EcoCast: Interactive, Object-Oriented Macroprogramming for Networks of Ultra-Compact Wireless Sensor Nodes

Yi-Hsuan Tu, Yen-Chiu Li
[1]Department of Computer Science
National Tsing Hua University, Taiwan
{cindyduh, joyce7216}@gmail.com

Ting-Chou Chien, Pai H. Chou[1,2]
[2]Center for Embedded Computer Systems
University of California, Irvine, CA USA
{tchien, phchou}@uci.edu

ABSTRACT

EcoCast is an execution framework for macroprogramming of wireless sensor networks. Users access sensor nodes as dynamic objects in Python by invoking methods on them without being concerned with network protocols, and type marshalling and demarshalling ensure proper data access. EcoCast extends Python's functional programming primitives `map()`, `reduce()`, and `filter()` to macroprogramming with several synchrony semantics and job-control options. EcoCast can compile Python lambda expressions and functions to run on the nodes at native speed without requiring most users to write code in C or assembly, and it patches the firmware transparently without rebooting. The use of Python also facilitates host-side application development by enabling developers to take full advantage of the rich code libraries and data structures in Python. Experimental results show the reprogramming and execution latencies of EcoCast scale well over the size of the network while occupying a small memory footprint.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems—*wireless sensing systems*; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*

General Terms

Design, Languages

Keywords

wireless sensor network, macroprogramming, dynamic loading

1. INTRODUCTION

Macroprogramming may be a crucial technology in making wireless sensor networks (WSN) a truly useful tool for a wide range of applications. In contrast to traditional programming for individual sensor nodes, macroprogramming entails writing a program for groups of nodes in the network. It promises to enable the user to command a large number of nodes at the high level without having to express error-prone details at the low level. Moreover, the high-level knowledge captured by a macroprogram can potentially lend itself to global optimization opportunities that would otherwise be difficult to discover with only node-local knowledge.

As appealing as the idea may sound, macroprogramming still remains a challenge today. To be truly successful, macroprogramming must strike a balance between its technical merits and practical considerations. The runtime system must be able to perform firmware patching over a wireless link, and we believe it should be able to execute code *natively* for efficiency and *interactively* for direct feedback. Perhaps the most important consideration is the choice of the language: it should be sufficiently expressive (i.e., Turing Complete) with both node-level and macro-level constructs in a clean syntax and can be smoothly integrated into a well-supported programming environment with a large user base. Moreover, the runtime support should be lightweight enough to run on some of the more resource-constrained WSN platforms.

To meet these challenges, we propose EcoCast, an interactive object-oriented macroprogramming framework. From a command-line shell on a host computer, one can issue a macroprogramming command to run on multiple nodes as a group and individually. We choose the Python language for its familiar syntax, dynamic object orientation, interactive scripting and batch execution, macroprogramming constructs, and rich library support for general-purpose application development. Nodes and groups of nodes in the network are represented as objects in Python on the host computer and are accessed via method calls. Python code can also be compiled and loaded on-demand transparently and executed on the nodes immediately, without having to rely on a different language for node programming. For macroprogramming, we extend *functional programming* constructs to working with groups of nodes: `map` (returns a list of return values of invoking a function on a group of nodes), `reduce` (returns the result of reduction operation on a group of nodes), and `filter` (returns a subgroup of nodes satisfying a given condition). EcoCast also supports different synchrony semantics for parallel execution.

This paper is organized as follows. We survey previous work in related areas, followed by an overview of the proposed system. We explain the scripting methodology, compilation, reprogramming, and multi-hop networking. Experimental results show EcoCast to be effective in terms of scalability of the number of nodes, round-trip latency, topology discovery, and multi-hop latency, especially considering the very limited resource on the sensor nodes.

2. RELATED WORK

Our proposed EcoCast system requires integration of several key features: remote firmware update, interactive scripting, and macroprogramming. This section surveys related work in these individual areas. Most systems proposed to date support node-level, compilation style of software development, whereas EcoCast supports interactive style of macroprogramming by scripting.

2.1 Remote Firmware Update

Although macroprogramming can be done in the traditional style of compiling the entire program and writing the firmware before execution, it is impractical as the cost of correcting even a minor error in the firmware can be prohibitive. Instead, it is almost mandatory to be able to update firmware remotely. Remote firmware update can be divided into full image replacement and binary patching. The former completely replaces the firmware image (except for the bootloader) and requires a reboot, and this has been done for single-hop [8, 9] and multi-hop [20, 21, 34] networks, including epidemic dissemination protocols [16]. However, full image replacement can be costly, and patching techniques have been proposed, by transmitting a *diff script* or an *edit script* containing patching commands. Proposed techniques use diff [18, 32], block-level comparison [17], by finding fixed-size shared blocks [36], or by leveraging higher-level program structure to generate more concise scripts [19]. Another alternative to program image replacement is to use loadable modules for reprogramming. Loading a module entails resolving references to symbols and can be divided into *pre-linking* [11], *dynamic linking* [10, 25], and *dynamic loading* [6, 13]. Some remote update methods may be more structured and limited to constant modifications [2] rather than general code replacement.

To be applicable to interactive execution, the remote reprogramming technique should update multiple nodes at a time [8, 9, 16, 20] rather than a single node at a time [18]. It should not require rebooting after a new program image has been installed or patched, or else program state will be lost. Also, it should be able to confirm successful reprogramming before proceeding to the next step in a macroprogram [8].

2.2 Shells and Scripting Systems

Scripting has been shown to enable programmers to achieve 10× the productivity over *system programming* languages such as C or Java [29]. Several recent works are suggesting *scripting* as a new trend towards programming and control of WSN. A shell is an interactive command-line interpreter for a scripting language, while a scripting language is one whose primitives consist of higher-level functions or programs rather than low-level instructions. Scripting languages may be executed interactively or in batch.

Trying to adapt the idea of scripting to WSN, however, requires much more than porting code and subsetting the language. For instance, SensorWare [4] includes an interpreter for a subset of the Tcl [28] scripting language on the sensor node itself. This is very powerful but requires a 32-bit CPU with several hundred kilobytes of program memory plus data memory, too costly for most sensor nodes. Several other systems also support shell on the nodes, including Mantis, Contiki, and TinyOS. These are useful for users to invoke commands interactively, but they are add-on features and occupy additional program and data memory.

One way to overcome the resource limitation on the nodes is to run the shell on the host computer, rather than on the node. LiteOS [5] provides LiteShell, a command-line interpreter that supports UNIX-like commands and maps the sensor network to the file system for invoking commands and updating firmware. The commands are also available as APIs for developing user applications on the host PC. The idea of *fat client, thin server* is also central to Marionette [37] and EcoExec [15]. Both provide a Python-based shell on the host while the node implements primitives for function invocation and memory access primitives. The former is for debugging during development phase and relies on an external reprogramming scheme [16] for whole image replacement; whereas the latter can patch on-demand, by involving an optimizing compiler in the loop if necessary, and continue executing commands interactively without rebooting. Both Marionette and EcoExec require users to write code in a separate language (nesC and C, respectively) at the node level, but neither supports macroprogramming.

Another issue is synchrony: nodes may take a different amount of time to finish executing a function, or the results may be invoked at different times due to the different number of hops that the packets may need to travel. Whether it is necessary to synchronize the nodes before the next command is allowed to proceed becomes an important consideration in such a parallel language. EcoExec with its serial semantics by default assumes synchronous (fully blocking) semantics, which is the most conservative and at the same time slowest.

2.3 Query Systems and Macroprogramming

Query-like interfaces have been proposed for WSN [3, 23, 38], including an SQL-like interface [24] for retrieving data from the WSN as if it were a database. All query systems are interactive. Similar to a macroprogram, a single query can potentially involve many nodes in the network and would otherwise be complex to specify at the node-level abstraction. However, query systems are limited to propagating data from the nodes upstream to the host and are not designed to support programming in general. First, a query system has no concept of compiler or linker to properly handle program fragments. Second, even if the query mechanism can disseminate code patches as data downstream to the nodes, doing so over a general query protocol would be highly inefficient compared to one designed specifically for remote reprogramming purpose, such as the protocol described in Section 6.2.

Macroprogramming systems support the use of high-level programming constructs on groups of nodes, rather than lower-level constructs for individual nodes. Several works impose a task graph or dataflow graph model of computation on top of a set of nodes [1, 30, 31]. They serve to abstract away the communication details and capture the data or control dependency among individual or groups of nodes; however, the actual functionality at the system level and node level needs to be expressed in another language.

Unlike graph models, Regiment [26] and Region Streams [27] support macroprogramming using *functional reactive programming* construct both spatially (over regions of nodes) and temporally (samples over time). They support arithmetic operations and pure functions on sensed data but not invoking functions in general or performing operations with side effects on the nodes. Kairos [12] also provides a C-like macroprogramming environment for vehicle tracking. The use of a special language is expressive at the macroprogramming level, but integration with the rest of the software development (such as GUI) would require more effort. MacroLab [14] uses a Matlab-like language, and its functions can be invoked remotely, though incremental patching is not a built-in feature. Frameworks such as EnviroSuite [22], which maps nodes to objects in a programming language, can facilitate such software integration efforts, but they provide less macroprogramming support. Our proposed EcoCast attempts to balance the expressivity of functional style macroprogramming with the practicality of node/group-to-object mapping in the popular Python language.

3. SYSTEM OVERVIEW

EcoCast is a lightweight, interactive, object-oriented macroprogramming framework that spans a host computer and a WSN plat-

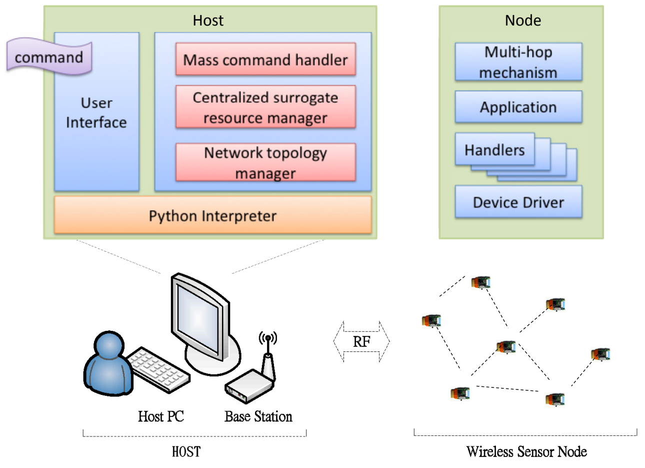


Figure 1: System overview.

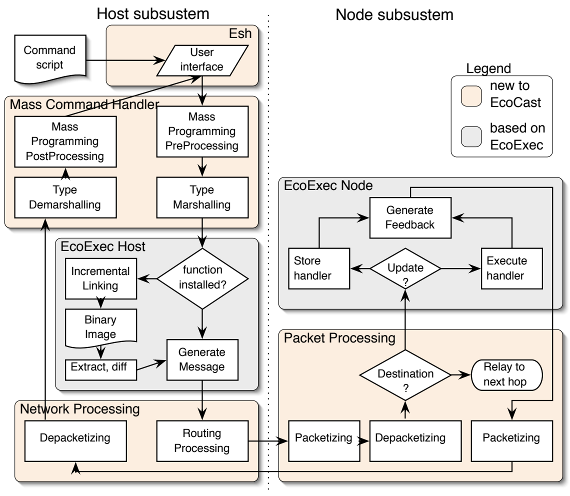


Figure 2: Execution Flow of EcoCast.

form. We use the source code of EcoExec [33] as a starting point. This section first describes our first target platform, followed by an overview of the execution flow.

3.1 Wireless Sensing Platform

EcoCast can be ported to a variety of wireless sensing platforms as long as they can be organized as three subsystems: *nodes*, *base stations*, and the *host*. An overview of these subsystems is shown in Fig. 1. Although EcoCast contains features of EcoExec, the newly added macroprogramming and multi-hop features could not fit in the 4KB program memory of the original Eco platform. Therefore, we ported the code and implemented EcoCast on a different platform that is less resource-constrained but still considered ultra-compact. This section describes this platform.

3.1.1 Nodes

We implemented EcoCast on the EcoSpire [7] sensor nodes. The MCU of EcoSpire is the Nordic nRF24LE1, which contains an 8051-compatible core integrated with the nRF24L01 radio in the 2.4 GHz ISM band, a multichannel analog-to-digital converter, and general-purpose I/O pins. The nRF24L01 radio supports auto-ack and auto-retransmission, and the nRF24LE1 MCU contains 1 KB on-chip data RAM and 16 KB flash as program memory. EcoSpire contains an on-board triaxial accelerometer, and it includes an expansion connector for a variety of sensor and actuator modules.

3.1.2 Base Station

The hardware of the base station in the experiment is built by connecting a Nordic nRF24L01 2.4 GHz RF transceiver module to a Freescale DEMO9S12NE64 evaluation board via SPI . The MCU on this board is the HCS12 (16-bit) with 64K bytes of flash and 8K bytes of RAM. It supports TCP/IP via its Fast Ethernet (10/100 Mbps) MAC/PHY transceiver.

3.1.3 Host Computer

The host subsystem runs on a conventional PC that can communicate with the nodes via the base station(s). The host computer runs the wireless shell that enables programmers to interact with the nodes over the air. The host also maintains a suite of tools that help the shell keep track of the state of the nodes as well as code generation and optimization. The tools include the compiler, linker, runtime estimator, node database, and version control. These tools are invoked automatically and transparently to the users as they type commands into the shell. The execution sequence is explained next.

3.2 Execution Flow

The user can either type in a command interactively or run an application program (script) written in Python to call our API library. In either case, the same command is invoked, and EcoCast handles them in exactly the same way. The steps in the execution are node-handle creation, command processing, and message issuing. The flowchart of EcoCast is shown in Fig. 2.

3.2.1 Joining and Leaving a Network

When the host system starts up, it first searches for available sensor nodes to join the network. We assume that every node has been programmed with a unique ID out of factory. On power-up, a node attempts to join a network governed by a host computer. Once a node and a host agree to connect, the host constructs a *node handle* object in Python with the node's unique ID and creates a new entry for it in the node database. Subsequently, all operations on the node, including function invocation, code update, configuration changes, etc, are all done through the node handle object. This idea is analogous to the concept of *file handles* used by most programming languages for reading and writing files, except these calls translate into wireless communication messages between the host and the node. If the host cannot reach a node after a timeout period, then the host system removes the corresponding node handle object and marks so in the node database. Attempts to use such a handle causes an exception to be raised.

3.2.2 Language Wrapping

Similar to other language wrappers for middleware systems, our wrapper also performs *type marshalling* and *demarshalling* when translating between Python code and the node's native code. Native code includes not only drivers at the low level and library routines but also user-written Python code that is translated into C and subsequently compiled. In all cases, function prototype information (types of the parameters and return value) is either explicitly written, inferred, or name-encoded. For each type of target MCU, we create an architecture profile in terms of the endian and word size to enable EcoCast to handle the checking and conversion or to raise an exception if it cannot convert.

Table 1: Comparison between EcoExec and EcoCast

Feature	EcoExec	EcoCast
Remote firmware patching	single node	group
Code memory	EEPROM	Flash, RAM
Function code	C	C, Python
Remote function invocation	Yes	Yes
Job control, scope	No	Yes
Functional programming on nodes	default (serial)	parallel
Type marshalling and demarshalling	No	Yes
Multi-Hop Networking	No	Yes

3.2.3 Code Swapping and Dynamic Compilation

In addition to serving as a language wrapper for function invocation and type marshalling, EcoCast also performs several OS functions on the host on behalf of the nodes, the most important of which is host-assisted code swapping. Dynamic compilation is also performed as a step if necessary. Before invoking a function, EcoCast checks the state of the node's firmware image to see if the target function is in memory. If not, then it attempts to swap in the code first, if the binary exists. If the binary does not exist, then it attempts to compile and link the code from either Python code fragments or library source code. It needs to perform incremental linking on the new function to produce a new binary image.

3.2.4 Communication and Synchronization

Ultimately, everything the host needs to do to a node is done by sending messages via the base station that the node is associated with. Messages from the host can either specify that new code be installed or a target function be invoked. A single command in Python may turn into commands to multiple nodes and gathering results from their invocations. EcoCast not only attempts to optimize such communication patterns by broadcasting and scheduling, but also supports several different user-specifiable semantics, including fully blocking and nonblocking execution styles.

4. SCRIPTING

This section describes the details of interactive execution involving the host-side scripting environment. EcoCast provides a scripting environment as the primary way for users and application programs (e.g., GUI) to interact with the sensor nodes. It consists of a class library at a higher level for the user to access the sensor network, a shell called Esh for interactive access, and a class library at a lower level for runtime support. Table 1 summarizes key enhancements of EcoCast over EcoExec.

4.1 Scripting Language

We use Python[1] as our scripting environment for its clean syntax and rich feature support. It can be run two ways: interactive mode and batch mode. In interactive mode, users can enter commands as Python statements directly into Esh (Section 4.2). Variables in Python need not be declared before they are used, and they track the references to objects that are self descriptive (and thus considered dynamically typed). Typing the name of a variable in interactive mode causes the object to be "rendered" in its string representation. Thus, in interactive mode,

```
>>> x = 3   # no need to declare x
>>> x       # displays its value as text
3           # interpreter calls int's __repr__ method
```

[1]We mean Python 2.6 as of this writing, instead of Python 3.0, which is an *intentionally backwards incompatible* release.

The runtime system keeps track of the types of the objects dynamically and enforces their consistent use. One direct advantage is that the same code is thus reusable over a wide range of types without *templates* (C++) or *interfaces* (Java). This enables Python's built-in *list* data structure to contain objects of any type, simply by enclosing them in square brackets, such as `["hello", 3, 2.95]`.

Python is object-oriented with its support for classes, inheritance, method invocation, and instantiation. Objects may be instantiated simply by calling the constructor with the parameters, in the syntax of function calls. The rest of this subsection explains our use of data structures and macroprogramming constructs in Python.

4.1.1 Node Handles and Group Handles

A *node handle* is a data structure through which the program can access the node, analogous to a *file handle*. To create a node handle object in EcoCast, one instantiates the `ecNode` class with the statement `var_name = ecNode(id)`, where `var_name` is the variable name that represents this node instance, and `id` is the node's network address. Subsequently, an application program can access the node by making Python method calls on the node handle object. Operations on attributes of such a node-handle object will have the same effect as directly manipulating them in the application program during runtime. One can use the `object.method()` and `object.attribute` syntax to invoke functions on the node and to access (*get* and *set*) attributes on the node. These accesses are translated into a sequence of actions on the host, ultimately reaching the nodes, and getting response back, as explained in Section 3.2. By treating nodes as objects, programmers can build complex applications such as graphical user interface or data analysis programs easily without getting bogged down with the details of WSN programming.

In interactive mode, the Python shell renders an expression as a string to give the user instant feedback. For built-in types such as `int` and `str`, the values are rendered in the literal form such as `3` and `"hello"`. For user-defined types (namely classes), the Python shell calls the special `__repr__` method, which can be defined by the user to return a string representation for the object's value. In our case, we define it to be the string in the constructor syntax with the value of the node ID. For example,

```
>>> x = ecNode(123)    # instantiate a node handle
>>> x                  # prints the string returned by
ecNode(123)            # the x.__repr__() special method
>>>
```

Note that in the current EcoCast implementation, node handles are supported only at the macroprogramming level rather node level. That is, user-defined functions to run on a node are not to reference other nodes, although library and synthesized code written in C may reference other nodes at a lower level using node IDs.

In EcoCast, a *group handle* is used to construct a group of nodes. The `ecGroup` class is instantiated with the statement `g = ecGroup(L)` where `L` is a list of node IDs or handles. The nodes in the group listen to a specific channel, and each node is assigned a group ID. The `ecGroup` handle can be reused in several macroprogramming constructs without having to reconstruct a new group each time.

4.1.2 Functional and Macroprogramming Constructs

Python provides `map`, `filter`, `reduce` as constructs for functional programming. EcoCast provides the corresponding functions for macroprogramming sensor nodes. In Python,
$\boxed{\texttt{map(f, A)}}$ is equivalent to `[f(A[0]), f(A[1]), f(A[2])...]`, that is, forms a new list with the return values of calling function `f` on every element of the list `A`. A more general form of `map` takes mul-

tiple lists and is equivalent to calling `f` with arguments taken from the corresponding elements of the lists,
e.g., `[f(A[0], B[0], C[0]), f(A[1], B[1], C[1]), ...]`.

`filter(f, L)` computes the subset of list members that satisfy a condition. That is, it returns a new list of members *x* of `L` such that `f`(*x*) evaluates to true.

`reduce(f, L[, Init])` performs a binary operation `f` on the first two members of `L`, and applies the `f` operator on the resulting value with the subsequent member of `L` for the rest of the list. For example, if `L` has five elements, then it evaluates to
`f(f(f(f(L[0], L[1]), L[2]), L[3]), L[4])`.
If the optional third parameter `Init` is provided, then `L[0]` in the expanded expression is replaced with `f(Init, L[0])` instead.

Functional programming constructs often make use of lambda expressions for defining anonymous functions for convenience. For instance, `lambda x,y: x+y` is an anonymous function that returns the sum of its two arguments. It can be passed in place of `f` to the `reduce` example above to compute the sum of the list members.

We extend the concept of functional programming on lists of data to macroprogramming on groups of nodes.

`ecMap(f, GH)` is equivalent to `[GH[0].f(), GH[1].f(), ...]`, that is, the list of return values from calling method `f` on every node in `GH`, which may be either a group handle or a list of node handles. A more general form of `ecMap()` takes additional lists whose members are passed as parameters to the corresponding calls.

`ecFilter(func, GH, f)` constructs a list from those elements `GH[i]` of `GH` for which `func(GH[i].f())` returns true. Note that if the `f` has arguments, the parameters can be added following the `f`.

`ecReduce(func, GH, f)` calls function `func` on two arguments cumulatively over `func(GH[i].f())` from left to right, so as to reduce the list to a single value. Note that if the `f` has arguments, the parameters can be added following the `f`.

Note that in the `ecMap` operator, `f` is "local" to each node by referencing a method by name. Both `ecFilter` and `ecReduce` also include a call to method `f` whose result is used for filtering and reduction purposes, respectively. On the other hand, `func` is "global" as either a function passed by reference or a lambda expression. We show a simple example using the commands above.

```
1  >>> listAll()
2  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
3  >>> GH = ecGroup([1,2,4,5,7,9]) # group based on IDs
4  >>> GH
5  ecGroup([1, 2, 4, 5, 7, 9])
6  >>> ecMap(readTemp, GH)
7  [25.5, 25.3, 25.3, 25.5, 25.2, 25.3]
8  >>> ecFilter(lambda x: x < 2000, GH, readADC)
9  [ecNode(1), ecNode(2), ecNode(4), ecNode(9)]
10 >>> ecReduce(lambda x, y: x + y, GH, readTemp)
11 177.5
```

On Line 1, the function `listAll()` lists the all IDs of nodes that are found and bound to the host during the start-up process of the Host subsystem. Line 3 constructs a group `GH` of nodes with IDs `[1,2,4,5,7,9]`. Line 6 calls `readTemp` method on each node in `GH` to read the temperature and return a list of the results. Line 8 constructs a list of nodes from `GH` whose result of calling `readADC` method is less than 2000. On Line 10, the `ecReduce` command sums the result of calling `readTemp` method on every node in `GH`. An average could be obtained easily by dividing it by `len(GH)`.

In this example, `readADC()` is a lower-level function that returns the raw value from the ADC. In contrast, `readTemp()` is a host-local method that invokes a lower-level function on the node to obtain the raw ADC value (a 12-bit int in this case) and converts it to a floating-point value. In either case, most users need not be concerned about whether the floating point conversion was done on the node remotely or on the host locally.

As another example, with these functional programming constructs, a tracking application can be written as simply as

```
def PEG():
  # Request all nodes to do one minute sensing
  # n = nodes that have average readings > THRESHOLD
  n = ecFilter(lambda x: x > THRESHOLD, GH, readADC1Min)
  # Request nodes to do ACTION
  ecMap(ACTION, n)
GH = ecGroup(getNodes())
ecMap(setSamplesPerSec, GH, [20] * len(GH))
task = repeated(1, PEG)  # obj to spawn a new thread / sec
task.start()
time.sleep(60)
task.stop()
```

These examples show nodes to be passive by executing commands upon host request and returning the results to the host. In general, nodes may be active in periodic data sampling or actuation, but these actions are decoupled from serving host requests.

4.2 Esh Constructs

EcoCast provides a command line interpreter named Esh (for "EcoCast shell"). It accepts the full Python syntax in interactive mode, plus extended syntax for the purpose of job control and scoping for macroprogramming and source-code interception. Although users can directly load the class library and run everything from the Python prompt, the reason for this additional layer is to intercept constructs that must be processed before invoking the underlying Python interpreter. Specifically, inline C code can be extracted and preprocessed this way. The rest of the EcoCast classes and API can be used directly by any other Python program without going through Esh.

4.2.1 Job Control in Esh

Esh supports running background jobs while executing interactively. To run a job in the background, the user simply types the command as usual, followed by `&`, similar to most Unix-style shells. The `jobs` command can then be used to list the command names, execution status, and the result of all the background jobs. The `getResult(id)` command returns the execution result of a specific background job with *id*. For example:

```
>>> n.readADC() &
>>> jobs

2 Background Job(s)
 No.          Command      Status    Result
============================================
  1 ecMap(readTemp, nodes)    Running   None
  2          n.readADC()       Done     2344
>>>
```

4.2.2 Scoping commands

Esh provides commands that change the default scoping of the symbols so that those inside a designated node can be visible and accessed directly without having to qualify them with the node instance first. These can be very useful for executing a series of commands associated with a node or a list of nodes.

Table 2: Summary of EcoCast commands

Command	Type	Meaning
`listAll()`	EcoCast API	list IDs of found nodes
`getNodes()`	EcoCast API	obtain a list of node instances
`ecNode`	EcoCast class	open connection to one node
`ecGroup()`	EcoCast class	open group connection(s) to node(s)
`ecMap()`	EcoCast Macro-prog. API	`map` function for node(s)
`ecFilter()`	EcoCast Macro-prog. API	`filter` function for node(s)
`ecReduce()`	EcoCast Macro-prog. API	`reduce` function for node(s)
`&`	Esh qualifier	run a command as a background job
`jobs`	Esh command	show the information of background jobs
`scope...end`	Esh block	set symbol scope to node or nodes
`extern`	Esh qualifier	enable access global variable in symbol scope of node or nodes

`scope` [*node instance* | *list of node instances* | `ecGroup` *instance*]
    list of statements
`end`

When user enter the `scope` command followed by *a node instance* or *a list of node instances*, all the functions or variables in the commands entered later are mapped to the attribute on the nodes.

While inside a `scope` block, the `extern [`*variable name*`]` command enables access to a *variable* in the global scope. Here is an example:

```
1  >>> t = 0           # global
2  >>> n = ecNode(1)
3  >>> scope n         # set scope to n
4  ... extern t        # access global
5  ... t = readTemp() # n.readTemp()
6  ... t
7  25.4
8  ... end
```

Line 1 initializes a global variable `t`. From Line 5 to 8, all symbols except `t` refer to those inside node `n`, as specified by the `scope` statement on Line 3 and `extern t` on Line 4. `readTemp()` on Line 5 is equivalent to `n.readTemp()` without the `scope` command. Note that if `n` on Line 3 is replaced with a list, then each statement in the `scope` block turns into an `ecMap` of the method to the elements (nodes) of the list.

Table 2 shows the classes, macroprogramming primitives, and APIs used in EcoCast. The node level methods are not list here, and users can get the details using `dir()`.

4.3 Method Dispatching and Attribute Access

EcoCast performs two tasks as the language wrapper: type marshalling and code swapping.

4.3.1 Type Marshalling

Type marshalling is the task of ensuring that the sender or caller's data type is properly matched with that of the recipient or callee, by conversion at runtime if necessary. EcoCast has access to all the prototype information for the functions and global variables, and it also can look up the architecture model for information such as the endian, word size, and their mapping to Python ones. Every native and non-native (e.g., 12-bit ADC int) type of every architecture can be modeled as a Python class with the proper conversion operators. For instance, EcoCast can provide an unsigned, 16-bit, big-endian integer class as follows:

```
class UInt16big: # unsigned 16-bit, big-endian int
    def __init__(self, n): # constructor
        if (type(n) == type(1)): # type of 1 is int
           # ensures value in range or throw exception
           # store in self._rawData
        elif # other types
           # handle other types
    def __int__(self): # typecast to built-in int
          # form int from self._rawData and return
```

Note that by representing MCU data types this way in Python on the host, the underlying Python interpreter automatically invokes the proper type conversion operator accordingly. In the example of `UInt16big`, any value going to the node can be specified using just a generic `int` in Python; and any returning value automatically can be cast into the proper type, all without burdening the user with having to know all the type variants. For instance,

```
>>> n.setSamplesPerSec(20)
>>> n.bitResolution
12
>>> n.readADC()
15
```

The user does not need to be concerned with whether the samples per second parameter is represented as an 8-bit, 16-bit, or 32-bit signed or unsigned int. If the value is not one that can be handled by the underlying hardware, then the user gets an exception. Conversely, the user also does not need to worry about the data type returned by the `readADC` function, which may return 12-bit or 14-bit values – it automatically gets cast into the proper type as needed.

4.3.2 Code Wrapping

Code wrapping is a way for the node-handle object to abstract away implementation details as to *where* the code associated with a node object is implemented. Some code and attributes may be maintained (or "cached") on the host and therefore can be executed efficiently without incurring expensive wireless communication. In the example above, the `setSamplesPerSec` and `readADC` functions reside on the node, but both may be locally intercepted and optimized. This is because if there was a previous call to `setSamplesPerSec` with the same value, and no other host has altered the samples per second attribute, then EcoCast can simply return without actually making the remote call. Similarly, if `readADC` is called five times within one second, but it had been configured to be sampling at once per minute, the user has the option of returning the cached value without communication. These are analogous to standard I/O buffering on general-purpose computer systems.

Another form of abstraction is the support of convenient syntax for attribute access. If `n.bitResolution` is used as an R-value (i.e., on the right-hand-side of an assignment, or passed (by-value) as a parameter), then the underlying Python interpreter automatically calls the special method `n.__getattr__("bitResolution")` to compute the value. Similarly, if `n.bitResolution` is used as an L-value (i.e., in the context of `n.bitResolution = 12`), then Python calls the special method `n.__setattr__("bitResolution", 12)` automatically instead. This mechanism enables the node handle to appear as if it were the node itself.

5. COMPILATION AND LINKING

EcoCast performs compilation on two types of code for the node: C source code and Python code. C programs are written in a style that is native to the node's MCU architecture and is inherited from EcoExec. Python code is normally interpreted in the context of the shell on the host PC itself. EcoCast adds the ability for the user to write Python code to be executed on the node. EcoCast would rewrite the Python code into C and invoke the C compiler. Then, EcoCast processes the `.map` file to perform linking incrementally so that a small patch file can be generated.

5.1 Compilation from Python

Instead of going to C language, the user can actually write Python functions and expressions that are then translated into C to be compiled to run on the node. This is especially useful for specifying conditions such as threshold or triggering conditions, and these functions are often passed as the first parameter to a macroprogramming construct. In addition to expressions, assignment statements, named functions and anonymous functions (lambda), we also support structured control flow in the form of if/else and for loops with known iteration bounds. A *named function* in Python has the syntax

```
def function_name(parameter_list):
        statements
        return expr
```

One problem is that Python parameters are dynamically typed and cannot always be inferred. In those cases, our solution is to require the use of naming convention to encode the types of the parameters and the function. We put a prefix before parameter name to indicate the datatype:

Prefix	C_	UC_	I_	UI_	F_
C Type	char	unsigned char	int	unsigned int	float

The types of all local variables are automatically inferred and so as the return value. To facilitate type inference, we requires that the type of a variable to remain constant. Once the types are known, then it is easy to convert most expressions and statements from Python to C.

5.2 Incremental Linking and Code Swapping

Linking is the step of assigning addresses to symbols after the program source code has been compiled into object code. To minimize the size of the patching script, linking should be performed incrementally by considering the way the program has been linked in the previous version. For each node, EcoCast maintains a directory of the source files, memory map file, and command script of the node.

The map file contains information on memory usage, list of function segments, segment sizes and locations in program memory, and the symbol table. The function call tree is also assumed to be available with the proper compiler directives. Information obtained from the memory map file can only reveal the size and address of each function but not the actual function name, function parameters or return value. The information extracted from the map files are used by EcoCast for type marshalling and dispatching. To allow another host access to the framework, the map file can be retrieved either from the original host or the nodes.

To minimize the size of the patching script, the incremental linker attempts to keep these assigned addresses unchanged between versions. The addresses include not only code but also data, constants, and library routines. If the linker can determine that two pieces of code are never invoked together, then it may consider overlaying them when necessary to fit in the very limited amount of program memory while reducing the reference patching. In the current implementation, each code segment is given a priority based on the call tree generated by Keil C LX51 linker. The higher the priority, the smaller the number, starting from 1, and priority 0 is assigned to driver code segments. For each segment, the larger its call depth, the higher its priority. This ensures that segments that are least referenced will be overlaid first. A least recently used (LRU) replacement algorithm is used to select among segments with the same priority.

5.3 Version Control and Patching Scripts

Conceptually, the host subsystem maintains a database entry for keeping track of the memory layout and source files of each sensor node. However, it is inefficient and not scalable to a large number of nodes, because we expect some or most nodes to have identical memory layouts. To eliminate redundant entries, we maintain the information according to the firmware version. Nodes with the same version share the same entry, and a new entry is created when functions are installed or updated, resulting in distinct memory layouts. This organization makes it efficient to re-compile, re-link, and updating code to a group of nodes at a time. Moreover, to encourage experimentation, EcoCast integrates version control (Subversion in our case) for the node firmware to enable roll back to any previous version of the firmware as simply as an “undo” command.

Upon generation of a new program image, binaries of the new function are extracted while pre-existing segments that are updated are *diff*ed with the original binaries to generate the patching binaries. These patches are then wrapped into store messages (Section 6.1) for installation or update on the target node.

6. REPROGRAMMING AND EXECUTION

One user command on the host subsystem can be expanded into a sequence of actions, including some that turn into messages that are received and executed by the nodes, followed by returning values to the host or other nodes. The host first sends patching scripts to the targeted nodes if necessary, and then sends the message to invoke the user’s desired function. For macroprogramming, EcoCast attempts to parallelize the communication and execution as much as possible. This section first describes message handling by a node, followed by group reprogramming over the air, and group invocation of functions with specific discussion on synchrony issues.

6.1 Execution Mechanism on the Node

The Node subsystem consists of two software components: *Store* and *Execute*. They make use of the same underlying messaging mechanism for wireless communication.

6.1.1 Store Component

The Store component is responsible for processing incoming data to the node. This happens during function installation or variable updates. A STORE message contains a 2-byte target address, 1-byte nonvolatile flag, a 1-byte length field, and the data payload. The nonvolatile flag indicates whether the data should be stored in nonvolatile memory such as EEPROM or flash, or if it should be written to RAM only. The difference is that the RAM content is lost when the device is rebooted. It is up to the programmer to decide whether the program is to be permanently updated or temporarily changed. In our case, the code for `lambda` expressions is kept in RAM only.

6.1.2 Execute Component

The execute component invokes target functions plus basic functions for memory access. It contains a set of *get* handlers that return memory content at a given address, and this mechanism can be further extended as a debugging utility. Unlike EcoExec, which assumes that the code must be copied from external EEPROM into

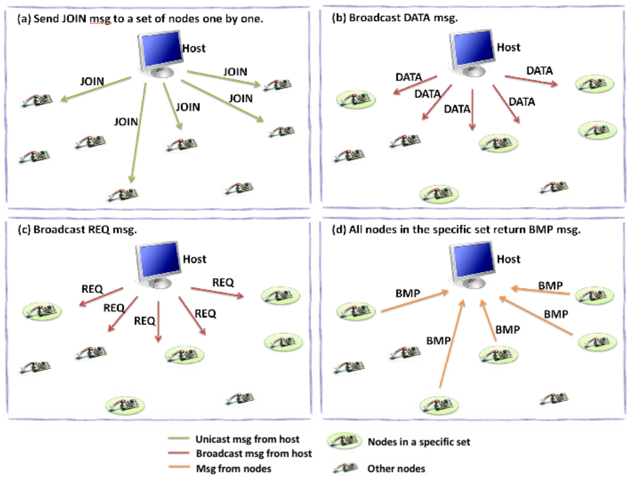


Figure 3: Reprogramming scenario

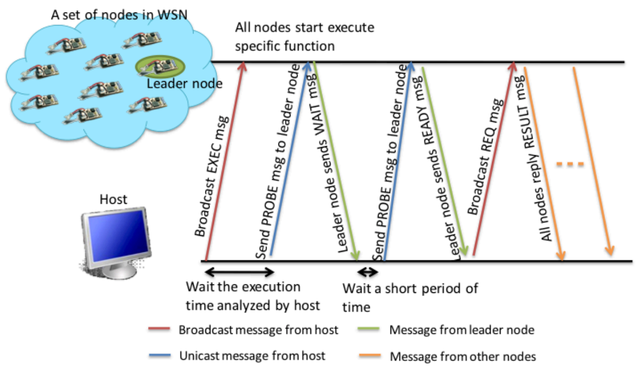

A set of nodes in WSN
Leader node
Host
All nodes start execute specific function
Broadcast EXEC msg
Send PROBE msg to leader node
Leader node sends WAIT msg
Send PROBE msg to leader node
Leader node sends READY msg
Broadcast REQ msg
All nodes reply RESULT msg
Wait the execution time analyzed by host
Wait a short period of time
Broadcast message from host
Unicast message from host
Message from leader node
Message from other nodes

Figure 4: Function execution sketch

on-chip RAM before it can execute, EcoCast assumes *execute-in-place* (XIP) model for program memory, although it is also possible execute from RAM for code that is meant to be run temporarily and discarded, such as most lambda expressions. In any case, the firmware is structured such that no rebooting is needed after patching. We achieve this by setting the highest priority on the code segment for the EcoCast runtime support such that it cannot be modified at run time.

6.2 Group Reprogramming of Nodes

To reprogram (patch) a set of nodes, EcoCast performs the steps similar to Telescribe [8] as shown in Fig. 3, with the difference that EcoCast does it incrementally rather than full image replacement. This scheme is robust to interruption during reprogramming. In Step 1, the host individually sends a JOIN message to ask the nodes that need to be reprogrammed to join the group. This group of nodes (in ovals in Fig. 3(b)) would listen to a specific channel and every member node keeps a bitmap to track packet loss. In Step 2, the host broadcasts the DATA messages via the specific channel on which only the group members can receive, as shown in Fig. 3(b). The DATA message includes the code (as data) and its memory address. Each time a node receives a DATA message, it stores the code to the specific memory location and sets the corresponding bit in the bitmap. After all DATA messages are sent, the host broadcasts a REQ message (Fig. 3(c)) in Step 3, and all the member nodes reply with a BMP message containing their bitmap to the host to indicate missing DATA messages (Fig. 3(d)). A simple TDMA technique is applied to the BMPs from the nodes, where each node waits until its time slot as determined by its group ID and sends the BMP back to the host. The host rebroadcasts the lost DATA packets according to the BMPs and repeats the steps above until all members are reprogrammed correctly, unless the user decides to abort.

6.3 Group Execution of Functions on Nodes

To ask a node to execute a function, the host just needs to send an EXEC message to the node and wait until the node replies with the result. To ask multiple nodes to execute a function, such as the case with `map`, it would be inefficient to simply iterate over the nodes one at a time. Instead, the following protocol is used, as sketched in Fig. 4. Initially, the host broadcasts an EXEC message, and the nodes execute the specified function. However, at this point the nodes should not reply as soon as they are done, because it is likely to run into collision if not coordinated. The host is not blocked but may send a PROBE message to see if a node has completed the previous call. Instead of sending a PROBE to each node, the host sends it to only the node acting as the *leader*, since it is expected that all nodes would complete execution around the same time. The leader replies with either a WAIT if it is still running and the host repeats; or the leader replies with a READY if it has finished. Then, the host broadcasts a REQ message, and all the nodes return their execution results similar to the group reprogramming in Section 6.2. The nodes reply with a RESULT message in a simple TDMA scheme, instead of the bitmap.

One question is how long the host should wait before trying a PROBE. The overhead for the leader is high if we probe the leader too often, but if the time interval between successive probes is too long, then the response latency increases. For this reason, we estimate the execution time using a timing analysis tool named Bound-T [35]. It computes an upper bound on the worst case execution time (WCET) of a program. Using the estimated execution time, EcoCast can determine an effective time for probing.

6.4 Background Job Management

To handle a background job, the interpreter assigns a thread to execute the command in the background, and the *background jobs manager* records the command, execution status and the information of the corresponding thread. When multiple jobs exist in the system, EcoCast is responsible for mapping the incoming packets from the nodes to the associated jobs on the host. Since every incoming packet is related to a node instance on the host system, the associated job is in charge of the packet. Note that we apply a locking scheme where an attribute of a node instance that resides on node must be serialized to ensure correct execution on the node. The `jobs` command is provided by EcoCast for the user to monitor the execution status and access the result after the background job is done.

7. MULTI-HOP NETWORKING

We use the proposed functional programming constructs to build our initial multi-hop network protocol. We implement a stack with minimal complexity by pro-active routing on the host, and this approach is effective for those networks whose topologies do not change rapidly. Our bootstrapping multi-hop protocol can be separated into *topology discovery* phase and *normal transfer* phase. Both make use of six types of messages, as shown in Table 3.

Table 3: Opcodes for our Multi-hop Protocol

Discovery Phase

Code	Type
2	Host Broadcast Request
3	Node Broadcast Request
4	Node Broadcast Ack
5	Host Broadcast Ack

Transfer Phase

Code	Type
0	Host Data Request
1	Node Data Ack

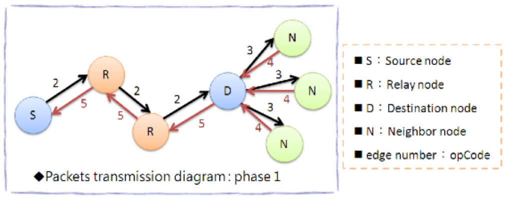


Figure 5: Illustrative example for Discovery phase

7.1 Phase 1: Topology Discovery

Every node can use a node broadcast request message (type 3) to to discover its neighbors by receiving type-4 messages from them. The host can use type-2 message to ask one certain node to discover its neighbors and report with a type-5 message. In this phase, EcoCast starts neighbor discovery from node #0, the default virtual root node represented by the base station, and it explores the neighbor nodes until the entire network has been discovered. The host maintains the neighbor list for each node for path selection. In this phase, when a node receives a packet, it checks if it is the recipient, and the response is either to process it, relay it, or ignore it. Fig. 5 shows an illustrative example for requesting a node to report its neighbor information.

After all neighbors have been discovered, the host can compute the network topology and select the path for each node. For simplicity, we use BFS (breadth-first search) as our path selecting policy. This protocol is designed to be simple on the node by moving the complexity to either the host or a base station in the field. In this case, path selection is done on the host in Python, but users can always replace it with other algorithms or make a distributed version.

7.2 Phase 2: Packet Transmission and Relay

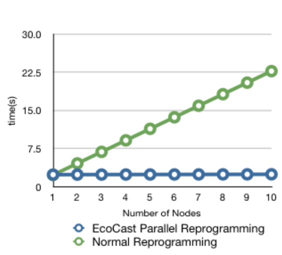
After neighbor discovery and path selection, phase 2 is for the normal operation of the network. It entails packet transmission and relay. To do this, two types of packets are used: OpCode 0 is the "downstream" packet type from the host to the nodes, and OpCode 1 is the "upstream" type. The packet format includes the hop length and the IDs of the nodes along the path, followed by the payload.

8. EVALUATION

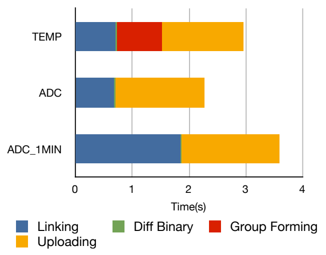
We evaluate the performance of EcoCast in terms of reprogramming latency, execution latency, and memory footprint.

8.1 Experimental Setup and Test Cases

Our setup consists of a PC acting as the host connected through an Ethernet base station to ten nodes as described in Section 3.1. The nodes are all equipped with an on-board triaxial accelerometer, and different nodes have their output pins connected to actuators for lighting control. At the beginning, all nodes have been programmed with only the essential drivers, including SPI, flash memory, and RF, but no application code. Each application is incrementally linked and uploaded by EcoCast on demand.



(a) Reprogramming time of ADC.



(b) Ratio of different steps of latency in reprogramming 10 nodes

Figure 6: Reprogramming Latency

We created several WSN applications using EcoCast, and their details are shown in Table 4. For single-hop experiments, the RF transmission power is 0 dBm; for multi-hop ones, the RF transmission power is set to −20 dBm.

8.2 Reprogramming Latency

EcoCast performs group reprogramming (Section 6.2) to minimize the reprogramming latency for multiple nodes. Fig. 6(a) shows the comparison of latency between sequential reprogramming and group reprogramming by EcoCast for installing the ADC application on-the-fly. Installation of a new application entails several steps, including linking, binary diff'ing (i.e., patch generation), uploading, and group forming if reprogramming a group of nodes. We demonstrate the reprogramming latency of applications TEMP, ADC and ADC_1MIN by group reprogramming of EcoCast where the group size is 10 nodes. The applications are uploaded in sequence. Fig. 6(b) shows the latency of different steps. Note that ADC and ADC_1MIN can reuse the same group as TEMP, and therefore their group formation times can be completely eliminated. The linking time of ADC_1MIN is longer since EcoCast tries to keep the shared functions of the two ADC applications at the same locations. On average, it takes about 6.15 ms to upload one byte and write it into flash memory. In our experiments, the compilation-linking-installation-confirmation procedure require less than 4 seconds total for 10 nodes, which is acceptable. Several other wireless reprogramming schemes actually do not confirm if the nodes have been programmed successfully.

8.3 Round-Trip Command Execution Latency

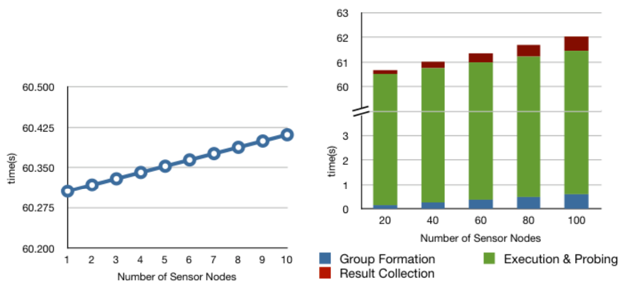
The round-trip command execution latency, also called the response time, is defined from issuing the command to receiving the results back on the host. We measure the response time of TEMP and ADC by serial execution iteratively (i.e., for-loop) and in parallel (using `ecMap`) over a range of group sizes from 1 to 10. We execute each application 50 times and record the average response time. The result is shown in Fig. 8.

The measured runtime scales linearly for serial execution. For parallel execution, we plot the worst case that includes the additional phase of group forming before execution, even though for the common case the same group can be reused. With this assumption, the response time of serial execution exceeds that of parallel execution for group sizes of 6 and higher in both TEMP and ADC applications. This means the group overhead can be amortized for a modest number of nodes even for very short execution delays. Fig. 9 shows the breakdown of parallel (`ecMap`) execution times of ADC into three phases: *group formation*, *execution and probing*, and *result collection*. The group formation time can be eliminated

Table 4: Test cases of functional programming

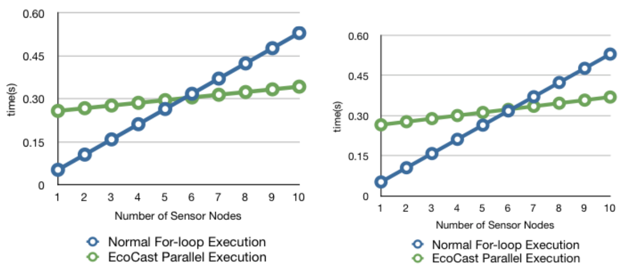
Name	Description	Size[1]	Iterative command	Macroprogramming statements
TEMP	Read the output of temperature sensor and return to host.	202 bytes	`g = getNodes([0,1,2,3,4,5,6,7,8,9])` `for n in g:` `    n.readTemp()`	`g = ecGroup([0,1,2,3,4,5,6,7,8,9])` `ecMap(readTemp,g)`
ADC	Reads the output value for the digitalized output of triaxial accelerometer return to host.	212 bytes	`for n in g:` `    n.readADC()`	`ecMap(readADC,g)`
ADC_1MIN	Samples the output value of triaxial accelerometer in 25Hz for one minute and return the average of the outputs.	372 bytes	No iterative version	`ecMap(readADC1Min,g)`
LIGHTING	Control the switch of the light(on/off)	12 bytes	`for n in g:` `    n.turnOn()` `time.sleep(5)` `for n in g:` `    n.turnOff()`	`ecMap(turnOn, g)` `time.sleep(5)` `ecMap(turnOff, g)`

[1] The uploaded size is the size of patching binary generated by diff'ing the original binary with the new binary.



(a) Response time (b) Ratio of different phases

Figure 7: Response time of ADC parallel execution of EcoCast



(a) TEMP (b) ADC

Figure 8: Response time of serial execution and parallel execution of EcoCast

by reusing the same group, while the *execution and probing* phase remains nearly constant. The *result collection* phase would still grow linearly but at a much lower slope. For applications with long execution times, the times of *group formation* and *result collection* phases become negligible. The measured result of ADC_1MIN shows in Fig. 7(a). By following the TDMA schedule, we estimate the response time with large group sizes according to the trend of Fig. 7(a) and show the time of different phases in Fig. 7(b). We observe that *group formation* and *result collection* phases together take a total of only 1.18 seconds in a group with 100 nodes.

8.4 Memory Footprint

A system with a small footprint gives users more room for applications. EcoCast has a small footprint, making it particularly applicable to resource-constrained platforms. Table 5 shows the comparison of memory footprints of runtime systems with interactive shell support. The core of EcoCast occupies 1425 bytes of program memory and 161 bytes of data memory. With essential drivers, EcoCast takes 3942 bytes of program memory and 215 bytes of data memory.

EcoCast is significantly smaller than most of the other works while being similar in size to Marionette [37], which also takes advantage of Python for the fat-client, thin-server organization. However, Marionette needs to be built on top of TinyOS, requires external support for remote reprogramming, is intended for debugging rather than deployment, and does not support macroprogramming. A related system, TinyBasic, also has some limitations. For instance, it only supports data operations for signed integers, whereas EcoCast maximizes the host assist and node flexibility to minimize the memory footprint on the node.

8.5 Multi-Hop Reprogramming Latency

We first evaluate the time it takes to find a path for each node. As shown in Fig.10, it takes about 16.6 ms to reach a node one-hop away. Then, we can find subsequent routes through this node. The number of hops is proportional to the latency since we start a whole discovery for every node.

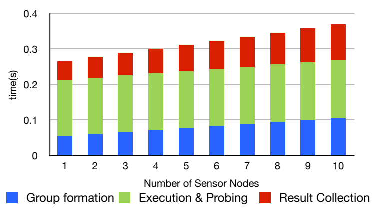


Figure 9: Ratio of different phases to execution time of ADC via parallel execution of EcoCast.

Table 5: Comparison of Memory footprint

Runtime System	Program memory	Data memory
SensorWare	$\leq$ 180KB	$\leq$ 64KB
Maté	16KB	849B
LiteOS	30KB	$\approx$ 1.6KB
Mantis	14KB	$\leq$ 5KB
Marionette	$\leq$ 4KB	153B
EcoCast	3.94 KB	215B

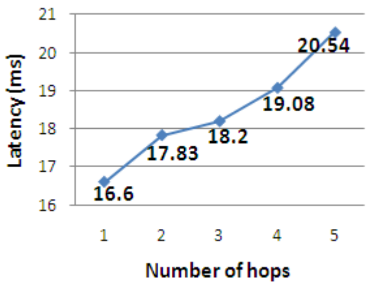
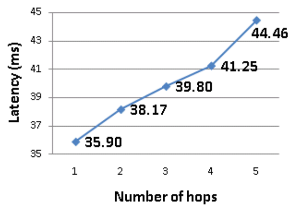
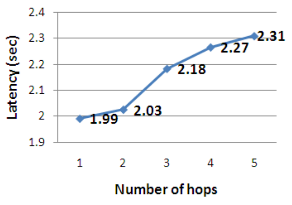


Figure 10: Latency of finding a path for each node



(a) Light control



(b) Door monitoring

Figure 11: The latency of adding functions

In multi-hop network configuration, we have one test application called LIGHTING for light control. It was not in the firmware of the nodes at the beginning of the experiment. Upon invoking the `turnOn()` and `turnOff()` functions for LIGHTING, EcoCast prepares the binary for the functions, each of which is six bytes long, and transmits two packets to each target node, which patches its own firmware and executes. As shown in Fig. 11(a), it takes 35.9 ms to relay a packet and reprogram the node that is one hop away from the host, with an average of 2 ms for each additional hop. Fig. 11(b) shows another application, DOORMON, which is 417 bytes in size and takes 1.99 seconds to reprogram a node that is one hop away and 2.31 seconds for five hops away.

8.6 Discussion

The experimental results show parallel execution of functional programming constructs to work well for single-hop networks; the performance is adequate for a modest-sized multi-hop network, but there is plenty of room for optimizations. One way to speed up is pipelined execution, although this is much more difficult in practice than in theory due to wireless communication. Another way is to explore asymmetric routes. If some nodes are not part of the targeted group for function invocation, then perhaps they can play more of a relay role, and reply packets need not take the reverse routes as request ones. The more even distribution of workload may also have energy efficiency benefits. Many more optimizations are possible for `filter` and `reduce` constructs in a multi-hop network. We have demonstrated the feasibility of combining macroprogramming, interactivity, and integration with a general-purpose language, and many further optimizations can now be built on this framework.

9. CONCLUSIONS AND FUTURE WORK

EcoCast brings the elegance and power of Python to enabling interactive macroprogramming by scripting of resource-constrained wireless sensor networks. Interactivity encourages experimentation by beginners, by providing instant feedback to functions of interest while being forgiving by supporting undo with an underlying revision control system. As Python is not just a "beginner's language" but is actually used by expert programmers in production-quality systems due to support of constructs at a much higher-level of abstraction, we expect EcoCast to also boost the productivity of expert programmers in a similar way. EcoCast relieves the user of the burden of the edit-recompile-relink and reprogramming process by transparently performing incremental compilation and patching, if necessary, before invoking functions on a group of nodes, all with negligible delays and is considered practically interactive. The use of node handles with type marshalling enables seamless integration of the sensor network as objects with the rest of the application code such as graphical user interfaces and data processing code. Moreover, host-assisted demand code paging enables EcoCast to run on some of the most resource-constrained platforms.

Several directions for future work remain. First is a smarter replacement policy for code by assigning priority to those segments that are either used more frequently in the near future. Second is further optimization of macroprogramming constructs, in particular `filter` and `reduce`, to complex network topologies. One way is to map the computation to the specific network topology; another is to enable pipelined execution of commands while keeping the results synchronized.

Acknowledgments

This work was sponsored in part by a National Institute of Standards and Technology (NIST) Technology Innovation Program (TIP) Grant 080058, National Science grant CBET-0933694, and a National Science Council grant 99-2625-M-007-001. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

[1] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming heterogeneous sensor networks using COSMOS. *Operating Systems Review*, 4(3):159–174, 2007.

[2] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[3] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14. Springer, 2001.

[4] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys '03*, May 2003.

[5] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The LiteOS operating system: Towards unix-like abstractions for wireless sensor networks. In *IPSN '08*, pages 233–244, Washington, DC, USA, 2008. IEEE Computer Society.

[6] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *IPSN '07*, pages 148–157, New York, NY, USA, 2007. ACM.

[7] C. Chen, Y. Chen, Y. Tu, S. Yang, and P. Chou. EcoSpire: an application development kit for an Ultra-Compact wireless sensing system. *Embedded Systems Letters, IEEE*, 1(3):65–68, 2009.

[8] M.-H. Chen and P. H. Chou. TeleScribe: A scalable, resumable wireless programmable approach. In *Proc. International Conference on Embedded Software*

(EMSOFT), Scottsdale, AZ, USA, October 24-29 2010.

[9] Crossbow Technology. Mote in-network programming user reference version 20030315. http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf, 2003.

[10] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06*, pages 15–28, New York, NY, USA, 2006. ACM.

[11] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[12] R. Gummadi, R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. *Distributed Computing in Sensor Systems*, pages 126–140, 2005.

[13] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05*, pages 163–176, New York, NY, USA, 2005. ACM.

[14] T. W. Hnat, T. I. Sookoor, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *SenSys '08*, pages 225–238, New York, NY, USA, 2008. ACM.

[15] C.-H. Hsueh, Y.-H. Tu, Y.-C. Li, and P. Chou. EcoExec: An interactive execution framework for ultra compact wireless sensor nodes. In *SECON '10*, pages 1 –9, jun. 2010.

[16] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04*, pages 81–94, New York, NY, USA, 2004. ACM.

[17] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 25–33, Oct. 2004.

[18] J. Kim and P. H. Chou. Remote progressive firmware update for flash-based networked embedded systems. In *Proc. International Symposium on Low Power Electronics andDesign (ISLPED)*, pages 407–412, San Francisco, CA, USA, August 19-21 2009.

[19] J. Koshy. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the second European Workshop on Wireless Sensor Networks*, pages 354–365. IEEE Press, 2005.

[20] S. S. Kulkarni and L. Wang. Mnp: Multihop network reprogramming service for sensor networks. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS*, pages 7–16, 2005.

[21] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[22] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. EnviroSuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Computational Logic*, 5(3):543–576, August 2006.

[23] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[24] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(1):122–173, 2005.

[25] P. J. Marron, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A flexible and efficient code update mechanism for sensor networks. In *EWSN '06: Proceedings of the third European Workshop on Wireless Sensor Networks (EWSN 2006*, pages 212–227, 2006.

[26] R. Newton, G. Morrisett, and M. Welsh. The Regiment macroprogramming system. In *IPSN '07*, pages 489–498, New York, NY, USA, 2007. ACM.

[27] R. Newton, R. Newton, and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *Proceedings of the 1st International Workshop on Data Mangement for Sensor Networks*, pages 78–87, Toronto, Canada, 2004. ACM.

[28] J. K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the USENIX Winter 1990 Technical Conference*, Berkeley, CA, 1990. USENIX Association.

[29] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[30] A. Pathak and M. K. Gowda. Srijan: A graphical tookit for sensor network macroprogramming. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of software engineering*, 2009.

[31] A. Pathak, L. Mottola, A. Bakshi, V. K. Prasanna, and G. P. Picco. Expressing sensor network interaction patterns using data-driven macroprogramming. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops '07. Fifth Annual IEEE International Conference on*, pages 255 –260, March 2007.

[32] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, New York, NY, USA, 2003. ACM.

[33] SourceForge. Source code of EcoExec. http://ecoexec.sourceforge.net/, 2010.

[34] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical report, UCLA, Los Angeles, CA, USA, 2003.

[35] Tidorum Ltd. Bound-T user guide. http://www.tidorum.fi/bound-t/manuals/user-guide.pdf, April 2009.

[36] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 2000.

[37] 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 *IPSN '06*, pages 416–423, New York, NY, USA, 2006. ACM.

[38] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD record*, 31(3):9–18, 2002.