

# Towards Dynamically Composed Real-time Embedded Systems

Leandro Batista Ribeiro and Marcel Baunach

Institute of Technical Informatics - ITI  
Graz University of Technology, Austria  
{lbatistaribeiro|baunach}@tugraz.at

**Abstract.** Embedded systems are present virtually everywhere, from simple home appliances to complex aerospace systems, and the establishment of the Internet of Things (IoT) increases this ubiquity. Naturally, after deployment, updates might be necessary due to bugs found in the original implementation, improvement of tasks, security holes, etc. However, some domains require the deployment of a big number of devices, which often cover a vast area or are unreachable after deployment, and lead to very expensive, or even impossible updates with physical access to the devices. Therefore, remote updates are of utmost importance. In this paper, we present the existing approaches for remote updates, and analyze the overheads each one causes in the devices and in the update server. Furthermore, we present our concept for coping with dynamically composed real-time systems. Our main goal will be schedulability analysis during dynamic updates.

## 1 Introduction

Embedded systems are already present in virtually all application domains (automotive, robotics, home automation, smart production and logistics, etc.). In addition, the establishment of the IoT will introduce tons of new smart devices in the embedded world and heavily increase the number of connected devices.

As the number of devices and application domains grow, the need for remote update mechanisms arises, in order to i) keep the systems always up-to-date in the long run: bug fixes, security patches, etc. are always needed after deployment and it is important to keep every device with the latest versions of software; ii) reduce maintenance costs: physical access to individual devices can be very expensive and time consuming (mainly if it is about millions or billions of units), or even impossible sometimes, what reinforces the need of remote updates; iii) provide more convenience to final customers: no need to visit service centers or allow physical access to their devices or properties.

Remote updates are easily applied to devices that perform non-critical tasks; the operation can be interrupted to update the system and the timing behavior is not crucial. On the other hand, some embedded systems perform real-time tasks in (life) critical situations, as pacemakers, automotive/aerospace systems, etc. Thus, their operation can not be interrupted without drastic consequences.

Hence, it is necessary to update devices without disrupting their normal operation, i. e., perform dynamic (on-the-fly) updates. However, dynamic systems have an inherent overhead compared to static systems, due to the metadata necessary to perform the dynamic procedures (symbol table, relocation entries, etc.). Therefore, we aim at providing several solution variants, by sharing this overhead with different proportions between the update server and the target devices.

Due to the increasing complexity and integration density of embedded software, it will also become essential to dynamically compose the systems from independent modules and keep them partially updatable at runtime. This will be specially important because, in the future, there will be many software providers offering similar services to different classes of devices, similarly with what happens nowadays with general purpose computers. Thus, manufactures can choose the solutions that best fit their requirements (energy efficiency, response time, low memory overhead, etc.).

Nevertheless, since those modules are independently developed, without any knowledge about the target systems, it is necessary to carefully analyze their compatibility with the system, i. e., check if all symbol references can be resolved and make sure the real-time capability is not violated; compatible updates are allowed to be deployed, non-compatible ones are rejected. Our ultimate goal is to cope with dynamically composed real-time systems, i. e., systems that are partially updated on-the-fly and that stay real-time capable during and after any update. The concept presented in this paper is part of a holistic solution for dynamically composed RTOS and MCU, presented by Martins Gomes et al. [16].

The remainder of this paper is organized as follows: Section 2 presents the current approaches for remote updates in embedded systems and Section 3 introduces our concept for building a dynamically composed real-time system.

## 2 State of the Art

The update process of a device can be divided in four steps: choice of update unit, size reduction, transmission and installation. The chosen approach in each step will affect the size of the data to be transmitted and the processing and memory overheads in the device, which will impact the amount of energy spent – a very important metric for battery powered devices. It is also worth to mention that approaches that demand a reboot are not suitable most safety-critical real-time systems, since the delay might cause real-time tasks to miss deadlines.

### 2.1 Update Units

**Monolithic image:** A monolithic image is built in the update server and sent to the devices, which just need to load the image to memory for execution. Therefore there is low in-node processing overhead. However, the amount of data transmitted and the memory overhead are the highest from all approaches. Another drawback is that the entire SW stack/source must be available for compiling/linking at the update server. Furthermore, the replacement of the

full image demands a reboot, so this approach is not suitable for dynamically composed systems. Deluge [9] provides full reprogramming support for TinyOS [14], including the dissemination protocol, loading and reboot mechanisms. Upon an update, it transmits both the full image and the reprogramming protocol, so there is an additional transmission overhead.

Stream [21], on the other hand, does not need to transmit the reprogramming protocol; it pre-installs the protocol on the external flash memory of devices.

**Loadable modules:** Component-based systems provide the possibility of a more flexible update mechanism. Since components/modules are independently developed, it is possible to update single modules instead of the whole program. When only compiling a module an object file, a relocatable Executable and Linking Format (ELF) file, is produced. Relocatable files contain references to symbols whose addresses are unknown at compile time. The addresses are defined through linking and relocation. Linking resolves the addresses of external symbols and relocation resolves the addresses of internal symbols. Relocatable files also contain metadata to describe where the references to each unresolved symbol occur, so that the file can be properly modified and become an executable binary. Nevertheless, the metadata overhead in relocatable files is quite large, from 45 % to 55 % of the object file [19]. One strategy for reducing this overhead is using Position Independent Code (PIC), which uses relative references for all internal symbols. However, PIC demands compiler and CPU architecture support, adds processing overhead, and modules might be subject to size restrictions.

All implementations of this approach, which do not demand a reboot after an update, are suitable to dynamically composed systems. The processing and memory overhead of the device, as well as the size of the transmitted data depends on how the metadata storage and processing are divided between device and update server. The more metadata the server stores, the more it can tailor updates for specific devices, so less data will be transmitted and the overhead in the devices will be lower. The less metadata the server stores, the more generic is the update. Thus, the devices themselves must tailor their own updates. It is worth to mention that every global symbol whose address is not known on the server side must be stored in the device, leading to higher memory overhead.

Some solutions implemented with PIC loadable modules are SOS [7], which is able to dynamically load modules without a reboot, and [10], which focuses on safety-critical systems. Without PIC, Contiki [5] is also able to update applications without a reboot, but not the system core. A module is pre-linked with the core symbols in the update server, but the remaining linking is performed in-node. On SenSpireOS [3] the server receives ROM and RAM loading addresses from the device and pre-relocates the module before transmitting. FlexCup [15] adds modular update capability to TinyOS, but it demands a reboot after updates. Additionally, in order to cope with module dependencies and versioning, FiGaRo [18] proposes a solution on top of Contiki.

**Virtual machine:** Virtual machine bytecode is much more compact than native code. Therefore, it is often used to reduce the size and cost of transmission. The memory overhead of each application is lower than native code, but the virtual machine itself must also be considered. In fact, the processing overhead due to code interpretation at runtime mostly outweighs the costs saved in the transmission [6].

Maté [13] is a virtual machine built over TinyOS. It is able to update applications, but no lower level binary code (drivers, kernel, etc.), since virtual machines for resource-constrained devices usually are not able to perform operations on registers. Since it does not reboot after an update, it is also suitable for the application layer of dynamically composed systems.

## 2.2 Size Reduction

**Optimizations on standard ELF:** The standard ELF format is designed to work on 32-bit and 64-bit architectures. Therefore all the internal data structures are defined with 32-bit data types [1]. When the target device is 8-bit or 16-bit, the high 16 bits of the fields are unused. In order to avoid this waste of memory, some compact versions of ELF were proposed.

Contiki [5] uses the Compact ELF (CELF) format for dynamic linking and loading. A CELF file contains the same information as an ELF file, but with 8 and 16-bit data types. SenSpireOS [3] proposes Slim ELF (SELF), which changes the data types, just like CELF, and tailors the ELF relocation, string and symbol tables. The result is a format 15%–30% of standard ELF, 38%–83% of CELF.

**Delta files:** In the so-called incremental approach, only the differences between consecutive versions are transmitted with delta files. These files are actually diff-scripts, which are basically composed of two types of commands: COPY and ADD. The delta file and the old version are used to build new version in the device. COPY commands simply copy a chunk from the old version into the new version, and ADD commands insert content nonexistent in the old version. However, a small delta file does not mean low processing overhead in the device. For example, if the only change is the removal of the first byte of a module, a naive solution may shift the whole module in memory. Furthermore, whenever a symbol is placed in a different address, every piece of code that has references to it must be patched. Therefore, some work has been devoted to increase the similarity between two consecutive versions of code and avoid function shifts.

The RMTD [8] is a byte-level differencing algorithm, suitable for small pieces of code (the  $O(n^2)$  space requirement makes it unsuitable to scale to large programs – 1.3 GB when comparing 100KB files, and around 5GB when comparing 200KB files [2]). R3 [2] proposes a method to achieve large similarity with low metadata overhead. To mitigate the function shift issue, Zephyr [20] uses a function indirection table (all function calls are indirected via fixed jump table slots to their implementations) and [11] uses slop regions after each function so that the address does not change when a function grows within the slop region.

**Compression:** This is yet another alternative for size reduction and can be used in addition to the aforementioned approaches. Design issues involved in adapting compression algorithms for energy-constrained devices are investigated in [22]. An implementation of the gzip algorithm on sensor nodes is done in [24].

## 2.3 Data Transmission

Often code dissemination over the network is necessary, as in Wireless Sensor Networks (WSNs). Efficient code dissemination protocols can save energy on transmission, as well as speed up the transmission process.

Deluge [9] uses a three-way handshake and NACK-based protocol for reliability, as well as segmentation into pages and pipelining for spatial multiplexing. MNP [12] addresses the problem of concurrent senders with a sender selection algorithm, that attempts to guarantee that at most one source at a time transmits in a neighborhood. ECD [4] considers link quality of 1-hop neighbors to improve the sender selection algorithm.

## 2.4 Installation

Installation is everything that happens after the update is fully received by a device and before the new version is ready to run. It can be divided in four major steps: i) make sure that the data is not corrupted and that it is a legit update; ii) configure related control blocks (from tasks, resources, etc.); iii) if the file is not yet an executable, build it based on the update (build new version from delta file and old version, and/or link and relocate the code); iv) load the executable file and add it to the ready queue.

## 2.5 Real-Time Awareness

The techniques listed so far are sufficient to dynamically compose non-critical systems in general, but fail on real-time systems, since none of them takes timing into consideration. In order to ensure that a system will remain real-time capable during and after any update, the update process must not interfere with any real-time tasks and must also ensure that no deadlines will be missed after the update. In other words, it must be non-intrusive and perform schedulability analysis.

Both [17] and [23] propose solutions for rate-monotonic scheduled systems. [17] is extremely simplified: the Worst Case Execution Time (WCET) of a new module is assumed to be lower or equal the old one and new modules are stored in the heap. On the other hand, [23] is more realistic: upon every update, there is a schedulability test to determine whether or not a new module can be accepted and the update process is assured to be finished within two hyper-periods. However, to the best of our knowledge, there are no approaches considering resource sharing or tasks synchronization mechanisms in the schedulability analysis.

### 3 Our Concept

As listed in Section 2, there are a number of works handling the different steps of remote updates. We will focus on the topic where there is lack of research: real-time capability assurance during and after updates, which can be an extra step in the installation. We will define a solution with a set of steps necessary to do it, and implement some variants of this solution, by assigning some steps to the device and others the server. Finally, we will define the minimum requirements of devices that are able to run each of the variants. For the update unit, we will use loadable modules without PIC.

#### 3.1 Requirements

**Direct server-device communication:** Some of our solutions variants will send generic data to any device of a network; this is fine also on multi-hop networks, since the data would have to be disseminated just once. However, other variants will send data linked and relocated to fit in specific devices; a multi-hop network would thus have to disseminate individual data to each of the modules to be updated. Since that would lead to a very high traffic, what might not be acceptable, we propose a direct server-device communication.

**Unintrusive update:** As mentioned in Section 2.5, it is essential not to interfere with real-time tasks while updating the system, so the update task must run with lower priority. However, if the priority is too low and the CPU utilization is high, the update process might starve. In mixed criticality systems, we can assign an intermediate priority to the update task, higher than either all soft real-time tasks or all the best-effort tasks. The trade-off between update completion time and user experience will be taken into consideration to define the priority.

**Low execution overhead:** Ideally, the execution time of the dynamically composed system should be exactly the same as for the static system. However, indirections are inevitable, for example, to isolate the kernel from the applications (kernel jump tables) or to access global variables or functions using PIC. We will minimize this overhead by using as few indirections as possible, so our first design decision is not using PIC. For other indirections, we will analyze the trade-offs as we progress with the implementation.

**Loose coupling:** Kernel and applications will be completely isolated, so that an update in the kernel does not require the applications to be rebuilt. A classical strategy to obtain it is through a kernel jump table in a fixed position in memory, where pointers to all system calls are stored. Thus, upon changing the kernel, only the jump table must be updated. We will also investigate the feasibility of relocating the application code instead of using the jump table; it would be a more expensive update process, but we would save the indirections during normal operation.

**Portability:** We won't rely on special compilers or architectures feature; this is another reason why we will not use PIC. Also, we will stick to standards: we do not intend to modify the standard ELF format, compiler or linker.

**Schedulability analysis:** It must be possible to define whether or not a system will remain schedulable after any update. If an update is to break schedulability, it should be rejected. This is the step that demands the highest processing power; in a future work, we will analyze what are the minimum requirements of a device, so that it is able to run our schedulability analysis algorithms without compromising the normal operation of the system.

**Easy programming:** We aim to avoid specific restrictions and rules inherent to our approach, in order to reduce the learning curve of developers. Ideally, writing a program with our approach would be exactly the same as in the static approach. Some changes are inevitable though, for example, modules versioning and dependencies declaration.

### 3.2 Update Protocol

In order to avoid unnecessary traffic, there is firstly metadata exchange between update server and device to check the compatibility of the update; only if it is compatible, the binary is finally transmitted.

Compatibility is ensured by two properties: *pluggability* and *interoperability*. Pluggability is about checking if modules fit in the system; a new module is pluggable if there is enough RAM and ROM to load it and every symbol it references can be resolved; similarly, a module can not be removed if it is currently used by other ones, since they would become unpluggable. Interoperability is more related to the execution behavior; here is where the schedulability is checked, as well as possible starvations and deadlocks.

During the full procedure, memory, processing and transmission overhead will depend on how the steps are divided between servers and devices. Table 1 shows some possible implementations. Tasks that must be accomplished in every possible implementation are not considered as overhead, for example, loading the module on the device or storing the modules on the server. Some slight changes in the update protocol can decrease the overhead on devices, as shown in cases 2 and 3: instead of just signaling if there is enough memory, sending the load positions to the server enables it to perform relocation, so the devices are spared from this step. Due to the high memory and processing overhead, it is very unlikely that the last case will be used in real-time embedded systems.

Building customized modules at the server side for individual devices might not be the ideal solution if billions of devices are deployed – a realistic magnitude with the advent of the IoT. But storing too much metadata in resource constrained devices is not feasible either. Therefore, a trade-off between device overhead and server overhead must be met, and it depends on how powerful the devices and server are and how energy efficient the system must be.

**Table 1.** Examples of implementation variants

Case	Server knowl.	Server overhead	Device overhead	Data transmission
1	Full	Full metadata, compatibility checks, linking and relocation	None	Executable module
2	Global symbols	List of symbols per device, compatibility checks and linking	Memory layout information and relocation	Module size, enough memory flag and executable module
3	Global symbols	List of symbols per device, compatibility checks, linking and relocation	Memory layout information	Module size, memory load positions and executable module
4	None	None	Full metadata, compatibility checks, linking and relocation	Module metadata and executable module

### 3.3 Implementation

**Metadata extraction:** The metadata for linking and relocation is automatically generated by the compiler. Nevertheless, information about execution time and synchronization mechanisms still need to be extracted. We will write custom tools to extract such information from the source code and standard ELF file. From an individual loadable module (ELF relocatable file), it is possible to extract i) its priority and deadlines: explicitly declared in the code; ii) a parametrized number of cycles needed for execution: this number is obtained by summing the amount of cycles each instruction needs to run – the exact number of cycles can not be defined because the amount of iterations in a loop can not always be directly predicted, so the programmer will still need to provide some parameters in order to specify the WCET; iii) waiting points: every system call that can put the task in waiting state. From the source code, it is possible to check for inconsistencies in the synchronization mechanisms, like getting a resource and never releasing it. We are currently also working on related theoretical foundations and implementation for this purpose.

**Execution model:** The information extracted from individual modules will be used as base for an execution model. When the information about all modules that compose a system are put together, and the scheduling algorithm and resource management protocol are defined, it is possible to extract the remaining information, like waiting time (time waiting for a resource or event), interference (time in ready state, but not running due to higher priority tasks), and Worst Case Response Time (WCRT).



**Execution simulation:** The execution model will be used to simulate execution scenarios and define the WCRT for every task. If in at least one of the scenarios, the WCRT of any task is higher than the respective deadline, the system is not schedulable.

## 4 Conclusion

We have presented an overview on the state-of-the-art approaches to perform remote updates. The only approach not suitable for dynamically composed systems is replacing the full image of the devices with a new monolithic image, because it always demands a reboot, what might cause an unacceptable delay. There is still lack of research on dynamically composed real-time systems (systems that remain real-time capable during and after any update) and that is exactly our focus. Our main goal is the schedulability analysis during an update, considering resource sharing and synchronization mechanisms. Several solutions variants will be implemented, with different trade-offs among overhead in the devices, overhead in the server and size of the data transmission.

## Acknowledgement

This research project was partially funded by AVL List GmbH and the Austrian Federal Ministry of Sciences, Research and Economy (bmwfw).

## References

1. TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. TIS Committee, 1995.
2. W. Dong, C. Chen, J. Bu, and W. Liu. Optimizing relocatable code for efficient software update in networked embedded systems. In: *ACM Transactions on Sensor Networks (TOSN)*, 11(2):22, 2015.
3. W. Dong, C. Chen, X. Liu, J. Bu, and Y. Liu. Dynamic linking and loading in networked embedded systems. In: *Mobile Adhoc and Sensor Systems*. MASS'09. IEEE 6th International Conference, pp. 554–562, 2009.
4. W. Dong, Y. Liu, C. Wang, X. Liu, C. Chen, and J. Bu. Link quality aware code dissemination in wireless sensor networks. In: *Network Protocols (ICNP)*, 19th IEEE International Conference, pages 89–98, 2011.
5. A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In: *Local Computer Networks*, 29th Annual IEEE International Conference, pages 455–462, 2004.
6. 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*, pages 15–28. ACM, 2006.
7. 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*, pages 163–176. ACM, 2005.
8. J. Hu, C. J. Xue, Y. He, and E. H.-M. Sha. Reprogramming with minimal transferred data on wireless sensor network. In: *Mobile Adhoc and Sensor Systems, MASS'09*. IEEE 6th International Conference, pages 160–167. 2009.

9. 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*, pages 81–94. ACM, 2004.
10. 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)*, 20th IEEE International Conference and Workshop, pages 92–99, 2013.
11. J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In: *Wireless Sensor Networks*, Proceedings of the Second European Workshop, pages 354–365. IEEE, 2005.
12. S. S. Kulkarni and L. Wang. MNP: Multihop network reprogramming service for sensor networks. In: *Distributed Computing Systems (ICDCS 2005)*, Proceedings of the 25th IEEE International Conferencen, pages 7–16, 2005.
13. P. Levis and D. Culler. *Maté: A tiny virtual machine for sensor networks*. ACM Sigplan Notices, 37(10):85–95, 2002.
14. P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. *Tinyos: An operating system for sensor networks*. Ambient intelligence, 35:115–148, 2005.
15. P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In: *European Workshop on Wireless Sensor Networks*, pages 212–227. Springer, 2006.
16. R. Martins Gomes, M. Baunach, L. Batista Ribeiro, M. Malenko, and F. Mauroner. A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems. In: *13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPRT'17)*, June 2017. pages 41-46.
17. J. Montgomery. *A model for updating real-time applications*. Real-Time Systems, 27(2):169–189, 2004.
18. L. Mottola, G. P. Picco, and A. A. Sheikh. Figaro: Fine-grained software reconfiguration for wireless sensor networks. In: *Wireless Sensor Networks*, pages 286–304. Springer, 2008.
19. R. K. Panta and S. Bagchi. Hermes: Fast and energy efficient incremental code updates for wireless sensor networks. In: *INFOCOM 2009*, pages 639–647. IEEE, 2009.
20. R. K. Panta, S. Bagchi, and S. P. Midkiff. Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In: *Proc. of USENIX Annual Technical Conference*, 2009.
21. R. K. Panta, I. Khalil, and S. Bagchi. Stream: Low overhead wireless reprogramming for sensor networks. In: *INFOCOM 2007*. 26th IEEE International Conference on Computer Communications, pages 928–936, 2007.
22. C. M. Sadler and M. Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In: *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 265–278. ACM, 2006.
23. H. Seifzadeh, A. A. P. Kazem, M. Kargahi, and A. Movaghar. A method for dynamic software updating in real-time systems. In: *Computer and Information Science*. ICIS 2009. Eighth IEEE/ACIS International Conference, pages 34–38, 2009.
24. N. Tsiftes, A. Dunkels, and T. Voigt. Efficient sensor network reprogramming through compression of executable modules. In: *Sensor, Mesh and Ad Hoc Communications and Networks*, SECON'08. 5th Annual IEEE Communications Society Conference, pages 359–367, 2008.

Logistik und Echtzeit

Echtzeit 2017

Halang, W.A.; Unger, H. (Hrsg.)

2017, VIII, 142 S. 60 Abb., Softcover

ISBN: 978-3-662-55784-6