GITAR: Generic extension for Internet-of-Things ARchitectures enabling dynamic updates of network and application modules

Peter Ruckebusch[a,*], Eli De Poorter[a], Carolina Fortuna[a], Ingrid Moerman[a]

[a] *Ghent University - iMinds*
Dept. of Information Technology (INTEC)
Gaston Crommenlaan 8 Bus 201
9050 Ghent, Belgium

Abstract

The Internet-of-Things (IoT) represents the third wave of computing innovation and relies on small, cheap and/or energy efficient devices, densely deployed in various spaces. Automatically managing, updating and upgrading the software on these devices, particularly the network stacks, with new, improved functionality is currently a major challenge. In this paper we propose GITAR, a generic extension for Internet-of-Things architectures, that enables dynamic application and network level upgrades in an efficient way. GITAR consists of four design concepts which can be applied to any operating system running on IoT/M2M devices. The proof of concept implementation in this paper uses the Contiki OS and the evaluation, based on analytical and experimental methods, shows that GITAR i) is up to 14% more efficient in terms of memory usage and ii) has less or similar run-time CPU overhead as state of the art solutions while offering upgrade functionality down to the network level and iii) can reuse existing Contiki network protocols for dynamic updates without requiring modifications to the code.

Keywords: GITAR, internet of things, IoT, network update, run time reprogramming, over the air reprogramming, protocol stack, machine to machine, wireless, reconfiguration, Contiki

*Corresponding author
Email address: Peter.Ruckebusch@intec.ugent.be (Peter Ruckebusch)

1. Introduction

Currently, in the developed world, individuals own a limited number of wireless devices including a mobile phone, a PC, a tablet, camera and a TV. The applications running on most of these devices are able to perform automatic updates to relieve the user from a relatively difficult and time consuming upgrade process. IoT brings a third wave of computing innovation, following the personal computer and mobile phone wave. The Internet of Things (IoT) is typically envisioned as the integration of *small, cheap and/or energy efficient* wireless radios in everyday objects. Most IoT devices are constrained in terms of processing power and memory storage to keep the unit price low, as well as in terms of energy since they are typically battery powered. This trend will lead to a further increase in i) the number of devices per person and ii) the number of devices per square meter, thereby introducing the need for efficient upgrade ability and reconfigurability of embedded devices, as recognized by Atzori [1], De Poorter [2], Fortuna [3] and the ETSI RRS WG 2 [4].

In addition, whereas wireless technologies and standards are changing at a rapid pace, embedded IoT devices will have a longer lifetime, typically a decade or more. To cope with such a huge amount of devices, deployed in rapidly changing wireless environments, we envision the need for update mechanisms that not only allow new IoT applications, but also allow to download new or updated network functionality, including MAC implementations, over the air from a repository (similar to an app store). This would, for example, allow to switch between a ZigBee [5] and WirelessHART [6] protocol stack, both of which use the same physical layer.

The most popular wireless sensor network/IoT operating systems, TinyOS [7] and Contiki [8], were not designed to support such dynamic upgradeable environments. In these operating systems, while the applications can be altered using partial code upgrades, altering the network stack can only be achieved using full firmware upgrades. This approach has several major drawbacks. i) Distributing a new firmware over-the-air consumes a significant amount of energy. ii) Extending or modifying large network stacks is time consuming and error prone. (iii) The lack of code re-usability slows the overall progress in the area. Other systems, such as SOS [9], allow more modular upgradeability but (i) have not been demonstrated to support network stack upgrades, ii) require modifications to existing network protocols and iii) do not use standard upgrade mechanisms and file formats.

In this paper, we claim the following three contributions. The first and most important contribution of the paper is GITAR, a Generic extension for Internet of Things ARchitectures that enables run time updates of network stacks and applications on constrained devices using standard file structures, tools and methods. GITAR supports position independent dynamic linking of components at run-time from the application level down to the MAC layer. The second contribution is a proof of concept implementation that is fully backwards compatible with the existing Contiki network protocols, thus extending an already popular IoT solution. The third contribution is the theoretical and experimental evaluation of the reference implementation which demonstrates that GITAR (i) is up to 14% more efficient in terms of memory usage; (ii) has less or the same run-time CPU overhead as the state of the art while offering beyond state-of-the-art network level upgrade functionality; and (iii) has a lesser or equal energy cost for deploying the dynamic components.

The remainder of this paper is organised as follows. First, Section 2 describes and analyses scenarios in which run-time software upgradeability is required. Section 3 gives a brief overview of related work. Next, Section 4 described the architecture of the proposed system that enables all the scenarios identified in Section 2. The reference implementation of the architecture is discussed in Section 5 while Section 6 evaluates the architecture both theoretically and experimentally. Finally, Section 7 concludes the paper.

2. Scenarios for (re)programmable wireless networks

Software updates are required in many scenarios and can happen in different stages over the "lifetime" of a device (e.g. development, testing, deployment, maintenance, etc.). We distinguish three levels at which (re)programmability is needed: system level, network level and application level.

2.1. Application level programmability

Application level (re)programmability refers to adding or changing a part of the program code that implements application functionality such as sensor reading and measurement compression. Such updates are currently already supported by major operating systems for constraint devices as further discussed in section Section 3.

2.2. Network level programmability

Network level (re)programmability refers to adding or changing a part of the program code that implements networking functionality, ranging from media access control (MAC) layer to transport layer. Such re-programmability is seldom used today since most of the transceivers for constrained devices feature a black box protocol stack implementation that conforms to one of the multitude of available standards. This black box cannot be changed and its performance is often "best effort" - meaning that if the operating environment is different from the one it was designed to operate in, it cannot be adapted. In dynamic environments, IoT devices have to be adaptive, thus able to replace their networking functions with new ones available in app-store like repositories. In order to achieve this, network level re-programmability should be supported independently of system level and application level re-programmability.

Network level programmability support will also decrease the development cost of IoT/M2M related networking solutions where the commercial developers will be able to (i) take the open source code implementing network functionality, (ii) adapt it for commercial purposes subject to license compliance and (iii) quickly deploy, test and evaluate on a real testbed. It will also significantly reduce the time and effort invested by the research community to develop new network functionality since there will already be a modular and re-usable code base to start from. This process is similar to the way Unix OS development is happening for over 20 years, and web development is happening in the last 5 years.

2.3. System level programmability

System level (re)programmability refers to adding or changing a part of the program code that implements basic functionality provided by the constraint device. Examples are code fragments that implement timers, schedulers, memory allocation, etc. This code is most commonly part of the core operating system running on the device. It may also include network and application functionality, especially when legacy operating systems for sensor networks are concerned. In the case of future networks of high number/high density devices, it is expected that the system level code is relatively well tested and stable, thus needing seldom upgrades.

2.4. Summary

The need for (re-)programmability in each level can be estimated by considering the frequency of updates in different scenarios. In Table 1, the frequency of the system, network and application level (re)programmability are compared using three gradations for the update frequency (Often, Regular and Seldom). Both pre- and post-deployment scenarios are considered.

Table 1: Quantitative comparison between system, network and application level for different re-programming scenarios.

	Pre-deployment		Post-deployment		
Level	Development	Experimentation	Additions	Updates	Extensions
System	Often	Seldom	Seldom	Seldom	Seldom
Network	Often	Often	Often	Often	Often
Application	Often	Regular	Often	Regular	Seldom

Pre-deployment scenarios include each code update that is necessary during development (small scale) and experimentation (large scale). Since development is an iterative process, the code is continuously updated in all levels. In large scale experiments, when the code is optimized for deployment, generally more fine-tuning is required in the network level. This is because applications are less complex and system level code is re-used in many cases. Partial code updates are not essential to support programmability in pre-deployment scenarios. Nevertheless, they can speed-up the experimentation process significantly because a firmware update and device reboot can be avoided.

Post-deployment, firmware updates cause a significant amount of overhead (in terms of energy consumption, downtime, ...) compared to partial code updates. The *core system functionality* should at this stage be based on well-tested and stable source code, thus only sporadic changes are required. Much like mobile applications today, IoT deployments will frequently require additional applications which will be updated regularly. In many future scenarios, especially for constraint devices located in robots, we expect the number of *application upgrades* to be smaller than the upgrades of the network functionality. While the applications are upgraded when additional functionality is added or when bugs are fixed, the network functionality will be changed when the surrounding environment changes and the device will need to communicate with other devices using different network functionality.

Currently, the need for post-deployment updates to the *networking program code* is still small, it will rise when i) the number of devices increases, ii) legacy devices are combined with newly installed IoT devices and iii) the developer community becomes larger and more dynamic as is happening today with the development of mobile app. The updates post-deployment will be triggered by new and improved protocols offering better wireless performance, by new requirements, or by the need to integrate new devices.

3. Related work

In general, there are four approaches for performing runtime code updates in already deployed WSNs: i) existing *script languages* are ported to WSN platforms; ii) *virtual machine approaches* allow the injection of intermediate code that is interpreted at run-time; iii) *full-image based approaches* replace the entire firmware at once; iv) *component or module based approaches* divide the firmware of a device in small code blocks that can be added or updated at runtime. Many of these are inspired by solutions already existing in Unix and adapted for sensor/actuator devices with limited memory and battery capabilities. An extensive survey of existing solutions is given in [10], and below we briefly summarize the first three approaches and look into more detail in the fourth that is more relevant to our work.

3.1. Script interpreters

Script interpreters allow the execution of a script by interpreting the statements inside the scripts at run-time. Because of the runtime interpretation, scripts can be added or updated after deployment. Some well-known scripting languages like Python [11, 12] were already ported to embedded devices. Also lightweight versions of the Unix bash environment were created for WSN platforms in the Contiki shell and in LiteOS [13]. Due to the string representation of statements, script interpreters require a substantial amount of memory and CPU overhead. Hence, scripts are not suited for the low-capability hardware platforms targeted in this paper.

3.2. Virtual machines

Virtual machines use an intermediate code format that contains generic instructions (i.e. not CPU/machine dependent). The intermediate code is translated into machine code at run-time by a code-interpreter and then executed. Since intermediate code is written at a higher abstraction level than

machine code, the program code for a virtual machine is smaller than the program code for physical machines. Consequently, the size of the code fragments that need to be distributed will also be smaller and hence require less transmission power. Therefore, many solutions [14, 15, 16, 17, 18] were proposed for wireless sensor networks that provide partial code updates using virtual machines. However, due to the run-time code interpretation, they exhibit a significantly higher run-time overhead compared to executing native machine code. In addition, due to memory constraints, these virtual machines are highly optimized for specific (niche) applications rather than being generic to support many application domains. Finally, since the network layer is included in the virtual machines, the scope for updates is always limited to the application level.

3.3. Image-based approaches

To avoid the high run-time overhead of both virtual machines and script interpreters, native machine code execution is preferred in scientific literature. Native machine code can directly be executed by the micro-controller of the constrained device and does not require interpretation at run-time. By far the easiest and most used approach for post-deployment code updates replaces the entire image. All source code is compiled into a single image and installed on each device. If an update is required, a new image must be compiled and distributed to all nodes. There are many examples using this approach of which XNP/MNP [19] and Deluge [20] are perhaps the best known. Because the entire image is reprogrammable, all levels (system, network and application) can be updated. Once installed, no run-time overhead is introduced. Amongst all approaches, the deployment overhead is the highest since the entire image must be distributed.

To make the update process more efficient, binary differential patching techniques [21, 22, 23, 24] were proposed that highly reduce the size of the image that needs to be transferred. Patching techniques are only efficient in homogeneous networks where each node uses the same image. If this is not the case, a different patch is generated for each node. The patching techniques can also be applied on component based solutions as demonstrated in [25].

3.4. Dynamic loadable native code components

Finally, *component or module based approaches* divide the firmware of a device in small code blocks that can be installed after deployment and

executed directly at runtime. Run-time upgradeable components are typically compiled and distributed as Executable and Linkable Format (ELF) objects [26]. An ELF object contains the compiled code and data sections of a component. During installation, the necessary ROM and RAM memory must be allocated for these sections. The relative addresses used in the code and data sections can then be relocated to the correct physical address offset using the relocation entries also included in ELF file. If the code and data sections contain undefined symbols (e.g. functions or data defined in other code blocks), there is also a linking step required in which each undefined symbol is linked to the correct physical address. To enable relocation and linking, component based solutions always require operating system support. Hence, in such solutions there is always a static part in the firmware, commonly referred to as the kernel, and a dynamic, component upgradeable part. Since the individual components are smaller, the deployment overhead is lower compared to image based solutions. The disruption is also lower because this approach does not require a system reboot and therefore state information can be transferred between updates. Component based solutions can be categorized by the binding model they use. The binding model defines how code blocks are linked post-deployment to the external functionality (functions, shared memory, ..) provided by code blocks already present. Currently, two models are applied: i) strict binding and ii) loosely coupled binding.

3.4.1. Strict binding

Probably the best known example of a *strict binding model* is the dynamic linker approach in Contiki [27]. This solution allows to add or update applications. The dynamic linker uses a symbol table that contains all global symbols in the Contiki system. The symbol table is generated before deployment and cannot be extended afterwards. Because of this, additions and updates are restricted to the application level. A solution that partially overcomes this problem is proposed in Dynamic TinyOS [28]. The static symbol table is replaced with a dynamic symbol table that can be extended. Now code blocks with unidirectional, outward dependencies can be added on top of each-other. Due to the strict binding model however, the possible update scenarios are still limited to the top-level components. In practice, Dynamic TinyOS also restricts additions and updates to the application level.

3.4.2. Loosely coupled approach

To overcome this limitation, a *loosely coupled approach* is required. In general there are two approaches for realizing loose couplings: i) event based solutions and ii) indirect function calls.

Event based solutions. Event-based interaction between dynamically loaded components allows the use of event producers and consumers that are linked using an event-bus. FiGaRo [29], LooCI [30] and Remora [31] are examples of event-based systems. They are implemented on top of the standard Contiki system and network stack. Applications can interact with other applications through an event bus but still require strict bindings with the system level. Since event-based solutions use an event bus, they introduce a non-negligible amount of run-time delay (in the order of milliseconds) that is not deterministic. They are hence not suited for providing network level re-programmability.

Indirect function calls. Another possibility is to implement an indirect function call mechanism that uses jump tables to redirect a function call to the correct function address. The solutions that use the indirect function call mechanism are more efficient because they add a limited and deterministic number of CPU cycles (in the order of microseconds). To redirect function calls, function pointers are used. When updating a component, the function pointers to that component used by other components also need to be updated. Hence only the bindings in other components to that component require an update.

In SOS [9, 32] the first example of such an approach for WSNs was demonstrated. Each dynamic SOS module encapsulates its external functions in function control blocks (FCBs). Other modules require a pointer to the FCBs and rely on the SOS kernel to execute the function pointer call, introducing a substantial amount of CPU overhead. In Enix [33] and RemoWare [34], the function pointers are maintained in a jump table with a 1-to-1 relation between function pointer and function name (or ID). Components also delegate the function pointer call to the kernel but with less overhead.

To reduce the memory usage for the FCBs in SOS and the jump table in Enix and RemoWare, they combine a strict binding model for using system level functionality and a loosely coupled binding model in the dynamic component level. The combination of both models introduces a fixed mem-

ory overhead for providing the system functions regardless of the number of dynamic components.

Therefore Lorien [35] allows to scale the memory overhead with the number of components, by embedding a jump table in each component. The jump table is instantiated by the kernel when a component is added and renewed when components are updated. The size of a component in Lorien is however drastically increased. Since in Lorien each component can be updated separately, the overall memory overhead will be much higher.

3.5. Differentiators of GITAR

GITAR is different than the related work in that it overcomes the aforementioned problems by combining several design concepts proposed in the related work into a generic architecture that can be applied on existing operating systems. More precisely, GITAR:

- Incorporates a loosely coupled binding model inside both the network and application level, *enabling partial code updates of protocols and applications.*
- Proposes an indirect function call mechanism that *does not require source code modifications in existing protocols and applications.*
- Maintains the function pointers required by the indirect function calls in each separate component allowing to *scale the overhead with the number of dynamic components.*
- Decreases the dependencies with the system level *defining and enforcing a system level boundary.*
- Further *reduces the overhead by linking components to components* rather than linking components to individual functions.

The combination of the first 4 contributions is novel for constrained IoT devices. Moreover, the last contributions is a unique feature when considering the discussed IoT/M2M landscape that uses devices with limited capacity.

4. Architecture

In this section, a number of design concepts are introduced that are essential for supporting network level (re)programmability. Using these design concepts, an architecture is proposed that can be applied to existing operating systems, making them component upgradeable.

4.1. Design concepts

Separation of concerns: define a clear boundary for the system level. All partial code update approach utilize a separation between a static code part and a dynamic code part. The static part, commonly referred to as the system level, can only be updated by reinstalling the entire firmware. In the dynamic part, separate code blocks can be added or updated. The level of upgrade-ability is defined by the functionality that is included in the system level. In order to achieve the desired level of flexibility (e.g. upgrading network protocols), the system level boundary must be placed just below the network protocols (including MAC).

Reduce coupling: use a loosely coupled binding model. In the dynamic part, there is still a choice between a strict and loosely coupled binding model. Using a strict binding model, components are linked using physical memory addresses. This implies that when updating a piece of code, all access to that code in other components also needs to be updated. When using a loosely coupled binding model, components can be dynamically (re-)linked, enabling updates without affecting other components. A loosely coupled component can only interact indirectly with other components which necessitates support for redirecting interactions.

Reduce dependencies: restrict interactions by enforcing hardware abstraction. In order to increase the portability of code, most operating systems for constrained devices provide guidelines for implementing a hardware abstraction strategy. By enforcing the already available hardware abstraction strategy, dependencies between dynamic components and the system level can be reduced. This will make the update management of components above the system level much easier. Ideally, the system level only offers portable (hardware independent) interfaces.

Increase modularity: group common functionality in components. In order to support efficient update strategies inside the dynamic part, there must be a rationale for dividing source code into update-able code blocks. Generally this is achieved by increasing the modularity of source code. For this, common functionality must be grouped in components that have a well-defined interface and implement a coherent set of functions. Since it must be possible to update single network protocols, the modularity of a component must also be implemented on a protocol level.

The implementation of each of these design concepts will be discussed in Section 5.

4.2. High level architecture overview

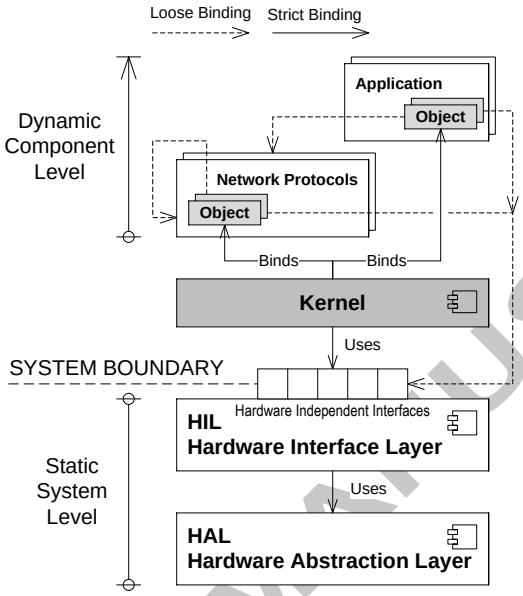


Figure 1: High level overview of the proposed architecture. Necessary components for upgradeability are depicted using grey boxes. Dynamic components use a loosely coupled binding model to interact. This is indicated by the dashed arrows. The strict bindings with the kernel and inside the system are depicted using full arrows.

The architecture that follows the above design concepts is illustrated in Figure 1 and consists of i) a static system level, ii) a dynamic component level part and iii) a kernel linking both levels.

System level. The system level only implements device drivers and core OS functionality. A firmware upgrade is required for updating the system level as discussed in Section 2.3. In order to reduce dependencies, the system level is further divided in a hardware abstraction (HAL) and a hardware interface (HIL) layer.

Kernel level. A clear system level boundary is introduced between the static system and the dynamic component part by adding a kernel. The role of the kernel is to bind dynamic components to each other and to system level functionality at run-time. The kernel is able to get the run-time references

to existing components because each new component is initially registered in a database that is part of the kernel. The database also keeps track of the dependencies between components in order to orchestrate the dependencies between components in future updates.

Component level. Above the system level, network protocols (or applications) are grouped in dynamically update-able components (see Sections 2.1 and 2.2) that can only interact indirectly with each other using *component objects* (grey boxes in Figure 1). Component objects contain meta-data and control functions used to enable indirect interactions between different components. The meta-data contains *empty* run-time references to other components and is updated by the kernel using the control functions with *actual* run-time references. Once the metadata is updated, the component object is able to redirect a call to any other component (or system level functionality) without needing the kernel.

There are two types of interactions enabled by the proposed architecture: loose and strict. The loose interactions (dashed arrows in Figure 1) happen between two different components *or* between a component and the system. This way of incorporating the loose interactions is specific for the proposed architecture and enables the automatic conversion of static source to dynamic components. The strict interactions are only used inside the system and kernel where no partial updates are required.

The required additional functionality of the proposed architecture that enables partial updates, compared to a static system, is depicted using grey boxes in Figure 1. It can be seen that only the kernel, that manages the code updates, and the component objects, that enable the loosely coupled binding model, are required for updating components.

4.3. Modifications to the standard boot and update process

The architecture introduces overhead in terms of extra actions during the system boot and update. Figure 2 presents the sequence diagram of the *boot-phase* interactions between the kernel, system level and components. The interactions are numbered sequentially and the grey boxes on the lifelines indicate the overhead in terms of additional interactions compared standard systems.

When a device boots, the kernel first initializes the system (1). Then it transforms system level functionality (i.e. the platform independent interfaces) into HIL component objects and stores them in the database (2).

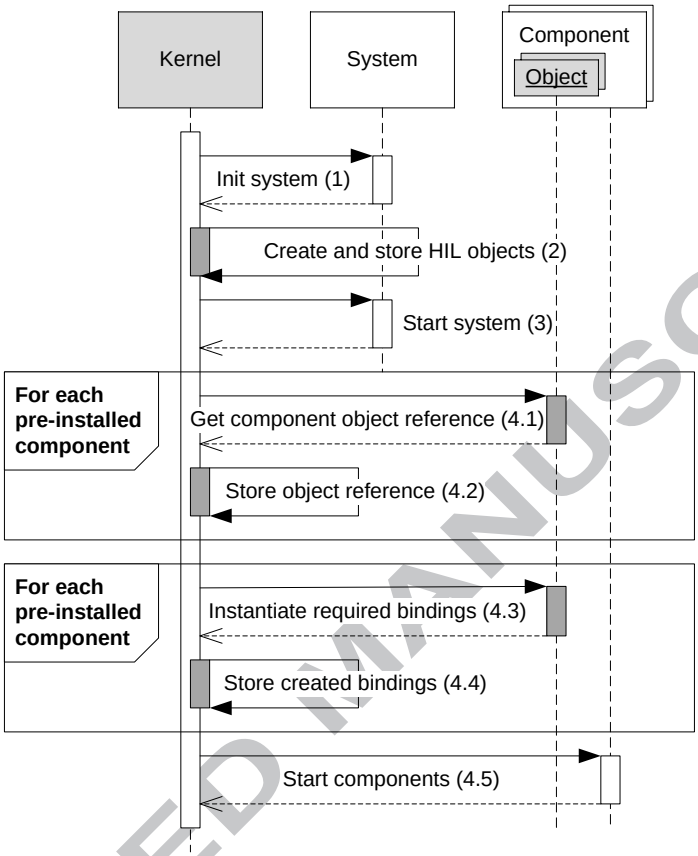


Figure 2: Step-by-step overview of the boot phase in the architecture (each step is numbered and the added functionality is depicted by grey boxes on the lifelines).

This enables the components in the network and application level to use the system level indirectly. Finally the system is started (3). After booting the system level, the kernel initializes the bindings for each pre-installed protocol and application (4). To this end, the kernel uses the control interface embedded in each component object and requires two iterations over all component objects. In a first iteration, the kernel retrieves the reference of each component object (4.1) and stores it in the database (4.2). In the second iteration, the component object metadata is used to identify the required bindings and instantiate them (4.3). The created bindings are registered in order to enable future updates (4.4). Then the components are started normally (4.5). The

kernel is only involved at boot-time or when new components are loaded, in all other cases, the components are executed normally.

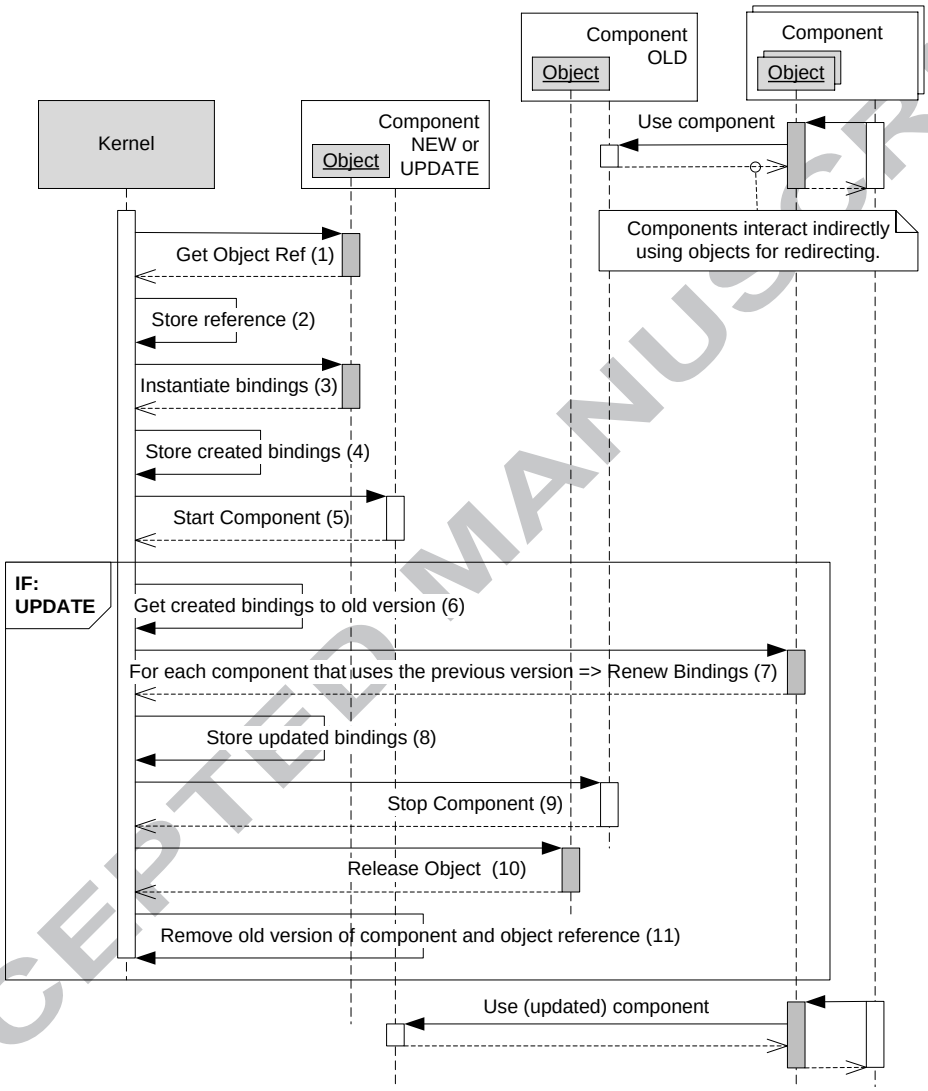


Figure 3: The interactions between the kernel and components when they are added or updated. The upper right part illustrates how component interact indirectly. The grey boxes on the lifelines indicate the interactions required for applying a loosely coupled binding model.

While the system is running, new component can be added and old ones

can be updated following the sequence diagram depicted in Figure 3. The kernel first needs to obtain (1) and store (2) a reference to the embedded component object. From this reference, it can identify the required bindings and instantiate them in the object (3). Then the created bindings are stored (4) and the component is started (5). When updating a component some additional steps are required for renewing the bindings in other component objects that use it. First, the kernel searches the database for existing bindings to the old version of the component (6). Then, these bindings are renewed using the reference to the new component object (7) and updated in the database (8). After replacing the bindings, the previous version of the component can be stopped (9) and the corresponding object can be released (10). Finally, the kernel can safely remove the old component and all references to its object (11). Now, the other components use the new version instead of the old one.

5. Implementation

This section demonstrates how presented architecture can be applied to Contiki[1], to enable dynamic code updates in the network level while maintaining backwards compatibility with the existing Contiki network stacks. The implementation of following design concepts will be discussed: i) loose coupling, ii) system boundary, iii) hardware abstraction and iv) grouping of functionality

5.1. Loosely coupled binding model: component objects

As mentioned in Section 4.2, component objects are introduced to enable loosely coupled interactions in the dynamic component level. A component object uses **indirect functions calls** instead of **direct function calls** to interact with other components. In order to invoke external functions, indirect function calls use function pointers that can be retrieved and re-assigned at runtime. In order to implement this efficiently, **function pointers are aggregated on a component level** in function pointer arrays included in the meta-data of each component object. Component objects can invoke external functions indirectly using a reference to the component object that provides the function array. The exact structure of a component object is

[1]The implementation is based on the main Contiki release 2.7 (`http://sourceforge.net/projects/contiki/files/Contiki/`).

illustrated in Figure 4 and includes i) component object definition, ii) object control functions and iii) component object dependency array.

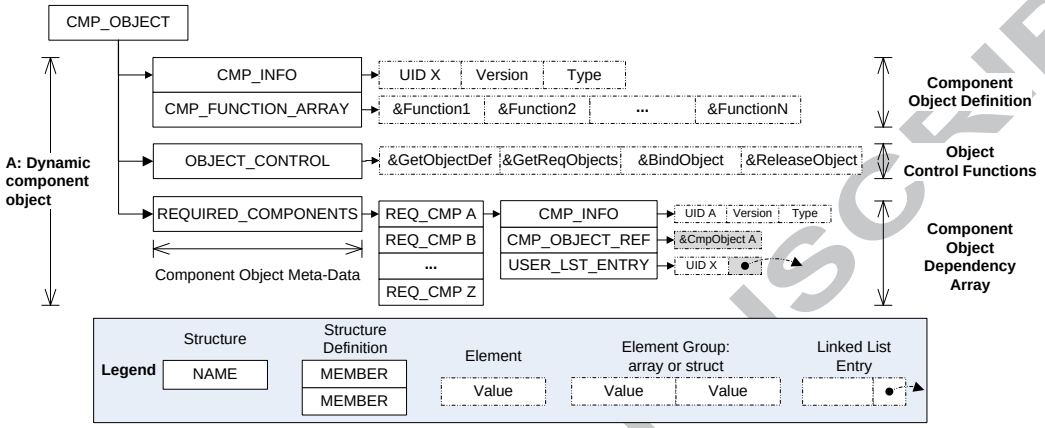


Figure 4: Definition of the component objects. A component object contains structures that enable other components to use this component. From top to bottom: i) component object definition, ii) component object control functions and iii) component object dependency array. The meta-data manipulated by the kernel is indicated using grey boxes.

The component object definition consists of meta-data used by other components to identify, and interact at run-time with, the component. It first contains component information such as unique ID (UID), version and type allowing to identify each version of a component. The UID is a unique hash value generated using the component string name. To allow access to external functions, the component object also contains a component function array that holds a function pointer for each external function implemented by the component.

The object control functions are used by the kernel to (re)-bind a component automatically. For this, they implement specific functions such as *GetObject* that returns a reference to the component object definition that can be stored in the database, *GetRequiredObjects* that returns an array with the dependencies defined by the component object and used by the kernel to bind the object, *BindObject* that triggers the object to instantiate (update) the dependency array and create (renew) the required bindings and *ReleaseObject* that triggers the object to release the dependency array.

The component object dependency array is used by the kernel to identify

and create the required bindings with other components. It consists of the following parts. First, it stores a copy of the component information used by the kernel to search the required object references in the database. Second, it stores a reference to the component object instantiated (updated) by the kernel after binding. And third, it stores a user list entry that is necessary for storing the created (renewed) binding in the component database.

Using the object reference of another component, its function pointer array can be used to call the components external functions indirectly. To ensure that existing protocols can be reused, a two-phase compilation in the build system is used. This process **automatically generates and embeds component objects in each component without requiring source code modifications and replaces external function calls with stub function calls using stub headers**, as explained in Section 5.4.

5.1.1. Component objects and stub headers: an example

The difference between a strict binding model, such as the one used in standard Contiki, and a loosely coupled binding model that is used in our architecture, is illustrated in Figure 5 using a simple example in which the Rime [36] unicast component calls the Rime broadcast component.

When applying a strict binding model (upper part of the figure), unicast uses the actual memory addresses to make direct calls to the functions implemented by broadcast. The addresses are assigned during linking and cannot be changed afterwards. This implies that when updating or replacing the broadcast component, also the unicast component must be updated.

In order to apply a loosely coupled approach (lower part of the figure), the unicast component includes an automatically generated *stub header* in which each function definition is replaced by a stub version with an identical signature. The stub functions are implemented by the *unicast object* and use a refernce to the broadcast object to make indirect function calls. An indirect function call uses a function pointer that can be changed at runtime.

In the current example, the unicast object will use a reference to the broadcast object to access the function pointers that correspond to the functions implemented by broadcast. The reference to broadcast is instantiated by the kernel when unicast was loaded by filling in the corresponding empty element in the meta-data as discussed in Section 4.2. The kernel hence requires that each component object defines meta-data (see Figure 4) enabling the kernel to automatically (re-)bind components at run-time.

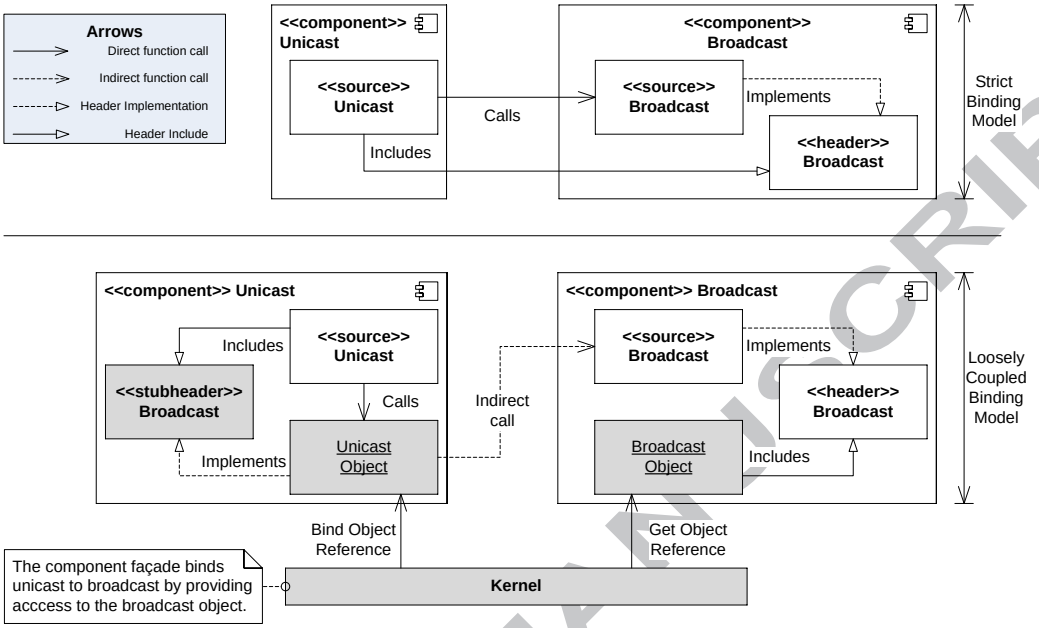


Figure 5: An example illustrating the indirect interactions between components. The upper part demonstrates a strict binding model (used in Contiki), the lower part illustrates the proposed loosely coupled binding model. The additional building blocks are depicted using grey boxes.

5.1.2. Component object and stub headers: benefits and limitations

The use of component objects and stub headers has a number of benefits, the following three being the most prominent.

i) *Limited CPU overhead.* Using indirect function calls always introduces extra execution delay for making a function pointer call instead of a direct call. The proposed stub functions are able to redirect the function calls in $O(1)$ complexity using the reference to the component object that implements the required function. This is much less then the overhead in similar solutions [9, 32].

ii) *Distributed memory overhead.* Because each component object defines its own metadata, the memory requirements scale with the number of components. A component database, thus, only needs to hold a reference to the component object and maintain a dependency list for each component.

iii) *Trade-offs between ROM and RAM memory are possible.* Because most of the metadata of a component object is read-only it can be placed in

ROM or RAM. Since component objects and stub headers are automatically generated, the build system can let the user choose which type of memory is allocated.

There are two main constraints to allow the loosely coupled binding model to be applied without requiring source code modifications.

i) *Avoid direct memory access.* Since direct memory access cannot be transformed into indirect memory access, shared memory access needs to be replaced with getters and setters. In Contiki, shared memory is used quite often, meaning that 1) in all source code that defines shared memory, getters and setters need to be provided and 2) in all source code using shared memory, the access needs to be changed to the provided getters and setters. This typically should require only minor source code modifications. Note also that direct memory access is typically considered as a bad programming habit.

ii) *There can be no static dependencies in the system level on the dynamic component level.* This means that there can be no direct references in the system level to dynamic components, otherwise updating such a component would require an update in the system level. This sounds straightforward but in Contiki many such dependencies exist, namely for the sensor device drivers and network drivers included in netstack. Section 5.4 provides more details on how this was solved in the current implementation.

5.2. Defining a clear system level boundary: the kernel layer

A second major design concept of the architecture focuses on the system level boundary. The kernel *enforces the system level boundary* by restricting interactions to the hardware independent (HIL) system interfaces and is essential for enabling a loosely coupled binding model in the dynamic component level. Introduced in Section 4.2, the kernel layer is further divided in i) *a component database*, ii) *a component façade* and iii) *a system façade* as illustrated in Figure 6.

i) *The component façade* is responsible for creating bindings between preinstalled components during the boot-phase and adding (updating) bindings when a component is added (updated) during run-time. It uses the object control functions embedded in each component to (a) obtain the object reference provided by the component (using the UID); (b) resolve the dependencies with other components; (c) create (update) bindings by instantiating (updating) the object references required by the component; and (d) release the bindings when a component is removed.

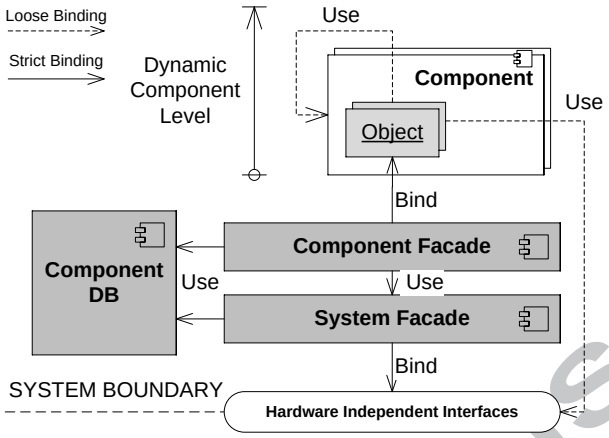


Figure 6: The implementation of the kernel requires three blocks: a component database, a component façade and a system façade. The extra functionality is depicted using grey boxes. The loosely coupled bindings are depicted using dashed arrows, strict bindings with full dashes.

ii) *The system façade* creates component objects for all hardware independent interfaces provided by the system level and adds the objects in the component database. This way, system level functionality is available for dynamic component linking. By introducing a system façade, *the system level boundary can be enforced transparent for the system level* (e.g. no source code modifications are required in the system level to use the kernel) and interactions can be restricted to HIL interfaces.

iii) The *component database* maintains a database to store references to each component object. Using these component objects references, components can interact loosely coupled with each-other and the system level. The component database includes (a) a fixed sized array with references to HIL objects and (b) a dynamic array with references to each component object in the network and application level. The objects from the fixed array provide access to the static system level functionality in the hardware interface layer (HIL). The dynamic array also contains a user list for each stored component. The user list of a component contains the UIDs of all component objects that use this component. In order to resolving dependencies during component updates, all the active bindings are stored in the component database.

5.3. Restrict interactions: use a hardware abstraction strategy

A *hardware abstraction strategy* is used to restrict interactions in the system level. This approach supports the kernel in *enforcing the system level boundary*, but requires a restructuring of the current Contiki system, as depicted in Figure 7.

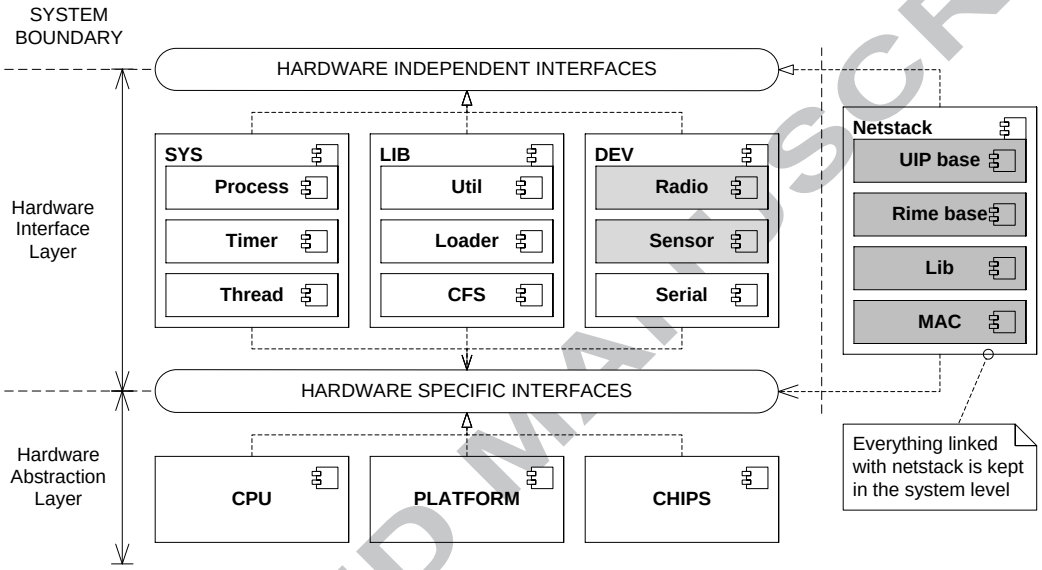


Figure 7: Overview of the refactored Contiki core, grey boxes denote limitations to the hardware abstraction architecture.

The system level is divided in hardware interface layer (HIL) and a hardware abstraction layer (HAL). The HIL transforms the hardware dependent interfaces provided by the HAL into HIL interfaces and provides additional HIL interfaces for core operating system features. The HAL provides the necessary abstraction for the cpu, platform and chip (e.g. sensors, radios) functionality. In order to provide network level re-programmability, all networking functionality, currently included in the static Contiki core system, needs to be moved to the dynamic component level. The system level boundary is placed above the remaining Contiki core which includes the much used process management, timers and libraries like the ELF loader and the contiki file system (CFS). As stated, dynamic components can only use system level HIL interfaces indirectly, using the HIL objects provided by the kernel. There can be no dependencies from the system level to the component level.

In theory, this only requires a small re-structuring of the Contiki core. Everything in \lib, \sys and \dev with hardware specific interfaces was moved to either \cpu, \platform or \chips. The former three directories contain all HIL (hardware interface layer) functionality. The latter three directories contain all HAL functionality. In practice, applying the hardware abstraction strategy and removing the network stack from the core system turned out to be more difficult than expected because there are many hidden dependencies inside the Contiki system.

The Contiki system initialises and starts all pre-installed network protocols from the main function and hence contains direct dependencies with these components. To resolve this, such functionality was moved to the init function of the component façade.

Some system level functionality is provided through shared memory (e.g. node address, event ids, sensors, etc ...). Since dynamic components cannot use direct memory access, getters and/or setters need to be provided. Inside the system level, direct memory access can still be used.

The sensor and radio *device drivers in the Contiki HAL also provide a HIL-like interface* by instantiating a generic driver struct that contains function pointers to the hardware dependent functions. Again, these use shared memory which can be resolved by providing getters. The added functions however, can not be added to the HAL interface but are decoupled and provided through HIL interfaces. This modification is indicated on the figure by the grey boxes inside \dev.

Harder to solve was the problem that *Contiki requires static dependencies between the network drivers* again using shared memory. This can only be resolved using major source code modifications to the mac and base network drivers included by *netstack.h*. For maintaining backwards compatibility, these are kept in the system level in the proposed implementation (see grey boxes in Figure 7). A large amount of networking functionality can however still be moved to the dynamic component level. This is indicated at the right side of Figure 7.

It is worth noting that these changes needed to be made only once and the not impact the design or reuse of other network functionality.

5.4. Group common functionality in components

The final design concept is to *group common functionality in components*. These concepts are applied on existing Contiki network protocols and applications. This is necessary for automatically generating the component objects

that enable a loosely coupled binding model. In order to demonstrate the network level (re)programmability, the Contiki Rime stack was chosen. Since Rime is very modular in design, it is easier to transform the Rime modules to upgradeable components and to analyse the effect of the design concepts.

In Rime, network functionality is divided in small modules (i.e. primitives) that can be stacked/combined to implement a routing protocol. A new routing protocol can be created by adding new modules on top of the existing ones (e.g. tree routing, multihop mesh). In the Contiki OS, all Rime modules are inside the system level and require strict bindings, thus they can not be updated separately.

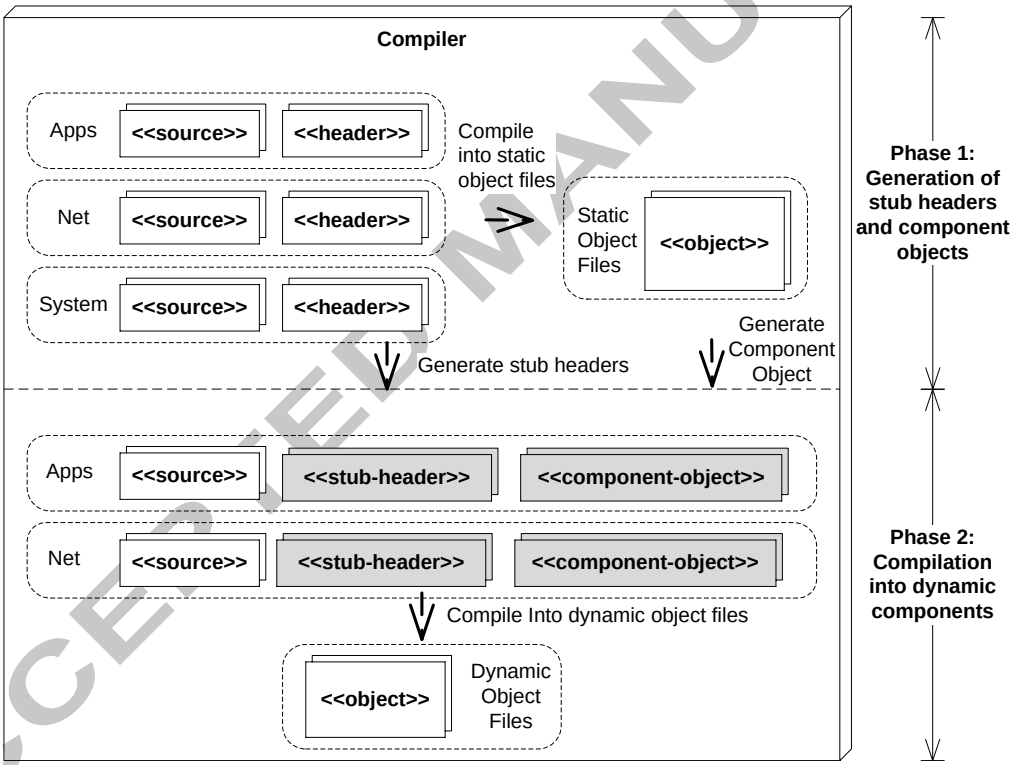


Figure 8: An overview of the build-system. There are two stages, first source code is compiled into static objects. Then using the objects and the original headers, component objects and stub headers are generated.

In the proposed reference implementation the modularity of Rime is fully exploited. Each Rime module is a loosely coupled dynamic component that

can be updated separately. The current implementation includes each Rime module above anonymous broadcast (which works directly above the MAC driver provided by *netstack.h*). The current implementation of the build system allows to transform Rime modules transparently by embedding auto-generated component objects during compilation. This requires a two phase compilation process as illustrated in Figure 8. First, the source code is compiled into static object files. Based on these object files, component objects are generated using custom tools. The original headers are transformed into stub headers. The original directory structure is also maintained to allow automatic inclusions in the next compilation phase. Second, the source code is compiled again, now including the stub headers and component object, resulting in dynamic object files that can be added or updated after deployment. The generation of component objects requires a one-on-one link between header and source file. Because each Rime module implements a header with a well defined interface this is not a problem. For systems where a one-to-one link between header and source file is not available, the build system can be modified accordingly since this is not a conceptual constraint of the proposed architecture.

6. Evaluation

This section presents the experimental and analytical evaluation of the reference implementation of GITAR. To demonstrate the improvements compared to prior art, the results are compared with a standard Contiki [27], SOS [9] and RemoWare [34]. The remainder of this evaluation section is structured as follows. i) First, each system is analyzed from a *functional viewpoint* in Section 6.1, identifying how dynamic components and interactions are implemented. ii) Next, in Section 6.2, an *experimental evaluation* is performed to analyse the memory overhead and the processing overhead of several network protocols and compare these with the overhead in the SOS and RemoWare approaches. iii) Third, a *mathematical model* is created in Section 6.3 that enables the calculation of the kernel level and dynamic component overhead in terms of ROM, RAM and processing overhead for any number of components with a known number of function calls for all discussed architectures. iv) Finally, the energy cost for deploying the dynamic components is analysed in Section 6.4.

6.1. Functional analysis

In this subsection, RemoWare and SOS are analysed from a functional viewpoint, focusing on three aspects: a) how can dynamic components interact with each-other; b) how can dynamic components interact with the system level functionality; and c) how can dynamic components be updated. For each system, a high level overview of this analysis is given in Figure 9. The figure illustrates how the Blink application interacts dynamically with the Led component (full black arrows) and how each system adds or updates a component (dashed arrows). Note that the available examples in both systems were limited to very simple applications so we had to introduce more complex function calls to enable a thorough comparison.

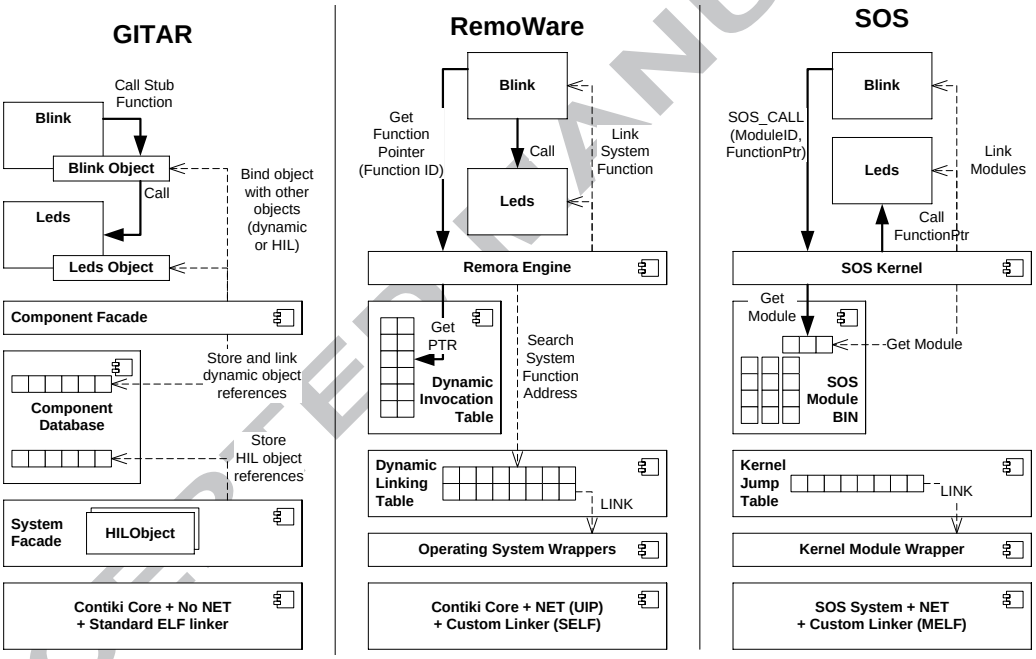


Figure 9: high level overview of the compared systems. The dynamic interactions for making a function call are depicted using full black arrows. Actions required for updating components are indicated with dashed arrows.

As explained in Section 5.1.1, GITAR does not rely on the kernel to redirect the dynamic function calls. This is because normal functions are replaced with stub functions in which the correct function pointer is called. The function pointer is obtained using the reference of the called component.

In RemoWare and SOS however, the actual call to the function pointer is executed by the kernel. For this RemoWare maintains a dynamic invocation table (DIT) that contains an entry for each dynamic function storing the function pointers and unique function IDs. In SOS, each dynamic component provides function control blocks (FCB) for each of the dynamic functions they implement and requires pointers to FCBs to use dynamic functions provided by other components. To make a dynamic call, a pointer to the called FCB is passed to the kernel. This is necessary because SOS mandates that a component is scheduled before it is called.

In GITAR, system level functionality is also accessed using dynamic functions. As discussed in Section 5.2, the system façade offers HIL objects for this purpose. Again, this is different in RemoWare and SOS. In RemoWare, a dynamic link table (DLT) is added to the firmware of each device. This DLT contains the unique ID and function address of system level functions. SOS uses a somewhat similar structure (called a jump table) which only contains function addresses. A unique ID is not necessary since the jump table is always placed at the same memory offset and the order of functions is always fixed. Another difference between GITAR and the other systems is that the HIL objects contain function pointers to the actual system functions while the DLT and the jump table contain the addresses of wrapper functions.

Updating dynamic components in GITAR requires to update each reference to that component in other components. This is done by the component façade using the information stored in the component database (see Section 5.2) . A component is linked to both the system level and dynamic components using component object references. When a component is updated in RemoWare, all the entries in the DIT corresponding to the dynamic functions provided by that component also need to be updated. No updates are required in other components. The DLT is used to link components to the system level by replacing undefined function symbols with correct addresses. In SOS, a component update requires that all the pointers to the function control blocks provided by that component are also updated. This is done by iterating each component in the module bin and checking if the updated function is used. Since the jump table is always stored in the same order at the same memory offset, SOS module does not require linking with system level functions. Both RemoWare and SOS hence statically link dynamic components to the system level.

6.2. Experimental evaluation

The evaluation focuses on three examples that use the Rime network stack: example-collect, example-mesh and a combination of both example-collect-mesh. The collect example uses the pro-active tree routing Collect module which only has a limited number of dynamic components (6) but incorporates a several dynamic functions (36). The mesh example uses the re-active mesh routing Mesh module that requires more dynamic components (9) but implements less dynamic functions (40) relative to the number of components. To analyse the effects of the architecture on a larger example they were combined into the collect-mesh example in which both applications run in parallel. As the three examples are only implemented for Contiki and GITAR, the evaluation starts from the Blink application that is available in all four systems, and extends these applications with functional calls of the same complexity as used in the three Rime examples. This offers an objective ground for comparing with SOS and RemoWare. Note that the calculated memory usage of GITAR is a perfect match with the memory usage observed in the experiments.

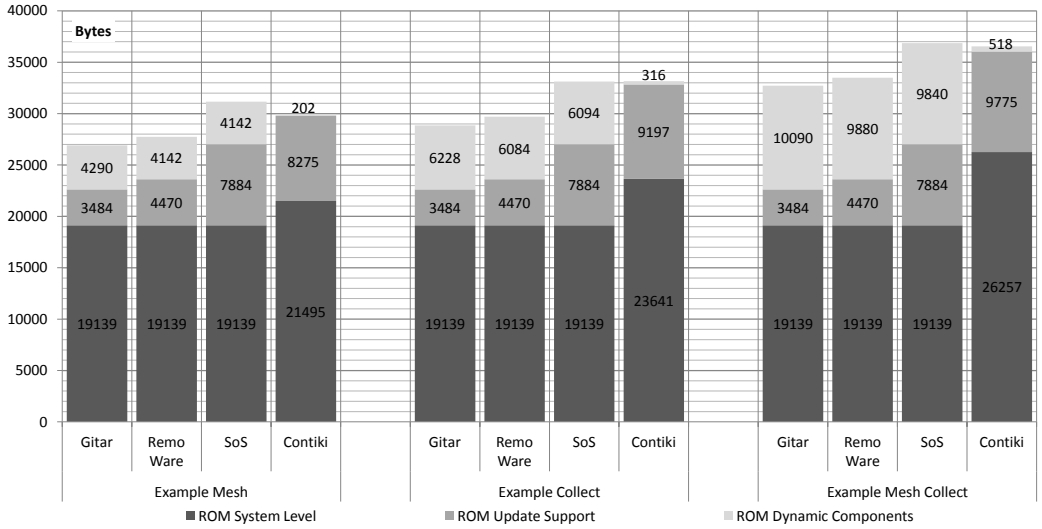
The reference implementation of the proposed architecture used Contiki version 2.7 and was validated on three different msp430-based platforms: TelosB (F1611 MCU: 8MHz, 48kB ROM, 8kB RAM), Zolertia z1 (F2617 MCU: 16MHz, 92kB ROM, 8kB RAM) and RM090 (f5437 MCU: 18MHz, 256kB ROM, 16kB RAM). All existing Rime modules in */contiki/core/net/rime* and examples in */contiki/examples/rime* could be converted automatically to dynamic modules without requiring source code modifications by using our adapted build system.

To verify that the kernel and dynamic interactions work, each example was tested on a small (5 nodes) setup. During these experiments, the CPU overhead of each dynamic interaction could be logged using a logic analyser by toggling GPIO-pins. Also the memory usage of each different part in the proposed architecture could be determined exactly using standard mspgcc-binutils tools (size, nm and readelf).

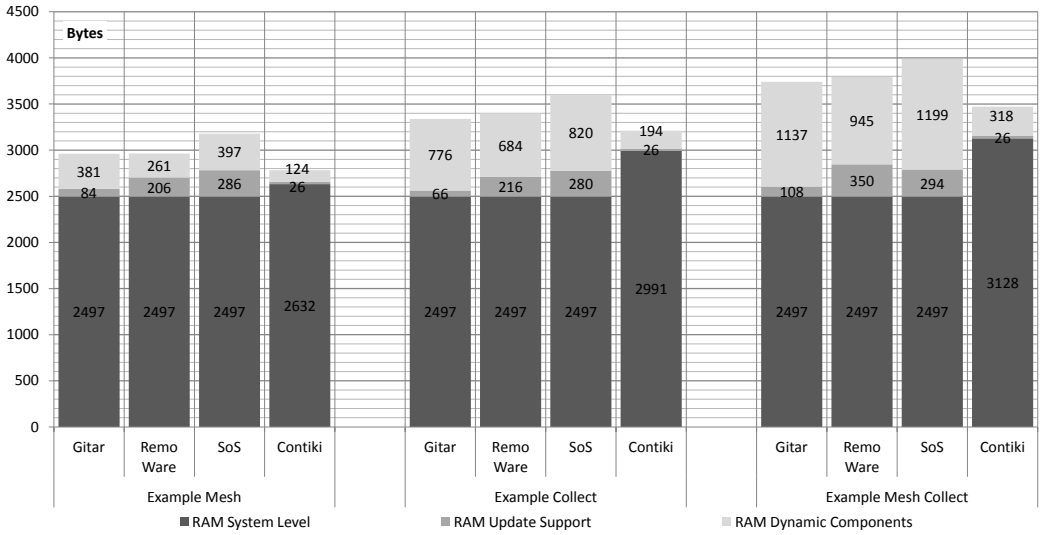
6.2.1. Overall memory Footprint

The overall memory usage for implementing the example applications in the reference implementation of the proposed architecture, SOS, RemoWare and standard Contiki is depicted in Figure 10. For each example and system, the memory usage is divided in system level functionality (dark grey), kernel update support (grey) and the dynamic Rime network level components and

example applications (light grey). Inside each bar the memory usage is given. The following conclusions can be made.



(a) Overview of the ROM usage



(b) Overview of the RAM usage

Figure 10: a comparison of the overall memory used in GITAR, RemoWare, SOS and standard Contiki. The memory usage is divided in: system level (dark grey), update support (e.g kernel level, grey) and dynamic component level (light grey).

i) From the figures it is clear that the ROM and RAM usage of the *system level* is always identical for the systems that support dynamic components (i.e. GITAR, RemoWare and SOS). This can be explained by the fact that they exclude the network protocols (in this case the Rime modules) and always implement the same functionality.

ii) In contrast, the ROM and RAM for *update support* is always less in GITAR compared to RemoWare and SOS that also support dynamic updates. The ROM for update support is larger in standard Contiki due to the use of a very large symbol table method used for supporting dynamic code updates in only a tiny part of the firmware, the application level. On the other hand standard Contiki uses less RAM for this purpose.

iii) The ROM and RAM usage of GITAR, RemoWare and SOS for *dynamic components* are comparable. In general, the ROM usage of GITAR is slightly higher and the RAM usage is only slightly lower in RemoWare. Of course, in standard Contiki the ROM and RAM usage is lower since only applications can be updated.

iv) The slightly higher memory usage of GITAR for dynamic components is more than compensated in the kernel level. Compared to RemoWare, the ROM (RAM) usage of the GITAR kernel is, on average, 986 (171) bytes less . For this, GITAR only requires, on average, 167 (135) bytes more in the dynamic component level. Compared to SOS, the average ROM (RAM) usage of the GITAR kernel is 4300 (201) bytes lower while the average ROM used in the dynamic component level is only 177 bytes higher. The average RAM usage is also 40 bytes lower in GITAR.

Overall, it can be seen that **in RemoWare and SOS, the ROM usage is *3%, respectively 14% higher* and the RAM usage is *1%, respectively 7% higher* compared to our GITAR for providing an *equal level of flexibility*. Compared to standard Contiki, *13% less ROM is required* for providing much more flexibility, i.e. *dynamic network level update functionality*, at the cost of *6% extra RAM***.

6.2.2. Runtime CPU Overhead

Using dynamic functions always results in efficiency overhead in terms of delay (CPU usage) compared to standard direct calls. This is because a dynamic invocation requires a function pointer that needs to be obtained at run-time and is necessary for enabling post-deployment updates in the network level.

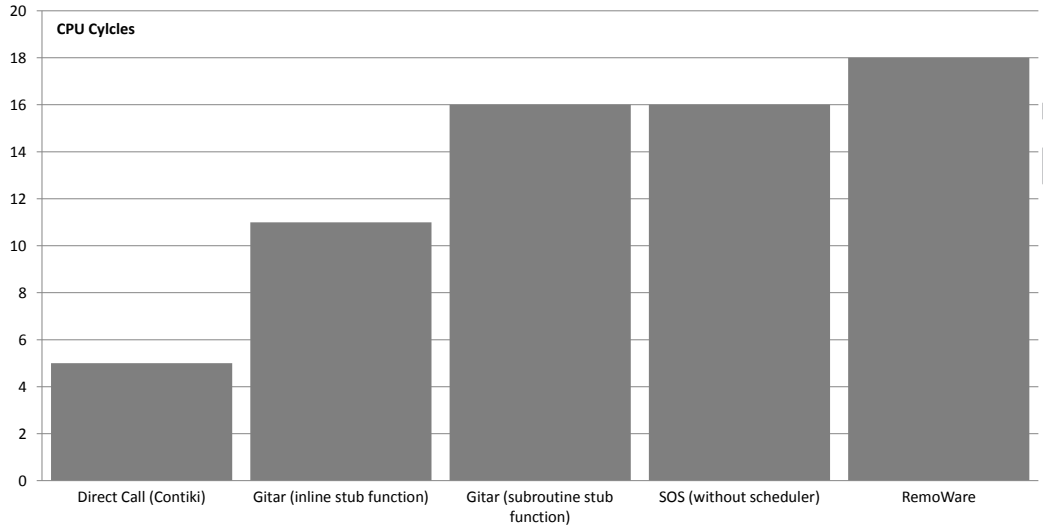


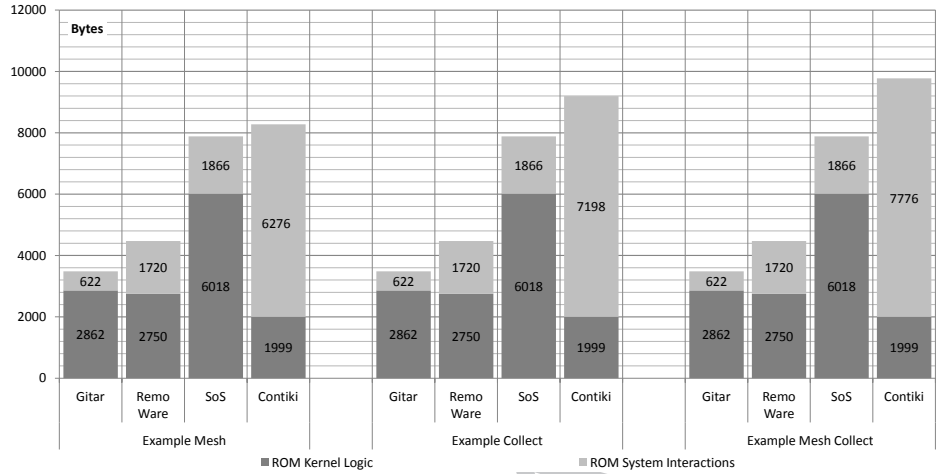
Figure 11: compares the minimal execution delay (CPU cycles) for a dynamic function call in Gitar (inline and subroutined) with RemoWare and SOS. Also the delay for a static call in Contiki is depicted. The function has no parameters and no return value.

The number of CPU cycles, required in each system, for executing a function call with zero parameters and no return value is depicted in Figure 11. It can be seen that the static (direct) call in standard Contiki is the most efficient requiring 5 CPU cycles. In GITAR, a dynamic (indirect) stub function call requires 11 CPU cycles if it can be in-lined (this is the case when only 1 call to the function is required) or 16 CPU cycles otherwise. In any case, the former is more efficient, while the latter is equal to the execution delay of a dynamic call in SOS and requires 2 cycles less then RemoWare. Note that for the overall results, the delay for scheduling a component in SOS and the variable delay for copying the function parameters in RemoWare is not accounted. **Overall, GITAR is at least as performant as SOS and RemoWare in terms of CPU usage, and outperforms them for in-line function calls.**

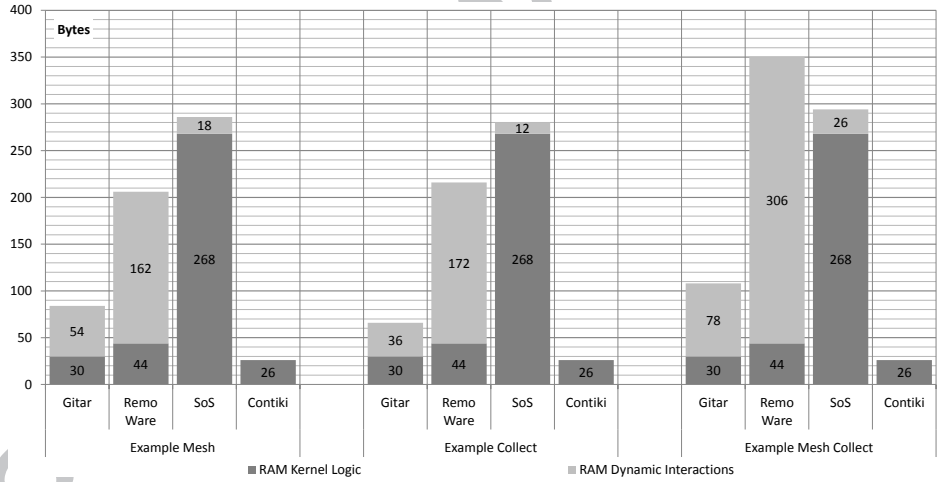
6.2.3. Detailed evaluation of the kernel level

Whereas the previous sections discussed the total overhead of a protocol stack, this section investigates in more detail the exact overhead that results from the kernel design. In each of the compared systems kernel level support

is necessary for enabling dynamic components to interact with each-other and with the system level functionality.



(a) ROM required to provide update support in the kernel.



(b) RAM required to provide update support in the kernel.

Figure 12: a comparison of the memory usage between GITAR, RemoWare and SOS. The memory required for the standard Contiki ELF loader and symbol table are also depicted as a reference. The memory usage is divided in kernel functionality (dark grey, e.g. the component linker, loader and runtime engine) and ROM required for providing access to the system level (light grey).

A distinction will be made between ROM and RAM overhead of the kernel. The exact memory usage is depicted on Figure 12.

(i) The kernel ROM requirements are compared in Figure 12a. The ROM requirements for implementing the kernel logic are depicted in dark grey. The ROM requirements that correspond with the example protocols are indicated in light grey. The former (dark grey parts) are determined by compiling each system, the latter (light grey parts) can be theoretically calculated using the formulas which will be derived in Section 6.3, together with the parameters defined in Table A.3. The standard Contiki ROM requirements for the ELF loader (dark grey) and static symbol table (light grey) are added as a reference. From the figure it is clear that GITAR (2862 bytes) and RemoWare (2750 bytes) require significantly less ROM memory than SOS (6018 bytes) for implementing the kernel logic. This is because the SOS kernel also implements a scheduler similar to the process scheduler provided by Contiki. Because the SOS linker relies heavily on the SOS scheduler, it could not be removed from the comparison. The main difference in the kernel logic of each system is the linker and loader. In GITAR, the standard ELF linker and loader is used, while the other systems require a custom, more complex, linker and loader. Another observation that can be made is that the ROM required for providing system level functionality in GITAR (622 bytes) is much lower then in both RemoWare (1720 bytes) and SOS (1866 bytes). This is a direct result of not using a dynamic linking table (RemoWare) or a system jump table (SOS) for this purpose. Instead, GITAR groups system level functionality on a component level, drastically reducing the required memory. Another reason for the lower ROM usage is that in the proposed architecture no wrappers are required for system functions.

(ii) The RAM requirements for the compared systems in each example are depicted in Figure 12b and divided in a fixed part (dark grey) required by the kernel logic and a variable part (light grey) required to enable dynamic interactions between components and to enable subsequent component updates. In the dark grey part, more RAM is always required by RemoWare (44 bytes) and SOS (268 bytes) compared to GITAR (30 bytes). The additional overhead of RemoWare and SOS can again be explained by the fact that they are using a non-standard ELF linker. The much higher RAM usage for implementing the kernel logic in SOS is anew introduced by the scheduler which masks the lower variable RAM usage of SOS. More precisely, it is due to the complex data structures (module stack and heap) used in the scheduler that less RAM is needed in the variable part. The higher RAM usage for

enabling dynamic interactions in RemoWare (average 213 bytes) compared to GITAR (average 56 bytes) and SOS (average 19 bytes) is introduced by the dynamic invocation table (DIT), that requires 4 bytes per dynamic function. In GITAR and SOS, this information is maintained by each component separately. The higher RAM in GITAR versus SOS is caused by the user list maintained for each component which allows for a faster update process.

In conclusion, the kernel used by GITAR is significantly more efficient in terms of ROM/RAM than the one used in comparable architecture (RemoWare and SOS respectively), and even compares favourably in terms of ROM with the default Contiki approach although in GITAR more flexibility is supported.

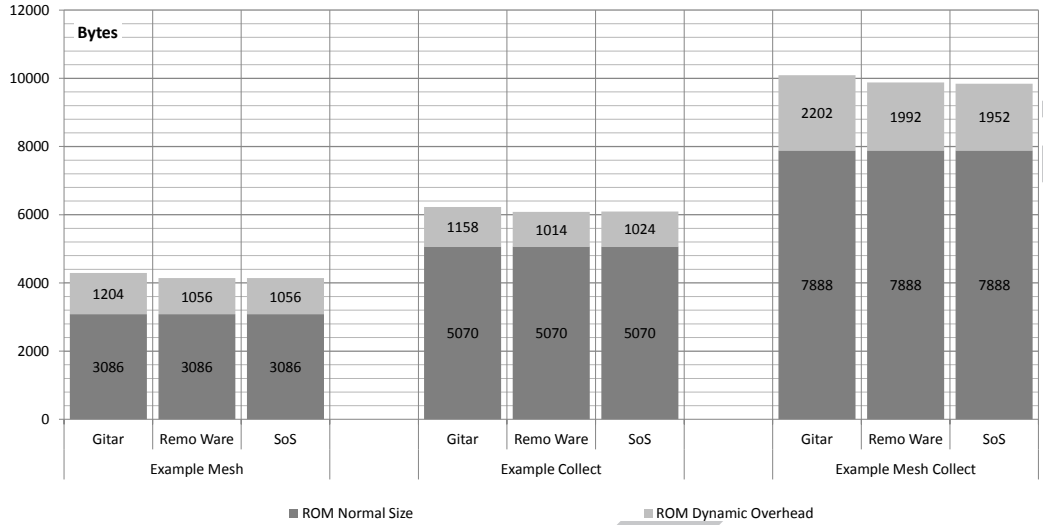
6.2.4. Detailed evaluation of the dynamic component level

The added flexibility for the Rime primitives and example applications comes at a cost. Compared to the static components used in Contiki, more ROM and RAM is required for the dynamic components in GITAR, RemoWare and SOS. Also run-time delay is inserted when using a dynamic, indirect, function instead of the direct functions used in Contiki.

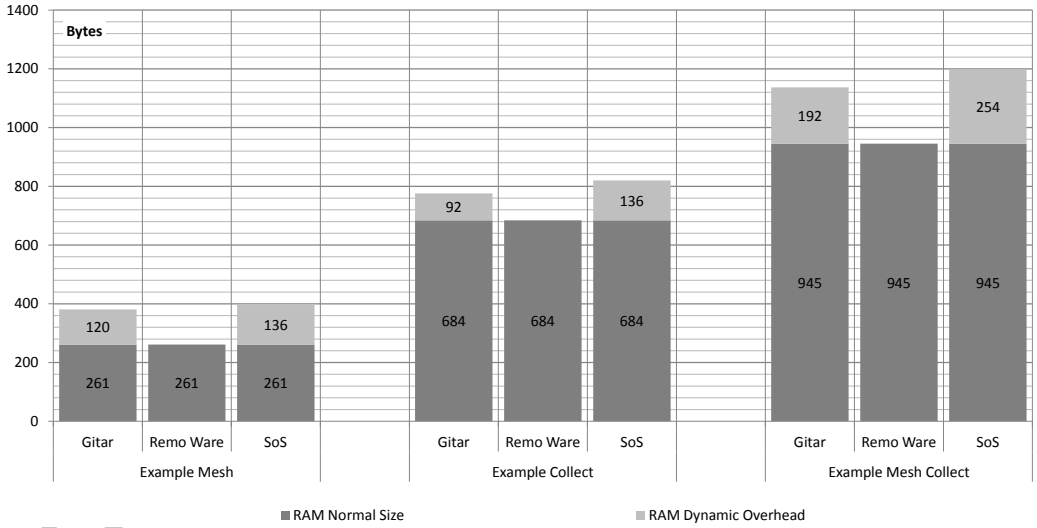
The ROM and RAM overhead per application and per architecture are shown in Figure 13. The dark grey part corresponds to the normal memory requirements in the standard Contiki examples. The light grey part represents the component overhead required for each of the compared systems. In the following discussion only the overhead (light grey) introduced by each system will be analysed.

From the results it can be seen that **GITAR introduces slightly more ROM overhead per component compared with RemoWare and SOS**. This is a consequence of not providing a dynamic invocation table in the kernel but storing it inside each component, thereby moving the overhead from kernel to component. Also more ROM is used for accessing system level functionality through dynamic functions (in RemoWare and SOS static calls are used for accessing system functions). On average GITAR requires 1521 bytes extra for all dynamic components versus 1354 in RemoWare and 1344 bytes in SOS. **However, due to the significantly reduced kernel overhead, the total ROM usage for the whole system is lower in GITAR**, as illustrated previously in Figure 10a.

When considering the combined RAM overhead for all dynamic components, GITAR requires less RAM then SOS. This is mainly because SOS links components to functions (using function control blocks or FCBs) while



(a) Overall ROM usage in the dynamic component level.



(b) Overall RAM usage in the dynamic component level.

Figure 13: overview of the overall ROM and RAM required by the dynamic components used in each example. The dark grey bars represent the normal memory usage for the Contiki modules. The light grey illustrates the extra memory usage, required by the dynamic extensions in GITAR, RemoWare and SOS.

in the proposed architecture components are linked to components. Since the number of components is lower then the number of functions, this is more efficient and hence GITAR requires on average only 10 RAM bytes per component while SOS needs 14 bytes. RemoWare also links components to functions but maintains the information (function IDs and pointers) and bindings in the kernel and hence does not introduce RAM overhead in the dynamic components. In the kernel level however, on average 17 bytes per component is needed by RemoWare. **Because GITAR links components to each-other rather then to functions less RAM is required overall to enable dynamic bindings.**

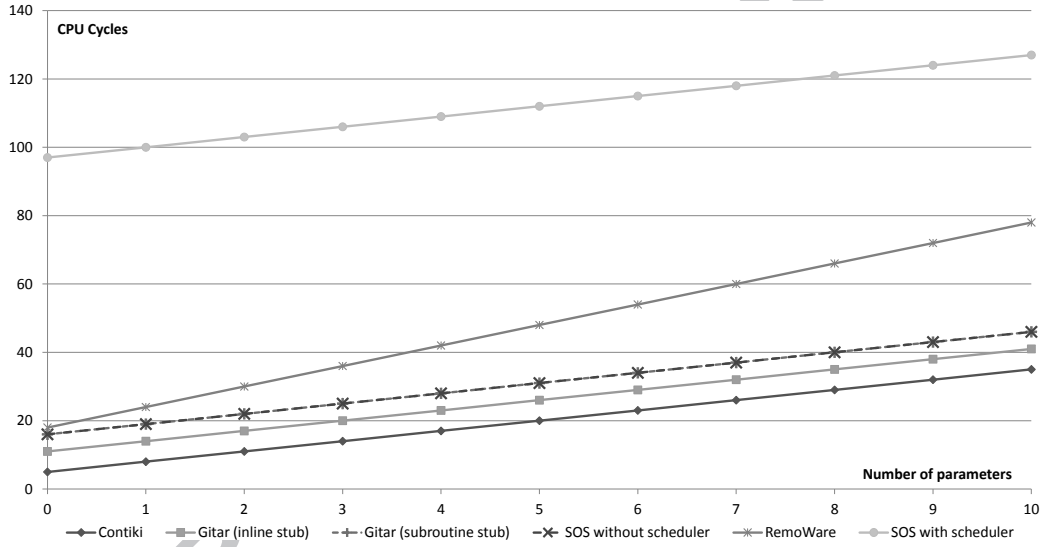


Figure 14: illustrates the delay for using dynamic functions in GITAR, RemoWare and SOS for different number of function parameters per function call. For GITAR both the inline and subroutine stub functions are depicted. For SOS also the actual delay with scheduler is given.

Finally, the use of indirect function call introduces *CPU overhead* for each call to an external function. The total CPU overhead depends on the number of external function calls as well as the number of parameters of each function call. In Figure 14 the delay for a dynamic function call is shown, varying the number of parameters n_{Param} from 0 to 10. For GITAR both the inline and subroutine stub functions are depicted. For SOS, the overhead with and without the scheduler is given. As indicated before, direct calls are

always more efficient than indirect, dynamic calls. A direct call requires 5 cycles plus 3 cycles per parameter. The inline stub function only requires two extra move operations to obtain the correct function pointer resulting in 11 cycles plus 3 cycles per parameter. The subroutine stub function is less efficient because an extra call is required to the stub subroutine, adding another 5 cycles (i.e. 16 cycles + 3 per parameter). This is identical for SOS if the the scheduling is not accounted (note that scheduling is always required and intertwined with the indirect call mechanism). If the scheduling of SOS is also considered, the CPU overhead is drastically higher (97 cycles plus 3 cycles per parameter). Opposed to the dynamic calls in GITAR and SOS, RemoWare requires to copy the function parameters twice (6 cycles per parameter). This explains the steeper slope for the delay of RemoWare. Also the initial delay is higher (18 cycles).

6.3. Mathematical analysis

Finally, this section will provide mathematical formulas that can be used to calculate the ROM, RAM and CPU overhead for arbitrary components, based on the number of external functions and the number of function parameters. Using these formulas, the experimental results obtained in the previous section can mathematically be recreated. All formulas are shown in a very generic form, with Appendix A and Appendix B listing the exact values that can be used for the different architectures to calculate the kernel overhead and the component overhead respectively.

6.3.1. Kernel memory overhead

The memory overhead of the kernel is divided in: (a) ROM required for implementing the kernel logic ($B^{ROM}_{KernelLogic}$) and exposing system components (B^{ROM}_{SysCmp}) and functions ($B^{ROM}_{SysFnct}$); and (b) RAM required for the kernel logic ($B^{RAM}_{KernelLogic}$) and for maintaining dynamic components (B^{RAM}_{DynCmp}) and functions ($B^{RAM}_{DynFnct}$). The kernel ROM overhead is fixed and depends on the number of system level components(n_{SysCmp}) and functions($n_{SysFnct}$). The kernel RAM demands is variable and depends on the number of dynamic components (n_{DynCmp}) and functions ($n_{DynFnct}$).

Using the above it is possible to express the kernel ROM and RAM over-

head using Equations (1a) and (1b) respectively:

$$b^{ROM}_{Kernel} = B^{ROM}_{KernelLogic} + n_{SysCmp} \times B^{ROM}_{SysCmp} + n_{SysFnct} \times B^{ROM}_{SysFnct} \quad (1a)$$
$$b^{RAM}_{Kernel} = B^{RAM}_{KernelLogic} + n_{DynCmp} \times B^{RAM}_{DynCmp} + n_{DynFnct} \times B^{RAM}_{DynFnct} \quad (1b)$$

To determine the exact overhead of the kernel, the constants (B^{MEM}_x) and the number of variables (n_x) in equations Equations (1a) and (1b) need to be determined. This can be done by analyzing the source code and inspecting the sizes of each memory object in the compiled binaries for SOS, RemoWare and GITAR. The constants (B^{MEM}_x) and parameters n_x are defined in Tables A.2 and A.3 and are discussed in Appendix A. Using these formulas and parameter values, the exact same overhead values can be derived as shown in the experiment evaluation section.

6.3.2. Component memory overhead

The memory overhead of a dynamic component is divided in overhead required for enabling the kernel to link (or bind) the component with other components or system functionality and overhead introduced per occurrence of a system or dynamic function call in source code.

The binding memory overhead is further divided in: (a) MEM required for providing component information ($B^{MEM}_{DynCmpInfo}$) and providing dynamic functions ($B^{MEM}_{ProvFnct}$); (b) MEM for using system components($B^{MEM}_{ReqSysFnct}$) and functions ($B^{MEM}_{ReqSysCmp}$); and (c) MEM for using dynamic components ($B^{MEM}_{ReqDynCmp}$) and functions ($B^{MEM}_{ReqDynFnct}$). Note that MEM is used because sometimes both RAM and or ROM are required. The binding memory overhead thus depends on the number of provided functions $n_{ProvDynFnct}$, required system components $n_{ReqSysCmp}$ and functions $n_{ReqSysFnct}$, and required dynamic components $n_{ReqDynCmp}$ and functions $n_{ReqDynFnct}$. It can be expressed as in Equation (2) for ROM memory and Equation (3) for RAM memory.

$$\begin{aligned} b^{ROM}_{CmpBinding} =& B^{ROM}_{DynCmpInfo} + n_{ProvDynFnct} \times B^{ROM}_{ProvFnct} \\ &+ n_{ReqSysCmp} \times B^{ROM}_{ReqSysCmp} + n_{ReqSysFnct} \times B^{ROM}_{ReqSysFnct} \\ &+ n_{ReqDynCmp} \times B^{ROM}_{ReqDynCmp} + n_{ReqDynFnct} \times B^{ROM}_{ReqDynFnct} \end{aligned} \quad (2)$$

$$
\begin{aligned}
b^{RAM}_{CmpBinding} =& B^{RAM}_{DynCmpInfo} + n_{ProvDynFnct} \times B^{RAM}_{ProvFnct} \\
&+ n_{ReqSysCmp} \times B^{RAM}_{ReqSysCmp} + n_{ReqSysFnct} \times B^{RAM}_{ReqSysFnct} \\
&+ n_{ReqDynCmp} \times B^{RAM}_{ReqDynCmp} + n_{ReqDynFnct} \times B^{RAM}_{ReqDynFcnt}
\end{aligned} \tag{3}
$$

The overhead introduced by component interactions has to be expressed differently in our architecture compared with RemoWare and SOS. Our approach for enabling dynamic function calls uses stub functions and makes no distinction between system and dynamic calls (a distinction which is present in the other two architectures). The stub functions do not rely on the kernel to redirect the call as explained in Section 5.1 and can be inlined to increase speed at the expense of ROM. The associated overhead is given by Equation (4) where the ROM for each call to a stub function subroutine is denoted by $B^{ROM}_{StubCall}$ and to an in-line stub function by$B^{ROM}_{InlineStubCall}$. The overhead also depends on the number of calls to subroutine stubfunctions $n_{StubCall}$ and inline stub functions $n_{InlineStubCall}$.

$$
b^{ROM}_{CmpInteraction} = n_{StubCall} \times B^{ROM}_{StubCall} + n_{InlineStubCall} \times B^{ROM}_{InlineStubCall} \tag{4}
$$

In Remora and SOS, ROM overhead can be split in (a) ROM per system call in the source code ($B^{ROM}_{SysCall}$); and (b) ROM per dynamic call ($B^{ROM}_{DynCall}$) and dynamic parameter copy (($B^{ROM}_{DynCallParam}$)). The interaction overhead depends on the number of calls to system functions $n_{SysCall}$, calls to dynamic functions $n_{DynCall}$ and parameters per dynamic function $n_{DynParam}$ as shown in Equation (5).

$$
\begin{aligned}
b^{ROM}_{CmpInteraction} =& n_{SysCall} \times B^{ROM}_{SysCall} \\
&+ n_{DynCall} \times B^{ROM}_{DynCall} + n_{DynParam} \times B^{ROM}_{DynCallParam}
\end{aligned} \tag{5}
$$

Using equations Equations (2) to (5), it is possible to express the ROM and RAM overhead of each dynamic component in each of the systems $b^{ROM}_{DynCmp}, b^{RAM}_{DynCmp}$ using Equations (6a) and (6b) respectively:

$$
b^{ROM}_{DynCmp} = b^{ROM}_{CmpBinding} + b^{ROM}_{CmpInteraction} \tag{6a}
$$

$$
b^{RAM}_{DynCmp} = b^{RAM}_{CmpBinding} \tag{6b}
$$

Similarly, the component overhead obtained in Section 6.2.4 can be produced by using Equations (2) to (5) where the parameters n_x and constants B_x^{ROM} are determined again by analysing the source code, object files and assembler output. The exact values are defined in Tables B.4 and B.5 and explained in Appendix B. Using these formulas and parameter values, the exact same overhead values can be derived as shown in the experiment evaluation section.

6.3.3. Calculating the CPU overhead

There is also a *run-time penalty for using indirect interactions* between dynamic components compared to direct interactions with the system level. The delay overhead is expressed in CPU cycles. Note that this delay is added every time an external call is made at run-time. The delay for a direct call $d_{DirectCall}^{CPU}$ is expressed in Equation (7a) in terms of the number of parameters n_{Param}. For simplifying the equations, the return value is also counted as a parameter. A direct call requires to move all parameters to the stack, call the external function at the physical address provided and copy the return value from the stack.

$$d_{DirectCall}^{CPU} = D_{CALL}^{CPU} + n_{Param} \times D_{MOV}^{CPU} \tag{7a}$$

$$d_{StubCall}^{CPU} = d_{DirectCall}^{CPU} + D_{CALL}^{CPU} + \ 2 \ \times D_{MOV}^{CPU} \tag{7b}$$

$$d_{InlineStubCall}^{CPU} = d_{DirectCall}^{CPU} + \ 2 \ \times D_{MOV}^{CPU} \tag{7c}$$

$$d_{RemoWare}^{CPU} = d_{DirectCall}^{CPU} + D_{CALL}^{CPU} + \ 2 \ \times D_{MOV}^{CPU} + n_{Param} \times D_{MOV}^{CPU} \tag{7d}$$

$$d_{SOS}^{CPU} = d_{DirectCall}^{CPU} + D_{Scheduler}^{CPU} + D_{CALL}^{CPU} + \ 2 \ \times D_{MOV}^{CPU} \tag{7e}$$

where $D_{CALL}^{CPU} = 5$, $D_{MOV}^{CPU} = 3$, $D_{Scheduler}^{CPU} = 51$

An indirect (dynamic) call always requires the overhead of a direct call plus the overhead to make the call redirection. Since the indirect function call is implemented differently in each system, different equations are necessary.

The delay overhead ($d_{StubCall}^{CPU}$) for a sub-routined stub call in our reference implementation is given inEquation (7b). Extra delay is added to make a call D_{CALL}^{CPU} to the stub subroutine and for moving (D_{MOV}^{CPU}) the required object reference and function pointer to the stack. When the stub function is inlined (Equation (7c)) only two move operations are required.

In RemoWare, the delay overhead (Equation (7d)) additionally requires two move (D^{CPU}_{MOV}) operation for the function id and function pointer and one call operation (D^{CPU}_{CALL}) to the retrieved function pointer. RemoWare also requires to move each function parameter again ($n_{Param} \times D^{CPU}_{MOV}$).

In SOS, the dynamic function call delay (Equation (7e)) is identical as in our sub-routined stub call when only considering the function redirection. However, since SOS additionally requires to schedule the called component ($D^{CPU}_{Scheduler}$), this also needs to be taken into account.

Finally, it is possible to calculate the CPU run-time overhead per external function call for using dynamic (indirect) functions in GITAR, RemoWare and SOS by using Equation (7). Using this formulas, the same overhead CPU values can be derived as shown in Section 6.2.4.

6.4. Energy usage analysis

The transformation from static to dynamic components employed in GITAR adds extra RAM and ROM bytes. This will also have an influence on the size of the compiled ELF object file and hence may impact the energy needed to distribute and install the dynamic components. By analysing the deployment overhead (e.g. the overhead for distributing and installing the ELF object files) it is shown that the transformation from static to dynamic components does not necessarily increase the ELF file size or energy consumption. This is because on one hand, the size of the ELF file will *increase* due to the additional ROM/RAM bytes and relocation entries. On the other hand, the size of the ELF file will *decrease* because it contains fewer string symbols in the symbol and string table. The ELF file size will therefore be higher for some components, and smaller for others.

To evaluate the energy usage during deployment we considered two examples. First we compare adding or updating the collect application and corresponding Rime modules in GITAR and Contiki. Then, we compare adding or updating mesh application and the corresponding Rime modules in GITAR and Contiki. For each case, we consider the standard ELF file and two possible ELF file reduction methods available in the literature: SELF object files [37] used in RemoWare [34] and ELF files compressed with the Lempel-Ziv algorithm (LZ77) [38, 39]. Another possible compression approach is represented by compressed ELF (CELF) [27] which replaces the 32-bit ELF headers with 16-bit CELF headers. The SELF files considered in our evaluation compress CELF files further by removing unused elements in

the 16- bit CELF headers. We disregard the MELF file format which is SOS specific and can hence only be used on SOS modules.

By using standard compression techniques available in the literature, some of which are also used by architectures to which we compare against such as RemoWare and Contiki, we are able to provide a good estimation of the trade-off in energy consumption. However a direct comparison with RemoWare and SOS as done in the previous evaluation subsections is not possible because the Contiki Rime modules cannot be compiled in RemoWare or in SOS and hence the actual size of the resulting ELF object files cannot be determined.

Figure 15 presents the results of our evaluation by comparing the cost in energy (mJ) to transfer all required ELF object files over a single link without protocol overhead. The file size and energy costs of each component is given as appendix in Table C.6 and Table C.7 respectively. The energy cost was determined using the same method and input data as in [27, 38, 39]. The energy usage was derived by multiplying the number of bytes per file with the energy required to receive one byte (0.0048 mJ) using the CC2420 transceiver[2].

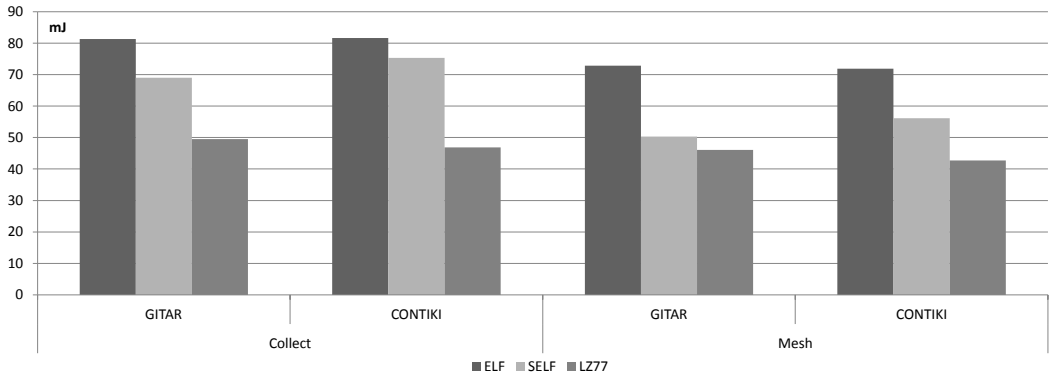


Figure 15: Comparison of the energy required to receive mesh (or collect) application and required Rime modules in GITAR and Contiki. The energy usage is given for three file formats: standard ELF file, the smaller SELF file and for ELF files compressed with LZ77.

[2]The CC2420 transceiver is used on both Zolertia Z1 and TMote Sky sensor nodes. The energy cost per byte was determined experimentally in [27] and also used in [38, 39].

From Figure 15, it can be seen that: (i) When using standard ELF files the extra ROM and RAM usage introduced in GITAR is compensated by reducing the size of the string table and symbol table. This is possible because the symbolic names of undefined symbols (i.e. required functions and variables) do not need to be included in the ELF file produced by GITAR. (ii) Using the SELF file format offers a higher ELF object file reduction in GITAR compared to standard Contiki because GITAR ELF object files contain more relocation entries that can be optimized. Conversely, LZ77 compression offers higher reduction in standard Contiki because Contiki ELF object files contain more trailing zeros in the symbol table. (iii) Using a general purpose compression algorithm achieves a higher energy overhead reduction compared to SELF files without sacrificing compliance with standard ELF file format. The drawback is that the update latency will increase due to the additional decompression step.

In conclusion, the method used in GITAR to transform static components into dynamic components does not have a negative impact on the energy used during deployment. Moreover, the energy cost can be further reduced by applying a different ELF file format (SELF) or using a general (lightweight) compression algorithm like LZ77.

7. Conclusion

Updating the network stack of constrained devices after deployment is essential for the long-term sustainability and maintainability of IoT networks; especially when these networks need to support future standards and adapt to new requirements. This paper proposed a generic architecture that incorporates fundamental design concepts for enabling partial code updates in the network level. The architecture can be applied on existing IoT operating systems (e.g. Contiki, TinyOS) without requiring major source code modifications. As a proof-of-concept, the architecture was applied on Contiki thereby enabling dynamic updates of single Rime modules. The solutions were implemented and validated on multiple embedded hardware platforms.

The combination of the design concepts proposed in GITAR is essential for providing such level of re-programmability. Moreover, they also ensure compatibility with standard tools and existing source code while being more efficient with respect to the current state-of-the-art as illustrated in the evaluation. Compared to standard Contiki, 13% less ROM is required for allowing

updates in a much larger portion of the firmware. Using an overhead analysis performed on RemoWare and SOS, two comparable architectures, it was also demonstrated that GITAR consumes 3% and 14% less ROM and 1% and 7% less RAM compared to RemoWare and SOS. In addition, mathematical formulas are provided to calculate the exact overhead of each of these architectures for other evaluation scenarios.

By extending the build-system with a two-phase compilation process it was possible to apply most design concepts on Contiki without requiring source code modifications. The build-system is currently able to automatically transforms static Rime primitives in dynamic components without producing larger ELF object files. Because of this, the deployment overhead of dynamic components will be similar to their static counterparts. In future work, the build-system should be further extended and integrated with more generic management tools. Future versions should also integrate a refactored Contiki netstack component which allows to dynamically add and update protocol layers further extending the level of upgrade-ability.

Acknowledgements

This work was was partially supported by the Fund for Scientific Research-Flanders (FWO-V), project "ASN: wireless Acoustic Sensor Networks", grant #G.0763.12, and the agency for Innovation by Science and Technology Flanders (IWT-V), project "SAMURAI: Software Architecture and Modules for Unified RAdIo control", and the iMinds IoT Strategic Research program, and European Commission Horizon 2020 Programme under grant agreement n645274 (WiSHFUL).

Appendix A. Kernel level evaluation constants and parameters

This appendix lists the exact parameter values that can be used for calculating the kernel overhead for the different architectures when using the formulas derived in Section 6.3. The parameters n_x vary for each example (Collect, Mesh, Collect-Mesh) because in each example a different (subset) of dynamic components are used. For each example they are listed in Table A.3. Since all our examples use the same system level functionality as base the parameters n_{SysCmp} and $n_{SysFnct}$ are identical in each example.

Table A.2: Defines the constants B_x^{MEM} used to determine the kernel level memory overhead in GITAR, RemoWare and SOS.

GITAR			
Symbol	Size (bytes)		Source
B_{SysCmp}^{ROM}	6	2 2 2	HIL object ref HIL object UID HIL object function array
$B_{SysFnct}^{ROM}$	2	2	HIL object function array entry
B_{DynCmp}^{RAM}	6	2 4	Dynamic component object ref Dynamic component user list
$B_{DynFnct}^{RAM}$	0	0	*stored in component
RemoWare			
Symbol	Size (bytes)		Source
B_{SysCmp}^{ROM}	0	0	*no system component abstraction
$B_{SysFnct}^{ROM}$	8	2 2 4	DLT system function pointer DLT system function ID DLT system function wrapper
B_{DynCmp}^{RAM}	2	2	Dynamic component reference
$B_{DynFnct}^{RAM}$	4	2 2	DIT dynamic function pointer DIT dynamic function ID
SOS			
Symbol	Size (bytes)		Source
B_{SysCmp}^{ROM}	18	2 16	SOS system module header ref SOS system module header
$B_{SysFnct}^{ROM}$	6	2 4	SOS system jump table entry SOS system function wrapper
B_{DynCmp}^{RAM}	2	2	SOS dynamic module header ref
$B_{DynFnct}^{RAM}$	0	0	*stored in component

Table A.3: Lists the parameters n_x required by the tested examples and used to determine the kernel level memory overhead in GITAR, RemoWare and SOS.

	mesh	collect	mesh-collect
n_{SysCmp}	32	32	32
$n_{SysFnct}$	215	215	215
n_{DynCmp}	9	6	13
$n_{DynFnct}$	36	40	70

Appendix B. Dynamic component constants and parameters

This appendix lists the exact values that can be used for calculating the overhead of the components for the different architectures when using the formulas derived in Section 6.3. The constants B_x^{ROM} and B_x^{ROM} used in the component memory model are defined in Table B.4 for each of the compared methods. In each method, a dynamic component requires a fixed amount of ROM ($B_{DynCmpInfo}^{ROM}$) to provide the kernel enough information to bind the component with other components or the system level.

In GITAR and SOS function pointers are stored in each component, hence a dynamic component requires ROM $B_{ProvDynFnct}^{ROM}$ for each provided dynamic function. In RemoWare, the function pointers are stored in the kernel (dynamic invocation table).

In GITAR, components are linked using component references. For each required dynamic component, 6 bytes in ROM ($B_{ReqSysCmp}^{ROM}$) is needed to store its information and 6 bytes in RAM ($B_{ReqSysCmp}^{ROM}$) to store the reference and an entry for the required component user list.

In contrary, RemoWare and SOS link components on a function level. For this RemoWare only requires 2 bytes in ROM ($B_{ReqDynFnct}^{ROM}$) for each required function ID, needed to retrieve the function pointer in the DIT. SOS requires 8 and 2 bytes in ROM ($B_{ReqDynFnct}^{ROM}$) and RAM ($B_{ReqDynFnct}^{RAM}$) respectively to store and use the function control blocks.

Since RemoWare and SOS link dynamic components directly to the system level no overhead is required for using a system function. In GITAR, dynamic components also use indirect interactions to call a system function. For this only 2 bytes in ROM ($B_{ReqSysFnct}^{ROM}$) and RAM ($B_{ReqSysFnct}^{MEM}$) are required.

Table B.4: Defines the constants B_x^{MEM} used to determine the RAM and ROM size of dynamic components in GITAR, RemoWare and SOS.

	GITAR		RemoWare		SOS	
Constant (Bytes)	ROM	RAM	ROM	RAM	ROM	RAM
$B^{MEM}_{DynCmpInfo}$	8	0	8	0	16	0
$B^{MEM}_{ProvDynFnct}$	2	0	0	0	8	0
$B^{MEM}_{ReqDynCmp}$	6	6	0	0	0	0
$B^{MEM}_{ReqDynFnct}$	0	0	2	0	8	2
$B^{MEM}_{ReqSysCmp}$	2	2	0	0	0	0
$B^{MEM}_{ReqSysFnct}$	0	0	0	0	0	0
$B^{ROM}_{DynCall}$	n.a.	n.a.	8	0	8	0
$B^{ROM}_{DynParam}$	n.a.	n.a.	4	0	0	0
$B^{ROM}_{StubCall}$	12	0	n.a.	n.a.	n.a.	n.a.
$B^{ROM}_{InlineStubCall}$	8	0	n.a.	n.a.	n.a.	n.a.
$B^{ROM}_{SysCall}$	n.a.	n.a.	0	0	0	0

In RemoWare and SOS, 8 extra ROM bytes ($B^{ROM}_{DynCall}$) are required for making a dynamic call. RemoWare also requires an extra 4 bytes in ROM ($B^{ROM}_{DynParam}$) per parameter (or return value) required by the called function (e.g. parameters need to be passed twice). No overhead is required in ROM for the system functions.

Since in GITAR stub functions are used, different constants are required. For each stub function 12 extra bytes in ROM ($B^{ROM}_{StubCall}$) are required. If the stub function can be inlined (e.g. only one call in source code), only 8 ROM bytes ($B^{ROM}_{InlineStubCall}$) are required.

The parameters n_x are different for each dynamic component and mainly depends on the number of dependencies and interactions with other components or the system level. They are defined in Table B.5.

Table B.5: Defines the parameters n_x for each Rime primitive used to determine the RAM and ROM size of each dynamic component in GITAR, RemoWare and SOS.

Binding and linking overhead					
Component	$n_{ProvFnct}$	$n_{ReqSysCmp}$	$n_{ReqSysFnct}$	$n_{ReqDynCmp}$	$n_{ReqDynFnct}$
broadcast	3	4	8	0	0
unicast	3	3	5	1	3
ipolite	4	4	11	1	3
netflood	3	3	11	1	4
multihop	4	3	9	1	3
mesh	4	2	5	3	10
route	10	4	13	0	0
route-discovery	3	3	8	3	8
collect	8	10	36	3	17
collect-neighbor	19	5	15	1	5
collect-link-estimate	6	1	0	0	0
example-mesh	0	1	1	1	3
example-collect	0	5	9	1	4
Dynamic function call overhead					
Component	$n_{SysCall}$	$n_{DynCall}$	$n_{DynCallParam}$	$n_{StubCall}$	$n_{InlineStubCall}$
broadcast	7	0	0	0	7
unicast	4	3	5	0	7
ipolite	19	4	6	4	10
netflood	17	5	12	5	9
multihop	15	5	10	6	5
mesh	7	13	22	3	12
route	26	0	0	7	6
route-discovery	22	12	29	9	6
collect	138	33	56	27	24
collect-neighbor	35	8	10	9	11
collect-link-estimate	0	0	0	0	0
example-mesh	2	4	7	2	2
example-collect	15	4	12	3	10

Appendix C. Energy usage analysis input data

This appendix lists the exact file size for each dynamic (static) component in GITAR (Contiki) after compilation to standard ELF object files, after tranforming the standard ELF object files to SELF object files and after compressing the standard ELF files using LZ77. The results are summarized in Table C.6. The energy required to receive each ELF object file (with or without SELF/LZ77 compression) is summarized in Table C.7. The values listed in the table were obtained by multiplying the number of bytes per file with the energy required to receive one byte (0.0048 mJ) using the CC2420 transceiver (used on both Zolertia Z1 as TMote Sky).

Table C.6: Comparison of the file sizes in bytes for each dynamic component in GITAR and their static counterparts in Contiki. The size of the uncompressed ELF object file is listed first, then the size after applying the SELF file format reduction and finally the size using LZ77 compression on the standard ELF file. The first (second) *total* row gives the total number of bytes required to receive when adding collect (mesh) after deployment.

Component	GITAR			Contiki		
	ELF	SELF	LZ77	ELF	SELF	LZ77
collect	9280	8592	5737	9780	9479	5706
collect-link-estimate	1488	894	875	1200	820	725
collect-neighbor	3960	3230	2299	3788	3546	2069
example-collect	2204	1661	1343	2236	1838	1251
total	16932	14377	10254	17004	15683	9751
multihop	2008	1348	1238	1988	1503	1149
ipolite	2144	1456	1379	2176	1691	1297
netflood	2024	1353	1300	2120	1619	1261
route	2708	2062	1653	2496	2177	1481
route-discovery	2540	1843	1616	2492	2002	1491
mesh	2040	1320	1331	2132	1618	1272
example-mesh	1704	1100	1025	1568	1080	934
total	15168	10482	9542	14972	11690	8885

Table C.7: Comparison of the energy in mJ needed to receive each component over a single link without packet loss and procotol overhead. Again, the uncompressed ELF file, the SELF file and LZ77 file are considered, this both for the dynamic modules (GITAR) and their static counterparts (Contiki). The first (second) *total* row gives the energy required for adding collect (mesh) after deployment.

Component	GITAR			Contiki		
	ELF	SELF	LZ77	ELF	SELF	LZ77
collect	44.544	41.2416	27.70869	46.944	45.4992	27.40371
collect-link-estimate	7.1424	4.2912	4.230193	5.76	3.936	3.484526
collect-neighbor	19.008	15.504	11.10565	18.1824	17.0208	9.938921
example-collect	10.5792	7.9728	6.490012	10.7328	8.8224	6.009859
total	81.2736	69.0096	49.53455	81.6192	75.2784	46.83702
multihop	9.6384	6.4704	5.982657	9.5424	7.2144	5.519726
ipolite	10.2912	6.9888	6.662812	10.4448	8.1168	6.230393
netflood	9.7152	6.4944	6.280257	10.176	7.7712	6.060255
route	12.9984	9.8976	7.984721	11.9808	10.4496	7.11306
route-discovery	12.192	8.8464	7.807121	11.9616	9.6096	7.162391
mesh	9.792	6.336	6.432412	10.2336	7.7664	6.111458
example-mesh	8.1792	5.28	4.953548	7.5264	5.184	4.486395
total	72.8064	50.3136	46.10353	71.8656	56.112	42.68368

References

[1] L. Atzori, A. Iera, G. Morabito, The internet of things: A survey, Computer Networks 54 (15) (2010) 2787 – 2805. doi:http://dx.doi.org/10.1016/j.comnet.2010.05.010.

[2] E. De Poorter, I. Moerman, P. Demeester, Enabling direct connectivity between heterogeneous objects in the internet of things through a network-service-oriented architecture, EURASIP Journal on Wireless Communications and Networking 2011 (1) (2011) 1–14. doi:10.1186/1687-1499-2011-61.

[3] C. Fortuna, M. Mohorcic, A framework for dynamic composition of communication services, ACM Transactions on Sensor Networks 11 (2) (2015) to appear. doi:http://dx.doi.org/10.1145/2700268.

[4] M. Mueck, A. Piipponen, K. Kalliojarvi, G. Dimitrakopoulos, K. Tsagkaris, P. Demestichas, F. Casadevall, J. Perez Romero, O. Sallent, G. Baldini, S. Filin, H. Harada, M. Debbah, T. Haustein, J. Gebert, B. Deschamps, P. Bender, M. Street, S. Kandeepan, J. Lota, A. Hayar, ETSI reconfigurable radio systems : Status and future directions on software defined radio and cognitive radio standards, IEEE Communications Magazine 48 (9). doi:http://dx.doi.org/10.1109/MCOM.2010.5560591.

[5] S. Farahani, ZigBee Wireless Networks and Transceivers, Newnes, Newton, MA, USA, 2008.

[6] J. Song, S. Han, A. Mok, D. Chen, M. Lucas, M. Nixon, Wirelesshart: Applying wireless technology in real-time industrial process control, in: Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE, 2008, pp. 377–386. doi:10.1109/RTAS.2008.15.

[7] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, D. Culler, Tinyos: An operating system for sensor networks, in: W. Weber, J. Rabaey, E. Aarts (Eds.), Ambient Intelligence, Springer Berlin Heidelberg, 2005, pp. 115–148. doi:10.1007/3-540-27139-2_7.

[8] A. Dunkels, B. Gronvall, T. Voigt, Contiki - a lightweight and flexible operating system for tiny networked sensors, in: Local Computer Networks, 2004. 29th Annual IEEE International Conference on, 2004, pp. 455–462. doi:10.1109/LCN.2004.38.

[9] C.-C. Han, R. Kumar, R. Shea, E. Kohler, M. Srivastava, A dynamic operating system for sensor nodes, in: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services, MobiSys '05, ACM, New York, NY, USA, 2005, pp. 163–176. doi:10.1145/1067170.1067188.

[10] S. Brown, C. J. Sreenan, Software updating in wireless sensor networks: A survey and lacunae, Journal of Sensor and Actuator Networks 2 (4) (2013) 717–760. doi:10.3390/jsan2040717.

[11] Micropython, python for microcontrollers, accessed: 2014-12-23 (2014). URL http://micropython.org/

[12] Mpy: Microcontroller python, accessed: 2014-12-23 (2014). URL http://www.mpyprojects.com/

[13] Q. Cao, T. Abdelzaher, J. Stankovic, T. He, The liteos operating system: Towards unix-like abstractions for wireless sensor networks, in: Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on, 2008, pp. 233–244. doi:10.1109/IPSN.2008.54.

[14] P. Levis, D. Culler, Maté: A tiny virtual machine for sensor networks, in: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS X, ACM, New York, NY, USA, 2002, pp. 85–95. doi:10.1145/605397.605407.

[15] N. Brouwers, K. Langendoen, P. Corke, Darjeeling, a feature-rich vm for the resource poor, in: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys '09, ACM, New York, NY, USA, 2009, pp. 169–182. doi:10.1145/1644038.1644056.

[16] Y. Yu, L. J. Rittle, V. Bhandari, J. B. LeBrun, Supporting concurrent applications in wireless sensor networks, in: Proceedings of

the 4th International Conference on Embedded Networked Sensor Systems, SenSys '06, ACM, New York, NY, USA, 2006, pp. 139–152. doi:10.1145/1182807.1182822.

[17] R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis, M. Srivastava, Multi-level software reconfiguration for sensor networks, in: Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software, EMSOFT '06, ACM, New York, NY, USA, 2006, pp. 112–121. doi:10.1145/1176887.1176904.

[18] J. Koshy, R. Pandey, Vmstar: Synthesizing scalable runtime environments for sensor networks, in: Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems, SenSys '05, ACM, New York, NY, USA, 2005, pp. 243–254. doi:10.1145/1098918.1098945.

[19] S. Kulkarni, L. Wang, Mnp: Multihop network reprogramming service for sensor networks, in: Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on, 2005, pp. 7–16. doi:10.1109/ICDCS.2005.50.

[20] J. W. Hui, 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, SenSys '04, ACM, New York, NY, USA, 2004, pp. 81–94. doi:10.1145/1031495.1031506.

[21] N. Reijers, K. Langendoen, Efficient code distribution in wireless sensor networks, in: Proceedings of the 2Nd ACM International Conference on Wireless Sensor Networks and Applications, WSNA '03, ACM, New York, NY, USA, 2003, pp. 60–67. doi:10.1145/941350.941359.

[22] J. Jeong, D. Culler, Incremental network programming for wireless sensors, in: Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on, 2004, pp. 25–33. doi:10.1109/SAHCN.2004.1381899.

[23] P. von Rickenbach, R. Wattenhofer, Decoding code on a sensor node, in: S. Nikoletseas, B. Chlebus, D. Johnson, B. Krishnamachari (Eds.), Distributed Computing in Sensor Systems, Vol. 5067 of Lecture Notes

in Computer Science, Springer Berlin Heidelberg, 2008, pp. 400–414. doi:10.1007/978-3-540-69170-9_27.

[24] Z. Yang, M. Li, W. Lou, R-code: Network coding based reliable broadcast in wireless mesh networks with unreliable links, in: Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE, 2009, pp. 1–6. doi:10.1109/GLOCOM.2009.5426175.

[25] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, K. Rothermel, Flexcup: A flexible and efficient code update mechanism for sensor networks, in: K. Rmer, H. Karl, F. Mattern (Eds.), Wireless Sensor Networks, Vol. 3868 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 212–227. doi:10.1007/11669463_17.

[26] Executable and linkable format (elf), tool Interface Standards Committee and others (2001).

[27] A. Dunkels, N. Finne, J. Eriksson, T. Voigt, Run-time dynamic linking for reprogramming wireless sensor networks, in: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys '06, ACM, New York, NY, USA, 2006, pp. 15–28. doi:10.1145/1182807.1182810.

[28] W. Munawar, M. Alizai, O. Landsiedel, K. Wehrle, Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks, in: Communications (ICC), 2010 IEEE International Conference on, 2010, pp. 1–6. doi:10.1109/ICC.2010.5501964.

[29] L. Mottola, G. P. Picco, A. A. Sheikh, Figaro: Fine-grained software reconfiguration for wireless sensor networks, in: Proceedings of the 5th European Conference on Wireless Sensor Networks, EWSN'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 286–304. doi:10.1007/978-3-540-77690-1_18.

[30] D. Hughes, E. Caete, W. Daniels, R. Gowri Sankar, J. Meneghello, N. Matthys, J. Maerien, S. Michiels, C. Huygens, W. Joosen, M. Wijnants, W. Lamotte, E. Hulsmans, B. Lannoo, I. Moerman, Energy aware software evolution for wireless sensor networks, in: World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE

14th International Symposium and Workshops on a, 2013, pp. 1–9. doi:10.1109/WoWMoM.2013.6583386.

[31] A. Taherkordi, F. Loiret, A. Abdolrazaghi, R. Rouvoy, Q. Le-Trung, F. Eliassen, Programming sensor networks using remora component model, in: Proceedings of the 6th IEEE International Conference on Distributed Computing in Sensor Systems, DCOSS'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 45–62. doi:10.1007/978-3-642-13651-1_4.

[32] H.-P. Chang, Y.-C. Lin, D.-W. Chang, An online reprogrammable operating system for wireless sensor networks., J. Inf. Sci. Eng. 27 (1) (2011) 261–286.

[33] Y.-T. Chen, T.-C. Chien, 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, SenSys '10, ACM, New York, NY, USA, 2010, pp. 183–196. doi:10.1145/1869983.1870002.

[34] A. Taherkordi, F. Loiret, R. Rouvoy, F. Eliassen, Optimizing sensor network reprogramming via in situ reconfigurable components, ACM Trans. Sen. Netw. 9 (2) (2013) 14:1–14:33. doi:10.1145/2422966.2422971.

[35] B. Porter, U. Roedig, G. Coulson, Type-safe updating for modular wsn software, in: Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on, 2011, pp. 1–8. doi:10.1109/DCOSS.2011.5982140.

[36] A. Dunkels, F. Österlind, Z. He, An adaptive communication architecture for wireless sensor networks, in: Proceedings of the 5th International Conference on Embedded Networked Sensor Systems, SenSys '07, ACM, New York, NY, USA, 2007, pp. 335–349. doi:10.1145/1322263.1322295.

[37] W. Dong, C. Chen, J. Bu, C. Huang, Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Networks 11 (1) (2013) 473 – 489. doi:http://dx.doi.org/10.1016/j.adhoc.2012.07.007.

URL http://www.sciencedirect.com/science/article/pii/S1570870512001424

[38] N. Tsiftes, A. Dunkels, T. Voigt, Efficient sensor network reprogramming through compression of executable modules, in: Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on, 2008, pp. 359–367. doi:10.1109/SAHCN.2008.51.

[39] M. Stolikj, P. Cuijpers, J. Lukkien, Efficient reprogramming of wireless sensor networks using incremental updates, in: Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on, 2013, pp. 584–589. doi:10.1109/PerComW.2013.6529563.

Peter Ruckebusch received his B.S. and M.S. in Computer Science from Hogeschool Ghent faculty engineering, Belgium. Peter has many years of experience in developing network solutions of networked embedded systems, especially wireless senor networks. Since 2011 he has been a Ph.D. student at University Ghent, iMinds in the department of Information Technology (INTEC). He has been/is collaborating in several national and European projects. His research topics are situated in the low-end of the IoT mainly focussing on re-configurability and re-programmability aspects of protocol stacks for constrained devices in IoT networks.

Eli De Poorter is a postdoctoral researcher at Ghent University. He received his Master degree in Computer Science Engineering from Ghent University, Belgium, in 2006. He received his Ph.D. degree in 2011 at the Department of Information Technology at Ghent University through a Ph.D. scholarship from the Institute for Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). After obtaining his Ph.D., he received a FWO postdoctoral research grant and is now a post-doctoral fellow at the same research group, where he is currently involved in and/or research coordinator of several national and international projects. His main research interests include wireless network protocols, network architectures, wireless sensor and ad hoc networks, future internet, self learning networks and next-generation network architectures. He is part of the program committee of several conferences and is the author or co-author of more than 50 papers published in international journals or in the proceedings of international conferences. He is also the creator of the patented IDRA architecture (http://idraproject.net), a flexible communication framework for heterogeneous networked devices.

Carolina Fortuna is currently a post-doctorate researcher at the University of Gent, Belgium. She received her PhD in 2013 from the Jozef Stefan International Postgraduate School with the thesis entitled “Dynamic Composition of Communication Services”. Carolina is an expert in software architectures and stacks for embedded systems and also has experience with machine learning and data mining applied to networking problems. Carolina has actively written and collaborated on over 6 successful H2020, FP7 and FP6 research projects. Carolina also undertook two industrial internships at Bloomberg LP and Siemens PSE. The system she helped developing at Bloomberg was the most competitive in a US wide challenge and was subsequently selected for implementation on their web site.

Ingrid Moerman received her degree in Electrical Engineering (1987) and the Ph.D. degree (1992) from the Ghent University, where she became a part-time professor in 2000. She is a staff member of the research group on Internet-Based Communication Networks and Services, IBCN (www.ibcn.intec.ugent.be), where she is leading the research on mobile and wireless communication networks. Since 2006 she joined iMinds, where she is coordinating several interdisciplinary research projects. Her main research interests include: Sensor Networks, Cooperative and Cognitive Networks, Wireless Access, Self-Organizing Distributed Networks (Internet of Things) and Experimentally-supported research. Ingrid Moerman has a longstanding experience in running national and EU research funded projects. At the European level, Ingrid Moerman is in particular very active in the FP7 FIRE (Future Internet Research and Experimentation) research area, where she is coordinating the CREW project and further participating in IP OpenLab, IP Fed4FIRE, STREP SPITFIRE, STREP EVARILOS, STREP FORGE and IP FLEX. In the FP7 research area on Future Networks, she is involved in IP LEXNET and STREP SEMAFOUR. Ingrid Moerman is author or co-author of more than 500 publications in international journals or conference proceedings. She is associate editor of the EURASIP Journal on Wireless Communications and Networking and vice-president of the expert panel on Informatics and Knowledge Technology of the Research Foundation Flanders (FWO).

Eli De Poorter Photo

Carolina Fortuna Photo

Peter Ruckebusch Photo

Ingrid Moerman Photo