Towards Updating the Framework for Reprogramming Wireless Sensor Networks

Aliaa Essameldin <aeahmed@qatar.cmu.edu>
Professor Phil Gibbons, Professor Saquib Razak
## Contents

1. **Introduction**  
   1.1 Wireless Sensor Networks  
   1.2 Real-time operating Systems  

2. **Research Gap**  
   2.1 Reprogramming Wireless Sensor Networks  
   2.2 Current State-of-the-art  
   2.3 Defining the gap  

3. **Future Work**
Introduction

The early 2000s has witnessed significant advancements in sensory and communication technologies that permitted Wireless Sensor Networks to capture the research community’s attention. A big part of this research covered the question of secure reprogramming of Wireless Sensor Network. However, the work done in this area is fragmented. The fragmentation of information is problematic in two main ways: first, it makes it challenging for new embedded system developers to make an educated choice between firmware options for their WSN and, second, it makes it almost impossible to learn from existing systems to work towards a more advanced system. These problems are growing more relevant today with the rise of the Internet of Things (IoT) paradigm. This paradigm means that more devices such as wearables, cars and house appliances, will join the internet and become the equivalence of a sensor mote in a world-wide wireless network. By revisiting these problems today, we realized that embedded systems developers are in need of a new system for reprogramming sensor nodes that builds on the early work done in the area and makes it compatible for the new IoT systems.

This report explains how we arrived at this research gap. Section 1 summarizes all the background research that has lead to finding this research gap. It starts by introducing Wireless Sensor Networks then covers the most common operating systems that are deployed on top of them. Most importantly, the section highlights the three main requirements of any Wireless Sensor Network system. Section 2 focuses more on the specific problem at hand: reprogramming wireless sensor networks. It starts off by explaining the significance of reprogramming WSNs, then briefly covers the previous work done in that area and compiles a current-state-of-the-art framework that does the task while addressing the three system requirements highlighted in section 1. Finally, section 3 lists some future steps that can be taken towards covering that gap and creating a new IoT-compatible framework for reprogramming WSNs.

1.1 Wireless Sensor Networks

The proliferation of Micro-Electro-Mechanical Systems (MEMS) technology over the past decade has facilitated the development of smart sensors. Smart sensors are composed of a micro controller, a radio communication interface and one or more mechanical, thermal, biological, chemical optical or magnetic sensors. A group of (hundreds or maybe thousands) smart sensors which communicate together and to a base
station (e.g., a laptop, a personal handheld device, or an access point to a fixed infrastructure) via wireless connections to accomplish a certain task is known as a Wireless Sensor Network (WSN). Such networks that can serve a wide variety of application domains ranging from health and education to green computing and property management.

The significance of work done in the area of WSNs has been steadily increasing. A new computing paradigm, known as the Internet of Things, predicts that the sensor nodes will move from becoming grouped in single-use WSNs into becoming a part of the Internet. Since processors, communication modules and most electronic components are diminishing in size and price, the IoT vision has come a lot closer to reality. To help bring this vision to reality, there are three main aspects of WSNs that need to be studied and improved upon:

**Durability:** The durability of a sensor node is important across all domains. In the industrial realm, a client would expect their smart gadget to last just as long as the old gadget did, especially that they paid more for it. In the scientific realm, scientists use smart nodes in locations that are difficult to reach, so they would not want the nodes to easily break since they are difficult to reach (More in that in section 1.2). Therefore, any WSN node should be designed to be as durable as possible. This means it should be 1. Fault tolerant and 2. Easily updatable.

**Feasibility:** As previously mentioned, a WSN can consist of thousands of nodes, so an increment of a single dollar in the price of a node would scale to thousands of dollars per system. Even as IoT products, it is always preferred that the price of the gadget is kept to a minimum. Minimizing the cost of a sensor node means two things: 1. Minimizing the hardware cost and 2. Minimizing energy consumption. We can minimize hardware cost by optimizing memory usage, since more memory usually means higher cost of node and designing firmware that would not need a memory control unit, which also significantly impacts price. Typically, there is a trade-off between energy and memory consumption and the final call should be application-specific.

**Scalability:** The predicted world-wide IoT will include millions of nodes world-wide. Thus, any improvements to WSNs must highly consider scalability as a desired characteristic. Namely, innovations in this area should avoid solutions that may require intensive communication (or broadcasting) or a central database management unit.

### 1.2 Real-time Operating Systems
The dramatic growth in the sensor nodes market that we have witnessed over the past decade was met with a growth in firmware and operating systems. Since these operating systems are expecting to use the sensors to collect data and immediately react to this data (either by sending it to other nodes or by using an actuator), they are known as Real-time Operating Systems. In this chapter I will introduce some of the most commonly used ones that tried to meet the three WSN requirements: durability, feasibility and scalability.

### 1.2.1 TinyOS

TinyOS is a component-based operating system written in nesC. The programming language nesC permits dividing the whole system into compartmentalized components running cooperative tasks and processes. Some of these are hardware and/or common abstractions such as packet communication, routing, sensing, actuation and storage. All these components are statically linked together using interfaces then compiled into a small binary images (Levis, Culler, & Gay). The images can run on a range of hardware platforms and has been used on several generations of sensor motes. Supported processors include the Atmel AT90L-series, Atmel ATmega-series, and Texas Instruments MSP-series processors. TinyOS includes hardware support for the RFM TR1000 and Chipcon CC1000 radios, as well as several custom radio chipsets (Levis, 2005).

TinyOS applications may be compiled to run on any of these platforms without modification.

“TinyOS is fully non-blocking: it has one call stack. Thus, all input/output (I/O) operations that last longer than a few hundred microseconds are asynchronous and have a callback. To enable the native compiler to better optimize across call boundaries, TinyOS uses nesC's features to link these callbacks, called events, statically. While being non-blocking enables TinyOS to maintain high concurrency with one stack, it forces programmers to write complex logic by stitching together many small event handlers. To support larger computations, TinyOS provides tasks, which are similar to a Deferred Procedure Call and interrupt handler bottom halves. A TinyOS component can post a task, which the OS will schedule to run later. Tasks are non-preemptive and run in first in, first out order.” - (Wikipedia, 2016)

TinyOS satisfies the durability requirement because this simple concurrency model helps it prevent most memory faults. Also, TinyOS supports a code-propagation protocol called Trickle, which permits updated code to propagate from one node to the rest of the network, making the network easily updatable (the developer does not need to access every single node). It is also relatively memory efficient since a typical image needs no more than 15Kbytes of memory (Levis, Culler, & Gay).

### 1.2.2 mBed and Arduino
mBed and Arduino are two very widely used quick-prototyping Real-time operating systems (Wikipedia, 2016). They both run on ARM processors and provide user-friendly IDEs with a hidden scheduler. They are both centered around making Smart-sensor-node technologies accessible to a wide audience with limited technical background. Thus, they mainly compete in terms of IDE and online support. The table below summarizes and compares their features.

<table>
<thead>
<tr>
<th></th>
<th>Arduino</th>
<th>mbed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Support</strong></td>
<td>HUGE and active community Great and comprehensive documentation.</td>
<td>Also a large and active community, but not as coherent or easy to navigate as the Arduino</td>
</tr>
<tr>
<td><strong>Hardware</strong></td>
<td>Can only be used with Arduino boards. Tons of hardware available for almost any function you'd want; &quot;shields&quot; are pretty much plug-and-play Hardware is limiting for more sophisticated tasks, though the Due makes up for some of this - including power concerns</td>
<td>Can only be used with mBed boards. Powerful hardware; this was a focus from the get-go. A heavier focus on &quot;serious&quot; applications rather than hobbyist putzing around. The Arduino Due is very recent so it'll take some time for the community to catch up to the new capabilities Lower diversity of hardware. There are far more Arduino &quot;shields&quot; than there are mbed shields. If you normally breadboard stuff, this won't matter. I'd wager that most popular Arduino shields have an mbed library somewhere.</td>
</tr>
</tbody>
</table>
| **IDE** | Simple IDE with lots of examples, but it is very limited in functionality once you become more advanced.  
In the Arduino IDE can get cumbersome to manage multiple libraries and projects because each project is in a window. | IDE supports exporting to full-featured ARM toolchains whenever you are ready.  
The interface is more mature, allowing better libraries and projects management through a tree structure.  
Online IDE. Meaning you must be connected to the internet, the IDE is accessed through your browser. That raises security and practicality concerns. But it makes it super easy to import a library. Find a library you like on the mbed site, click "Import," and it's done. |
|---|---|---|
| **Deployment** | A bit more "coherent" as an entire platform. Generally designed for beginners, less space to mess-up the application.  
Moving from prototype -> production is not really a focus of the Arduino and therefore not quite as smooth as with the mbed. | Heavier focus on moving to production, final robustness is the key.  
Putting code on the MCU takes an extra step.  
A binary file is downloaded after compiling that you then move to the mbed (it shows up as a drive, similar to a USB flash drive). The mbed will then load this code automatically on the next reset. |

Despite them being so easy to use, they are not good for WSNs or IoT applications in general since they do not satisfy the WSN node requirement. First, they are not durable since they are not easily updatable, the user has to update each node manually. Second, they are not feasible; they only support their respective hardware boards which cost from 10$ up to 30$ per unit without the cost of the shields.
1.2.3 Tock

Tock's design is centered around protection (and thus, durability), both from potentially malicious applications and from device drivers. Tock uses two mechanisms to protect different components of the operating system. First, the kernel and device drivers are written in Rust, a systems programming language that provides compile-time memory safety, type safety and strict aliasing. Tock uses Rust to protect the kernel (e.g. the scheduler and hardware abstraction layer) from platform specific device drivers as well as isolate device drivers from each other. Second, Tock uses memory protection units to isolate applications from each other and the kernel.

It is bounded by 64 Kbytes of memory as a maximum (Levy, et al., 2015), does not assume the presence of a hardware memory management unit and all memory management is done at compile time which saves a lot of energy. However, the protection model placed too many restrictions on the users of Tock which made Tock’s applications very limited in nature, which defies the purpose of a general RTOS for WSNs and IoT applications. These limitations will be discussed in section 2.1.1.
Research Gap

2.1 The Problem: Reprogramming Wireless Sensor Networks

To narrow down the study, I decided to focus on one WSN-related problem: reprogramming sensor nodes. As explained in section 1.1, the ease of updating a sensor node is vital to its durability. Additionally, it improves the system’s scalability because, even if you could access hundreds of nodes to update them, you will not be able to access thousands of them. Last, but not least, it must be kept in mind that the reprogramming framework must be as memory and energy efficient as possible.

Remotely updating code, however, introduces a new risk: if the code contains any memory bugs, it may lead to breaking the node's wireless interface and the node becomes no longer reachable. This risk exists because, unlike general use computers, embedded systems typically do not have separate user and kernel layers; so application has access to device drivers code. The reason embedded systems do not separate user and kernel layer is that this separation imposes significant processing and memory overhead that would hurt the WSN’s feasibility. We explored two mechanisms of avoiding such safety hazards: memory separation and language imposed memory protection.

Memory separation is the common approach used in personal computers; It is when the memory has a bootloader that loads new image from EEPROM into a space in RAM dedicated for the program. Of course, the process of receiving the image and storing it into EEPROM, then loading it from EEPROM into RAM are very energy consumptive. Also, the bootloader is occupying space in the RAM that could be used for the application running.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Power (nAh)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read data block</td>
<td>01.261</td>
</tr>
<tr>
<td>Write data block</td>
<td>85.449</td>
</tr>
<tr>
<td>Send a packet</td>
<td>20.000</td>
</tr>
<tr>
<td>Receive a packet</td>
<td>08.000</td>
</tr>
<tr>
<td>Idle listen (1ms)</td>
<td>01.250</td>
</tr>
</tbody>
</table>

Table 1. Energy cost across different
Table 1 above (Crossbow Technology Inc., 2003) summarizes some of these costs in Mica2 motes as a demonstration of the variance of energy consumption across the different tasks. A data block is the same size as a packet and power is measured in nAh (nano Ampere hours). As seen in the table, writing a data-block costs significantly more than any other process. So having to write the image twice (once to EEPROM and then to RAM) is a very costly operation.

So instead, we considered reading the new image packets into RAM directly and instead of having the boot-loader separated and keeping the node safe, we would rely on the programming language used to write the code image to prevent any fatal memory bugs. This approach was inspired by the work done by the Tock development team (Levy, et al., 2015). They used RUST to write Tock. Rust has no runtime overhead / no garbage collector, instead: at compile time it detects data races and unsafe memory accesses by:

- No null / dangling pointers
- Type system supports traits - facility for ad-hoc polymorphism
- The keyword Unsafe which allows programmers to separate code that may subvert the type system from other so code regions trusted by kernel can be minimized.
- Affine types known as Ownership which determines when memory can be accessed or freed
  - a value’s owner is its variable
  - a value is freed when its owner leaves the scope
  - aliasing is disallowed, each value has a single owner at a time
  - x = y would disallow y from accessing its data, until it’s passed back by x
  - values can be “borrowed”. A value can only be borrowed once, and the borrow variable cannot outlive the value borrowed.

However, these features lead to four major fallbacks:

1. Even though they do not need to, the Ownership memory management mechanism would track resources that would never be freed such as hardware peripherals and device drivers.
2. Ownership will not allow different modules to normally keep reference to the same module, even though that might be necessary at times
3. The Ownership model prevents resource sharing between closures and kernel code. Basically, closures cannot be used as callback functions because RUST would not allow them to capture shared state. Namely, for a closure to capture a variable it must either own it or complete before returning.
4. closures’ requirement for dynamic memory is problematic for embedded systems because they can neither be allocated dynamically because there is no dynamic memory allocation support nor statically because this will allow for re-entrance and multiple calls for a single closure would then lead to a single invocation.

To counter these fallbacks, Tock developers used the following work-arounds respectively:

1. The hardware interrupts will be considered software ready-tasks and enqueue them in the main scheduler loop.
2. The safety-precautions of Ownership will be over-ruled by letting an unsafe piece of code borrow a static instance of the module/resource.
3. Tock abandoned closures completely and used other, less efficient and more complex, event-driven approaches.

Overall, the developers of Tock believed that RUST is still in need of some major modifications before it can support an embedded real-time operating system. Therefore, despite its relatively high memory and energy cost, we decided to stick with the normal memory separation approach for safe reprogramming of WSN nodes. In the next section, 2.2, we will study how the memory separation model can be used in a general framework to optimize overall memory need and energy consumption.

### 2.2 Current State-of-the-art

Let’s recall our three requirements for any good Wireless Sensor Network: durability, feasibility and scalability. In a wireless reprogramming framework this implies the following characteristic:
1. **Scope Selection**: To further optimize energy consumption in the system, a concept called Scope Selection is applied. This means that packets are marked for certain nodes, nodes that do not need the update do not have to encode or reload their images.

2. **Multi-hop routing**: Communication over the network is kept efficient by using *Code Propagation* (Levis P. A., 2005). As shown in the figure to the right, that means that the base station would send to the nearest node then those nodes would forward the image to phase 2 nodes and so on until the package has reached all nodes. This propagation requires multi-hop routing protocols to support it.

3. **Sends Updates Only**: Another thing that would minimize the size (and number) of packets sent is having the base station send only the image updates that the nodes in question have not received before.

4. **Encoding/decoding**: To minimize packet sizes, and thus optimize energy and memory consumption, the packets are encoded before transmission. This allows them to dramatically decrease in size. This can also be potentially used to secure communication between the nodes.

5. **Pipelining Support**: To allow for complete re-tasking, as opposed to only bug-fixing, the system should support fragmentation of the code image into separate packets that can be pipelined to the selected nodes. After a node has received all the image fragments in the EEPROM, the bootloader then switches the new image into the user application space in RAM and the node sends back a *Completion Validation* to the base station.
2.3 Defining the Research Gap

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Scope Selection</th>
<th>Multi-hop Routing</th>
<th>Sends Updates only</th>
<th>Encoding/Decoding</th>
<th>Pipelining Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>XNP</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Trickle</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Deluge</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>DiCode</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SenSeOP</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>AdapCode</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>

First, despite the presence of a robust theoretical framework for efficient wireless reprogramming of sensor nodes, there are still multiple trade-offs that prevented a single protocol from fully implementing this framework. Second, no system sends encoded updates, they all either encode entire image or send updates only, which reflects a big area of improvement in network usage and, thus, energy consumption. Last, writing a data-block to memory is almost 10 times more expensive than receiving a packet over wireless in stereotypical sensor node devices which indicates that using a bootloader is might be costlier than receiving bigger packets and writing them to application space directly.

In addition to these conclusions, that could be used to improve the framework for reprogramming WSNs, the framework also needs to be changed to adapt to the IoT trend described in section 1.1. If you notice in the framework above, the reprogramming process always starts from the base unit. In an Internet of Things composing of millions of end-points, there cannot be a single base-unit that can initiate the process.
Future Work

There are three things that can be done to map the existing WSN reprogramming framework into a durable, feasible and scalable solution to reprogramming devices and gadgets on the IoT:

1. **Re-design** the framework and the propagation methods as a distributed model by:
   - I. Finding a distributed alternative for Nodes Base
   - II. Designing an application-based node selection scheme
   - III. Limiting the number of message exchange and broadcasts

2. **Implement** a framework that can combine all the features in the theoretical current-state-of-the-art.

3. **Optimize** memory consumption by studying memory-efficient ways to load the code without using a bootloader
References


