

HiNRG Technical Report: 21-09-2008

A Modular Approach for WSN Applications

Răzvan Musăloiu-E. Chieh-Jan Mike Liang Andreas Terzis
razvanm@cs.jhu.edu cliang4@cs.jhu.edu terzis@cs.jhu.edu

Abstract

We propose a mechanism for introducing loadable code to the TinyOS 2 framework. Unlike previous proposals that use virtual machines, the loadable code runs natively on the mote avoiding the overhead associated with interpretation. Loadable programs interact with the rest of the system through a *Proxy*, a nesC component that exposes abstract versions of system resources including the sensors, flash memory, communication, and timers. We leverage the abstractions provided by the Proxy through *TinyJavaScript*, a subset of the popular JavaScript language that can be used to develop wireless sensor network applications. TinyJavaScript offers familiar syntax, type inference, and the ability to simulate and debug mote programs in a browser. TinyJavaScript programs are translated to equivalent C code and compiled to *MicroExe* binaries that are dynamically linked by *TinyLD*, a dynamic linker/loader that runs on the mote. We compare loadable code and TinyJavaScript to monolithic programs developed in nesC/TinyOS. Our experimental results show that offering system modularity and the ability to program applications in a high-level language incurs only moderate overhead in terms of increased memory footprint and runtime performance.

1 Introduction

Developing applications for sensor networks is a challenging task, currently mastered by a small group of seasoned developers. However, if wireless sensor networks (WSNs) are to reach their full potential, programming them has to be more accessible to the average programmer who does not have the time or interest to become proficient with the intricacies of a new programming paradigm before developing a WSN application. This problem is compounded by the requirement to re-program WSNs after they have been deployed, either to correct programming errors or to evolve the application based on experience from the deployment. Doing so efficiently, requires minimizing the size of code updates to minimize the energy cost of delivering the update to the network's motes. This imperative in turn calls for assembling applications from modular components and transmitting only modified components.

Virtual machines (VMs) [11, 12] were proposed in the past to address the two challenges described above. In this paradigm, a developer writes the application in a high-level language that is translated to bytecode for a virtual machine. This bytecode is then executed by an interpreter running on the mote. While VMs offer both the ability to develop WSN applications using high-level languages (e.g., subsets of Java) and compact program representations, they suffer from high execution overhead associated with the bytecode interpretation process.

Depending on the duty cycle of the application, this overhead and the energy consumption associated with it, can considerably reduce the network’s operational lifetime.

In this paper we present a different methodology for developing and evolving WSN applications in the field. Our approach is driven by the observation that application developers rarely need to modify system services such as the wireless radio, sensors, and flash memory but can instead interact with *service abstractions*. These abstractions are offered by a *Proxy*, that in turn interfaces with the corresponding low-level system components. The level of abstraction provided by the Proxy offers the opportunity to develop WSN applications in a high-level language that abstracts the intricacies of developing code for motes. We present *TinyJavaScript*, an example of such a language that offers type inference and a simplified debugging and testing process.

Raising the abstraction level however, should not be done at the expense of efficiency. Instead, we provide a mechanism by which a TinyJavaScript program is translated to C and eventually object code. *TinyLD*, a dynamic linker/loader we developed, links the application object code to the system-level code. The result is a single binary that runs natively on the mote. In this way, we achieve the best of both worlds: simplifying application development and achieving execution efficiency comparable to that of monolithic programs.

The two parts of our approach (i.e., Proxy/TinyLD and TinyJavaScript) are independent. One can leverage the ability to dynamically evolve application behavior by exploiting the services exported by the Proxy without using TinyJavaScript, opting to use a system-level language like C. At the same time, the proposed architecture provides a basis for the development of multiple high-level languages that can raise the level of abstraction for programming sensor networks.

Said differently, we advocate for a bottom-up approach to the problem of WSN programming. Exposing system services is the basis upon which multiple novel programming paradigms can be built upon. We take a modest step in this direction by presenting a simple, yet expressive scripting language for developing WSN applications, but expect that other more ambitious languages will continue down the same path.

Our experimental results show that the proposed reprogramming mechanism is efficient, generating moderate overhead in execution performance and resource usage. The increase in resource usage is mostly due to the addition of the Proxy and TinyLD components. For the Tmote Sky platform the overhead we measured is less than 19.6% of the available ROM and less than 4.7% of the available RAM. Overhead in execution time is limited to the cost of one or two additional function calls for the services provided by the Proxy and copying of buffers between the application and the Proxy. Finally, linking and loading a new binary requires less than 1.5 seconds for the applications we implemented in our framework.

This paper makes the following three contributions. First, we propose an efficient way for WSN reprogramming while leveraging the existing code base of TinyOS 2 [13] and the community built around it. Second, we present TinyJavaScript, a simple scripting language that can be used to develop WSN applications, and finally we show how TinyJavaScript programs can be translated to native code and provide a dynamic linker and loader (TinyLD) for loading such programs to the mote.

This paper has eight sections. In the section that follows we frame our work in the context of the related literature. Next, we present an overview of our approach for providing efficient reprogramming in TinyOS. Section 4 describes TinyJavaScript, an example of a scripting language customized for writing WSN applications, while Section 5 describes the steps required to translate TinyJavaScript to executable code that runs natively on motes. We evaluate the performance and efficiency of our approach in Section 6. Finally, we outline future work in Section 7 and close in Section 8 with a summary.

2 Related Work

The need for flexible programming models has inspired the use of virtual machines (VMs) for sensor networks. Maté was the first VM-based approach proposed for this task. Maté programs are contained in sequences of *capsules* delivered to the mote and interpreted by Bombilla, a stack-based VM optimized for resource constrained environments [11]. Subsequent work on Active Sensor Networks extends on Maté by providing a method for building application specific virtual machines (ASVMs) [12]. ASVMs provide a flexible boundary between native and interpreted code. VM^{*} takes a similar approach, customizing the service layer on a per-application basis [9]. VM^{*} programs are written in Java and import Java libraries to interact with the underlying hardware. Our approach also uses a high-level language for programming WSNs (TinyJavaScript in our case, TinyScript and Mottle for ASVMs, and Java for VM^{*}). On the other hand, TinyJavaScript programs translate to native code avoiding the overhead of interpretation. Given a mote’s limited resources, we argue that virtualization at the function level, as provided by our Proxy, is preferable to VMs.

A number of previous proposals have convincingly argued about the advantages of using loadable code to dynamically evolve WSN applications. Among them, SOS is a dynamic operating system for sensor networks that allows a high degree of modularity [7]. SOS modules are fragments of position independent code (PIC) that can be inserted and removed without rebooting. Even though SOS reuses some parts of TinyOS, it is incompatible with it. The system Proxy we propose offers similar features with the SOS API in the context of the TinyOS framework. While SOS uses C for application development, we use TinyJavaScript, a high-level language that offers type inference and the ability to simulate and debug WSN applications in a browser.

Contiki is another sensor operating system that supports loadable objects [4]. Contiki uses ELF/CELF binaries which are similar to the MicroExe binaries we use. While the MicroExe binary format we developed also derives from ELF it is not compatible with it (like CELF). On the other hand, MicroExe is more compact thus reducing program transfer cost. The approach followed by Contiki is more general in the sense that it allows arbitrary loading and unloading. TinyLD works in more restrictive environments because it does not assume the existence of a byte-level external memory (the ESB platform used in Contiki has a 64KB EEPROM that is used by the loader to store the binary image during the linking and relocating phase).

The work that is closest to ours is FlexCup [20]. FlexCup also allows dynamic load-

ing of TinyOS components. There are however many differences between Flexcup and our approach. First of all, FlexCup uses a different linking and loading method that requires rebooting the node for the new image to run. Moreover, the application halts during the linking and loading process. On the other hand, using TinyLD, the application can continue its operation during that time and it can restart immediately after the load process terminates. Finally, in addition to TinyLD, we provide a system Proxy that abstracts system resources and forms the basis for high-level programming languages such as TinyJavaScript.

A number of domain-specific languages for WSNs have been proposed in the past, spanning the design space from in-network query processors to macroprogramming ([3, 17, 19, 6, 22, 23, 2] among others). TinyJavaScript is a modest proposal in this space, offering a familiar environment for non-specialists to develop WSN applications. By using a syntax that is similar to C and retaining split-phase operations from TinyOS it is efficient and easy to translate to native code. At the same time, it removes low-level constructs such as pointers and explicit types. Moreover, TinyJavaScript offers an interesting opportunity for prototyping and testing WSN applications. We expect that more ambitious languages will emerge based on our approach but until then, TinyJavaScript offers a feasible alternative to programming WSN applications in nesC.

3 Overview

Our approach is motivated by the observation that while the fine-level control and high expressivity provided by a *system-level* language such as nesC [5] are invaluable to *system-level* developers, i.e., programmers that develop services such as MAC and time synchronization protocols, it is seldom used by *application-level* developers, who are tasked to write applications such as environmental monitoring. The distinction between system and application-level programmers is crucial. While the former need to intimately control hardware resources (e.g., radio, sensors, and storage), the latter can interact with *abstractions* of these resources.

Our goal is to raise the level of abstraction for application developers who can code their applications using *abstract* versions of the system’s resources. Given these abstractions, application developers can use a high-level (scripting) language to encode application logic. At the same time, we do not want to trade simplicity in the development process with poor execution performance. For this reason, we require the high-level application code to run natively on the mote’s hardware. This is a distinction with previous proposals in which application scripts are interpreted by a virtual machine running on the mote [11].

Figure 1 provides a graphical overview of the proposed architecture. The user-level code is dynamically loaded on the mote on which it runs natively. This code accesses system resources by calling *functions* provided by the *Proxy*, a nesC component whose purpose is to expose resources such as the radio interface, flash memory, onboard sensors, etc. We retain the split-phase paradigm from TinyOS to avoid blocking during long operations. To do so, the application code registers *callback functions* which the Proxy calls when certain operations complete.

In turn, the Proxy implements the exposed services by internally connecting to other

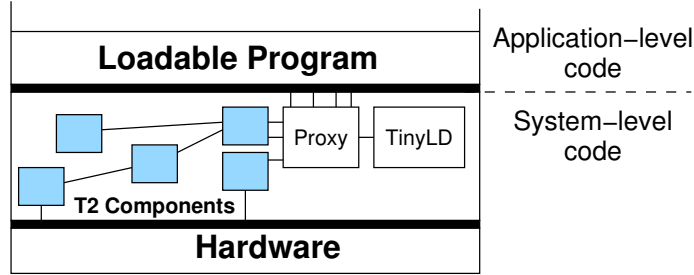


Figure 1: Code organization on a running mote including loadable application code, the system Proxy, and the TinyLD linker/loader.

TinyOS components and calling the corresponding functions provided by these components. When a split-phase operation completes, the Proxy closes the loop by calling the callback function registered by the application code. Note that the application code does not interact with any other nesC components, effectively separating the application from the rest of the system. At the same time, we do not provide any explicit mechanisms to ensure separation between the application and system code. An application can take full control of the mote (e.g., by going into an infinite loop) or corrupt system code by overwriting physical memory. In other words, we implicitly trust that the application running on the mote is well behaved. In this respect we follow the security model used in TinyOS today.

Figure 2 outlines the two parallel paths involved in the proposed code generation process. The path to the right summarizes the generation of *system* code that includes the Proxy and other TinyOS 2 components. It starts with the definition of the resources (e.g., radio, sensors, etc.) that will be provided by the Proxy. We envision a graphical front-end in which the user inputs the required components and that automatically generates a binary customized for a specific deployment that can be installed on a mote. Creating such a customized binary minimizes the Proxy’s memory footprint. The second output of the right path is a set of helper `.h` files used for the compilation of the application code.

Moving to the left path, the application code written in a scripting language is first translated to equivalent C code during the meta-compilation phase. The generated C code is subsequently compiled into an object file for the target platform (e.g., the MSP430 microcontroller used in the Tmote Sky mote [21]). During the pre-processing stage that follows, the object file is encapsulated into a *MicroExe* binary, a compact binary format we developed for embedded programs. The resulting binary cannot be executed as-is because it contains unresolved references to external functions (i.e., the functions provided by the Proxy) and unallocated variables. All these issues are resolved during the linking and loading phase. At the end of this phase, the application code is properly patched, moved in the internal flash ROM of the microcontroller and is ready to run in combination with the existing code.

While the first three stages in the system path of Figure 2 occur on the programmer’s desktop, linking and loading occur on the mote after the application object code is placed in its external flash memory. The code is transferred either through a serial connection or via an over-the-air code distribution mechanism such as the one used in Deluge [8].

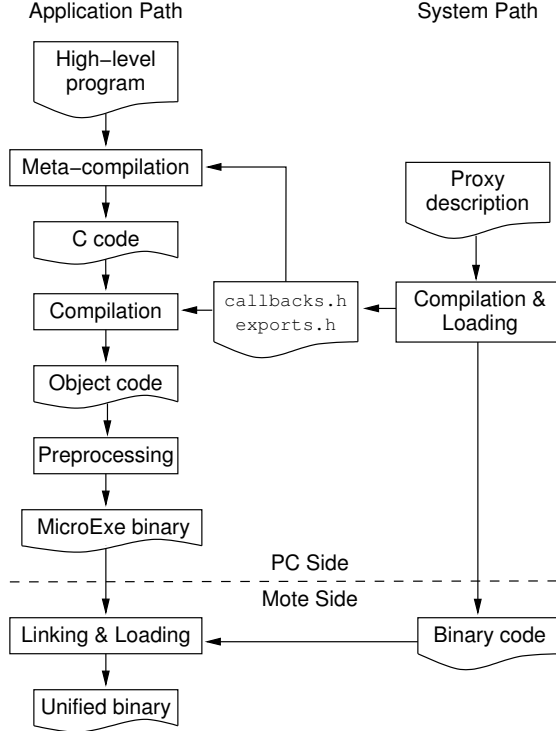


Figure 2: The different stages of the code generation process.

Note that when one needs to modify the application running on a deployed sensor network, only the application code needs to be transmitted over the air instead of the complete binary image. This decreases the propagation cost and subsequently decreases the amount of energy required for reprogramming.

4 The TinyJavaScript Language

The goal of TinyJavaScript is to simplify the development of loadable WSN applications. As its name implies, it is a derivative of the popular JavaScript scripting language. This choice has the advantage of preserving the expressiveness of C, without the incurring burden of pointer arithmetic and explicit typing information associated with it. Moreover, as we describe in Section 4.4, using JavaScript provides an easy way to simulate WSN applications.

TinyJavaScript programs interface with a global object called TinyOS. This object contains all the objects and functions exposed by the Proxy mentioned in Section 3. More specifically, a TinyJavaScript program can perform the following actions: **(1)** implement callback functions that complete the two-way interfaces offered by the objects provided in the global TinyOS object; **(2)** define local variables; **(3)** define local functions.

Appendix A.1 presents an example of a complete application written in TinyJavaScript that is equivalent to the Blink application in TinyOS. Briefly, the application sets three

Objects	Functions	Callbacks
TinyOS	nodeId(), post(fun)	init()
TinyOS.Leds	set(value), get()	
TinyOS.Leds[<i>n</i>]	on(), off(), toggle()	
TinyOS.Timer[<i>n</i>]	stop(), startOneShot(interval), startPeriodic(interval)	fired()
TinyOS.Radio	start(), stop()	startDone(error), stopDone(error)
TinyOS.Radio[<i>am</i>]	send(addr, msg, len)	sendDone(error)
TinyOS.Serial	start(), stop()	startDone(error), stopDone(error)
TinyOS.Serial[<i>am</i>]	send(addr, msg, len)	sendDone(error)
TinyOS.Sensor[<i>n</i>]	read()	readDone(error, val)
TinyOS.Volume[<i>n</i>]	getSize(), read(addr, buf, len), write(addr, buf, len), erase(), crc(addr, len)	readDone(error), writeDone(error), eraseDone(error), crcDone(crc, error)
TinyOS.Systime	get()	
TinyOS.Random	get16(), get32()	

Table 1: The objects available in TinyJavaScript through the TinyOS object. All callbacks and functions from TinyOS.Leds, TinyOS.Leds[*n*] and TinyOS.Timer[*n*] return void. Most of the other functions return an error code.

timers with different timeout values and toggles one of the mote’s LEDs when each of the timers fires.

4.1 TinyOS Object

Table 1 summarizes the service abstractions and objects available through the TinyOS object. Most of the table’s entries are self explanatory, so we focus on the ones that deserve further explanation.

Some objects offer only functions (e.g., TinyOS.Systime), while others offer only indexed objects (e.g., TinyOS.Timer), and others have both (e.g., TinyOS.Leds). Indexing has different semantics for different objects. For example, in the case of the TinyOS.Radio and TinyOS.Serial objects, it represents the type of active messages sent over the radio and the serial interfaces respectively, while in the case of TinyOS.Timer it represents a separate instance of a Timer object. Applications can post tasks by using the TinyOS.post() function. The Proxy enqueues tasks posted by the application for execution. Due RAM resource constraints, the task queue might not be able to handle all the tasks posted. In this case, the Proxy returns an error to the application. Otherwise, after the Proxy regains control, it executes all queued tasks and clears the task queue. Finally, the TinyOS.init() function is used to define the application code segment that will run immediately after the application has been loaded to the mote.

While Table 1 represents the current system interface, it is straightforward to include additional TinyOS 2 services, such as Dissemination and Collection.

TinyJavaScript fragment

```
SENSOR_NO = 3;
readings = new Array(SENSOR_NO);
...
TinyOS.Sensor[0].readDone = function(error, val)
{
  if (error == SUCCESS) {
    readings[0] = val;
  }
  TinyOS.post(mainTask);
}
```

C fragment

```
#define SENSOR_NO 3
uint16_t readings[SENSOR_NO];
...
void _sensor0readDone(uint8_t _error, uint16_t _val)
{
  if (_error == SUCCESS) {
    readings[0] = _val;
  }
  _post(mainTask);
}
```

Figure 1: Type inference for arrays. The array size is determined by the constant `SENSOR_NO`. The type of the array (`uint16_t`) is inferred from the assignment of `readings[0]` to `val`, which is known from the predefined function `readDone`.

4.2 Language Constructs

Next, we elaborate on the features offered by TinyJavaScript and provide examples of their use.

Variables and Built-in Types. TinyJavaScript preserves the variable declaration method using `var` from JavaScript. Variable and function return types are not explicitly declared but are instead inferred. Type inference is possible due to the observation that the prototypes of the external and callback functions are known at compile time. Another hint that guides type inference is the type of the operation that a variable is involved in.

Arrays can be declared using the `new` operator on the object `Array`. The code snippet in Figure 1 provides an example of a variable definition in TinyJavaScript and explains how its complete type is determined.

Associative arrays in TinyJavaScript translate to C structures. If the types of a structure’s elements have to be precisely defined (e.g., when defining packets sent over the serial link or the radio), predefined values (`uint16_t`, `uint32_t`, etc.) can be used in dummy assignments to indicate the desired type. To declare a field without setting its type, the `undefined` type can be used. Figure 0 shows an example of explicit type definition for an associative array.

Operators. With minor exceptions (i.e., identity check, `===` and `!==`), all the JavaScript arithmetic and logic operators are present in TinyJavaScript and can be directly translated to C. The special operators `delete` and `typeof` are not supported, while the `new` operator is partially supported when used in array declarations.

Control structures. The control structures that TinyJavaScript provides are: `if`, `while`, `for`, `for...in`, `break`, `continue` and `return`. We do not currently support the `with` and `try...catch` clauses from JavaScript.

Functions and Objects. JavaScript comes with a relatively rich set of built-in functions (`eval`, `parseInt`, `parseFloat`, `escape`, `unescape`) and objects (`Array`, `Boolean`, `Date`, `Math`, `Number`). Even though it would be easy to implement some of them (i.e., `Boolean`, `Date`,

TinyJavaScript fragment

```
rpkt_in = {'key': uint32_t,  
          'version': uint32_t,  
          'value': uint32_t};  
  
TinyOS.Radio[0].receive = function(payload, len)  
{  
  rpkt_in = payload;  
  if (rpkt_in.key == curKey) {  
    ...  
  }  
}
```

C fragment

```
struct {  
  uint32_t key;  
  uint32_t version;  
  uint32_t value;  
} rpkt_in;  
  
void _radio0receive(void* _payload, uint32_t _len)  
{  
  memcpy(&rpkt_in, _payload, sizeof(rpkt_in));  
  if (rpkt_in.key == curKey) {  
    ...  
  }  
}
```

Figure 0: Translation of associative arrays to C structures. The `memcpy()` operation is necessary because the `_payload` pointer might be misaligned.

and Number) their usefulness is limited in this application domain and for this reason we chose not to do so. On the other hand, we do allow limited use of the `Array` object, in order to describe arrays from C.

Finally, object prototypes is a JavaScript feature that we do not support in TinyJavaScript.

4.3 Meta-Compilation

TinyJavaScript programs are translated to equivalent C code using a custom compiler we wrote in Python. Lexical and syntax analysis is done using PLY [1], an implementation of the standard `lex` and `yacc` tools for Python. Besides offering all the main features (e.g., empty productions, precedence rules, ambiguous grammars) of standard lexical and syntax analyzers, PLY offers a number of other useful features. It is compatible even with old versions of Python (2.1), it is contained in only two files, and it does not have any external dependencies. These features, coupled with the wide availability of the Python interpreter, mean that the meta-compiler we developed can run on a wide range of systems and configurations.

Due to the high degree of similarity between the syntax of TinyJavaScript and that of C (alluded by the code snippets in Figures 1 and 0), the translation from TinyJavaScript to C is relatively straightforward. The only real challenge is type inference. The meta-compiler performs this task by exploring the abstract syntax tree and augmenting it with the type information from external functions and operations in which the variables are used. Lastly, because all the meta-compiler logic is encoded in the two Python files, it is possible for third parties to augment its functionality.

```

1  #include <sys/inttypes.h>
2
3  extern void _led00n();
4  int i;
5
6  void inc() {
7      i = i + 1;
8  }
9
10 void _init() {
11     i = 1;
12     inc();
13     _led00n();
14 }

```

Figure 1: `demo.c`, a simple C program that increments a variable and requests services from the Proxy.

4.4 Simulation

Because TinyJavaScript is a subset of JavaScript, TinyJavaScript programs are also valid JavaScript programs. This allows us to build a simulation environment using any JavaScript enabled web browser. To demonstrate this capability, we developed `TinyOS.js`, a library that implements the objects from Table 1. Support for timers and tasks is provided by the JavaScript function `setTimer()`. Because packets are JavaScript objects, JSON¹ can be used to simulate their transmission and reception. By using an HTML page in conjunction with `TinyOS.js` we can quickly test and debug complete TinyJavaScript programs using a web browser as a graphical front-end.

5 TinyLD

TinyLD is the dynamic linker and loader we developed for TinyOS. The purpose of TinyLD is threefold: **(I)** Translate the object code produced by the C compiler to a loadable program. **(II)** Patch unresolved address references and link the loadable program to the Proxy. **(III)** Load the resulting program to the mote’s flash ROM. In the paragraphs that follow we elaborate on each of these three phases.

5.1 Background

Before we delve into the details of how dynamic linking and loading is done in our case, we provide background information on the steps required to link and load a program in general. Given that the `gcc` compiler for the MSP430 uses the Executable and Linkable Format (ELF), we use ELF to provide this information.

¹JSON (JavaScript Object Notation) is a lightweight computer data interchange format. Support for JSON was added to many languages and is used heavily in Ajax applications.

```

00000000 <inc>:
  0:  05 12          push  r5
  2:  04 12          push  r4
  4:  92 53 00 00    inc   &0x0000
  8:  34 41          pop   r4
  a:  35 41          pop   r5
  c:  30 41          ret

0000000e <_init>:
  e:  05 12          push  r5
 10:  04 12          push  r4
 12:  92 43 00 00    mov   #1,    &0x0000
 16:  b0 12 00 00    call  #0
 1a:  b0 12 00 00    call  #0
 1e:  34 41          pop   r4
 20:  35 41          pop   r5
 22:  30 41          ret

```

Figure 2: Disassembly of demo.o. Entries that need patching are highlighted, including the address location of variable i, the call to internal function inc() and external function _led00n. The push and pop instructions represent dead code which is removed when compile optimizations are enabled.

An ELF object file includes the ELF symbol table and the ELF relocation table that contain information about how to resolve address references in the code. Figure 1 lists a simple C program similar to the one generated by the TinyJavaScript meta-compiler. This program turns on LED 0 after incrementing the variable `i`. The C `extern` keyword on line 3 indicates that `_led00n` is an external function (defined in the Proxy). Function `_init` on line 10 is a reserved function name equivalent to the `TinyOS.init()` function in TinyJavaScript (see Table 1).

The result of compiling the C code in Figure 1 is an object file `demo.o` whose disassembled code is shown in Figure 2. As this figure shows, there are four unresolved address references at address offsets, `0x0006`, `0x0014`, `0x0018`, and `0x001c`. These references have a value of `0x0000` in the machine code. The goal of the linking and loading process is to correctly resolve these references before the program starts executing.

To help in this process, ELF files contain a symbol table and a relocation table, in addition to the machine code. As their names imply, the first table contains the names and types of all symbols used in the program, while the second contains all the offsets, relative to the starting address of the code, in which these symbols are used. Figures 3 and 4 show the corresponding symbol and relocation tables of `demo.o` respectively.

We focus first on entries 5-8 in the symbol table. Entries 5 and 7 indicate that `inc` and `_init` are defined at address offsets `0x0000` and `0x000e` in the machine code. Since `_init` is a callback function that will be invoked by the Proxy component, the linker will need to update the corresponding entry in the Proxy's callback table with the start address of the loaded code plus offset `0x000e`. Entry 6 describes the two-byte variable, `i`. Since this is a local variable, the loader will need to allocate two bytes in RAM for it. Entry 8 is a call to the external function `_led00n` provided by the Proxy which needs to be resolved during the linking process.

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	FILE	LOCAL	DEFAULT	ABS	demo.c
2:	00000000	0	SECTION	LOCAL	DEFAULT	1	
3:	00000000	0	SECTION	LOCAL	DEFAULT	3	
4:	00000000	0	SECTION	LOCAL	DEFAULT	4	
5:	00000000	14	FUNC	GLOBAL	DEFAULT	1	inc
6:	00000002	2	OBJECT	GLOBAL	DEFAULT	COM	i
7:	0000000e	22	FUNC	GLOBAL	DEFAULT	1	_init
8:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_led00n

Figure 3: The ELF symbol table of demo.o. Entries of interest are highlighted.

Offset	Info	Type	Sym.Value	Sym.Name+Addend
00000006	00000603	R_MSP430_16	00000002	i + 0
00000014	00000603	R_MSP430_16	00000002	i + 0
00000018	00000505	R_MSP430_16_BYTE	00000000	inc + 0

Figure 4: The ELF relocation table of demo.o.

Moving on to the relocation table, the first two entries in Figure 4 indicate that variable `i` is used in two address offsets, `0x0006` and `0x0014`. This information, combined with size knowledge from the symbol table as well as knowledge of the location in which the code will be placed in physical memory, help the loader to resolve the references to `i`. Similarly, the last two entries of Figure 4 specify function calls to `inc` and `_led00n` at address offsets, `0x0018` and `0x001c`. Also shown in Figure 4 are `addend`'s which are equivalent to array offsets. Because `demo.c` does not use any arrays, the `addend` values are always 0.

5.2 MicroExe

While widely used in Unix systems, using ELF for mote binaries is inefficient. For example, addresses are encoded as 32-bit values while mote platforms based on the MSP430 microcontroller, such as the Tmote Sky [21], have 16-bit address space. Moreover, symbol names are encoded as text strings for ease of use, thus increasing the size of the file. To avoid the inefficiencies associated with ELF, we designed *MicroExe*; a file format designed specifically for loadable programs in TinyOS. Addresses in the MicroExe file format are 16 bits wide and are encoded using the little-endian byte order to comply with the byte order used on mote platforms. We use a PC-side script to construct the MicroExe files. The script invokes the MSP430 tool-chain to extract the ELF symbol and relocation tables and re-arranges their information according to the MicroExe format described below.

As Figure 5 indicates, there are four sections in a MicroExe file: metadata, callback table, patch table, and finally the machine code of the loadable program.

Metadata. The metadata is the only section in the file with fixed size and provides the size of each other section. The first four fields in this section specify the number of entries in each of the subsequent sections, while the last field specifies the size of the machine code in bytes.

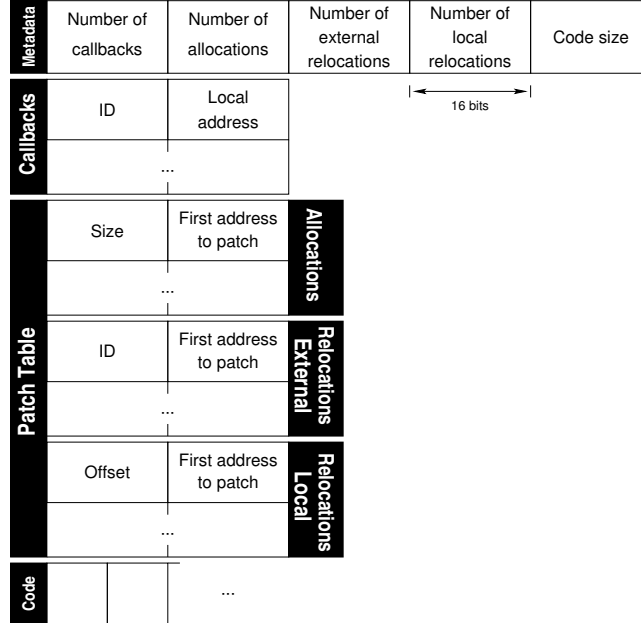


Figure 5: MicroExe file format.

Callback table. The callback table includes the address offsets of all callback functions defined in the loadable program. These are the functions that the Proxy component will call (*e.g.*, `readDone()`). Since their actual location in memory is known only during loading, the TinyLD linker will need to convert these address offsets before patching the corresponding entries in the Proxy component. To simplify table lookups and reduce the size of the callback table, callback functions are referenced by numerical identifiers instead of human-readable strings.

Patch table. The patch table contains information about the type and location of each unresolved address reference in the machine code. It contains three sub-tables: (1) The allocation table, which contains locally defined symbols (*i.e.*, global variables). (2) The external relocation table, which contains references to external functions called by the program, and (3) the local relocation table which contains references to variables and functions defined in the program.

To minimize the footprint of these tables, MicroExe employs a common compiler technique called *chained references* [10]. This technique takes advantage of the observation that references to unresolved addresses in the code contain useless values, or `0x0000` in the case of the MSP430 gcc compiler (see Fig.2). We can then create a chain or linked list of all the references to the same unresolved symbol. Specifically, a chain is created in the machine code by having each unresolved reference point to the next unresolved reference of the same symbol. By employing this technique, each entry in the patch table has only two columns: the symbol and a pointer to the first chained reference of that symbol.

The symbol column acquires a different meaning depending on the sub-table it belongs.

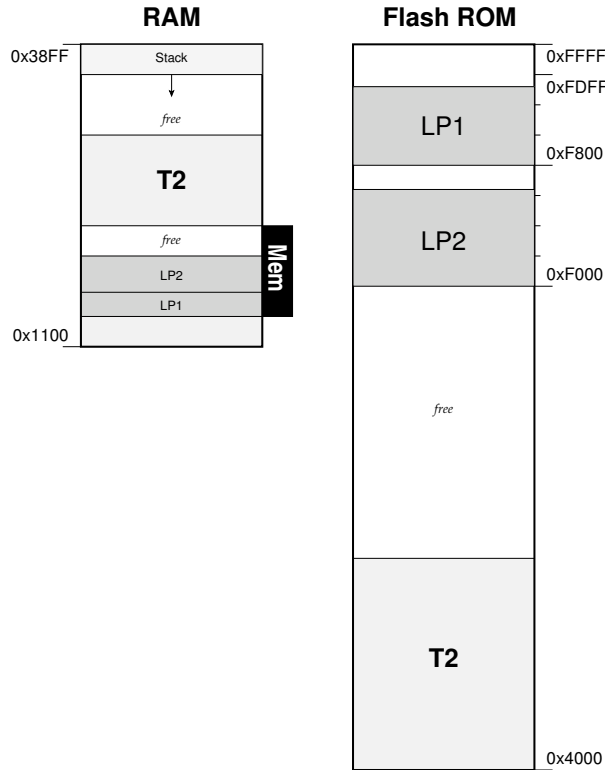


Figure 6: RAM and flash ROM layout. Two loadable programs, *LP1* and *LP2*, have been loaded to the flash ROM. In addition, space in the *Mem* region is allocated for the variables in *LP1* and *LP2*.

As Figure 5 shows, the allocation table interprets the symbol column as the symbol size. The external relocation table and the local relocation table interpret it as a Proxy service identifier and address offset within the machine code respectively. This design decision ensures that the patch table grows with the number of unique symbols in the loadable program, not with the number of times each symbol is referenced.

Machine code. The machine code section of the ELF file is copied to this section.

5.3 Dynamic Linking and Loading

Before a loadable program can run on the mote, TinyLD must link its machine code to the Proxy by patching unresolved addresses corresponding to calls for Proxy functions and callbacks, allocate memory space for variables defined in the program, patch references to local variables, and finally load the machine code to the flash ROM.

Figure 6 presents a graphical layout of the physical RAM and ROM of a mote in which the loadable programs reside after the linking and loading process terminates. Shown in the figure are two loadable programs *LP1* and *LP2*, whose machine code resides in ROM. Also shown in the figure is a pre-allocated area of the RAM (*Mem*), which TinyLD uses to store

the programs' local variables.

TinyLD starts this process by reading the metadata section of the MicroExe file. Knowing the size of the machine code, TinyLD calculates the flash ROM address in which the machine code will reside. This location information is crucial, as all addresses in the callback table and in the patch table are offsets with respect to the first address of the machine code. Contrary to TinyOS, MicroExe places the machine code at the end of the flash ROM (i.e., the highest address). However, special care must be taken because mote platforms such as the Tmote Sky reserve the last sector of the ROM for interrupt vectors. Figure 6 shows this area as memory range `0xFDFD` - `0xFFFF`. Another constraint is that the MSP430 can only erase the flash ROM at sector-level blocks (512 bytes). To allow unloading individual programs, MicroExe ensures that programs are sector-aligned. As Figure 6 shows, a disadvantage of this decision is that ROM fragmentation may occur.

Next, the callback table is copied from the external flash to RAM. Instead of allocating space in RAM, TinyLD temporarily stores the callback table in *Mem*. Since the new loadable program is not running at this point, the unused portion of *Mem* can be safely used for other purposes. Next, the callback identifiers are resolved, and their addresses are translated with respect to the start of the machine code. Finally, MicroExe patches the Proxy with these values.

The next step is to patch unresolved address references in the machine code. To do so, the patch table is first copied in RAM in the same way with the callback table. Then, TinyLD pre-processes the patch table because each table entry interprets the first column differently (cf. Fig. 5). Based on the size of local variables, TinyLD assigns space within *Mem* to each local variable and overwrites the size information in the patch table with the resolved addresses. Next, TinyLD translates the Proxy service identifiers in the external relocation table to the resolved addresses given by the Proxy. Finally, TinyLD adjusts the address offsets in the local relocation table with respect to the machine code location.

Because RAM use is at a premium, the machine code is not cached in RAM. Instead, the machine code is copied from the external flash in chunks, which are defined as code segments between two consecutive unresolved references. The process starts by searching the MicroExe patch table for the lowest unresolved address reference in the machine code. Then the code chunk up to this address is copied to the flash ROM. TinyLD then writes the resolved address reference to the flash ROM and the corresponding reference chain is updated with the next unresolved reference. The process repeats by finding the next lowest unresolved reference and terminates after all unresolved address references have been successfully resolved.

The section that follows presents an example to better illustrate the steps outlined above.

5.4 Linking and Loading Example

We use the simple C program from Figure 1, to demonstrate how a MicroExe file is created and all the steps that TinyLD follows.

Building the MicroExe binary file. The process begins with a script that parses the ELF symbol and relocation tables (shown in Figures 3 and 4). From the symbol table the

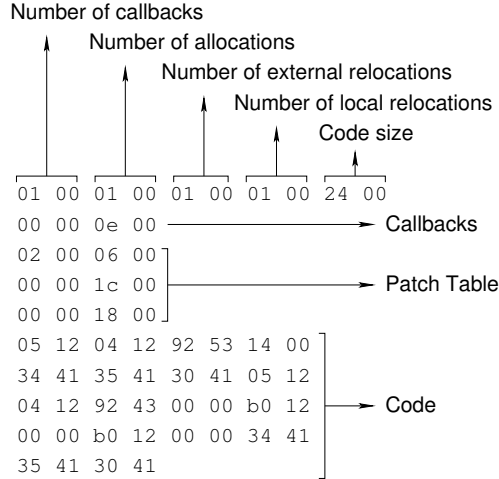


Figure 7: MicroExe binary file for demo.c.

script learns that `i` is a local variable with size of two bytes and inserts this information in the allocation table. Since the list of Proxy services is known in advance, the pre-processing script also knows `_init` is a callback function and fills the callback table with an entry for `_init` and address `0x000e`. Likewise, MicroExe fills the global relocation table with an entry of `_led00n`. The location of `_led00n` function call, `0x001c`, can be found in the ELF relocation table. To reduce space use, `_init` and `_led00n` are recorded using predefined numerical identifiers. Finally, `inc` and `i` are recorded in the local relocation table. Note that `i` has two entries in the ELF relocation table because it is called twice in the code. Instead of having two records for `i` in the local relocation table, we chain these two references and point the first reference to the second reference. Therefore, the first reference has the little-endian value, `0x1400`. Figure 7 shows the resulting MicroExe binary file of `demo.c`.

Linking and loading. Linking and loading starts by reading the metadata portion of the MicroExe file, which is then used to calculate the ROM location in which the machine code will reside. Because the last sector of the flash ROM is reserved for interrupt vectors and the code size is 36 bytes long, the machine code will be placed at address `0xfc00`, as Figure 8 shows. The callback table is first copied to *Mem*. Using the callback table, we can update the Proxy with the actual locations of all callback functions used by the program (`_init` at `0xfc0e`). The patch table is copied to RAM next. As mentioned before, entries in the table need to be offset differently. Figure 8 shows this step. First, address references to the functions that the Proxy exports are translated based on knowledge about the Proxy’s location in ROM. Next, local function references need to be offset with respect to the machine code location in the flash ROM. Last, local variables need to be offset with respect to *Mem*.

Patching unresolved address references starts by searching for the lowest unresolved address in the address patch table, which is `0x0006`. The code segment before `0x0006` is first copied from the external flash to the flash ROM. Then, the resolved address reference, or `0x2000`, is written to the flash ROM. Finally, the corresponding chain in the patch table is

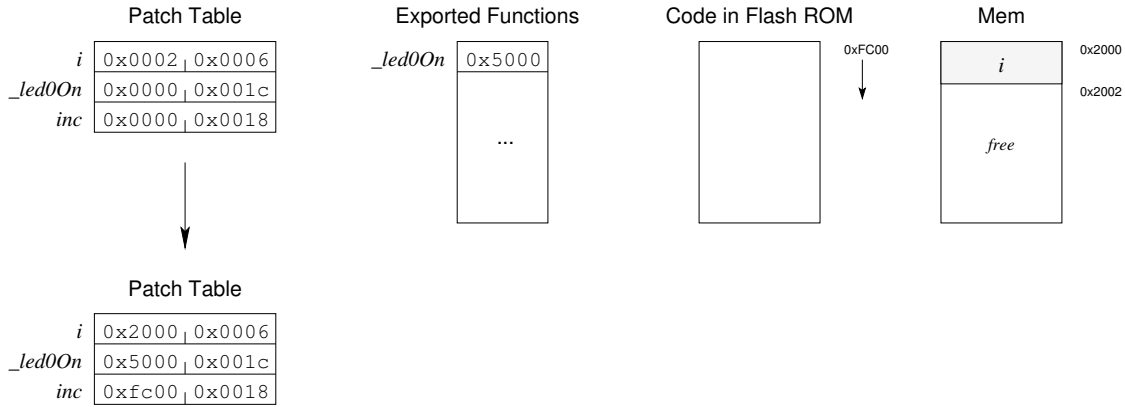


Figure 8: TinyLD pre-processes the patch table before linking and loading. References to exported Proxy functions are translated as given by the Proxy. Local functions are offset with respect to the machine code location in the flash ROM. Local variables are offset with respect to the location of *Mem*.

updated to the next unresolved reference, or 0x0014. The next lowest unresolved address is 0x0014. The code segment between 0x0008 and 0x0014 is copied, and the resolved address reference is written. This process repeats until there are no unresolved address references in the machine code. Figure 9 shows the final machine code in the flash ROM. Finally, the Proxy gives control to the loadable program by calling `_init`.

6 Evaluation

We evaluate the performance of the proposed approach using three metrics: *code size* of the loadable programs, *overhead* in terms of increased memory footprint and execution overhead, and *energy consumption* associated with the linking and loading operations.

We perform all our experiments using Tmote Sky motes [21]. The Tmote Sky is based on the MSP430-F1611 microcontroller with 10KB RAM and 48KB flash ROM. One notable characteristic of the MSP430 is that it follows the Von Neumann architecture, which places both RAM and flash ROM in the same address space. The Tmote Sky also features 1 MB of external flash which we use to temporarily store loadable programs before we copy them to microcontroller’s internal RAM and flash ROM.

6.1 Test Applications

To measure the performance across the three metrics defined above, we implemented a number of applications in TinyJavaScript and compared them with existing monolithic TinyOS programs. The selection of applications is meant to demonstrate the effects of the interaction between the loadable programs and the underlying Proxy services for different models of use.

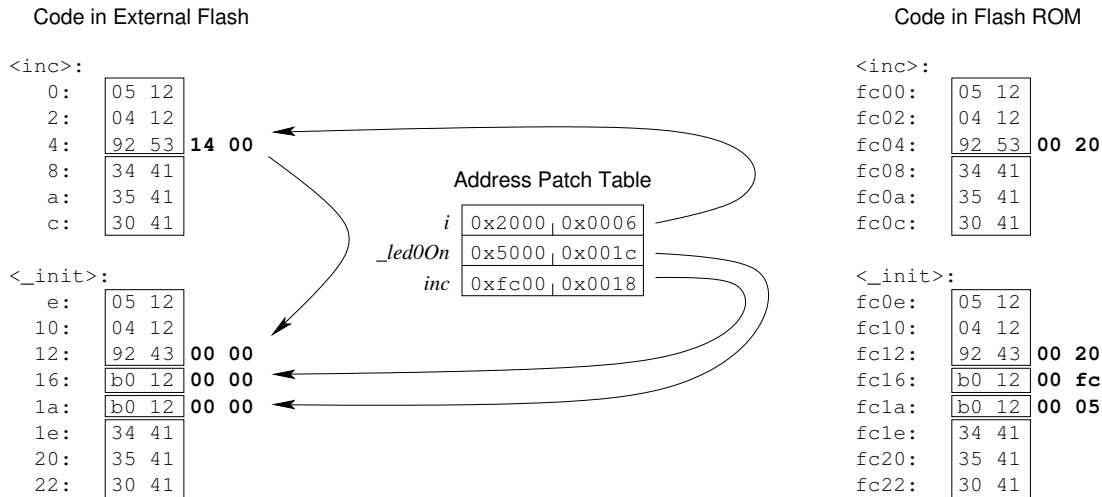


Figure 9: TinyLD chains references that resolve to the same symbol, demonstrated by the arrows. Patching is performed sequentially with respect to the location of unresolved address references in the machine code. The right side of the figure shows patched machine code in the flash ROM.

Blink. This program is a re-implementation of the standard Blink application from TinyOS 2 using loadable code . It uses three timers and three LEDs. The full source is available in Appendix A.1.

RadioCountToLeds. This is a re-implementation of another standard TinyOS 2 application. It uses one timer to implement a 4 Hz counter. Each time the timer fires, the counter is incremented and its value is sent via broadcast over the network. Each mote that receives a packet displays the last three bits of the binary value it contains on its LEDs.

RadioSenseToLeds. This is also a re-implementation of an existing TinyOS 2 application. As its name implies, it is similar to the previous program, except in this case the counter is replaced with a sample from a sensor (DemoSensor).

BaseStation. This program implements a simplified version of the standard BaseStation application from TinyOS 2. This version forwards packets with a single Active Message ID to and from the radio and the serial interface. We implement a queue of twelve packets as the original application does.

Dissemination. This program disseminates a value to all the motes in a network. It is similar to the TinyOS dissemination service [16], as it borrows the ideas of Trickle timers and advertisement suppression based on message overhearing [15]. Upon receiving a new dissemination value from either the serial port or the radio, the mote resets a Trickle timer to advertise the new value with maximum frequency. The full source of the program can be found in Appendix B.1.

Application	Code Size (bytes)	MicroExe Size		ELF Size		RAM Usage (bytes)	Relocations				Allocations	
		(ratio)	(bytes)	(ratio)	(bytes)		All	External Unique	Local Unique	All	Unique	
Blink	44	1.77	78	11.48	896	0	6	6	0	0	0	0
RadioCountToLeds	220	1.30	286	4.71	1348	1	11	10	0	0	3	1
RadioSenseToLeds	296	1.30	386	4.14	1600	5	11	11	0	0	11	2
BaseStation	572	1.23	706	4.02	2840	704	16	9	12	4	62	10
Dissemination	550	1.35	744	4.24	3156	68	12	7	2	1	119	12
Env. Monitoring	1140	1.23	1402	3.63	5092	68	52	22	4	4	119	13
Null	948	1.04-1.11	986-1050	2.73-2.57	2696	2	-	-	20	-	21	3
Blink	2548	1.02-1.13	2610-2886	2.63-2.38	6856	51	-	-	73	-	69	9
Sense	6912	1.01-1.14	7014-7878	2.54-2.26	17832	84	-	-	220	-	126	23
RadioCountToLeds	8064	1.02-1.17	8190-9426	2.86-2.49	23460	197	-	-	313	-	170	38
RadioSenseToLeds	12390	1.01-1.16	12552-14336	2.70-2.37	34000	249	-	-	450	-	226	51
BaseStation	10412	1.02-1.19	10682-12346	3.13-2.71	33456	1429	-	-	420	-	389	70
Oscilloscope	12568	1.02-1.16	12798-14570	2.71-2.38	34648	280	-	-	447	-	268	55
MultiHopOscilloscope	25316	1.02-1.16	25718-29326	2.84-2.49	73056	3075	-	-	906	-	571	122

Table 2: Code sizes of sample TinyJavaScript applications. The ratio in the MicroExe Size column is relative to the Code Size, while the ratio in the ELF Size column is relative to the MicroExe Size. The lower half presents the same data for several standard TinyOS 2 applications. The larger value in the range from the MicroExe Size column is obtained by considering the worse case scenario when all the relocations are unique.

Environmental Monitoring. This program is based on an environmental monitoring application we have deployed [18]. This application periodically samples three sensors: voltage, temperature, and humidity. The measurements are then written to a circular buffer implemented using two flash volumes. When one volume becomes full, the other is erased and then used for new measurements. The measurements stored in these volumes can be retrieved by sending the mote a download request over the radio. The source code for this application is omitted due to paper length limitations.

6.2 Code Size

We compare the size of an application encoded in our MicroExe custom file format to that of the equivalent ELF file. We also compare the size of the MicroExe file to the size of the raw machine code. The first comparison quantifies the benefits achieved by the space optimizations included in MicroExe, while the second indicates the overhead above the bare minimum file size that MicroExe generates.

The size of the MicroExe format depends on three factors: the size of the machine code, the total number of relocations (U_{Reloc}), and the total number of allocations (U_{Alloc}). The size of a MicroExe file is then given by:

$$MicroExe = Code + (U_{Reloc} + U_{Alloc}) \times 4 + 5 \times 2$$

The size of the machine code represents an unavoidable cost unless MicroExe employs data compression algorithms. The additive constant in the formula above corresponds to the fixed size of the metadata section in MicroExe.

The upper part of Table 2 presents code size comparisons for all the applications we developed in TinyJavaScript. As the table shows the overhead of MicroExe is lower than 35% of the raw machine code size (744 bytes vs. 500 bytes for Dissemination). Moreover,

the space optimizations in MicroExe are able to reduce the file size by a factor of three to eleven compared to ELF.

Next, we investigate the overhead of the MicroExe format for programs larger than the ones we developed. To do so, we build several TinyOS 2 applications (shown in lower part of Table 2) and extract the total number as well as the number of unique allocations these programs perform. Based on these values we can estimate the overhead required to build the MicroExe patch table. While the information about the number of allocations is accurate, no external relocations exist because these applications are not loadable programs. Furthermore, due to the same reason, most local relocations are relative to the beginning of the code. As the results in the lower half of Table 2 indicate, this makes the MicroExe format extremely efficient; its ratio to the raw machine code size is never larger than 1.04. We also present the worse case scenario, when all the relocations are unique. In this case, the ratio of MicroExe size over the raw machine code grows to a maximum of 1.19. The only parameter that we cannot estimate with this technique is the size overhead due to callbacks and external relocations, again because we derive the estimates from monolithic applications. However, this is not a big concern because the space they require is small and constrained by the API the Proxy provides.

The results from larger applications are encouraging because they never exceed the maximum overhead observed for the smaller TinyJavaScript applications we developed. Based on these results, we posit that the overhead of MicroExe files will not grow disproportionately to the size of TinyJavaScript applications.

6.3 Runtime Overhead

We measure two types of runtime overhead: execution overhead and overhead in memory use.

The first component of the execution overhead is the one indirection per each boundary crossing between the loadable code and the Proxy. The indirection from the loadable code to the Proxy exists because the address patched in the loadable code is that of a Proxy function and not that of the actual TinyOS component. However, this indirection is crucial when we want to limit the amount of information the Proxy passes to the loadable code (e.g., hiding the `message_t` structure). On the other hand, the overhead associated with other forward indirections as well as most of the callbacks is unnecessary. Nonetheless, we keep both types of indirection because they allow a clean integration of loadable code in TinyOS 2. Nevertheless, this indirection is cheap: it costs 8 cycles ($2 \mu s$ at 4 MHz for the MSP430) and is performed using a `CALL` and a `RET`.

The other source of execution overhead has to do with the buffer copy between the application and the Proxy that occurs when a packet is sent or received over the serial or radio interfaces. We chose to incur this buffer copy to avoid exposing the whole `message_t` structure to the loadable programs. The cost of this operation is proportional to the size of the buffer, with each byte copy requiring 11 cycles. For a buffer of 28 bytes, the default maximum packet size from TinyOS 2, the copy operation represents $77 \mu s$ of execution overhead. This value corresponds to 8.6% of overhead when sending the packet over the

	Blink	RadioCount ToLeds	RadioSense ToLeds
Code size	44	220	296
RAM usage	0	1	5
Proxy + TinyLD			
ROM	12288	21486	17334
RAM	538	761	697
TinyOS 2			
ROM	2618	12782	8452
RAM	55	300	236
Overhead			
ROM	9670	8704	8882
RAM	483	461	461

Table 3: RAM and ROM memory footprints for three applications implemented as loadable code and as monolithic applications. The additional memory used by loadable applications is attributed to the memory size of the Proxy and TinyLD. All the values are in bytes.

radio (896 μ s at 250 Kbps) or 4% when the serial port is used (1.9 ms at 115 Kbps).

The runtime space overhead corresponds to the memory used by the Proxy and TinyLD. This overhead can be further divided into the fixed cost of including the Proxy and TinyLD to the code running on the mote and the variable cost of providing different services through the Proxy.

In order to evaluate the fixed cost of including the Proxy and TinyLD, we compared the ROM and RAM usage of three TinyOS 2 applications for which we have perfectly equivalent implementations using TinyJavaScript. Table 3 summarizes the results of this comparison. We compute the overhead by subtracting the memory sizes for the monolithic implementation from their TinyJavaScript counterparts. As expected, the overhead is fairly consistent in all cases. Blink shows a slightly higher overhead most likely because the small size of the application makes the savings obtained by the global optimizations more relevant².

As mentioned earlier, developers can customize the Proxy to expose only the services that a particular WSN deployment might need. Table 4 shows the RAM and flash ROM overhead imposed by each of the services provided by a Proxy we developed for the Tmote Sky. Most services have a one-time cost associated with instantiating the underlying TinyOS 2 components that correspond to that service. For example, using TinyOS.Radio incurs a fixed cost for instantiating the TinyOS 2 components that start and stop the actual radio. Incremental cost captures the overhead of adding one additional instance of the service (e.g., adding transmission/reception of a new AM type).

²We compiled all the applications with the default compilation optimizations from the TinyOS 2 distribution.

Proxy Services	One-time cost		Incremental cost	
	RAM	ROM	RAM	ROM
LEDs	0	0	6	42
Timer	0	26	18	112
Radio	172	4426	10	332
Serial	54	260	8	234
On-board Sensors				
- Air Temp / Humidity	42	1644	8	184
- Light (S1087PAR)	62	3212	10	76
- Light (S10871TSR)	62	3212	10	76
- Internal Temp	58	3176	10	758
- Battery Voltage	58	3728	10	502

Table 4: RAM and flash ROM memory use by various Proxy services. All the values are in bytes.

Application	Time (ms)		Energy (mJ)	
	A	B	A	B
Blink	15.33	55.83	0.031	0.117
BaseStation	21.73	655.19	0.050	1.383
Dissemination	24.17	653.79	0.054	1.373
Environment Monitoring	30.11	1330.51	0.070	2.796

Table 5: Cost in time and energy for linking and loading. “A” columns represent the cost of pre-processing the patch and callback tables while “B” columns represent the patching and copying of the code.

6.4 Energy Consumption

We estimate energy consumption by measuring the voltage drop on a 10 k Ω resistor connected in series with the mote. The loadable program is initially stored in external flash when we trigger a linking and loading operation by pressing the mote’s user button. Figure 10 shows the current draw trace we captured for the Blink application, while other applications produce similar graphs. The two phases, pre-processing the patch and callback tables and patching and copying of the machine code, are clearly demarked by explicitly turning on all the LEDs at the end of each phase. In both phases the mote draws approximately the same current, however the second phase dominates power consumptions when the machine code size is large.

Table 5 summarizes the energy costs related to linking and loading for four TinyJavaScript applications. We calculate the total energy used during those two phases by integrating the current draw curve and multiplying it with the supply voltage (3 V).

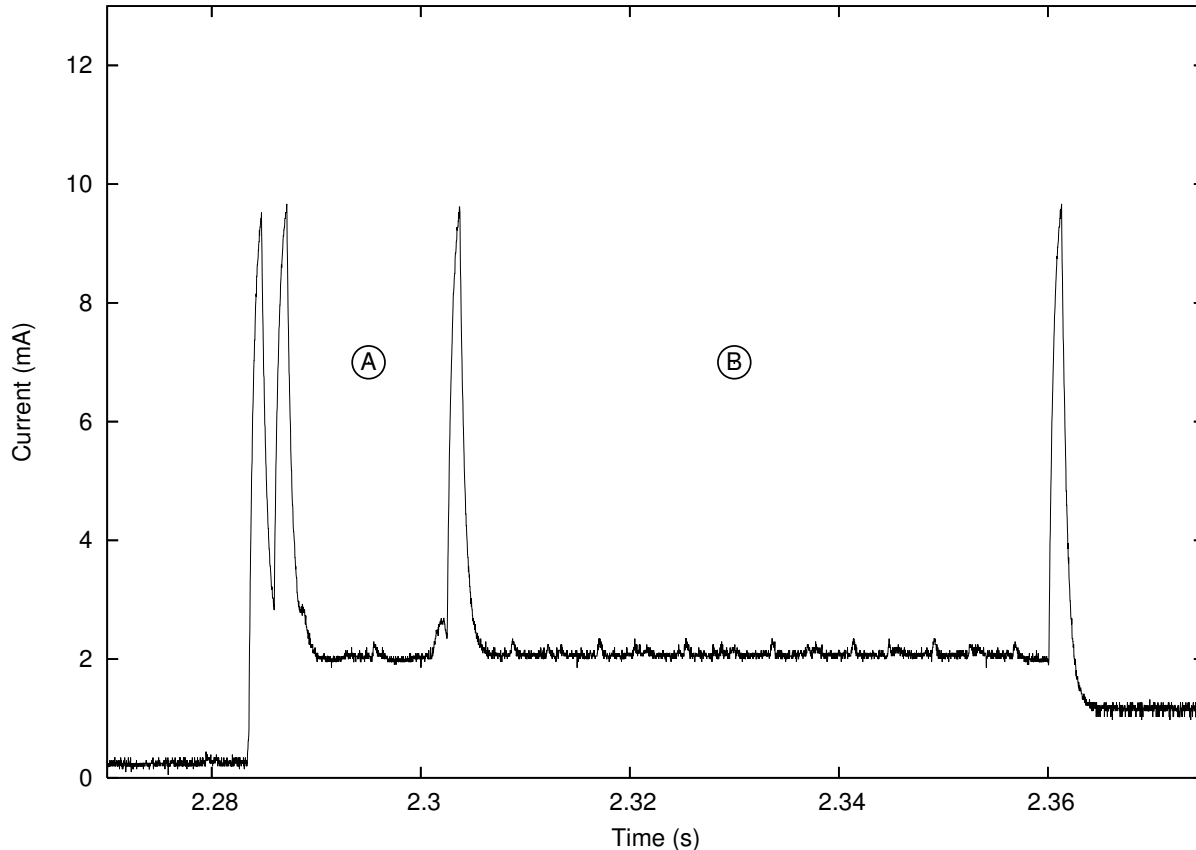


Figure 10: Current draw by a Tmote Sky for linking and loading the Blink application. Phase A (pre-processing the callback and patch tables) requires 15.33 ms, while phase B (patching and copying the code to the flash ROM) requires 55.83 ms. The current spikes are due to turning all the LEDs on.

7 Future Work

We intend to extend the work presented in this paper along a number of dimensions. First, we will develop and deploy additional applications written in TinyJavaScript to evaluate the expressivity of the language and discover any limitations that will require changes to the Proxy API. At the same time, we intend to add support for other execution platforms, among which the ATmega/Mica family of motes is our top priority. While the level of abstraction offered by the Proxy component is admittedly modest, it presents a conscious decision. We want to follow a bottom-up approach, incrementally raising the level of abstraction by exploring more ambitious language designs.

On a different topic, we will extend our work on the simulator outlined in Section 4.4. We believe that such a simulator will complement existing whole-system simulators such as TOSSIM [14]. We envision it being used earlier in the development process, providing a quick and easy way to validate application designs and resolve logic flaws at an earlier stage.

Last but not least, we plan to improve the existing runtime in two ways. First, we will investigate mechanisms to improve its performance by reducing the overhead associated with indirections and buffer copies. Second, we intend to extend the existing runtime with support for unloading programs as well as support for runtime resource checks.

8 Summary

We present a mechanism to introduce loadable application code to the TinyOS software framework by exposing abstractions of the system's resources through a Proxy component. Application code can access these resources by calling functions provided by the Proxy and receiving callbacks to functions it defines. External function resolution happens on the mote at runtime, using TinyLD—a dynamic linker and loader that we developed. The result of this linking process is a binary that combines system and application logic and runs natively on the mote's hardware.

As an example of a high-level language that can leverage the service abstractions that the Proxy exposes, we present TinyJavaScript; an subset of the popular scripting language that provides a more familiar programming paradigm for developing WSN applications. TinyJavaScript programs are translated to equivalent C programs and eventually to object code that runs natively on the target platform, avoiding the overhead associated with interpretation. The benefits of interfacing to the Proxy through TinyJavaScript instead of an existing language, such as nesC, include the availability of high-level languages features (e.g., type inference) as well as the ability to debug and simulate WSN applications in a web browser.

Our evaluation results show that providing these features generates only minor overhead in terms of increased memory footprint and runtime performance. Execution speed is minimally affected, incurring a single additional pointer indirection for each call to functions provided by the Proxy. Finally, runtime linking and loading a new binary requires less than 1.5 seconds, a small cost for an operation that we envision to be infrequent.

In summary, we present a feasible alternative to the way WSN applications are currently developed in TinyOS. While we do not expect that current developers will initially migrate to this paradigm, we hope that it will enable a new community of uninitiated developers to approach the field of wireless sensor networks.

References

- [1] D. M. Beazley. PLY (Python Lex-Yacc). Available at <http://www.dabeaz.com/ply/>.
- [2] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: Design and implementation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, Mar. 2004.
- [3] A. Boulis, C. Han, and M. Srivastava. Design and Implementation of a Framework for Distributed Embedded Systems. In *Proceedings of the First International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, May 2003.

- [4] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 15–28, New York, NY, USA, 2006. ACM Press.
- [5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003.
- [6] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming Wireless Sensor Networks using Kairos. June 2005.
- [7] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (Mobisys)*, June 2005.
- [8] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, Nov. 2004.
- [9] J. Kosy and R. Pandey. VM*: Synthesizing scalable runtime environments for sensor networks. In *Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys 2005)*, Nov. 2005.
- [10] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, 2000.
- [11] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.
- [12] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proceedings of the Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005)*, April 2005.
- [13] P. Levis, D. Gay, V. Handziski, J.-H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, R. S. Cory Sharp, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz. T2: A Second Generation OS For Embedded Sensor Networks. Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin, 2005.
- [14] P. Levis, N. Lee, A. Woo, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS Applications. In *Proceedings of Sensys 2003*, Nov. 2003.
- [15] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self-regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *Proceedings of NSDI 2004*, Mar. 2004.
- [16] P. Levis and G. Tolle. TEP 118 Dissemination. Available at <http://www.tinyos.net/tinyos-2.x/doc/html/tep118.html>.
- [17] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. State-Centric Programming for Sensor-Actuator Network Systems. *IEEE Pervasive Computing Magazine*, Oct. 2003.

- [18] Reference removed to preserve anonymity.
- [19] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of SIGMOD 2003*, June 2003.
- [20] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006)*, pages 212–227, February 2006.
- [21] Moteiv Corporation. Tmote Sky. Available at <http://www.moteiv.com/products/tmotesky.php>.
- [22] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of NSDI 2004*, Mar. 2004.
- [23] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In *Proceedings of the fifth international conference on Information processing in sensor networks (IPSN)*, 2006.

A Blink

A.1 TinyJavaScript Version

```
TinyOS.init = function()
{
  TinyOS.Timer[0].startOneShot(250);
  TinyOS.Timer[1].startOneShot(500);
  TinyOS.Timer[2].startOneShot(1000);
}

TinyOS.Timer[0].fired = function()
{
  TinyOS.Led[0].toggle();
}

TinyOS.Timer[1].fired = function()
{
  TinyOS.Led[1].toggle();
}

TinyOS.Timer[1].fired = function()
{
  TinyOS.Led[1].toggle();
}
```

A.2 C Version

```
#include <sys/inttypes.h>

extern void _timer0startPeriodic(uint32_t dt);
extern void _timer1startPeriodic(uint32_t dt);
extern void _timer2startPeriodic(uint32_t dt);
```

```

extern void _led0Toggle();
extern void _led1Toggle();
extern void _led2Toggle();

void _init()
{
    _timer0startOneShot(250);
    _timer1startOneShot(500);
    _timer2startOneShot(1000);
}

void _timer0fired()
{
    _led0Toggle();
}

void _timer1fired()
{
    _led1Toggle();
}

void _timer2fired()
{
    _led2Toggle();
}

```

B Dissemination

B.1 TinyJavaScript Version

```

var minPeriod;
var maxPeriod;
var curPeriod;
var suppCounter;
var counter;

var curKey;
var curVersion;
var curValue;

rpkt_out = {'key':uint32_t,
            'version':uint32_t,
            'value':uint32_t};
rpkt_in = rpkt_out;
spkt = {'key':uint32_t, 'value':uint32_t};

TinyOS.init = function()
{
    curKey = 0x08070605;
    curVersion = 0;
    curValue = 0;

    minPeriod = 1000;
    maxPeriod = 1024000;
    suppCounter = 1;
    curPeriod = minPeriod;
    counter = 0;

    TinyOS.Radio.start();
}

```

```

TinyOS.Radio.startDone = function(error)
{
  TinyOS.Timer[0].startOneShot(curPeriod);
}

function broadcast()
{
  rpkt_out.key = curKey;
  rpkt_out.version = curVersion;
  rpkt_out.value = curValue;
  TinyOS.Radio[0].send(0xFFFF, rpkt_out,
                      sizeof(rpkt_out));
}

TinyOS.Timer[0].fired = function()
{
  TinyOS.Led[0].toggle();
  if (counter < suppCounter) {
    broadcast();
  }

  counter = 0;
  curPeriod = curPeriod * 2;
  if (curPeriod > maxPeriod) {
    curPeriod = maxPeriod;
  }
  TinyOS.Timer[0].startOneShot(curPeriod);
}

TinyOS.Serial[0].receive = function(payload, len)
{
  spkt = payload;
  if (spkt.key == curKey) {
    curVersion++;
    curValue = spkt.value;

    if (curValue % 2 == 0) {
      TinyOS.Led[2].on();
    } else {
      TinyOS.Led[2].off();
    }

    counter = 0;
    curPeriod = minPeriod;

    TinyOS.Timer[0].startOneShot(curPeriod);
  }
}

TinyOS.Radio[0].receive = function(payload, len)
{
  rpkt_in = payload;
  if (rpkt_in.key == curKey) {
    if (rpkt_in.version > curVersion) {
      TinyOS.led[1].toggle();
      curValue = rpkt_in.value;
      curVersion = rpkt_in.version;

      if (curValue % 2 == 0) {
        TinyOS.led[2].on();
      } else {
        TinyOS.led[2].off();
      }

      counter = 0;

```

```
    curPeriod = minPeriod;
    TinyOS.Timer[0].startOneShot(curPeriod);
} else if (rpkt_in.version == curVersion) {
    counter++;
} else if (rpkt_in.version < curVersion) {
    broadcast();
}
}
}
```