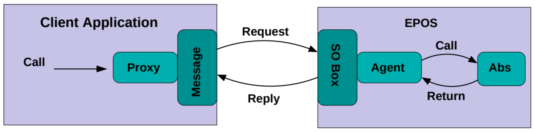

Figure 2. A method invocation with code update support.

The *Proxy* is responsible for sending a message with the abstraction method invocation to its *Agent*. A message contains the object, method and class IDs that are used by *Agent* to invoke the correct method, associating them to a method table. The object ID is used to get the correct object before the method call. The *Agent* receives the message and invokes the method through the *Adapter* class.

Adapter class is responsible for applying the aspects supplied by *Scenario* before and after of the real method call. Each instance of *Scenario* consults the abstraction's *Traits* to verify which aspects are enabled to that abstraction, aggregating the corresponding scenario aspect. When an aspect is not selected to abstraction, an empty implementation is used. In this case, no code is generated.

4.2 Support for Code Update

The framework infrastructure was extended according to Figure 2. The invocation of a component method of the client application passes through *Proxy* which sends a message to the *Agent*. After the method execution, a message with the return value is sent back to the application. With this structure, an indirection level is created among the application method call, making the *Agent* the only one aware of the component's position in the system memory. The *OS Box* at *Agent* controls the access to the component method through a synchronizer (*Semaphore*), avoiding the invocation of the component method while it is being updated.

A Thread created during the system's bootstrapping is responsible for receiving an update request and the new component's code (the code must be in an object file). This request is sent to the *Agent* and contains the class, method and object IDs, the new component code to be updated, the new relative addresses inside of the object file and the code size. The *Agent* has a vector with the methods' position related to the component methods (object file). The external references used in this new component are solved by making the linking of the new component with the old image system, that has not changed.

In the remote update scenario, the ID method is referring to the *Update* method. Inside this method, the *Agent* allocates memory for the new code, copies the received code to this new position, updates the received addresses of the

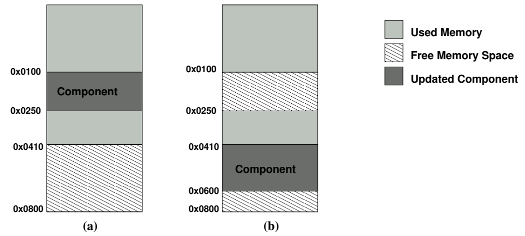

Figure 3. System memory example. (a) before updating (b) after updating.

new methods, destroys the old object, creates the new object and adds the new object to the objects table. Figure 3 shows the system memory before and after a component update. Although the creation of a new object demands a memory allocation/release, this process in EPOS does not aggregate much overhead to the application, as the memory management is designed to be extremely efficient, as exemplified in the next section comparison.

The framework infrastructure with the update system is transparent to application. However, each addition of a new component to the system requires that its method are placed in the framework infrastructure to allow the update support for this new component. With the update support enabled in the system, there is additional memory consumption and the components method call suffers a little delay.

5. Preliminary Results

In order to evaluate the applicability of the described infrastructure, was used a dining philosopher's application in the Mica2 Platform [10]. From the test application, were evaluated two metrics compiling the system with the GNU g++ 4.0.2: the memory amount needed by the framework and the performance overhead generated by the indirection level. Table 1 presents the framework memory usage for all non-empty sections. The framework adds 1710 bytes to the application, in which 362 bytes are added to bootloader section due to the function responsible for writing in the flash memory, 192 and 1132 bytes to data and text sections respectively, due to pointers and code and finally 24 bytes to bss section. When a new component is selected to support the update mechanism, the added memory space depends on the number of the component methods, because each method must have its address stored, increasing the table size that saves these addresses and consequently the framework size.

Table 1. Infrastructure memory consumption for the Thread component.

Section	*Without (bytes)*	*With (bytes)*	*Overhead (bytes)*
.bootloader	0	362	362
.data	476	668	192
.text	28386	29518	1132
.bss	278	302	24
TOTAL	**29140**	**30850**	**1710**

Table 2 shows the overhead generated by the remote update support infrastructure when a method of the *Thread* component is invoked. These values were measured by using the system *Chronometer* abstraction. The results are the representation of the values average obtained in 10 executions, disregarding the highest and lowest values. The *Thread* constructor time incurred an overhead of 100% of the original latency due to the *Thread* function passed to the framework and the *Thread* creation through the memory allocation. As the method's computation time increase, the influence of remote update infrastructure is minimized. It is what occurs in the *Yield* and *Pass* methods. This little overhead is achieved due to static metaprogramming which resolves all framework dependencies among system components, execution scenarios, and applications at compile-time.

Table 2. Time to invoke a Thread component method with and without code update support enabled.

Method	*Without (us)*	*With (us)*	*Overhead(%)*
Constructor	95	190	100%
Suspend	34	52	52.94%
Resume	69	86	24.64%
Yield	20,254.5	20,329	0.37%
Pass	7,715.5	7,728.5	0.17%

Figure 4 presents a comparison between the allocation and release memory functions in EPOS [7], *MantisOS 1.0* [1] and *SOS 2.0.1* [9] OSs. The number of bytes allocated/released were 8, 32, 64, 128, 256, 512 and 1024. *Contiki* was not evaluated because its version in the ATMega128 microcontroller is unstable. *SOS* was chosen because it implements remote update support and *MantisOS* was selected because it is widely known in the embedded systems scenario. TINYOS was not used in comparisons because it does not support memory allocation/release [11]. The test values were measured using the memory allocation/release and time measure functions present in each operating system. The results are the representation of the average of the values obtained in 1000 experimental runs. The Figure 4 shows that EPOS presented a better performance in terms of memory allocation. In terms of memory release the three OSs have similar behavior. *MantisOS* memory allocation has proven to be dependent on the number of bytes allocated. EPOS memory management does not depends on memory size and the good performance is achieved due to the C++ metaprogrammed list that is used to handle the memory free spaces. Therefore, the destruction and creation of a new object at the moment of an update is not a performance problem. Furthermore, static allocation presumes that all objects are always in memory and sometimes the embedded system cannot have enough memory for all objects.

6. Conclusion

This work presented a infrastructure for remote code update developed around EPOS component framework, which was modified in order to enable the confinement and isolation of

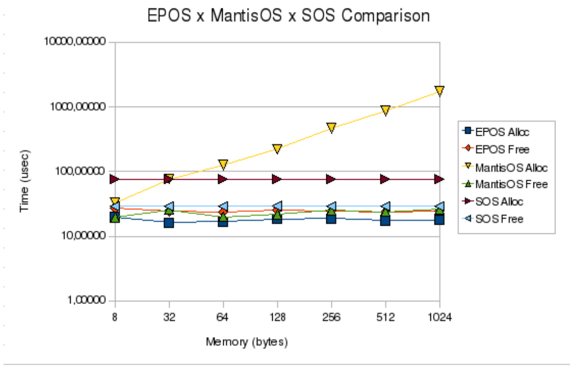

Figure 4. EPOS x MantisOS X SOS memory allocation and release comparison in the Mica2 platform.

system components in physical modules, thus making them memory position independent. The infrastructure also allows the developer choose which components can be updated.

A preliminary prototype evaluation has shown that the remote update infrastructure adds little memory and processing overhead to components marked as updatable and therefore does not compromise the availability of system resources and services at the application level. This is mainly due to the use of sophisticate static metaprogramming techniques, which enable the remote update infrastructure to resolve dependencies among system components, execution scenarios, and applications at compile-time.

The next step in the project is to extend remote code update mechanism to support code distribution over the network through a dissemination protocol focused on incremental differences between new and old versions of components.

References

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: system support for multimodal networks of in-situ sensors. In *Proc. of the 2nd ACM int. conf. on Wireless sensor networks and applications*, pages 50–59, NY, USA, 2003. ACM.

[2] Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. Multi-level software reconfiguration for sensor networks. In *Proc. of the 6th int. conf. on Embedded software*, pages 112–121, NY, USA, 2006. ACM.

[3] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. of the 1st int. conf. on Mobile systems, applications and services*, pages 187–200, NY, USA, 2003. ACM Press.

[4] Adam Dunkels, Bjrn Grnvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, USA, 2004.

[5] Friedrich-Alexander-Universitt Erlangen-Nrnberg. Technologie roadmap: Software-verwaltung im farzeug. Technical report, DaimlerChrysler, Erlangen, Germany, Sep 2003.

[6] Meik Felser, Rüdiger Kapitza, Jörgen Kleinöder, and Wolfgang Schröder-Preikschat. Dynamic software update of resource-constrained distributed embedded systems. In *International Embedded Systems Symposium*, 2007.

[7] Antônio A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, S. Aug., Aug 2001.

[8] Antônio A. Fröhlich and Wolfgang Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proc. of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

[9] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proc. of the 3rd int. conf. on Mobile systems, applications, and services*, pages 163–176, NY, USA, 2005. ACM.

[10] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. The platforms enabling wireless sensor networks. *Commun. ACM*, 47(6):41–46, 2004.

[11] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[12] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the 2nd int. conf. on Embedded networked sensor systems*, pages 81–94, NY, USA, 2004. ACM.

[13] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proc. of the second European Workshop on Wireless Sensor Networks*, pages 354–365, Jan 2005.

[14] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[15] Pedro Jos Marrn, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Proc. of the Third European Workshop on Wireless Sensor Networks*, pages 212–227, February 2006.

[16] Niels Reijers and Koen Langendoen. Efficient code distribution in wireless sensor networks. In *Proc. of the 2nd ACM int. conf. on Wireless sensor networks and applications*, pages 60–67, NY, USA, 2003. ACM.

[17] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical report, LA, CA, USA, 2003.

[18] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[19] Limin Wang. Mnp: multihop network reprogramming service for sensor networks. In *Proc. of the 2nd int. conf. on Embedded networked sensor systems*, pages 285–286, 2004.