μSETL: A Set Based Programming Abstraction for Wireless Sensor Networks

Mohammad Sajjad Hossain, A.B.M. Alim Al Islam, Milind Kulkarni, Vijay Raghunathan
School of ECE, Purdue University, West Lafayette, IN 47907
{sajjad, abmalima, milind, vr}@purdue.edu

ABSTRACT

Programming wireless sensor networks is a major challenge, even for experienced programmers. To alleviate this problem, prior work has proposed a paradigm shift from node-level *microprogramming* to *macroprogramming*, where the user specifies a distributed application using a single macroprogram that is automatically translated into a set of node-level microprograms. This paper makes the case that node-level microprogramming itself can be made much easier by using the right set of programming abstractions. To support this claim, this paper presents μSETL, a programming abstraction for sensor networks based on *set theory*. Sets offer a powerful formalism and high expressiveness, yet are a natural way of thinking about resource abstraction in sensor networks. In addition to the set abstraction, μSETL features programming constructs that enable event-driven programming at a high level of abstraction, thereby significantly simplifying node-level microprogramming. μSETL consists of a set-based programming language, a compiler that translates μSETL programs into node-specific application code, and a runtime environment that provides various services to support the set-based programming abstraction. μSETL has been implemented using the Contiki operating system and runs on the Telos motes. Experimental results demonstrate that μSETL enables programmers to write various sensor network applications in a natural and highly compact manner with minimal overheads.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming-Distributed programming

General Terms

Design, Languages, Performance

Keywords

Wireless Sensor Networks, Programming Abstractions

1. INTRODUCTION

Cyber-Physical Systems (CPSs) are poised to play a pivotal role in engineering new solutions to a variety of societal-scale problems, such as energy conservation, climate change, healthcare, transportation, *etc*. Networked embedded systems, such as wireless sensor networks (WSNs), form a crucial building block for realizing large-scale CPSs and have received considerable research attention. While this has resulted in numerous technological advances (*e.g.,* a plethora of tiny, cheap, and low-power sensor platforms is now available [1]), the problem of programming a distributed wireless sensor network still remains a major challenge and a potential show stopper to widespread adoption. This challenge is best exemplified by the following quote from a recent EE Times article [2]:

> *"Programming the software that manages applications running on wireless sensor and control networks is currently so technically intricate, complex and laborious that it can take months of work by specialized programmers just to deploy the simplest application. That process takes even longer for more complex deployments..." [2]*

Previous attempts at addressing the programmability problem in WSNs (see [3] for a comprehensive survey) have yielded the notion of macroprogramming, where the user specifies a distributed application using a single macroprogram that is automatically translated into a set of microprograms that execute on individual nodes. Such macroprogramming systems, discussed further in Section 6, play a key role in lowering the barrier to entry for application domain experts who may be novice sensor network programmers.

This paper adopts a philosophically different approach to simplifying sensor network programming, compared to macroprogramming approaches. We contend that the main issue with existing microprogramming techniques is not that they adopt a node-level perspective, but rather that the abstractions used to specify the node-level microprograms are not conducive to easy programming. Our assertion is that node-level microprogramming can be made significantly easier if programmers are provided the right abstractions to describe their applications and the corresponding runtime services to support these abstractions. In addition to relieving programmers of the burden of worrying about low-level system issues, these abstractions should also be expressive enough to easily compose distributed applications from a node-level perspective. To support our claim, this paper presents the μSETL programming abstraction for wireless sensor networks[1].

The cornerstone of μSETL is a new data structure, namely a *set*. Our choice of this data structure was motivated by the fact that a set is a natural way to think about resource abstraction in a wireless sensor network (indeed, while conversing about sensor networks, we often use phrases such as "a set of nodes", "a set of sensor values", *etc.*). Further, most people with a science, engineering, or mathematics background have some familiarity with set theory. In μSETL, although the set abstraction is used to hold network-level

[1]The μSETL programming abstraction is inspired by the SETL [4] programming language for computer systems.

IPSN'11, April 12–14, 2011, Chicago, Illinois.

information, the scope of a set is local to the node where the set is defined, and each set is operated on from a node-level perspective. In addition to the set data structure, μSETL also features two special programming constructs (a *periodic* block and a *monitor* block) to support event-driven programming at a high level of abstraction[2]. The periodic and monitor constructs allow us to trigger execution of handler functions based on timer events and mutating events (*e.g.*, when the contents of a set change), respectively. The μSETL system also features a compiler that translates μSETL programs into node-specific application code, and a run-time environment (RTE) that provides methods for μSETL programs to perform various set operations such as union, intersection, iterating over the members of a set, *etc*. The RTE is also responsible for populating and updating the contents of sets defined by μSETL applications.

μSETL has been implemented using the Contiki [6] operating system and runs on the Telos motes. We evaluate μSETL using three representative WSN applications, namely (a) periodic data collection, (b) defining user-level routing protocols, and (c) object tracking. Experiments conducted using the Cooja [7] simulator as well as Telos motes demonstrate that combining the expressiveness of sets with high-level event-driven programming enables WSN applications to be easily written as highly compact, yet efficient node-level microprograms. For example, we are able to express the Surge data collection application using just 3 lines of μSETL code.

2. BACKGROUND

The most common approach to programming WSNs is node-level programming or microprogramming. In this approach, programs are written for individual sensor nodes using a language such as C, nesC [8], *etc*. These programs interact with hardware and other node resources through low-level abstractions provided by the operating system. However, programming at this level of abstraction forces the application programmer to pay attention to many low-level system issues such as interrupts, sensing, node-to-node communication, *etc*. Further, it is difficult to express distributed applications and network-level functionality at such a low level of abstraction. For example, to generate a list of nodes that are a certain number of hops away from a given node, a programmer has to specify all of the program logic detailing how the list will be assembled and maintained. Virtual machines (*e.g.*, Maté [9]) create higher levels of abstraction on top of the node's operating system, but mainly with the goal of making node reprogramming easier. Macroprogramming languages [3], on the other hand, offer a different perspective by adopting a global or network-centric view. Programs written using these macroprogramming languages specify the behavior of the entire network or a group of nodes, either logical [10] or physical [11]. A compiler then decomposes a macroprogram into a set of microprograms that run on individual nodes. By creating different kinds of abstractions, macroprograms allow programmers to write concise scripts that hide the complexity of a lot of common operations. However, it is not possible to express node-level interactions using most macroprogramming languages, making them unsuitable for certain types of applications. For example, a mesh-routing protocol cannot be designed without specifying node-level interactions.

Set theory is a well-established mathematical discipline. Sets provide a concise yet natural syntax and have an inherent expressive power. The compactness comes from the declarative nature of set definitions. Thus, set-based abstractions can significantly enhance program compactness, clarity, and readability. As a result, a number of programming languages (*e.g.,* specification language Z [12], functional language MIRANDA [13], and procedural language SETL [4]) from different domains use sets as data abstractions and exploit the potential offered by set constructs. SETL is an interpreted language with a syntax similar to the language of set theory. For example, a set can be declared in the following manner:

```
A := {2, 3.5, 5,'Hi There!'};
```

SETL supports all the elementary set-operations commonly used in set theory. For example:

```
print('Set union = ', A+B);
print('Set intersection = ', A*B);
```

However, the added expressive power comes at the cost of some loss in efficiency. Set-based programming languages are known to be slower than their low-level counterparts. Hence, these languages are mostly seen as a tool for rapid experimentation with algorithms and design, and not for production use [4].

As mentioned in Section 1, sets are a natural abstraction to represent various distributed resources available in WSNs. Examples of such resources could be different types of sensors, storage, or even a node itself. In addition to these physical resources, sometimes it is useful to define abstract resources, for example, a set of nodes with temperature data above a threshold. Expressing these types of abstract resources or groups has always been a difficult task for programmers, especially in microprogramming models. We believe that this is where the set abstraction really shines. Hence, we strongly believe that a programming abstraction based on the powerful formalism of set theory will make sensor network programming significantly easier. To the best of our knowledge, μSETL is the first work to provide such a set-based abstraction for WSNs.

3. μSETL PROGRAMMING MODEL

Although μSETL is inspired by SETL [4], the scope of its applications is very different. μSETL programs are written for networked embedded systems, which have their own unique requirements and characteristics compared to their desktop counterparts. These systems are often limited in computing power and other resources. Hence, efficiency is a major concern here. Also, embedded systems such as WSNs are often used as data collection and aggregation systems. They can be viewed as sets of distributed resources that can generate these data as necessary. Any programming language or abstraction designed for these environments has to take these characteristics into account. While μSETL has syntactic similarity with SETL, semantically it is quite different. It also has new programming constructs tailored to the needs of WSN programs. This section provides a detailed description of the μSETL programming model.

3.1 Overview of μSETL Programming Model

Unlike SETL, μSETL adopts an event-driven programming model. Conceptually, a μSETL program is just a collection of event handlers and, optionally, initialization code that is executed at startup. Events can be triggered by timers or changes in state (including the receipt of messages from other nodes, such as a command from a base station). Event handlers are written using two new block constructs introduced in μSETL, the *period* block and the *monitor* block, which govern timer-triggered events and state-triggered events, respectively. These constructs make it natural to write event-driven applications using high-level idioms, and we describe them in greater detail in Section 3.2.

Notably, a μSETL program is written from the point of view of a particular node. This is in contrast to the network-centric view of-

[2]It is known that event-driven programming is well suited for reactive systems such as wireless sensor networks [5, 6].

Idiom	μSETL expression
Nodes with even IDs	`{N(i) | i % 2 == 0}`
Neighbors of the current node	`{i | distance(i) == 1}`
Two-hop neighbors of the base station	`{i | distance(base_station, i) == 2}`
Temperature data that are greater than x	`{N(i).temp | i IN all, N(i).temp > x}`
Nodes having a camera	`{N(i) | N(i).has(CAMERA)}`
The latest 3 data values received from the base station by the current node	`{x | receive(base_station, x)}:3`

Table 1: Expressing common WSN idioms using μSETL.

fered by macroprogramming languages [3]. By providing a node-centric programming model and a set-based abstraction, μSETL naturally captures common idioms that are used in most WSN applications. Examples of such idioms include the set of neighbors of a node, temperature data from all nodes, *etc*. The abstraction provided by μSETL also makes it possible to easily express other, more complex, idioms. For example, iterating over all the nodes with certain capabilities (*e.g.*, nodes having light sensors) can be easily expressed using custom-defined sets. Table 1 shows a few examples of how such idioms can be expressed using μSETL.

Even though a μSETL program is written from a node-centric view, it is possible to specify the behavior of a group of nodes. The μSETL constructs allow programmers to write code blocks specific to a set of nodes (Section 4.2.3). This encourages code reuse and allows a single piece of code to be written that behaves differently on different types of nodes (*e.g.*, an event handler that triggers different behavior on light sensor-equipped nodes than on temperature sensor-equipped nodes). The μSETL compiler detects this and generates node-specific code, thereby decreasing binary size and improving run-time efficiency (discussed further in Section 4.2).

3.2 μSETL Language Specification

μSETL adopts many of the language constructs found in SETL. However, to accommodate the needs of wireless sensor networks and other networked embedded systems, μSETL provides new language constructs to simplify the specification of event handling. In this section, we describe the key features of the μSETL language.

3.2.1 An Example μSETL Program: Object Tracking

We use a representative application - simple object tracking with actuation (henceforth referred to as *Object Tracking*) - to describe the various features of μSETL. The goal of *Object Tracking* is to track a light source in a WSN. The μSETL code for this application is shown in Figure 1. Each sensor node that has a light sensor measures light intensity periodically (every 4 seconds) and compares the measured value to a threshold. If the sensed value is above the threshold, the node reports the value to a base station (lines 13 – 18). The base station periodically (every 5 seconds) checks all the received data and selects the nodes that have the highest light intensity value. It then sends a command to the selected nodes to turn on their cameras (lines 4 – 10). The commands are executed by the corresponding nodes once they are received (lines 19 – 22). This program captures most of the key features of μSETL. In the following sections, we expand upon these features in detail.

3.2.2 μSETL Grammar

The μSETL language is defined by a context-free grammar, G, available in [14]. Currently, G defines three types of commonly-used sensors as resources (temperature, light, and magnetometer).

```
1  @base_station@ #
2    received := {[x, y] | receive(x, y)};
3    previous_on := {};
4    period 5000 do
5      targets_on := {i:u8 | [i, j] IN received,
           j == max({k | k IN received.second})};
6      target_off := previous_on - target_on;
7      send(target_off, CAMERA_OFF);
8      send(target_on, CAMERA_ON);
9      previous_on := target_on;
10   end
11 #
12 @{node | node  != base_station,
           has(node, LIGHT_SENSOR)}@ #
13   period 4000 do
14     reading := N(node).light;
15     if (reading > 400) then
16       send(base_station, reading);
17     end
18   end
19   command := {x:u8 | receive(base_station, x)}:1;
20   monitor command do
21     execute(command);
22   end
23  #
```

Figure 1: μSETL code for *Object Tracking*.

G can be easily modified to include other types of resources (*e.g.*, flash storage, other sensors). μSETL also does not currently support user-defined functions; instead, we provide a set of common function calls (Table 3). Resources and functions are simply hooks into built-in capabilities of the RTE, and to extend either, both G and the RTE must be modified accordingly.

3.2.3 Data Types, Sets, and Variables

There are six basic data types in μSETL. They are *integer* (both signed and unsigned), *float*, *char*, *string*, *set*, and *node*. All of the data types except *node* are available in SETL. *Node* is a data type in μSETL that represents a single node in the network and encapsulates various properties of a node. Examples of such properties include *location*, *identifier*, *etc*. *Node* also encapsulates various types of resources a node may have. These resources can be various types of sensors (light, temperature, *etc*.) or computing elements (CPU, memory, *etc*.). This makes the representation of a node and the manipulation of various information about it simple and intuitive. The μSETL grammar reserves the letter `N` to denote a node, and `N(i)` refers to a particular node *i*. Resources and different attributes of a particular node are accessed using the dot operator (*e.g.*, the light sensor on the node with ID *i* can be accessed using `N(i).light`). It should be noted that the dot operator cannot be used to access any local variables of a node.

Variables (*i.e.*, primitive data types and sets) in μSETL can be *volatile*, which means that they can change over the course of execution without the direct knowledge of the programmer. The runtime environment is responsible for monitoring the state of volatile variables. The μSETL compiler performs an analysis to determine which data should be considered volatile by the RTE (Section 4.2). Sets can be explicitly declared non-volatile by using the `novolatile` declaration (line 15 of [14]). In μSETL, sets are essentially multi-sets, since the same value can occur multiple times within a set. However, for the sake of simplicity, we refer to them as sets throughout the paper.

Table 2 shows a few examples of sets defined in μSETL. Sets must be finite and nested set definitions are allowed. Note that elements in volatile sets are implicitly associated with a time-stamp, according to when the RTE placed the element in the set. In addi-

Set	Description
`novolatile {2, 3, 4}`	(Non-volatile) set of integers
`{N(2), N(5), N(7)}`	Set of nodes
`{i | distance(i, x) == 1}`	One hop neighborhood of *x*
`{N(i).light | i IN y}`	Light data of the nodes in set *y*

Table 2: Examples of sets in μSETL.

tion to regular sets, μSETL allows us to define sets with an upper limit on their cardinality. For example, consider the set `command` defined in *Object Tracking* (line 19 in Figure 1). Here, the cardinality of `command` is restricted to 1. By restricting the cardinality of `command` to 1, the set contains the last command received from the base station. For a volatile set, if the size of a set is restricted to *x*, only the latest *x* members (determined by the time stamps of the members) of that set will be available. For a non-volatile set, only the first *x* members will be available according to the order in which they are inserted into the set. This language construct allows us to define singleton sets by restricting the cardinality to 1. It can also be used for debugging purposes where the size of a set growing beyond the specified size could indicate a possible bug.

3.2.4 New Syntactic Constructs

As mentioned in Section 3.1, a μSETL program is essentially a set of event handlers. Events can be triggered by an expired timer, the arrival of new sensor data, *etc*. To facilitate defining event handlers, μSETL includes two new constructs, *periodic block* and *monitor block*, described below.

Periodic Block: In WSNs, certain tasks often need to be executed periodically. To express such tasks using μSETL, we introduce the notion of *periodic blocks*. The body of a periodic block can be considered as an event handler that is executed whenever a timer expires. Note that the semantics of μSETL call for the timer to be automatically rescheduled upon expiry. The auto-generated code produced by the μSETL compiler contains the necessary timer code to handle the execution of the block. For example, consider the following code segment from *Object Tracking* (lines 4 – 10):

```
period 5000 do
  ......
end
```

Here, the periodic block is executed every 5,000 milliseconds. Optionally, the maximum number of invocations of a periodic block can also be specified in the block header [14]. Periodic blocks cannot be nested or enclosed in other blocks or loops.

Monitor Block: In WSNs, certain actions may need to be taken when the state of the system changes. For example, if the neighborhood of a node changes, the node's routing table may need to be updated. *Monitor blocks* make it easy to express such actions.

Recall that μSETL distinguishes between volatile and non-volatile sets; the state of volatile variables and sets is maintained by the RTE (Section 4.2.2). A monitor block is conditioned over such volatile data, and is executed whenever that data changes. Consider the following example:

```
monitor neighborhood, xyz do
  // Do something
end
```

Here, the block is executed every time there is a change in either `neighborhood` (a volatile set) or `xyz` (a volatile variable). In Section 4.3, we describe how the RTE detects any changes in the variables that are being monitored and notifies the corresponding application about the changes. Note that a monitor block can only be conditioned on volatile variables, a constraint that is checked at compile time. Similar to periodic blocks, monitor blocks are also non-nestable and can not be surrounded by other loops, blocks, or conditional statements. In *Object Tracking*, we use a monitor loop to check for, and execute, commands that are received from the base station (lines 20 – 22).

4. μSETL IMPLEMENTATION

To efficiently implement the μSETL programming model, we developed a compiler that translates a μSETL program into C code, and a run-time system that provides several of the key capabilities necessary to support the μSETL model (such as set management, timers, *etc*.). Section 4.1 provides an overview of the μSETL architecture, Section 4.2 discusses the μSETL compiler, and Section 4.3 describes the run-time environment we developed.

4.1 Overview of the Architecture

Figure 2 gives a high-level overview of the μSETL architecture and shows the interactions between its different components. At

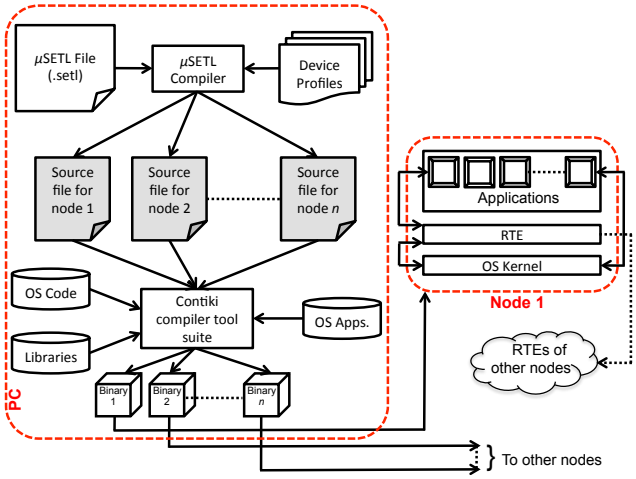

Figure 2: An overview of the μSETL architecture.

the heart of the architecture is the μSETL compiler, which takes as input a μSETL program (a `.setl` file) and a set of *device profiles*. The language specification for writing a μSETL program was described in Section 3. A device profile contains information about a device, such as available sensors, type of the device, *etc*. This information is used by the compiler to generate node-specific code (Section 4.2.3), if possible. If a new device joins the network, a new device profile is created for it. Similarly, if a device is removed from the network, its corresponding device profile is no longer used.

The μSETL compiler was implemented using `lex` and `yacc`, two common Unix tools. The compiler generates C code written for Contiki, a lightweight operating system for resource-limited, networked embedded systems [6]. It provides a thread-like programming style on top of an event-driven kernel through the use of lightweight protothreads [15]. Contiki also comes with a simulator, Cooja [7], which we used for simulating many of our experiments. Cooja is highly extensible through the use of external plugins. It can be used for network-level, OS-level, and instruction-level simulation of sensor nodes running Contiki.

The Contiki compiler suite takes the source files emitted by the μSETL compiler and generates binaries for the targeted nodes in the network. Every node that runs a μSETL program also has a μSETL run-time environment (RTE) installed (Section 4.3). The

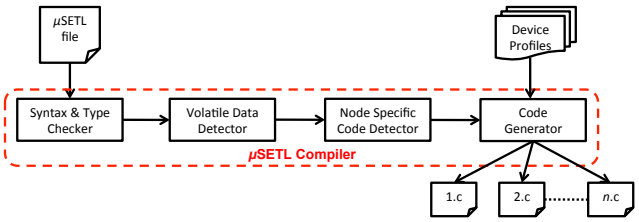

Figure 3: Key components of the μSETL compiler.

RTE provides the necessary communication and data collection mechanisms to applications.

4.2 μSETL Compiler

The μSETL compiler takes an input file (with `.setl` extension) and emits node-specific source code. The μSETL compiler is a source-to-source compiler, transforming μSETL programs into C code (which, as shown in Figure 2, is then compiled with the Contiki compiler tool suite). As part of this translation, the μSETL compiler performs two key tasks: *volatile data detection* and *node-specific code generation*. Figure 3 shows the key components of the μSETL compiler, which are described below.

4.2.1 Type Inference

μSETL is a statically and weakly-typed language where the programmer specifies the types of the variables with basic types. For variables that are of type *set*, types are automatically inferred by the compiler. The other sources of type inference are predefined macros and functions (Table 3). For example, if an untyped variable v is assigned the value of `base_station`, the type of v is inferred to be an unsigned integer. If the type of a variable cannot be determined (*e.g.*, the programmer did not specify the type and it cannot be inferred) or has a conflicting type (*e.g.*, a set is assigned to a variable with integer type), the compiler reports an error.

4.2.2 Volatile Data Detection

In μSETL, it is possible to define a set whose members change over time. For example, a set may be composed of sensor readings (*e.g.*, temperature) from other nodes, which may need to be periodically updated. Hence, evaluating the set membership only during the definition of the set is not enough. Each time the set needs to be used, there has to be a mechanism to ensure that the set's contents are up-to-date. We refer to such sets as *volatile sets*. Variables that are of types other than *set* can also be volatile. For example, the average of the temperature values in the aforementioned set is also volatile. The μSETL compiler can detect such variables and the run-time environment provides necessary support to ensure that the volatile data remain up-to-date (Section 4.3).

To determine whether a variable X is volatile, we consider two cases. In the first case, X is a set. We define a binary relation, $\leftarrow$, between two sets a and b. $a \leftarrow b$ means, the membership of a depends on the membership of b. Examples of such dependencies can be found in Section 5. We also define the *dependency closure* of X, $V^*(X)$, as follows:

$$
\begin{aligned}
S &= \text{The set of all sets defined or used in the program} \\
V_0(X) &= \{X\} \\
V_{i+1}(X) &= V_i(X) \cup \{y : x \leftarrow y \wedge y \in S \wedge x \in V_i(X)\} \\
V^*(X) &= \bigcup_{i \in \mathbb{N}} V_i(X)
\end{aligned}
$$

X is volatile if any of the following is true:

1. Membership of X is defined in terms of any attributes of other nodes that may change over time (*e.g.*, distance).
2. Members of X contain information about some resources that may change over time (*e.g.*, sensor reading).
3. X is a member of $V^*(Y)$, where Y meets criteria 1 or 2 above.

In the second case, where X is a non-set variable, X is volatile if either of the following is true:

1. X refers to some information that may change over time (*e.g.*, distance of a node, sensor reading, *etc*.)
2. X is defined in terms of other volatile variables.

It is possible to force the μSETL compiler to ignore the volatility of a variable explicitly by preceding the definition of that variable with the keyword `novolatile` [14].

Types of Volatile Data: We categorize volatile data in μSETL into three classes:

1. *Distance information:* A majority of WSN applications require inter-node distance information, such as finding the closest node, defining a neighborhood, *etc*. For example, the following code snippet defines the set of immediate neighbors of a node:

```
neighbors := {i | distance(i) == 1}
```

2. *Resource data:* This type of data contains information about various types of resources of a node, most notably, sensors that are present (*e.g.*, temperature, light). For example, the following code snippet defines a set containing temperature data from all the nodes in the network:

```
temp := {N(i).temp | i IN all}
```

3. *Data received using `receive()`:* A node can send various types of information (other than resource or distance-related data) to other nodes in the network using the `send()` function (Table 3), which recipients can receive using the `receive()` function. For example, the following code snippet defines a set that contains all the integers received from the base station:

```
received := {i:i8 | receive(base_station, i)}
```

More examples of using volatile data can be found in the code for *Object Tracking* (*e.g.*, `received`, `targets_on`, *etc*.).

4.2.3 Node-Specific Code Generation

The μSETL language constructs allow programmers to explicitly name the target node(s) for a block of code. For example, in *Object Tracking*, the μSETL code in lines 1 – 11 looks like the following:

```
@base_station@ #
    ......
#
```

The code between the hash signs is intended only for the base station. The μSETL compiler can detect such code and include it in the source code of the corresponding nodes. Node-specific code can also be generated by analyzing expressions that are *constant*. Only conditional statements (in our case, `if-then-else`) can create code segments that are node-specific. Figure 4 shows an example of node-specific code generation. As shown, the code segment B is specific only to the node with ID 0, identifiable by the surrounding `if` statement. The μSETL compiler retains this code segment only in the source code targeted for node 0, while other nodes do not have this segment. Complex conditional statements that have constant expressions (*e.g.*, `node_id == 0 || average(temp) > 25`) can also be handled by the μSETL compiler.

There are other types of expressions that may create opportunities for generating node-specific code. Such expressions are not

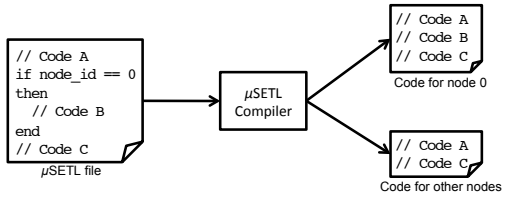

Figure 4: Node-specific code generation in μSETL.

constants, but change less frequently over the lifetime of a network. We refer to these expressions as *quasi-constants*. For example, a code segment may be conditioned on whether the node has a particular resource available (*e.g.*, `if N(node_id).has(CAMERA)`). That code segment should only be dispatched to nodes with the necessary resource available. However, this resource availability may change in the future, in which case an appropriate version of the code needs to be re-dispatched. The μSETL compiler detects quasi-constant expressions and generates node-specific code accordingly. The *device profiles* (Figure 3) of the nodes are used to evaluate quasi-constant expressions (*e.g.*, to check whether `has(CAMERA)` is `true` for a particular node).

4.2.4 Code Generation

After detecting volatile data and node-specific code, the μSETL compiler generates C code that can then be compiled to generate binaries for the Contiki operating system. In addition to translating the μSETL code into C code, the *code generator* needs to perform the following key tasks:

Code Optimization: Since the code is generated in C, many standard compiler optimization techniques are applied by the Contiki compiler tool suite. Our code generator performs some optimizations that will not be performed by the C compiler. For example, if there is an empty *periodic* block, there will still be a timer that will keep triggering periodically. Our code generator will detect and eliminate such empty loops.

Calculating a Variable: Due to the presence of *sets* and *volatile* data, calculating the value of a variable is non trivial. For example, a variable x may depend on a volatile variable v. So, before we use x, we need to evaluate v and re-calculate x. Volatile data detection makes it easier to identify these cases. However, a chain of such dependencies may complicate the detection and lead to unnecessary overhead. To address this problem, we construct the *volatile variable dependency graph* (VDG).

VDG = (V, E) is a directed acyclic graph where V is the set of vertices and E is the set of edges. If there is an edge $e \in E$ from $a \in V$ to $b \in V$, then the definition of b is dependent upon the definition of a. Hence, a needs to be evaluated before we can evaluate b. For example, consider the code shown in Figure 5. The figure also shows the VDG corresponding to the μSETL code. The VDG also includes temporary sets that are derived from the set definitions. These temporary sets are shown alongside the code.

To evaluate a volatile set or variable x, we construct a depth-first tree, T, rooted at x from the graph (V, E^T). We topologically sort the nodes V' in T such that if $a \in V'$ precedes $b \in V'$ in the sorted order, then the definition of b is dependent on the definition of a. Hence, a needs to be evaluated before b. The topological sort allows us to ignore redundant relationships in the VDG (shown as dotted edges in Figure 5). The VDG for a μSETL program is not maintained during run time. It is used only for code generation by the μSETL compiler. The VDG allows the compiler to generate code that evaluates variables in the proper order. Note that the

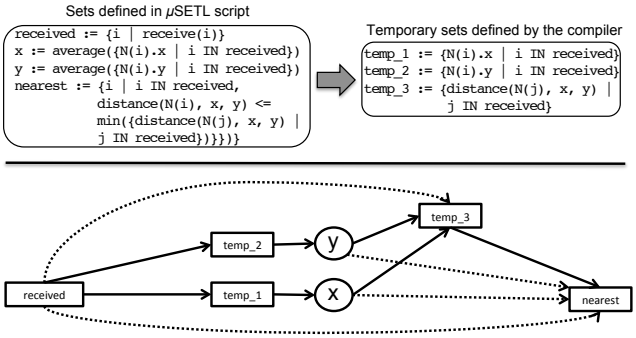

Figure 5: *Volatile Variable Dependency Graph* **generated from μSETL code. The corresponding μSETL code and the temporary sets are shown above the graph. The rectangles in the graph represent the variables that are sets while the circles represent other types of variables.**

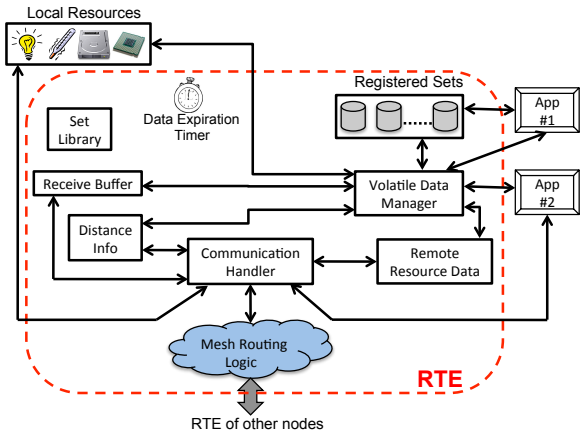

Figure 6: The run-time environment (RTE) in μSETL.

VDG only depicts the dependencies among volatile variables, as non-volatile variables do not need to be recalculated on use.

The code generator is also responsible for detecting and initializing timers based on the periodic loops present in the μSETL program, proper data initialization, locking and unlocking volatile variables in right places, adding Contiki specific code, *etc*.

4.3 Run-Time Environment

The μSETL *run-time environment* (RTE) is the component that provides the necessary run-time support for executing a μSETL program. For example, communication with another node, sensing remote data, set operations, *etc*. are all performed with the help of the RTE. The main components of the RTE and the interactions among them are shown in Figure 6. The following sections describe the key components of the RTE.

4.3.1 APIs Provided

The RTE provides a set of functions and macros that makes it easy to write complex applications. For example, to find the hop distance between the current node and a node with ID i, one can simply write `distance(i)`. The detailed mechanism for finding the distance is transparent to the application. Table 3 provides the list of macros and functions that are provided by the RTE. Note that we only list the most common functions and the ones used

	Name	Description
Macro	`all`	The set of all node IDs in the network
	`node_id`	The current node ID
	`base_station`	The base station of the network
Function	`distance(i)`	Hop distance to the node with ID *i* from the current node
	`distance(i, j)`	Hop distance from the node with ID *i* to the node with ID *j*
	`send(n, data)`	Sends `data` to the node with ID *n*. *n* can be a set of IDs as well, in which case, `data` is sent to all members of *n*
	`average(s)`	Calculates the average of the members of the set `s`. Return type depends on the type of the members of `s`
	`max(s)`	Returns the maximum element of the set `s`
	`min(s)`	Returns the minimum element of the set `s`
	`execute(c)`	Executes the command `c` locally (*e.g.*, `CAMERA_ON`, `CAMERA_OFF`). `c` can also be a set of commands
	`has(n, r)`	Returns 1 if node `n` has the resource `r` (*e.g.*, CAMERA, TEMP_SENSOR), 0 otherwise
	`print(s)`	Generates a `printf` statement that prints `s`. Here, `s` can be a set
	`receive(n, d)`	Gets the data `d` from the RTE sent by node *n* using `send()`
	`set_param(p, v)`	Sets the value of the RTE parameter `p` (*e.g.*, TS_PERIOD to set T_s, TD_PERIOD to set T_d) to `v`

Table 3: Common functions and macros provided by the RTE.

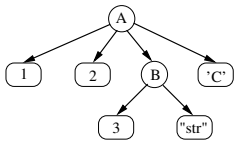

Figure 7: Set representation in μSETL. The set represented here is `{1, 2, {3, "str"}, 'C'}`. Circles represent a set while rectangles represent atomic elements of a set.

in our experiments. It is possible to extend this list with a minor modification of the RTE. The RTE also contains the *set library* that provides necessary support to handle all set-related operations (*e.g.*, set union, subtraction, *etc*.). Sets are represented using a tree-like data structure as shown in Figure 7.

4.3.2 Communication Handler

All communication to and from a μSETL application is handled by the RTE. Examples of such communication include sending a packet to another node, querying another node for a sensor reading, *etc*. Figure 6 shows how the *communication handler* coordinates all the communication for a node. It buffers all outgoing communication packets to avoid conflicts due to simultaneous requests from applications. Note that each node only has a single RTE, which is used by all the applications on the node. Also, a μSETL application cannot directly address another μSETL application.

4.3.3 Volatile Data Manager

The *volatile data manager* (VDM) in the RTE is responsible for managing the volatile data used by applications. Even though volatile data are stored by the applications themselves, the VDM provides necessary mechanisms for updating them. This includes notifying the application about an update, fetching a sensor reading from a remote node to update a variable, *etc*. A brief discussion of how the VDM performs these tasks is given below.

Set Registration: Each set that is volatile is registered with the RTE when an application is initialized. The RTE stores the set identifier (an integer), source Contiki process identifier, and the type of dependency it has. The type of dependency is determined by the type of volatile data used to construct the set at compile time. A set may have multiple types of dependencies. We do not register a primitive type of volatile data (*e.g.*, an integer). If such a variable is defined in terms of another volatile set, that set is already registered with the RTE. Upon exiting, an application de-registers all of its sets.

Data Update Procedure: In Section 4.2.2, we categorized volatile data into three types: *distance information*, *resource data*, and *receive data*. Each type of data has an associated *data expiration timer* (Figure 6) with it. We refer to the timers for these three types of volatile data as T_d, T_s, and T_r, respectively. To avoid any problems due to synchronized network access, a small random variation is added to the periods. For example, for sensor data, the time period of the corresponding data expiration timer is set to a random value in the range $[T_s(1-\alpha), T_s(1+\alpha)]$. We used $\alpha = 0.3$ for our implementation. It should be noted that these timers determine the duration for which the corresponding data is considered valid after arriving at a node. Thus, it does not include the time between when the data was generated at the source node and when it arrives at the destination node. When a timer expires, the corresponding data is marked as invalid. When distance information becomes invalid, a *ping request* is sent to the corresponding source. If a node receives a ping request, it sends a *ping reply* to the inquirer. Similarly, if resource data expires, a *resource request* is sent to the source and a *resource reply* is sent back as reply. If the resource is not available (*e.g.*, a node without a temperature sensor may receive a request for temperature data), an error code is sent instead. For local resource data, it is directly fetched from the current node. When *receive* data expires, no request is sent to the source since this type of data is only generated when the source invokes `send()`.

Every time the RTE has new volatile data, it finds all registered sets that are dependent on this type of data. Each application that owns such a set is notified via a callback function about the update. An application re-constructs a volatile set at its convenience (*e.g.*, whenever it uses the set). Note that an update notification for a set only means that new data of type similar to the type that the set is dependent upon are available. If an application decides to evaluate a volatile variable, it simply queries the RTE for the corresponding data. If the data is already stored, it is returned to the application. Otherwise, the RTE starts maintaining that data in an internal buffer by starting the corresponding data expiration timer.

The data update procedure described above provides several benefits over other update mechanisms. If data is updated by the RTE on demand (*i.e.,* only when an application needs the data), then applications will need to wait for the data to arrive. In contrast, the RTE caches and periodically updates remote data without the knowledge of the applications. While the freshness of data is compromised in this approach, it avoids a lot of delay and synchronization issues. However, there are certain types of applications (*e.g.*, time synchronization) where this freshness may prove to be crucial.

Data Consistency: Since μSETL does not have any concept of shared variables and volatile data is polled periodically rather than being pushed, there is the possibility of inconsistent data. This is true for any programming language that does not support shared variables. Some macroprogramming languages allow shared variables and use synchronous methods [16] or other locking mechanisms [17] to ensure consistency. The use of shared variables are known to introduce significant overheads due to the extra message passing needed to keep them consistent. There are different ways in which inconsistencies may arise in a μSETL program. Consider

the following code snippet:

```
1  temp := {N(i).temp | i IN all};
2  for i:f IN temp do
3    // Do something
4  end
```

First, different nodes running this code segment may have different values for the members of `temp`. Second, inconsistency can occur when a loop is conditioned upon a volatile set. For example, in the above code segment, the `for` loop in line 2 depends on the size of the volatile set `temp`. The RTE notifies the application of any changes in `temp` at run time. If `temp` is not updated carefully, unexpected behavior (*e.g.*, looping forever) may occur.

We previous discussed how volatile data can be dependent on three types of information: distance, resource data, and `receive()` data. Only the distance and resource data dependencies can cause the first type of inconsistency mentioned above. Data received using `receive()` are local properties of a node and hence, are immune to this problem. Let us consider two instances of a variable v in node A and node B. Since T_d and T_s are chosen randomly, they may be different for the two nodes. Let us assume the period of the data expiration timer for v to be T_A^v and T_B^v for nodes A and B, respectively. We define the *age* of v in a node n as the difference between the time when the value of v was created at the source and the time when v was used by the application in n. We also define L_n^v as the network latency to transfer a value of v from the source to n. Assuming expired values are not used, the expected maximum age difference between two used values of v in A and B can be $\max(E(T_A^v + L_A^v), E(T_B^v + L_B^v)) + p.E(\epsilon)$. Here, p is the probability that v is changed after the update, which obviates the need for recalculating any sets that have v as a member, and ϵ is the time necessary to do the recalculation. For a set, S, used in node, n, the maximum age difference between the members of S is $\max_{x \in X} E(T_n^x + L_n^x) + p.E(\epsilon)$.

To avoid inconsistencies that may arise from loops that are conditioned upon volatile variables, μSETL uses locks in appropriate places to delay updates of such sets until the loop execution is finished. The μSETL compiler automatically detects appropriate places to put locking and unlocking mechanisms in the auto-generated code. While calculating the age difference between two instances of a variable on two different nodes, we also have to consider the loop execution time in addition to the other components discussed above. For example, consider the following code snippet:

```
1  a := {i | distance(i) == 1};
2  b := {N(i).temp | i IN a};
3  for i:i8 IN a do
4    ...
5    for j:i8 IN {0...10} do
6      // Use b here
7    end
8  end
```

Here, set `a` is locked before the first `for` loop (line 3). Set `b` cannot cause any inconsistency and is, therefore, not locked. However, the calculation of `b` may use an old snapshot of `a`. After the outermost loop execution is finished (line 8), `a` is unlocked again.

5. EVALUATION

We evaluated μSETL using three applications developed in Contiki [6] and tested them using the Cooja [7] simulator as well as on real Telos motes. The three applications were a modified version of Surge (*Modified Surge*) [18], an implementation of a user-defined routing protocol (*Routing Table*), and tracking a light source with actuation (*Object Tracking*). These experiments demonstrate how μSETL can be used to develop simple and common sensor network applications (*e.g.*, Modified Surge) as well as more complex ones (*e.g.*, Object Tracking).

Program	Code size (Approximate # of lines)	
	Baseline Version	μSETL Program
Modified Surge	195	6
Routing Table	201	3
Object Tracking	135 + 115	11 + 12

Table 4: Code size comparison for μSETL applications. The code for *Object Tracking* shows the code size separately for the base station and the follower nodes.

μSETL significantly reduces programmers' effort in writing an application. Table 4 shows how μSETL programs compare against their baseline versions (versions that would have been written manually in Contiki) in terms of lines of code. In many cases, the μSETL program was less than 5% the size of the corresponding baseline version. While number of lines may not always be the best metric for judging the strength of a programming abstraction, it does give an idea of relative conciseness and ease of programming. Other works [19] have used metrics such as number of variables, number of functions, *etc*., to analyze code complexity. Unfortunately, such metrics are only useful when comparing programming models that adopt the same general paradigm (*e.g.,* comparing two imperative programming models). When comparing programming models that use different paradigms (*e.g.,* declarative *vs.* imperative), it is difficult to distinguish between advantages provided by the abstraction (μSETL's set and event abstractions) and those inherent to the programming paradigm (μSETL's set definitions are especially concise due to their declarative nature).

Table 5 shows the memory consumption (both ROM and RAM) of the applications we used in our experiments, both for the μSETL and the baseline versions. These statistics were collected from simulations running in Cooja for 2,000 seconds. If applicable, memory footprint for the node with the maximum consumption is reported. The table also lists the ROM size for the two key components of the μSETL runtime system: the RTE and the set library. These two modules are always included in a μSETL program. As a reference, a simple Contiki program having only a single `printf` statement uses 20,370 bytes of ROM when compiled using the default settings for TelosB platform. In all cases except *Routing Table*, the μSETL versions consumed less RAM than their baseline counterparts. The reason is that Routing Table contained a number of explicit and implicit set definitions which contributed towards the added overhead. Overheads other than the memory consumption (*e.g.*, number of packets transmitted by the RTE) and performance metrics are discussed in later sections as we describe the applications used in the experiments. All the programs used in our experiments were built using the default compilation settings for the TelosB platform in version 2.x of Contiki. All our experiments used the Rime [6] networking stack for communication. Unless otherwise specified, the values of T_d, T_s, and T_r were set to 60, 60, and 20 seconds respectively. Experiments were repeated five times and results averaged.

5.1 Case Study #1: Modified Surge

Surge [18] is a data collection application where a base station periodically gathers sensor data from a set of distributed follower nodes. The level of abstraction offered by μSETL makes it very easy to write Surge-like applications. Figure 8 shows how different versions of Surge can be implemented using only a few lines of μSETL code. In *Distributed Surge* (Figure 8(a)), each follower

Application / Module			Memory Usage (bytes) RAM	ROM
Set library			-	1,314
RTE			-	3,568
Modified Surge	Baseline		1,021	29,508
	μSETL		981	34,592
Routing Table	Baseline		1,475	26,262
	μSETL		1,848	34,960
Object Tracking	Baseline	Base Station	731	25,756
		Sender	160	25,594
	μSETL	Base Station	734	35,508
		Sender	646	34,486

Table 5: Memory footprint for different applications.

node independently samples its temperature sensor every 4 seconds and forwards the data to the base station. This simple version does not use any set and the RTE does not have to manage any volatile data. In the centralized version of Surge (Figure 8(b)), the base station actively gathers the data from the followers. It stores the collected data in a set (line 2) and whenever the set changes, the collected data is printed to the serial port. The change is detected by using a *monitor block*. To keep the data sampling rate consistent with the distributed version, the centralized version overrides (line 1) the default value of T_s and sets it to 4 seconds. However, in this experiment, we used a modified version of Surge, called *Modified Surge*, where a node periodically (every 4 seconds) collects temperature readings from its neighbors and forwards the average of the collected data towards the base station. This new version uses a lot of the services offered by the RTE and the set library. Therefore, it was useful in micro-benchmarking the μSETL architecture.

The number of lines in the μSETL code for *Modified Surge* was 97% less than the corresponding baseline version (Table 4). We successfully simulated this application in Cooja with a network consisting of 30 nodes. However, to benchmark μSETL to analyze packet loss, memory consumption (Table 5), energy consumption, *etc*., we used a linear topology with a smaller number of nodes as shown in Figure 9. Figure 10 shows the number of packets received at the base station from each of the nodes in the network in a 200 second time window. As shown in the figure, the packet delivery numbers for the μSETL version were very similar to the baseline version of Modified Surge. As expected, nodes that were further from the base station experienced more packet loss than nodes that were closer. This was true for both the μSETL and the baseline versions of Modified Surge. Figure 11 shows the energy consumed in a 200 second time window by each node other than the base station (the base station in μSETL and baseline version contained the same non μSETL program). As shown in the figure, the μSETL version consumed slightly more energy than the baseline implementation. This was mostly due to the overhead introduced by the additional communication performed (*e.g.*, sending *ping packets* to maintain connectivity information) by the RTE.

As mentioned in Section 4.3.2, the RTE is responsible for all the communication in a μSETL program. In addition to exchanging resource data (in this case, temperature) for the application, it also maintains up-to-date connectivity information to other nodes. Other than the packets sent by applications, the RTE itself sends four different types of packets: *ping request*, *ping reply*, *resource request*, and *resource reply*. Figure 12 shows a breakdown of the number of these types of packets sent from the RTE while running Modified Surge for 200 seconds.

Due to the overhead of set-related operations and volatile data management (Section 4.3.3), it is possible that the temperature data used in the μSETL program were older than the data used by the baseline version. We compared the average age of the temperature data used in both versions. *Age* is defined as the difference between the time when a data sample was sensed at the source node and the time when the data was used to calculate the average temperature. As shown in Figure 13, data used in μSETL version was slightly older than the baseline version.

T_d, T_s, and T_r are the three parameters used by the RTE to decide the lifetime of various types of volatile data (Section 4.3.3). Setting proper values for these parameters is crucial to achieve the desired level of consistency for these types of data. We analyzed the effect of T_s (the timeout for sensor data) on the age of temperature data used in Modified Surge. Four different values were used for T_s: 60, 30, 15, and 5 seconds. As shown in Figure 14, reducing the value from 60 seconds to 30 seconds reduced the average age for the temperature data (*i.e.*, fresher data is collected by the base station). Interestingly, reducing T_s to 15 seconds *increases* the average data age for some nodes. This counter intuitive result is easily explained: quick expiration of the temperature data leads to increased resource requests, increasing the network traffic, packet collisions and therefore, resource retrieval time. Reducing T_s to 5 seconds greatly exacerbates this effect. This shows that T_s can only be lowered up to a certain value due to the restrictions imposed by the network. Similar conclusions can be drawn for T_d. However, changing T_r will not affect the consistency across the network since *receive data* age is a node-local property (Section 4.3.3). Both the above experiments related to data age ran for 500 seconds.

5.2 Case Study #2: Routing Table

Macroprogramming languages provide a global view of the network instead of a node-centric view and hence typically can not express node-level interactions. As a result, they can not be used to perform certain types of tasks, for example, defining a routing protocol, time synchronization, *etc* [16]. On the other hand, microprogramming languages can design these applications, but at the cost of more programmer effort. By raising the level of abstraction of microprogramming, μSETL can express these applications naturally and concisely. In this experiment, we developed a simple mesh routing protocol using μSETL. Our routing protocol was a traditional minimum hop-count based routing protocol. Figure 15 shows the μSETL program that was used to generate the routing table for this protocol. The routing table *rtable* was a set of maps. If $[x, y] \in$ *rtable*, then y is the next hop on the minimum length path towards x from the current node.

We used nine nodes as shown in Figure 16 and simulated the routing table in Cooja for 500 seconds. Figure 17 shows the number of valid entries in the routing table for each node. After the initial phase (around 70 seconds), the number of valid entries in each node largely stabilized. It should be noted that `rtable` is a volatile set and hence, the routing table entries were periodically updated by the RTE. Therefore, this program will work for an environment where nodes are mobile. Instead of hop count, any other routing metric could also be used for this program. Since this protocol operates in the user space, it can be used to easily construct logical subnets on top of the physical network. For example, the routing metric could be based on the number of hops with the added restriction that the nodes on a route must have a light sensor. Thus, the logical subnet will consist only of nodes with light sensors.

5.3 Case Study #3: Object Tracking

The goal of this experiment was to track a light source in a sensor network, as we described in Section 3.2.1. We deployed eight TelosB motes in a room with normal lighting conditions as shown in Figure 18. We moved a flashlight along the dotted line shown

```
1  period 4000 do
2    send(base_station, N(node_id).temp);
3  end
4
5
6
```

(a) *Distributed Surge*

```
set_param(TS_PERIOD, 4000);
temp := {N(i).temp | i IN all};
monitor temp do
  print(temp);
end
```

(b) *Centralized Surge*

```
neighbors := {i | distance(i) == 1};
temp := {N(i).temp | i IN neighbors};
period 4000 do
  avg:i8 := average(temp);
  send(base_station, avg);
end
```

(c) *Modified Surge*

Figure 8: μSETL code for different versions of Surge.

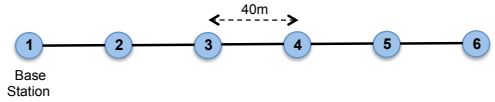

Figure 9: Topology used for *Modified Surge*. Nodes had a transmission range of 50m and an interference range of 100m.

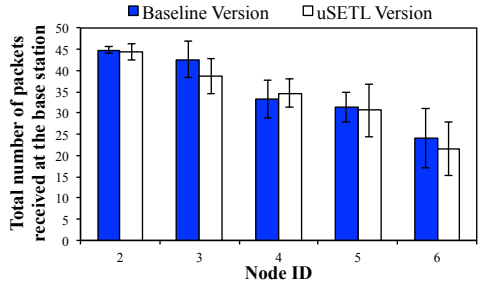

Figure 10: Total number of data packets received at the base station for *Modified Surge*.

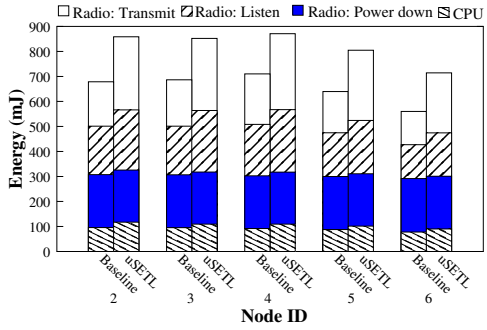

Figure 11: Energy consumed by the nodes for *Modified Surge*.

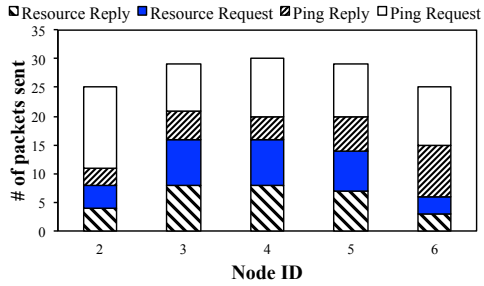

Figure 12: Number of non-data packets sent by the RTE on each node for *Modified Surge*.

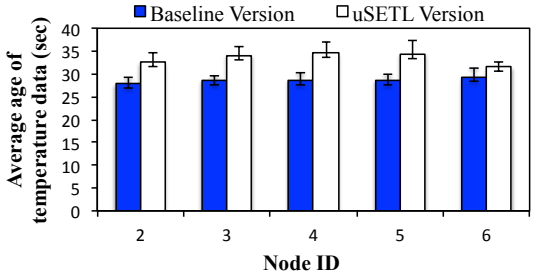

Figure 13: Comparison of the age of temperature data between the μSETL and the baseline versions of *Modified Surge*.

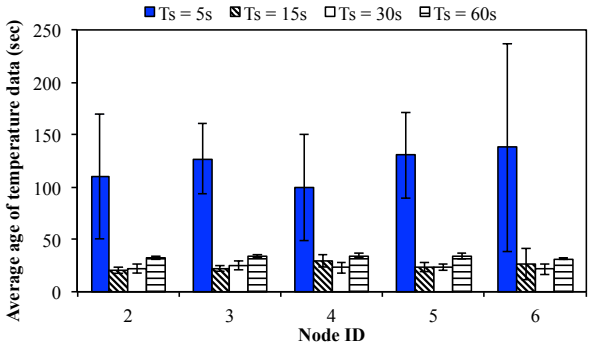

Figure 14: Effect of T_s on the age of temperature data for *Modified Surge*.

```
1  nodes := all - node_id;
2  neighbors := {i:i8 | distance(i) == 1};
3  rtable := {[x:i8, y:i8] | x IN nodes, y IN neighbors,
              distance(y, x) == min({distance(i, x)
              | i IN neighbors})};
```

Figure 15: μSETL code for *Routing Table*.

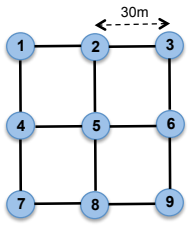

Figure 16: Topology used in *Routing Table*. Nodes had a transmission range of 35m and an interference range of 50m.

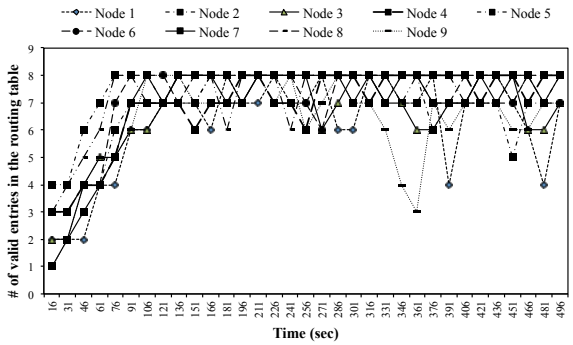

Figure 17: Number of valid entries in each node's routing table.

Programming framework	# of lines	Application
MacroLab [16]	17	Object tracking without actuation
Pleiades [17]	50	Finding an empty spot for street parking
Abstract Regions [20]	30	Object tracking without actuation
Regiment [21]	10	Plume monitoring
EnviroSuite [22]	40	Object tracking without actuation
μSETL	23	Object tracking with actuation
	7	Simple object tracking without actuation

Table 6: Approximate program size for *Object tracking* and similar applications in different programming frameworks.

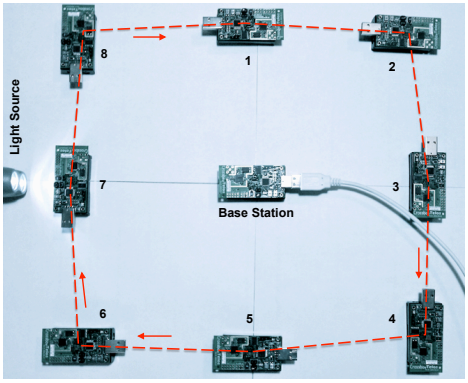

Figure 18: Experimental setup for *Object Tracking*.

in the figure. Since our nodes were not equipped with cameras, we only turned on or off LEDs to track the light source. The μSETL code that implements this application is shown in Figure 1.

Figure 19 shows how the base station selected and de-selected nodes as the closest ones to the light source. For this experiment, the flashlight was held close to a node for 15 – 40 seconds before it was moved to another node. This experiment demonstrated some key features of μSETL. The first was node-specific code generation (Section 4.2.3). As shown in Figure 1, the code for the base-station was separate from that of the follower nodes. Also, the follower nodes were required to have a light sensor (line 12). The μSETL compiler provided separate binaries for the base station and the followers, eliminating superfluous code. The other key μSETL feature that was used in this experiment was the use of a *monitor block*. The nodes were storing the command to control the camera (LED in our experiment) in a singleton set called `command` (line 19). Instead of polling for new commands, we used a monitor block for `command`. Whenever there was new data available for `command`, the RTE notified the application to take proper action (in this case, to invoke

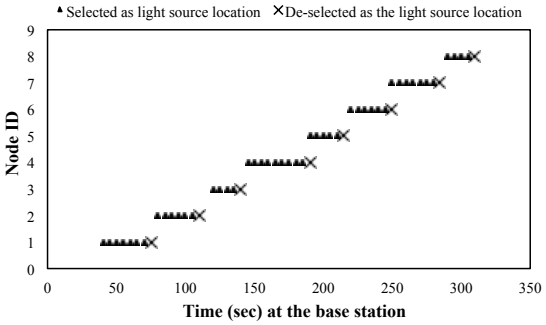

Figure 19: Locating the light source in *Object Tracking*.

```
1  period 4000 do
2     if (N(node_id).light == max({N(i).light |
                                  i IN all})) then
3       execute(CAMERA_ON);
4     else
5       execute(CAMERA_OFF);
6     end
7  end
```

Figure 20: Simplified μSETL code for tracking a light source.

`execute()`), avoiding the overhead associated with polling.

Object tracking in macroprogramming languages requires 20 – 50 lines of code without any actuation (Table 6). A microprogramming language (*e.g.*, nesC [8]) would require substantially more code; our baseline Contiki implementation took around 250 lines of code (Table 4). However, using μSETL, we were able to write a 23 line script that could do both object tracking and actuation.

In principle, object tracking can be made even simpler in μSETL. Figure 20 shows the μSETL program for simple object tracking without using any base station or actuation. In this program, a node turns on its camera if it has the highest light sensor reading in the network. Otherwise, it turns off the camera. Although this variant will have higher network traffic than the version shown in Figure 1, it is much simpler. The increased traffic is because all nodes need to know the light sensor readings of all other nodes in the network.

6. RELATED WORK

Depending on the intended application, high-level programming languages (*e.g.*, C, C++) offer different types of abstractions, such as control abstraction, data abstraction *etc*. Since most node-level programming languages for WSNs are variants of these general-purpose high-level languages, they too provide similar abstractions. Detailed surveys on the state-of-the-art in programming approaches for WSNs are provided in [3] and [23]. The choice of programming language for WSNs is usually dictated by the particular operating system used. For example, nesC [8] is used with TinyOS [5], and C with SOS [24] and Contiki [6]. These languages, combined with operating system hooks, provide access to the nodes' hardware and flexible control of the nodes' behavior. However, the level of abstraction offered by these languages is seldom enough to easily express complex applications and thus makes programming WSNs difficult. Virtual machines (*e.g.*, Maté [9], ASVM [25], VM-Star [26]) offer a higher level of abstraction, but their main focus is on ease of reprogramming wireless sensor networks.

To address this issue, a number of macroprogramming solutions have been proposed. MacroLab [16] is a vector-based pro-

gramming abstraction with a global view of the network. Vectors are stored in either a distributed, centralized, or reflected [16] manner based on an analysis of the cost for each representation. Kairos [27] is an imperative programming language where communication occurs by manipulating shared variables at specific nodes. Pleiades [17] extends Kairos' programming model by allowing a program to be partitioned into independent execution units called nodecuts. Different nodecuts may run on different nodes based on a cost analysis. During run time, the flow of execution moves from one node to another if the nodecuts are assigned to different nodes. Pleiades also uses locking and deadlock detection and recovery to ensure serializability. Envirosuite [22] is an object-based programming framework designed for monitoring and tracking applications. Some macroprogramming systems, such as Abstract Regions [20] and Hood [11], are specifically targeted for applications exhibiting spatial locality (*e.g.*, object tracking). Spidey [10] offers an abstraction that allows programmers to create logical neighborhoods based on certain logical properties of the nodes. TinyDB [28] and Cougar [29] provide SQL-like interfaces and view the sensor network as a relational database table. Other declarative macroprogramming systems have been proposed including Regiment [21], DSN [30], and Semantic Streams [31]. Section 2 briefly described the advantages and disadvantages of different programming paradigms for WSNs. μSETL is an attempt to bridge the gap between these paradigms. While allowing a programmer to write event-driven programs from a node-level viewpoint (similar to microprogramming), it also offers a high-level of abstraction (similar to macroprogramming) based on concepts from set theory.

7. CONCLUSIONS AND FUTURE WORK

The key role that CPSs are envisioned to play in our day-to-day lives has brought renewed attention to networked embedded systems such as WSNs. However, programming these systems still remains a major barrier to their widespread adoption and deployment. This paper introduced μSETL, a set-based abstraction for WSN microprogramming that exploits the powerful formalism and expressive power of set theory to address the programmability challenge in WSNs. Experimental results showed that programs written using μSETL featured a significantly decreased source code size (by more than 90%) compared to the corresponding baseline versions.

As part of future work, we plan to extend μSETL in three ways. First, nodes that run a μSETL program currently contain the same RTE. However, not all the components of the RTE are used by all applications. In the future, we would like to generate an application-specific RTE using the μSETL compiler. That will result in a smaller RTE binary. Second, one of the major benefits of μSETL is that programmers can write node-specific code. We would like to deploy a node-specific code dissemination protocol that will make reprogramming of nodes running μSETL applications more convenient. Finally, in the current implementation, the RTE uses a *pull model* to retrieve data from other nodes to update set contents. However, for some applications, a subscription-based *push model* might be more suitable. In that model, nodes can subscribe to other nodes for various kinds of data. Any update of the subscribed data will be pushed to the subscribers by the source, keeping the sets at the subscriber updated. In future versions of μSETL, we plan to provide both the pull and push models of data access as configurable options.

8. ACKNOWLEDGEMENTS

This work was supported in part by NSF grant CNS-0953468 and ECCS-0925851. The views expressed represent those of the authors and do not necessarily reflect the views of the sponsoring agency. We thank the anonymous referees and our shepherd, Luca Mottola, for their insightful comments, suggestions, and feedback.

9. REFERENCES

[1] List of wireless sensor nodes (Wikipedia). `http://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes`.

[2] T. Enwall, "Deploying Wireless Sensor Networks for Industrial Automation & Control." `http://www.eetimes.com/design/industrial-control/4013661/Deploying-Wireless-Sensor-Networks-for-Industrial-Automation-Control`.

[3] L. Mottola and G. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Computing Surveys*, 2011.

[4] J. Schwartz, R. Dewar, E. Schonberg, and E. Dubinsky, *Programming with sets: An introduction to SETL*. Springer-Verlag, 1986.

[5] The TinyOS operating system. `http://www.tinyos.net`.

[6] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *EmNets*, pp. 455–462, 2004.

[7] F. Osterlind, A. Dunkels, J. Eriksson, and N. Finne, "Cross-level sensor network simulation with cooja," in *LCN*, pp. 641–648, 2006.

[8] 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 *PLDI*, pp. 1–11, 2003.

[9] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," in *ASPLOS*, pp. 85–95, 2002.

[10] L. Mottola and G. Picco, "Logical Neighborhoods: A programming abstraction for wireless sensor networks," in *DCOSS*, pp. 150–168, 2006.

[11] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *MobiSys*, pp. 99–110, 2004.

[12] J. Spivey, *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.

[13] D. Turner, "Miranda: A non-strict functional language with polymorphic types," in *Functional Programming Languages and Computer Architecture*, pp. 1–16, Springer, 1985.

[14] M. S. Hossain, A. B. M. A. A. Islam, M. Kulkarni, and V. Raghunathan, "μSETL: A Set-based programming abstraction for wireless sensor networks, " Technical Report #TR-ECE-11-06, School of ECE, Purdue University, 2011. `http://docs.lib.purdue.edu/ecetr/`.

[15] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *SenSys*, pp. 29–42, 2006.

[16] 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*, pp. 225–238, 2008.

[17] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," *PLDI*, vol. 42, no. 6, pp. 200–210, 2007.

[18] A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," in *SenSys*, pp. 14–27, 2003.

[19] L. Mottola, "Programming storage-centric sensor networks with Squirrel," in *IPSN*, pp. 1–12, 2010.

[20] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *NSDI*, pp. 3–3, 2004.

[21] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *IPSN*, pp. 489–498, 2007.

[22] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic, "EnviroSuite: An environmentally immersive programming framework for sensor networks," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 3, pp. 543–576, 2006.

[23] R. Sugihara and R. K. Gupta, "Programming models for sensor networks: A survey," *ACM Trans. Sen. Netw.*, vol. 4, no. 2, pp. 1–29, 2008.

[24] C. Han, R. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *MobiSys*, pp. 163–176, 2005.

[25] P. Levis, D. Gay, and D. Culler, "Active sensor networks," in *NSDI*, pp. 343–356, 2005.

[26] J. Koshy and R. Pandey, "VMSTAR: synthesizing scalable runtime environments for sensor networks," in *SenSys*, pp. 243–254, 2005.

[27] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos," *DCOSS*, pp. 126–140, 2005.

[28] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.

[29] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *SIGMOD Rec.*, vol. 31, no. 3, pp. 9–18, 2002.

[30] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica, "The design and implementation of a declarative sensor network system," in *SenSys*, pp. 175–188, 2007.

[31] K. Whitehouse, F. Zhao, and J. Liu, "Semantic streams: a framework for composable semantic interpretation of sensor data," in *EWSN*, pp. 5–20, 2006.