An Operating System Runtime Reprogramming Infrastructure for WSN

Rodrigo Steiner, Giovani Gracioli, Rita de Cássia Cazu Soldi, and Antônio Augusto Fröhlich
Software/Hardware Integration Lab
Federal University of Santa Catarina
88040-900 - Florianópolis, SC, Brazil
{rodrigo,giovani,rita,guto}@lisha.ufsc.br

***Abstract*—WSNs, despite of its limited resources, are expected to operate without human interventions for a long period of time. Nevertheless, the environment might develop unpredicted characteristics or some network functionality might need some changes. Thus, it is necessary a mechanism which allows software reprogramming of the network nodes after its deployment. This paper presents the integration of a data dissemination protocol and ELUS, an OS support environment. The dissemination protocol is responsible for spreading the data across the network, while ELUS isolates the system components in memory position independent units, allowing their updating at execution time. We have evaluated our infrastructure, using real sensor nodes, in terms of memory consumption, dissemination, and reprogramming time.**

***Keywords*-Software reprogramming, data dissemination protocol, reconfiguration**

I. Introduction

A software reprogramming infrastructure for a Wireless Sensor Network (WSN) is composed of a data dissemination protocol and a structure capable of organizing the data in the system's memory. By using an embedded Operating System (OS) it is possible to provide for embedded applications an infrastructure to hide this data organization. Usually, the reprogramming structures present in OSs are composed of updatable modules. These modules are memory position independent and are replaced at runtime [1] [2].

In addition, it is essential that all new data of one or more modules is correctly received by all nodes involved in the reprogramming process. In order to provide safe data transfer, a data dissemination protocol is used together with the OS infrastructure.

Epos Live Update System (ELUS) is an OS infrastructure for software updating that has better performance in terms of memory consumption, method invocation time, and reconfiguration time when compared to related works [3]. Although this favorable result, ELUS memory consumption could still be improved. Furthermore, ELUS does not have any support for data dissemination.

In this paper we make the following contributions: (i) we develop and integrate a data dissemination protocol to ELUS; (ii) we improve the memory consumption of ELUS by using C++ templates specialization techniques; and (iii) we evaluate the new infrastructure in terms of memory consumption, and dissemination and reprogramming times.

Section II presents the design of the developed dissemination protocol. Section III presents the integration between the dissemination protocol and ELUS. The evaluation of the infrastructure is carried out in Section IV. Finally, Section V concludes the paper.

II. Data Dissemination Protocol

In general the network reprogramming process is divided in three steps, as shown in Figure 1. The first step is responsible for preparing the data to be disseminated. The second step encompasses the whole dissemination process. Finally, the OS reconfiguration mechanism interprets the received data and uses it to update the program memory.

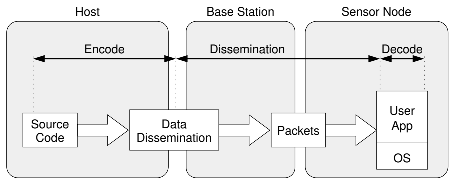

Figure 1. Network reprogramming process.

Data dissemination protocols are used to spread data over the network using its own nodes. In particular, these protocols must ensure accurate delivery of all data to all nodes. When designing a dissemination protocol some properties must be taken into account: low latency, low memory consumption, reliability, energy efficiency, tolerance to nodes insertion/removal, and uniformity [4].

Reliability and **uniformity** are mandatory properties, as they guarantee the correct functioning of the protocol. All other properties are only desirable. However, a protocol that ignores them would be of little use [5].

A. Design Choices

As some desirable properties come into conflict with others, existing protocols make design choices giving preference to some over others. The choices made in the protocol developed in this work are:

- Energy efficiency was the non-mandatory property considered most important, since all operation require energy, and in many embedded systems there is only a finite amount available.

- Memory consumption was considered the second most important property, since the dissemination protocol is not the main purpose of the node, but only a service offered by the OS. Thus, one should not limit the amount of memory available for applications.
- Finally, the latency. In order to optimize energy and memory consumption some characteristics that would decrease latency were not used (e.g. pipelining).

B. Implementation

Figure 2 shows the state machine of the developed protocol. It disseminates data in a neighborhood-by-neighborhood fashion, uses a sender selection mechanism based on publish/subscribe, use receptors for packet loss detection, performs unicast requests, broadcast retransmissions, and uses the sliding window mechanism for segments management.

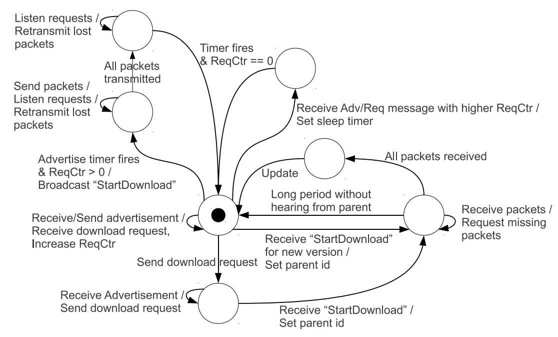
Listen requests / Retransmit lost packets
Timer fires & ReqCtr == 0
All packets transmitted
Send packets / Listen requests / Retransmit lost packets
Receive Adv/Req message with higher ReqCtr / Set sleep timer
All packets received
Advertise timer fires & ReqCtr > 0 / Broadcast "StartDownload"
Update
Long period without hearing from parent
Receive/Send advertisement / Receive download request, Increase ReqCtr
Receive packets / Request missing packets
Receive "StartDownload" for new version / Set parent id
Send download request
Receive Advertisement / Send download request
Receive "StartDownload" / Set parent id

Figure 2. Developed protocol state machine.

Nodes publish their versions periodically, and all interested parties request it. A potential sender maintains a variable *ReqCtr*, initialized to zero, and increments it for each new received request (intended for him) coming from a node not yet computed. Publish messages have two functions: to announce a new version, and prevent nodes with fewer requests of becoming senders. Publish messages have the version number, the nodes *id* and its *ReqCtr*. When a node receives a publish message that contains a new version, it will send a request message containing both the sender and its own *id*, and the value of *ReqCtr* received. As both publish and subscribe messages are broadcasted other nodes that are in contention to become a sender will also receive it. If a node receives a message and it has a lower *ReqCtr* it will go to sleep. The nodes *id* is used as a tie-breaker.

When a node becomes a sender it sends a *"StartDownload"* message and starts to send the data, packet by packet. Receivers define this node as its 'parent' and only accept packets from him. Each packet has a sequential unique identifier, and receivers maintain the number of the last received packet. Thus, upon receiving a new packet it checks if there is a gap between those numbers. When a loss is detected the receiver sends a unicast retransmission request to the sender. Requests for retransmission have a higher priority than normal packets, then a sender will first respond to all requests before continuing with the transmission.

III. Integration to the Operating System

A. ELUS *Overview*

Figure 3 shows the ELUS framework for software reprogramming. The PROXY and AGENT elements create an indirection level between the method invocations from the application and the OS. Thus, the AGENT is the only member in the system aware of the components' memory location. Consequently, the AGENT is able to locate and to update the code and data of the system components. Moreover, system components can be marked as reconfigurable or not at compile-time by setting a boolean value in its *Trait* class (*Traits<Component>::reconfiguration = true*). For those components that are not marked as reconfigurable, no overhead in terms of memory and processing is added in the final system image. The ADAPTER class is responsible for applying the aspects supplied by SCENARIO before and after the real method call [6].

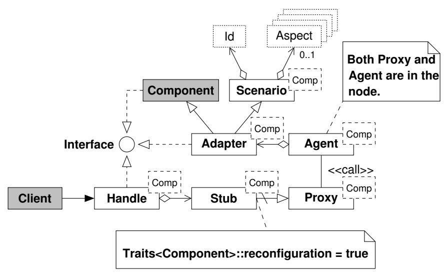

Figure 3. ELUS framework for software reprogramming.

B. Memory Optimization

Each component marked as reconfigurable generates an overhead in terms of memory consumption due to the ELUS metaprogrammed framework. In addition to the real component methods, it is included the code of the framework's methods. Moreover, the code for each framework method is replicated for each reconfigurable component. For example, the method `update`, responsible for updating the code and data of a component, is replicated for each component.

By using a template specialization technique it is possible to overcome this limitation. We observed that several methods from AGENT and SCENARIO classes can be replaced by template classes that receive a parameter with no type (*void*) [7]. With this simple technique we reduce the ELUS memory consumption without losing performance in terms of method invocation and reconfiguration times, as will be shown in Section IV.

C. Data Dissemination Protocol Integration

ELUS receives a reprogramming request for a component through a protocol, called ELUS TRANSPORT PROTOCOL

(ETP). This protocol defines the format of updating messages. A thread, created during the system's initialization, named RECONFIGURATOR, creates a data dissemination protocol instance and after receiving the new data, starts the reprogramming process. It is possible to add or remove a method, update the entire component (all methods), update the application, add attributes, and update a specific address. In case of updating/adding attributes values, the data state transfer between the new and old component's attributes is done by the *set* and *get* methods of each attribute.

This message structure allows an easy integration of a data dissemination protocol and the OS. The data dissemination protocol mounts messages in the ETP format and calls the AGENT informing an updating. This simple message structure is not found in the related works, but it is essential to abstract and provide a simple updating process for developers. Figure 4 shows the UML sequence diagram for the updating process. The RECONFIGURATOR starts the protocol by calling the method *run*. This method stays blocked until a new version is received by the node. After receiving the new data, the RECONFIGURATOR creates a message in the ETP format and passes the data to AGENT (*trapAgent*). The AGENT writes the new component code into the appropriate location and updates all tables, if necessary.

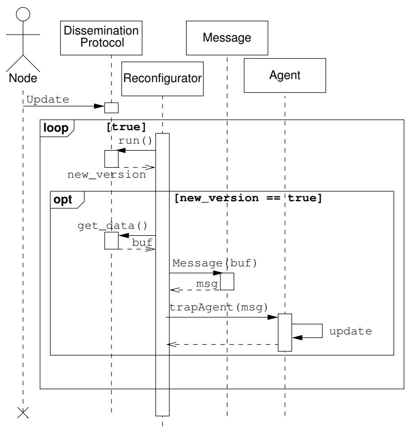

Figure 4. Integration between the data dissemination protocol and ELUS.

IV. EVALUATION

The infrastructure was evaluated in terms of memory consumption, latency of sending data across the network and reconfiguration time. These tests were performed using Mica2 nodes. The system was generated with the GNU compiler g++ 4.0.2 and memory consumption measured using the GNU *objdump* 2.16.1 tool. Latency and reconfiguration times were measured by the timer of the microcontroller.

A. Memory

Table I shows the memory consumption of all the framework's elements. For this test, the reconfiguration support

Table I
MEMORY CONSUMPTION OF THE STRUCTURE: ELUS AND DISSEMINATION PROTOCOL.

Structure Elements	Section size (bytes)			
	.text	**.data**	**.bss**	**.bootloader**
Dissemination Protocol	2536	0	21	0
RECONFIGURATOR	166	0	4	0
AVR Code Manager	36	0	0	416
ELUS	1648	26	68	0
Total	4386	26	93	416

Table II
MEMORY CONSUMPTION OF ELUS TO ENABLE SUPPORT FOR A RECONFIGURATION COMPONENT.

Framework Methods	Section Size (bytes)	
	.text	**.data**
Create	178	0
Destroy	136	0
Method without parameter and return value	90	0
Method with parameter and without return value	94	0
Method without parameter and with return value	104	0
Method with parameter and return value	122	0
Update	260	0
Dispatcher	0	2 X (n. of methods)
Semaphore	0	18
Minimum size	664	26

was enabled for a component that contains 4 methods. The dissemination protocol occupies 2536 bytes in code area and 21 bytes in the uninitialized data. It is important to notice that a buffer is created dynamically when the new code is received, and its size depends on the size of the update. The ELUS framework consumes 1648 bytes of code, 26 bytes of data and 68 bytes of uninitialized data due to the variables and tables required to store objects and methods.

Table II shows the memory consumption required by ELUS when a novel component is added to the system. The methods `Create`, `Destroy` and `Update` represent the constructor, destructor and update method for the component, and must always be present. Each component also requires a semaphore to control its exclusive access and prevent an update while the code is running. The minimum consumption to a new component added to the framework is composed of the constructor, destructor, `Update` method and a method without parameters and without returning value. Equation 1 summarizes the extra cost of memory.

$$Size_c = \sum_{i=1}^{n}(Method_i) + Create + Destroy + Update \quad (1)$$

Using the specialization through void pointers it was possible to reduce consumption of approximately 1.2KB (more than 50%). Currently the minimum size for a new component of the system is 664KB instead of 1.6KB from the previous ELUS implementation [3].

B. Latency

To measure the latency we used two topologies: (i) the base station can communicate with all nodes, and (ii) there are nodes outside the base station reach. In both topologies we repeated the dissemination process twenty times, in order to update a method of a system component, propagating 10 bytes of data (used in the update method) and 6 bytes of control information (used by the protocol).

Figure 5 shows the average time that the base station takes to propagate data to the nodes around it. We observed a standard deviation of 0.0233 seconds. This time did not change by changing the number of receivers between one and three. This happens because loss packet is highly correlated, so multiple receivers lose the same set of packets [8].

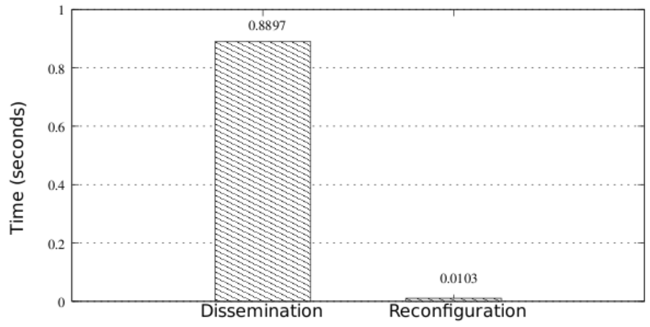

Figure 5. Dissemination and reconfiguration time.

Figure 6 shows the average time it takes to propagate the data from the base station to the intermediate nodes, and from these to the node out of range of the base station. It is possible to notice that the time required to propagate data between normal network nodes is approximately four times greater than the time spent by the base station. This is due to the fact that the base station does not perform the step of selecting a sender. This way it wastes no time publishing its version and receiving requests to finally become a sender and begin to disseminate the data. The time of intermediate nodes had a standard deviation of 1.1288 seconds.

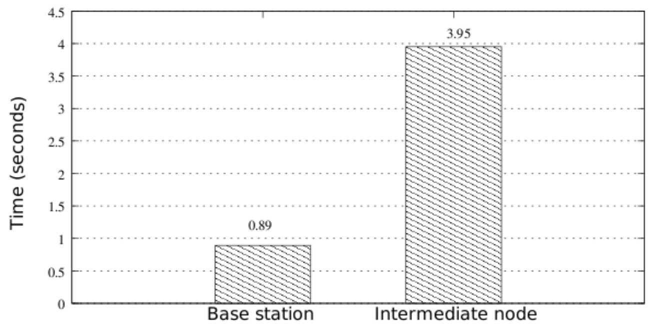

Figure 6. Dissemination time.

C. Reconfiguration Time

The reconfiguration time encompasses the time a node takes to update its program memory after receiving all the necessary data. In the proposed structure this time comprises: the call of RECONFIGURATOR, methods `p` and `v` of the semaphore, the call to the `Update`, the recovery of arguments passed on the ETP message, the recovery of the object to be updated in a hash table, find the address of the `vtable`, and writing data in flash. Figure 5 shows the average reconfiguration time obtained.

One feature of the Mica2 platform is that it is not possible to change only one byte at a time in its flash memory. It allows only writing in pages (of 256 bytes), and before rewriting a page it is necessary to delete its contents. Thus, in order to update a portion of memory, it is necessary to read the page content, store it in a temporary buffer, modify only the desired part of it, and finally write in the flash.

V. CONCLUSION

This paper presented an OS runtime reprogramming infrastructure for WSN which consists of a data dissemination protocol and ELUS. The infrastructure was tested in a WSN with real nodes, Mica2, and evaluated in terms of memory consumption, and dissemination and reprogramming time. Our results corroborate our design choices, with figures better than that present in related work. We have also reduced ELUS memory consumption by more than 50% by using C++ templates specialization techniques.

REFERENCES

[1] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, “A dynamic operating system for sensor nodes,” in *MobiSys*. New York, NY, USA: ACM Press, 2005.

[2] A. Dunkels, B. Grönvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *EmNetS-I*, Tampa, Florida, USA, Nov. 2004.

[3] G. Gracioli and A. Fröhlich, “Elus: A dynamic software reconfiguration infrastructure for embedded systems,” in *ICT*, April 2010.

[4] P. Lanigan, R. Gandhi, and P. Narasimhan, “Disseminating Code Updates in Sensor Networks: Survey of Protocols and Security Issues,” Tech. Rep., 2005.

[5] T. Stathopoulos, J. Heidemann, and D. Estrin, “A remote code update mechanism for wireless sensor networks,” Tech. Rep., Nov. 2003.

[6] A. A. Fröhlich and W. Schröder-Preikschat, “Scenario Adapters: Efficiently Adapting Components,” in *WMSCI*, Orlando, U.S.A., Jul. 2000.

[7] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[8] S. Kulkarni and L. Wang, “Mnp: multihop network reprogramming service for sensor networks,” in *SenSys*. ACM, 2004.