Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2012, Article ID 490826, 15 pages
doi:10.1155/2012/490826

Research Article

Hybrid Macroprogramming Wireless Networks of Embedded Systems with Declarative Naming

Chalermek Intanagonwiwat

Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok 10330, Thailand

Correspondence should be addressed to Chalermek Intanagonwiwat, chalermek.i@chula.ac.th

Received 16 February 2012; Accepted 11 May 2012

Academic Editor: Sherali Zeadally

Wireless Networks of Embedded Systems (WNES) are notoriously difficult and tedious to program. The difficulty is mostly originated from low-level details in system and network programming. This includes distributedly managing and accessing resources from a dynamic set of nodes in hostile and volatile networks. To simplify WNES programming, we propose *Declarative Resource Naming* (DRN) that abstracts out the mentioned low-level details by programming a WNES in the large (*i.e., macroprogramming*). DRN provides programming simplicity, expressiveness, tunability, on-the-fly reprogrammability, and in-network data aggregation for energy savings. None of existing macroprogramming paradigms supports all of the mentioned features. Furthermore, DRN is an integration of declarative and imperative programming. The low-level details are declaratively abstracted out, but the main algorithm remains procedural. This allows programming simplicity without an adverse impact on the expressiveness. We have implemented and evaluated DRN on two platforms: Smart Message and Maté. Our result indicates that DRN enables programmers to develop energy-efficient applications with the desired flexibility and quality.

1. Introduction

A WNES (e.g., a wireless sensor network) consists of a massive number of resource-constrained wireless nodes that are unattendedly deployed to collect data or to monitor the area in dynamic, hostile environments. Programming WNES for such applications is notoriously difficult and tedious because of the low-level details in system and network programming. These low-level details include distributedly discovering, managing, and accessing remote resources as well as routing in a dynamic set of nodes while maintaining low energy consumption and memory usage.

Several programming abstractions have been proposed in literature to hide these low-level details from the programmers. Of particular interest are approaches to program WNES in the large (i.e., macroprogramming). Unlike other abstractions, these macroprogramming abstractions allow programmers to take a centralized view of programming a distributed system rather than a distributed view. A macrocompiler is normally required for translating a centralized-view macroprogram into a distributed version for execution. The macrocompiler is also responsible for automatically generating the mentioned low-level details in the executable distributed code.

Macroprogramming abstractions can be divided into two subclasses: node-independent and node-dependent. In node-independent subclasses, WNES is declaratively programmed as a whole or a unit (e.g., a database). Examples of node-independent abstractions are TinyDB [1], Cougar [2, 3], and Sense2P [4, 5]. By abstracting a WNES as a database, WNES programming is reduced to database querying.

Conversely, in node-dependent subclasses, WNES are programmed as a collection of nodes. These abstractions enable programming tasks that are more complicated than database-like querying. Examples of node-dependent abstractions include Kairos [6], Split-C [7], SP [8, 9], Regiment [10], Macrolab [11], and EcoCast [12]. Most of these works (except Regiment and Macrolab) do not support in-network data aggregation for energy savings. Even though Regiment and MacroLab do, they do not address the on-the-fly reprogrammability issue.

In this paper, we propose Declarative Resource Naming (DRN) (an initial design of this work appears in Algosensors [13]), a hybrid macroprogramming approach that supports simple tasks (e.g., database-like querying), and difficult tasks as well as data aggregation mechanisms for energy savings. DRN is an integration between declarative and imperative programming. The low-level details are declaratively abstracted out whereas the core algorithm remains procedural. Our abstraction allows programmers to declaratively describe a dynamic set of nodes by their run-time properties and to map this set to a variable. To access the desired resources on nodes in the set, we can simply refer to the mapped variable. Therefore, remote resource access is simplified to only variable access that is completely network-transparent. DRN provides both sequential and parallel access to the desired set. Parallel access reduces the total access time and energy consumption because it enables data aggregation in the network. Additionally, we can associate each set with tuning parameters (e.g., timeout, energy budget) to bound access time or to tune resource consumption.

Given that WNES may be deployed in dynamic, hostile environments, and also that we may not be able to physically reach the nodes, it is necessary that we can remotely program these unattended nodes on the fly. Systems based on code migration are preferable because programs can be propagated to target nodes without human intervention or system rebooting. Examples of such systems include Smart Messages (SM) [14], SensorWare [15], and Maté [16]. Therefore, we have implemented our DRN run-time library on two mobile-agent platforms: SM and Maté. SM can run on iPAQs equipped with 802.11 radios whereas Maté can run on motes equipped with 802.15.4 radios.

In addition, we have implemented an object tracking application using our DRN runtime library to illustrate the model's viability. We have also evaluated our DRN runtime library and its tuning knob (i.e., resource binding lifetime). Our result indicates that the tuning knob enables the DRN application to save up to 55.2% of the bytes sent without significant accuracy degradation.

2. Related Work

WNES macroprogramming has been explored earlier by several research efforts, including TinyDB [1], COUGAR [2, 3], Semantic-Streams [17], and Sense2P [4, 5, 18]. The above node-independent abstractions propose programming WNES as a database. Thus, WNES programming is reduced to database-like querying with declarative languages. However, declarative languages are designed for expressing the desired data, but not for expressing the algorithmic details. As a result, they are not appropriate for complex tasks where the core algorithmic details cannot be automatically generated. Conversely, DRN is a hybrid between declarative and imperative languages. Thus, our work can easily support both types of tasks.

Other macroprogramming research efforts are node-dependent. These include Kairos [6], SP [8, 9], Regiment [10], Macrolab [11], and EcoCast [12]. Similar to Split-C [7] for parallel programming, Kairos provides a facility to sequentially access remote variables for WNES programming. Unlike Split-C and Kairos, DRN can access variables and other resources at declaratively-named nodes in parallel. Accessing resources in parallel significantly reduces the total access time and the overall energy consumption (by enabling data aggregation inside the network).

Our work is mostly influenced by Spatial Programming (SP). DRN and SP simplify resource access as variable access, exposing the space property to the programmers, hiding network details, and supporting imperative programming. However, SP supports only sequential resource access, whereas DRN supports both sequential and parallel access. Additionally, SP is purely imperative programming, but DRN is partially declarative and mostly imperative.

Parallel access is also supported by other works such as EcoCast, Regiment, and Macrolab. Surprisingly, even though EcoCast does access resources in parallel, it does not support in-network data aggregation.

Regiment is a spatiotemporal macroprogramming system based on functional reactive programming paradigms (one form of declarative programming). In Regiment, the whole program can be treated like a math equation that reacts to the input changes. Input is data from a set of nodes that are defined by their location. In this sense, Regiment is very similar to SP. However, Regiment is not welldesigned for applications with highly dynamic behaviors, nonreactive applications, short-lived queries, or mobile-agent-based applications. Conversely, DRN does not suffer from the above limitations.

Macrolab is a Matlab-like macroprogramming framework that provides deployment-specific code decomposition. In Macrolab, every deployment change requires recompilation and reinstallation. This can be troublesome as there is no explicit support for remote-reprogramming the system on the fly. In contrast, DRN is incorporated with two mobile-agent platforms. Thus, our work can certainly handle such changes with ease.

There exist several research efforts on a hybrid of declarative and imperative programming. Examples of such research include embedded SQL [19] and constraint-imperative programming [20]. In embedded SQL, SQL is mainly used for database access, and imperative programming is used for data processing. In a sense, resources in DRN are analogous to the database in embedded SQL where declarative accesses are appropriate. In constraint-imperative programming, variables are confined with conditions about their eligible value. Given that conditions are declaratively described, our resource variables are similar to their constrained variables. Despite the mentioned similarity, DRN, embedded SQL, and constraint imperative programming target different problems, platforms, and environments. Specifically, embedded SQL is designed for data processing on conventional databases, and constraint-imperative programming is designed for computing a solution that matches a particular constraint on traditional systems. In contrast, DRN targets resource naming on highly dynamic WNES.

Table 1: WNES macroprogramming system characteristics.

System	Characteristic				
	Programming model	Node dependency	Supported tasks	In-network data aggregation	On-the-fly reprogramming
Cougar	Declarative (SQL)	Node independent	Relational database queries	Yes	No
TinyDB	Declarative (SQL)	Node independent	Relational database queries	Yes	No
Semantic Streams	Declarative (logic programming)	Node independent	Service queries	No	No
Sense2P	Declarative and imperative (logic programming)	Node independent	Deductive database queries	No	Yes
SP	Imperative (procedural programming)	Node dependent	Space-centric	No	Yes
Kairos	Imperative (procedural programming)	Node dependent	Remote variable access	No	No
EcoCast	Imperative (object-oriented programming)	Node dependent	Interactive group access	No	Yes
Regiment	Declarative (functional programming)	Node dependent	Spatiotemporal	Yes	No
Macrolab	Imperative (Matlab-like)	Node dependent	Deployment specific	Yes	No
DRN	Declarative and imperative (procedural programming with declarative names)	Node dependent	Declarative resource access	Yes	Yes

Hybrid macroprogramming systems also exist in Internet (e.g., XTree [21]). Similar to TinyDB, X-Tree programs the whole system as a database but X-Tree is designed for wide-area sensor systems, not hostile dynamic WNES.

Nevertheless, our work has been influenced by directed diffusion [22–24] and LEACH [25]. This is seen most clearly in the energy savings gained by processing data in the network. Despite this influence on our parallel access, DRN shares several similarities with diffusion. Given that diffusion APIs [26] require declarative data description for publication and subscription, DRN and diffusion are examples of hybrid programming that effectively hides networking details. However, diffusion programming view can be somewhat distributed. This is probably why diffusion is not widely classified as macroprogramming in the community.

We summarize the differences of these macroprogramming approaches in Table 1.

3. What Is the Right Abstraction?

Traditionally, there are two programming styles in computer literature: declarative and imperative. Declarative programming fully abstracts out all algorithmic details. Programmers only specify what they want rather than how to algorithmically obtain the results. The translator and optimizer will then fill in the algorithms. Automatic generation of algorithmic details can be efficient for simple and specific tasks (e.g., database), but is questionable for others. Examples of such an SQL-based approach include COUGAR and TinyDB. Despite its simplicity, declarative programming is not applicable for every WNES application. Imperative programming is more appropriate for complex tasks where efficient algorithmic details are either not obvious, or not easy to generate automatically. For example, it is difficult or even impossible to implement Kalman filters or maximum likelihood algorithms for estimating object locations in SQL because SQL is not designed for expressing algorithmic details.

Declarative and imperative programming function well within their domain and complement one another. Integration of declarative constraints and imperative constructs can form a powerful programming paradigm suitable for both domains. In this paper, we propose that such integration is possible if the declarative abstraction is applied only to some parts of the program.

In general, potential targets for abstraction are (1) parts that are unrelated to the core algorithms (2) common to applications, and (3) tedious for programmers. To identify the abstractable parts, a basic understanding of WNES programs is required. Typically, programs are collections of operations on variables and resources. Given that variables are more frequently accessed, programming languages provide a simpler way to access variables than to access resources.

Not surprisingly, traditional resource access is more tedious, especially in networked systems where there exists a distinction between local and remote resources. Resources are normally bound to nodes that are known a priori. Therefore, in order to specify the remote resources that are of interest, node ids are required. If the node ids are not known, resource discovery is needed. As a result, programmers are required to work on several programming details (e.g., networking, resource discovering, resource accessing).

WNES programming is even more labor-intensive because the resources of interest are specified by their properties at run-time rather than node ids. For example, we may want to access sensors on a particular hill only when the temperature is more than 30 degrees Celsius. In this case, resource discovery in WNES becomes necessary and common rather than optional. The resource property is highly dynamic because the environment—where the temperature can drop below 30 degrees Celsius at any moment—is hostile and volatile. Some resource bindings or mappings may have to be invalidated because the bound resources may no longer match the desired property. But even if the resource property does not change, bound resources may not be accessible because of network dynamics such as node mobility. WNES programs are required to handle changes, invalidate bindings, discover equivalent resources, and bind the newly discovered resources. Given that the above events are frequent in WNES, these resource handlings (e.g., discovering, accessing, rebinding, and networking) are tedious to programmers. Therefore, the resource-related parts of the WNES program are reasonable choices for our declarative abstraction.

4. Declarative Resource Naming

To simplify the programming tasks for WNES, we propose a scheme that will program the WNES as a collection of nodes in a network-transparent manner. As a result, there is no notion of networking, being remote, or local.

4.1. Resource Variable. WNES programming can be simplified by making a resource access as simple as a variable access. In order to do this, we propose *resource variables* (i.e., variables that are mapped and referred to actual nodes). For example, one can write a program to read a light sensor and to control a camera as follows:

```
Resource R, X;
Printf "(light intensity = %f", R⟶ light);
X ⟶ camera = off;
```

In the above example, we assume that the resource variable R contains a light sensor and the resource variable X contains a camera. To read the light intensity, we can simply refer to $R \rightarrow light$. Similarly, the camera can be turned off by assigning off to $X \rightarrow camera$. There is no need for algorithmic detail of resource controls and operations. This example shows that our approach is not only for retrieving data and for pushing data to desired nodes but also for controlling them.

4.2. Declarative Constraint. Understandably, one may wonder to which physical nodes (or resources) these variables R and X) are precisely bound and how programmers know about the individual sensor types. Rather than specifying the node ids for binding, a target resource's desired property can be declaratively indicated with a boolean expression or a predicate. For example, we can specify that R will be bound to light-sensor nodes within the forest with temperatures greater than 30 degrees Celsius.

```
Resource R = ⟨within (location, forest) &&
        temperature >30 &&
        exist(light)⟩
Resource X = ⟨a(b,c)! = 0 && exist (camera)⟩
```

Given that more than one node can match a specified expression, a resource variable is referred to as a set of matching nodes rather than a single one. Location and temperature are local properties (of a node) that are used to determine the node's membership in the set R. Furthermore, we also allow user-defined boolean functions (e.g., function $a()$) in our expression. Such a flexible expression is generally powerful and sufficient for various complex conditions.

4.3. Resource Access. In this section, we illustrate the need for various types of DRN resource access that can be used in different situations. Their advantages and disadvantages are also provided as a guideline for selecting the resource access type that is most suitable for a particular task. We propose two approaches for accessing multiple matching nodes: *sequential* and *parallel*.

Sequential Access. Each element in a set can be referred to using an iterator (similar to an iterator in C++ standard template library). The iterator enables sequential and selective access of resources. For example, one can sequentially read the light intensity of each resource in the set R as follows:

```
Resource R;
Interator i;
Foreach i in R {
Print ("light intensity = %f\n", i ⟶ light);
}
```

However, the sequential readings cannot represent a snapshot of the desired target because the delay in accessing the whole set sequentially can be significant. In particular, the total delay is essentially the summation of all individual access time. Nevertheless, this individual approach is still useful, especially when only some elements in the set are accessed.

Parallel Access. Conversely, in this approach, all resources in the set are simultaneously accessed. This parallel access can be specified using a direct reference to the resource variable as follows:

```
Resouce R;
Printf ("light intensity =%f", R⟶ light);
```

In the above example, the program prints out the light intensity of all nodes in R. The total delay using this parallel approach is reduced to the longest delay of an access. The parallel approach not only reduces the total access time but also provides a much better snapshot of the desired target. Additionally, unlike the sequential approach, this parallel approach exposes an opportunity for the underlying system to perform in-network processing (e.g., data aggregation) that can significantly reduce a system's overall energy consumption [22–24, 27, 28]. An example of data aggregation functions is max(A) whereby the maximum element in A is returned.

```
Resource R;
printf (“max light intensity = %f”,
        max (R ⟶ light));
```

Ideally, the system expends energy only on delivering that max element, not on the others. This delivery can be practically approximated by in-network suppression of the elements whose values are less than that of the previously seen elements of the same access. Suppression will be ineffective or even impossible if the resources are accessed in sequence rather than in parallel.

4.4. Resource Binding. Our model supports two binding types: *dynamic* and *static*.

Dynamic Binding. In our paradigm, code does not need to be written to maintain binding between the physical resources and resource variables. Given that the resource property is constantly changing, rebinding the set of matching nodes is laborious. For example, the set of resources R at time t_1 can be completely different from the set of resources R at time t_2.

```
Resource R = ⟨expression1⟩
Time t1 = get_time ( );
x = Count (R);
. . .
Time t2 = get_time ( )
y = Count (R);
/∗Normally, x ! = y ∗/
```

Rather, it is desirable to simply provide the declarative expression that is associated with the resource variable to describe the resources of interest. In general, a reference to a resource variable implies a resource access. Our semantic of a resource-variable access is rather strict in a sense that the access is only performed on the resource that matches the declarative expression at the time of access. Furthermore, changes in the set of matching nodes do not require attention from programmers. (This is the main difference between our approach and the traditional approach that relies on node ids and OIDs of SNMP.) As a result, to conform with this strict semantic, the underlying system may need to spend significant overhead and excessive energy consumption for ensuring that this reactive binding is up to date. Therefore, we propose options or *tuning knobs* for lessening the semantic in order to save energy. For example, programmers can lessen the semantic by allowing access if the resource is bound in the last t seconds.

```
Resource R = ⟨expression,
                    last_bound_time > now-t⟩
```

Furthermore, programmers can even specify an energy budget to bound the energy consumption of a resource access.

```
Resource R = ⟨expression,
                    energy_budget = 100⟩
```

Other tuning knobs are currently under investigation.

Static Binding. Although the above dynamic binding of resources seems reasonable, one may notice that there are situations where dynamic bindings may not be appropriate. Specifically, we may want to access the previously matched resources that are no longer matched. For example, we may have turned on cameras in area A. However, after a period of time, we may want to turn them off, but some cameras have since been moved out of the area. If area A is included in our declarative expression, those cameras that have since been transferred will no longer match the expression. As a result, we may be unable to turn off the relocated cameras directly using the resource variable.

One solution to the above problem is to rely on the underlying system. For example, we could declare a new resource variable using a usual expression with an additional timing condition.

```
Resource R = ⟨expression1⟩;
Time t1 = get_time();
. . ..
Resource X = ⟨expression1 && time == t1⟩;
```

As long as we know the time of the matching, we can describe the desired resources. A similar solution is to provide the function *last*() that returns the previous set of matching nodes to the caller. Therefore, we can operate on the desired set even though it no longer matches the expression.

```
Resource R = ⟨expression1⟩;
Resource X = last (R);
```

However, both solutions incur excessive overhead as the system is required to maintain all changes of a set at all times.

An alternative solution is to provide explicit instructions for memorizing matching nodes. We propose two explicit mechanisms: the *static* resource and the *iterator*.

Using the static resource variable, we can specify which resources are statically bound. The static resource variable will not be rebound in any circumstances. Therefore, we can maintain any set of resources even though they are no longer matched to the expression.

```
Resource R1;
Static Resource R2 = R1;
/∗ R1 changes over time but R2 does not∗/
```

This explicit instruction is cheaper to implement than last() because the system no longer has to keep all previous values of every resource variable. (Programmers might write “last(last(. . .last((R). . .)))”.) Furthermore, the static resource is intended for memorizing the entire set of matching nodes. To memorize only one resource, an iterator is more appropriate. The value of an iterator does not automatically change without an explicit assignment.

```
Iterator i1 = R1 ⟶ first_element;
```

4.5. Access Timeout. Regardless of binding type, there is no guarantee that every WNES resource access will succeed. Unfortunately, WNES resource access time is unbound, and access failures are usually unavoidable because of network dynamics. Given that there is no response after unbound access time and failures, they cannot be easily differentiated. (This problem is similar to that of TCP. Packet loss and unbound acknowledgment delays are handled using time-out.) Timeout is usually a common technique for handling such problems. Therefore, we propose associating a resource variable with an access timeout. In this model, the access time is monitored for each access. Once an access has timed out, an exception is raised (similar to Java exceptions). It is necessary that the method for handling a time-out is explicitly specified in the *catch* statement.

```
Resource R = ⟨expression1, timeout = 10⟩
Iterator i = R ⟶ first_element;
try {
printf (“light intensity = %f”, i⟶light);
} catch(TimeoutException) {
Printf (“cannot access the light sensor”);
}
```

5. Implementation

There are currently two implementations (of all or part) of this abstraction: DRN on SM and *TinyDRN* on Maté. DRN on SM provides full features described in this paper. SM is a mobile-agent-based reprogramming middleware that runs on IPAQs equipped with 802.11 radios. We have also implemented *TinyDRN*, a bare subset of this abstraction on Maté. Maté is also a mobile-agent-based reprogramming middleware, but it runs on motes equipped with 802.15.4 radios. DRN and TinyDRN are implemented as libraries. Thus, there is no need for macrocompilers. In this section, we briefly overview the SM architecture and its advantages before we explain the detail of our implementation on SM and Maté.

5.1. Smart Message Architecture. Smart Messages (SMs) are mobile agents for wireless networks of embedded systems. They consist of code and data sections (called *bricks*), and a lightweight execution state. Unlike request/reply paradigms, SM applications need to migrate to nodes of interest and execute there. To do this, SMs execute a routing algorithm, carried as a code brick, for determining the next hop toward a node of interest. The code bricks are cached by nodes along the way to reduce the cost of transferring the same code in the future. Over time, this cost is amortized because of temporal and spatial locality of SM applications.

SM architecture consists of Smart Message Virtual Machine (SMVM) and Tag Space. SMVM is basically Sun’s K Virtual Machine (KVM) that is modified to support Tag Space and program migration. SMVM is suitable for mobile devices with resource constraints and with as little as 160KB of memory [29]. Tag Space is a name-based memory region that unifies an interface to I/O and memory on SMVM. I/O and memory can only be accessed through an object called a *tag*. Direct-access instructions will not be recognized by SMVM.

Given that there is no Java thread or preemption in SM, the SM execution model is quite simple. Only one SM is active at a time on an SMVM while the other SMs wait in the queue until the active SM terminates, migrates, or suspends itself.

There are several advantages using SM as our target platform. One significant advantage is on-the-fly reprogrammability. New aggregation functions and predicates can be deployed on the fly. Another advantage is the SM tag, the unified interface to memory and I/O. This feature tremendously simplifies our implementation, especially our variable-like access to resource.

5.2. DRN Implementation Using SM. DRN is implemented as SM run-time libraries. Given that we only use regular SM commands to implement our libraries, we do not have to modify the SM Virtual Machine (SMVM) and do not have to implement a DRN macrocompiler. In the SM platform, each node is a pocket PC running an SMVM. Therefore, each node has interfaces to interact directly with a user. Consequently, we can use any node in the system as a user node.

Our macroprogram is implemented as a Smart Message that is injected at the user node. Generally, an SM program can migrate and execute at any node but, in our implementation, the main SM macroprogram do not migrate. To acquire data from other nodes, this main SM will create child SMs that migrate to those nodes and bring the data back.

When a resource variable is declared with a predicate and a binding lifetime, the main SM does not immediately bind the variable. Instead, the resource variable will be bound on demand when the variable is referred. Given that the set of matching nodes changes over time, binding the variable too early may not be useful. The variable is likely to be rebound at the time of access and overhead in early binding is wasted. This concept is similar to on-demand routing in wireless ad hoc networks.

To bind the variable, nodes that match the predicate have to be discovered (see Algorithm 1). The main SM creates a Discovery SM that contains the given predicate to discover nodes and their routes. The default target region is the whole network. If no target region is specified in the predicate, the Discovery SM floods the network by duplicating itself and migrating to all neighbors of the current node until all nodes are visited.

If there is geographical information about the target region in the predicate, the Discovery SM will migrate to only the neighbor that is closest to the target (Line 1–5). Upon reaching the region, the Discovery SM floods all nodes in the region to check if those nodes match the predicate (see Algorithm 2).

On each visited node, the Discovery SM creates a marking tag for differentiating visited nodes from others (Line 4). The Discovery SM will terminate if it arrives on a node with the marking tag (Line 1–3). In addition to the

```
1: while not in the target region do
2:    create marking tag
3:    migrate to the neighbor closest to the region
4:    create route-to-user tag point to previous node
5: end while
6: call Flood Migration
```

Algorithm 1: Resource discovery.

```
1: if marking tag exist then
2:    exit
3: end if
4: create marking tag
5: create route-to-user tag point to previous node
6: for each neighbor in the target region do
7:    create child Discovery SM with given predicate
8:    if this SM is the created child Discovery SM then
9:       migrate to the neighbor node
10:      call Flood Migration
11:      exit
12:   end if
13: end for
14: if this node matches the predicate then
15:    migrate back to user node
       (also create route-to-id and route
       to previous node along the way)
16:    add this node in a set of bound node
17: end if
```

Algorithm 2: Flood migration.

marking tag, the Discovery SM also creates a route-to-user tag on the current node for memorizing the previous hop (Line 5). Therefore, this route-to-user tag contains the next hop toward the user node. These route-to-user tags on all nodes form an aggregation tree for gathering data from all matching nodes.

Once a matching node is found, the Discovery SM migrates back along the aggregation tree to notify the main SM (Line 14–17). On the way back to the user node, the Discovery SM creates a route-to-id tag and a route-to-resource tag on the current node for memorizing the previous hop. Route-to-id tags form a path toward a matching node for sequential access whereas route-to-resource tags form a multicast tree for sending access request to matching nodes in parallel.

Upon reaching the user node, the Discovery SM notifies the main SM about the matching node. The main SM then adds the reported node into the set that is bound to the resource variable. Additionally, the main SM resets the binding timer of the resource variable to its lifetime.

In this implementation, an iterator access (i.e., sequential access) does not cause resource discovery. The main SM creates an Access SM that migrates toward the bound node using the corresponding route-to-id tags. Conversely, a resource-variable access causes resource discovery if the binding timer expires or the variable is not bound (see Algorithm 3). Once the variable is bound, the main SM creates Access SMs that migrate toward the bound nodes along the multicast tree in parallel (see Algorithm 4). Upon reaching the bound nodes, the Access SMs perform instructed operations and carry the results back to the user node (Line 1–8).

Accessing resources in parallel enables data aggregation that results in energy savings. Therefore, on each branching node along the aggregation tree, the Access SM waits for other SMs until SMs from all branches arrive or the waiting timer expires (Line 18). All arriving SMs are merged into one and only the resulting SM migrates to the user node (Line 19–23). Our waiting timer in this implementation is fixed and quite naive. A better implementation is to use the depth of the branching node to proportionally set its waiting timer. Undoubtedly, the deeper node requires the longer timer. The depth of the branching node can be computed when Discovery SMs migrate back to the user node. Given that a matching node is a leaf of our aggregation tree, its depth is zero. Each Discovery SM carries this depth counter and increments it by one for each hop that the SM migrates. The Discovery SM will also creates a depth tag on each node along the way for maintaining the current depth of the node if there is no such tag yet. Both the depth tag (on the current node)

```
1: if binding expired or not bound then
2:    call Resource Discovery
3:    restart binding timer
4: end if
5: call Access Migration
```

Algorithm 3: Resource access.

```
1:  if on a bound node then
2:     access the specified object
3:  end if
4:  if on a leaf node and exist parent node then
5:     migrate back to the parent node
6:     notify the parent SM
7:     return
8:  end if
9:  for each child on multicast tree do
10:    create child Access SM
11:    if this SM is the created child Access SM then
12:       remember this node as parent
13:       migrate to the child node
14:       call Access Migration
15:       exit
16:    end if
17: end for
18: wait for results from all child Access SMs
19: merge the results
20: if exist parent node then
21:    migrate back to the parent node
22:    notify the parent SM
23: end if
```

Algorithm 4: Access migration.

and the depth counter (on the SM) are compared and set to the greater value between them.

Furthermore, this implementation can also be improved by merging the Discovery SM and the Access SM into a Discovery&Access SM. This new SM behaves like the Discovery SM but, once it finds a matching node, it immediately accesses the resource on the node and migrates back to notify the main SM about the matching node as well as the access result in one step. This merging can improve energy savings and reduce delay of our system. We plan to implement this merging and depth computing in our future work.

5.3. TinyDRN. TinyDRN is a subset of our abstraction, retaining only resource variables and static binding features. TinyDRN is implemented as new bytecode instructions on Maté, a tiny virtual machine for sensor networks. In the Maté platform, each node is a mote running a Maté virtual machine. Given no user interface on motes, we need a PC as our user node that connects to a gateway mote for relaying our commands. Each sensor node or mote has upto 128 KB ROM for instruction memory and upto 4 KB RAM for data whereas the K Virtual Machine (used in SM) targets devices with a memory budget of at least 160 KB. As a virtual machine, Maté is a bytecode interpreter implemented as a component in Tiny OS (an operating system of motes).

In a sense, a Maté program is simply a script consisting of Maté commands that are recognized, interpreted, and executed by a Maté VM. To implement TinyDRN, we need to modify the Maté VM so that the virtual machine knows how to interpret and to execute our new bytecode instructions.

We have also developed an application using TinyDRN to test our TinyDRN implementation. Our testing application turns on the LED of nodes in the area that is brighter than 400 units. To understand the overhead of our implementation, we have written the same application using the original Maté (without TinyDRN instructions). Based on our measurement, this application with the original Maté takes 42,976 bytes of ROM and 3,134 bytes of RAM. In contrast, this application with the TinyDRN-added Maté takes 44,586 bytes of ROM and 3,289 bytes of RAM. The result indicates that the TinyDRN version takes only 3.75% additional bytes of ROM and 4.95% additional bytes of RAM.

Even with the slightly bigger memory usage, the TinyDRN version surprisingly runs faster and sends fewer messages than the non-TinyDRN one does. This is due to the smaller script size (a benefit of new bytecode instructions).

```
Space sp=UNIVERSE;
Resource R1=< (within(Sp)==TRUE)&(motion>0)  >;
Location AverageLoc;

for(inti=1;i<= 25;i++) {
  AverageLoc = average(R1->Location);
  if (AverageLoc != NULL) {
    System.out.println(
        "Average("+i+")="+AverageLoc);
    sp.updateRegion(AverageLoc, 10);
  } else {
  System.out.println(
        "Average("+i+")= NOT FOUND");
  sp = UNIVERSE;
 }
 sleep(4000);
 }
```

Algorithm 5: Pseudocode for our object-tracking application.

Although the modified virtual machine is bigger, the application itself is smaller to propagate. This results in fewer messages and bytes to send over the network. Consequently, the code is propagated faster and the energy is consumed less. It is a classic example of using energy wisely on computation rather than on communication.

6. Evaluation

In this section, we conduct an experiment to evaluate a DRN application executed over our DRN runtime system. This section describes our methodology and considers the impact of a DRN tuning parameter on the application's performance.

However, only DRN is used in this evaluation because we intend to test the full feature of our abstraction. Our test on TinyDRN can be found in Section 5.3.

6.1. Goals, Metrics, and Methodology. We have implemented our object-tracking application (Section 6.2) using DRN. This application is evaluated on a network of 20 nodes. Each node is emulated using a Smart Message Virtual Machine (SMVM) that runs on a different port of a physical machine. (Given that the SMVM can run directly on an HP iPAQ [14], our DRN code can also run on the iPAQ without any modification.)

Our goals in conducting this evaluation study are twofold. First, it is necessary to verify the viability of the DRN model for macroprogramming WNES. Second, we would also like to understand the impact of resource-binding lifetime on the DRN application.

We choose two metrics to analyze the performance of our DRN application: *the number of application bytes sent* and *average distance error*. The number of application bytes that are sent measures the total bytes sent across the network. The metric roughly indicates the dissipated energy and implies the overall lifetime of WNES. Average distance error measures the distance between the actual object location and the reported location. This metric implies the accuracy of the tracking application; similar metrics were used in earlier work [30]. We study these metrics as a function of the resource binding's lifetime.

In our experiment, we study a multihop sensor field (of 20 nodes) that is generated by randomly placing the nodes in a 20 m by 40 m rectangle. Each node has a radio range of 10 m and a sensing range of 5 m. Such ranges enable a direct communication between two nodes that detect the same object. The transmission range also defines neighbors of each node (SMVM) in this emulation.

The DRN application tracks an object that moves at a rate of 0.25 m/s. The object moves clockwise along the edge of a 10 m by 30 m rectangle located in the middle of the sensor fields. This clockwise movement causes nodes in different regions to detect and track the object. The application estimated the object location on 25 different occasions during our experiment, or once every 4 seconds.

6.2. Object Tracking Application. Algorithm 5 shows the simple DRN pseudocode that accompanies our object tracking application. Essentially, the application tracks an object by acquiring the location of devices (i.e., resources) that detect motion within a region of interest. The average location of such devices is an estimation of the object location. At the beginning, there is no estimation of object location. The application first searches for the object throughout the sensor field. Once an object location is found, the region of interest for the next search is set to an area within 10 m of the estimated location. This approach limits the searching space and results in better energy efficiency, especially when the geographical routing is used in the underlying system. Later, if the object cannot be found in this dynamic circular region, the region of interest is reset to the whole sensor field.

The actual Java code for this application (listing 1) is very similar to the simple DRN pseudocode in Algorithm 5;

```
public class TrackingApp extends SmWrapper {

  private final static int timeout = 24000; // Binding lifetime
  private Space sp;
  private TrackingExpression tExp;
  private Resource resource;
  private LocationAverage agg;

  public TrackingApp(){
    super(“TrackingApp”);
  }

  public void run() {
    try{
      sp = new Space(null, −1); // sp = UNIVERSE
      tExp = new TrackingExpression(sp, ‘‘motion’’);
      resource = new Resource(tExp, timeout);
      agg = new LocationAverage();
       for(inti=1;i<= 25; i++){
        agg = (LocationAverage)resource.access(agg, 4000);
        System.out.println(“agg = ”+agg);
         Location average = (Location)agg.evaluator();
         if (average != null){
          System.out.println(“Average(“+i+”) = ”+average);
          sp.updateRegion(average, 10);
        } else {
          System.out.println(“Average(“+i+”) = NOTFOUND”);
          sp.updateRegion(null, −1); // sp = UNIVERSE
        }
         sleep(4000);
      }
    } catch(Exceptione){}
 }

 public static void main(String[]args){
   TrackingApp trackingApp = new TrackingApp();
   String[]types;
   types = new String[3];
   types[0] = “TrackingApp”;
   types[1] = “TrackingExpression”;
   types[2] = “LocationAverage”;
   trackingApp.initSM(types,trackingApp);
   trackingApp.run();
 }
}
```

Listing 1: Real Java code for our object-tracking application.

it is possible to achieve a one-to-one translation from simple DRN pseudocode to real Java code. In this Java code, our *TrackingApp* simply extends the *SmWrapper* that hides SM-related details from programmers. To conform with the Java syntax, we implement the resource expression (*TrackingExpression*) as a class (listing 2). (Automatic generation of this expression class from DRN pseudocode is part of our future work.) Each expression class contains an *evaluate()* method that needs to be executed on the device to determine if the device property is matched with the expression.

In this application code, resources are accessed in parallel. Parallel access provides an opportunity for in-network processing (e.g., data aggregation) that can significantly reduce the system's overall energy consumption [22–24, 27, 28]. Typically, in other systems, the code for in-network aggregation cannot be dynamically installed after network deployment. In some systems, an API may not be provided for writing a new in-network aggregation code. For example, it is not obvious how a new aggregation algorithm can be expressed in TAG using SQL, given that SQL is not designed for expressing algorithmic details. Furthermore, TAG is not

```
public class TrackingExpression extends Expression {

  private Space sp_;
  private String moTag_;

  public TrackingExpression (Space sp, String moTag) throws BadSMApiUsageException {
      sp_=sp;
      moTag_=moTag;
  }

  public boolean evaluate() {
     try {
         Integer moInt = (Integer)TagSpace.readTag(moTag_);
         GPSData gps = (GPSData)TagSpace.readTag("gps");
         if (!sp_.outside(new Location(gps.latitude, gps.longitude)) && (moInt.intValue() >0) ) {
             return true;
         }
     } catch(Exception e) {}
     return false;
  }
}
```

Listing 2: TrackingExpression class for matching resources.

a reprogrammable platform. Therefore, it is not clear how a new aggregation code can be deployed on the fly. Unlike other WNES programming approaches, DRN provides an *Aggregation* class that can be extended to implement a new dynamically-deployable aggregation technique.

Generally, a data aggregation technique is implemented using three functions: initializer *i()*, merger *m()*, and evaluator *e()*. The initializer *i()* specifies how to instantiate a data state record for a single sensor value. DRN will call this function on devices whose properties are matched with the declarative expression. This data state record will then be sent back toward the user node. During the return trip, this data state may meet other data states from the same set of desired resources. DRN will call the merger *m()* to aggregate these data states into one. Once the data state reaches the user node, the evaluator *e()* will compute the actual value of the aggregate.

In our application, we have shown how to implement a new aggregation technique called *LocationAverage* (listing 3) in DRN. To do this, we simply extend the *Aggregation* class and overload the three mentioned functions. We use $\langle \text{sum\_}x, \text{count\_}x, \text{sum\_}y, \text{count\_}y \rangle$ as our data state. Suppose the matching device is located at $(x1, y1)$. The initializer sets the data state record to $\langle x1, 1, y1, 1 \rangle$. The merger combines the state $\langle x1, cx1, y1, cy1 \rangle$ and the state $\langle x2, cx2, y2, cy2 \rangle$ into a single state $\langle x1 + x2, cx1 + cx2, y1 + y2, cy1 + cy2 \rangle$. The evaluator returns $\langle \text{sum\_}x/\text{count\_}x, \text{sum\_}y/\text{count\_}y \rangle$ as the average location.

6.3. Tuning Knob. Semantically, in our model, resource access is strictly performed on resources that match the declarative expression at the time of access. Changes in the set of matching nodes do not require attention from the programmers. Therefore, DRN must rebind resources transparently and dynamically. This strict semantic could incur significant overhead and excessive energy consumption for ensuring that this reactive binding is up to date. Not surprisingly, we propose tuning knobs for balancing strong semantics with energy savings. One of these tuning knobs is the *resource binding lifetime*. For example, using a binding lifetime of *t*, programmers can slightly lessen the semantic and allow access if the resource is bound in the last *t* seconds.

In this experiment, we study an impact of binding lifetime on energy consumption and tracking accuracy of an unoptimized version of our application. Specifically, Line 25 in listing 1 is removed. Therefore, searches for the object are always performed throughout the sensor field. An additional objective of this experiment is to show that, even though the declarative expression and related variables are not changed, the resource is dynamically and deservedly rebound.

Figure 1(a) plots the number of bytes sent as a function of the resource binding lifetime. As expected, the number of bytes sent is reduced (the line with black rectangles) as we increase the binding lifetime (i.e., reduce the number of resource discovery). Results indicate that it is possible to achieve meaningful energy savings without a significant degradation in tracking accuracy. Specifically, we can achieve a 51.5% savings in bytes sent with only small accuracy degration when we increase the binding lifetime from 4 to 16 seconds. The total number of bytes sent includes the overhead for installing this mobile-agent program on the fly (the line with white rectangles). When we factor out the bytes sent for injecting the application code into the network, the savings improve to 55.2%.

The average tracking error does not significantly increase until the binding lifetime is more than 16 seconds

```
public class LocationAverage extends Aggregate {

  private GPSData gps;
  double sum_x, sum_y;
  int count_x, count_y;

 public void initializer() {
    try {
        gps = (GPSData)TagSpace.readTag("gps");
         sum_x = gps.latitude;
        sum_y = gps.longitude;
        count_x = 1;
        count_y = 1;
     } catch (Exception e) {}
  }

  public void merger(Aggregate agg) {
      sum_x = sum_x+agg.sum_x;
      sum_y = sum_y+agg.sum_y;
      count_x = count_x+agg.count_x;
      count_y = count_y+agg.count_y;
  }

  public Object evaluator() {
       try {
          return new Location(sum_x/count_x, sum_y/count_y);
       } catch (Exception e) {
          return null;
       }
  }
 }
```

Listing 3: LocationAverage class for in-network processing.

(Figure 1(b)). The result is intuitive. If the object moves away from a bound sensor at the speed of 0.25 m/s, it will take at most 20 seconds to move beyond the bound node's sensing range. Conversely, if the object moves toward the bound sensor without changing its direction, it will take at most 40 seconds to pass out of range. Given the moving pattern in this experiment, we do not need to rediscover the resources within 20 seconds to achieve a reasonable accuracy. However, after 20 seconds, the accuracy will be significantly degraded. If we do not rediscover the resources after 40 seconds, we will no longer be able to track the object.

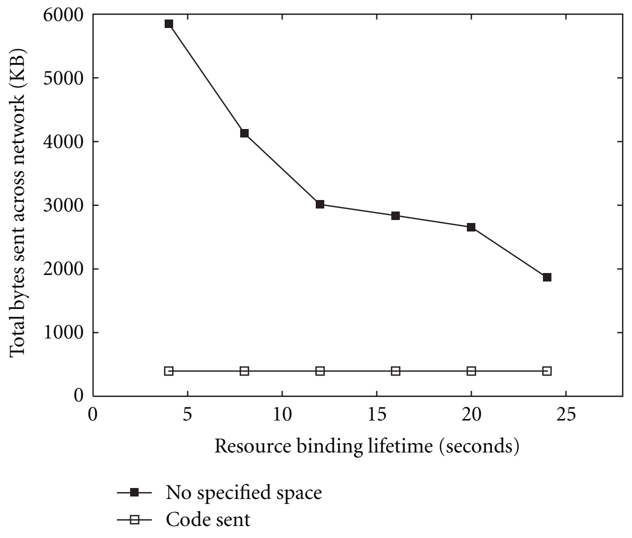
Tracking accuracy depends on several factors: estimation techniques, network density, and sensing range. The estimation error of 2-3 m in this experiment is considered reasonable, given our simple estimation technique, low-density network, and 5 m sensing range.

6.4. Space Scoping for Optimization. Like other programming paradigms, writing an efficient program requires understanding of the underlying system. For example, in virtual memory systems, programs should be written such that the number of page faults is minimized. To operate on an entire two-dimensional array in those memory systems, elements in the array should be accessed row-by-row rather than column-by-column. Similarly, our tracking application is more efficient when the searching space is specified because our run-time library supports geographic routing. Given a specified space, resource-discovery request is geographically routed to the space instead of flooding throughout the network.

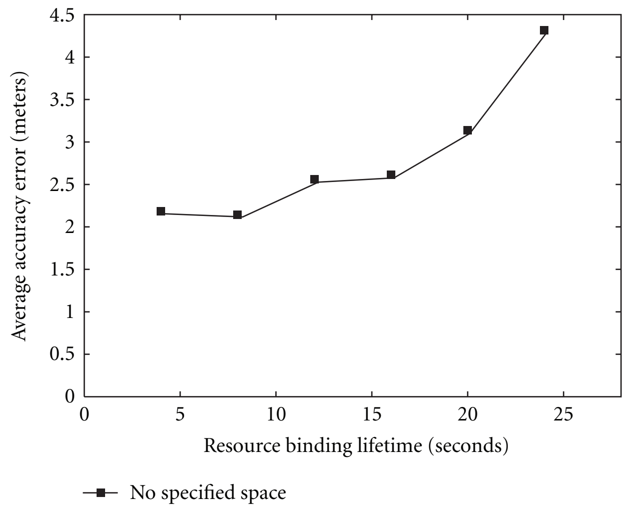
To study the impact of space scoping on our tracking application, we conduct an experiment similar to that of the previous section. The difference is that Line 25 in listing 1 is now included.

As the binding lifetime is increased, the savings is decreased due to the reduced number of resource discovery. Additionally, the tracking accuracy is not significantly degraded by space scoping (Figure 2(b)).

Our results indicate that we can achieve 42.5% savings on the number of bytes sent when we dynamically specify the target space (Figure 2(a)). Although this savings is significant, one may expect more savings because geographic routing is much more efficient than flooding. However, once the resource-discovery request is geographically routed to the specified space, the request is flooded within the space in order to discover all matching nodes. This scoped flooding incurs additional overhead and results in fewer-than-expected savings.

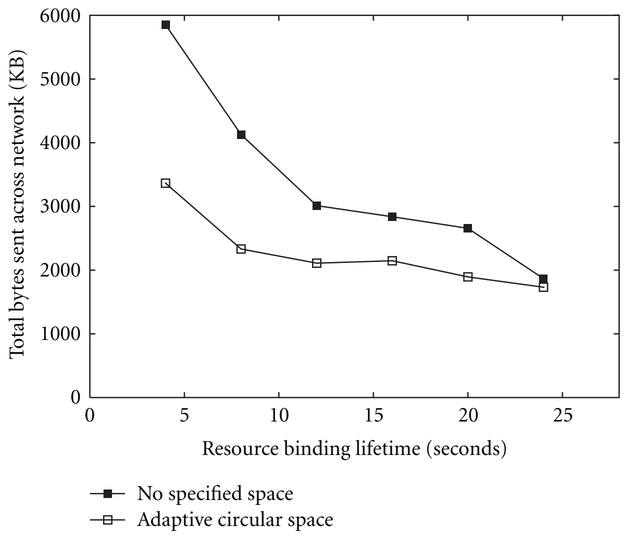


(a) Application bytes sent.

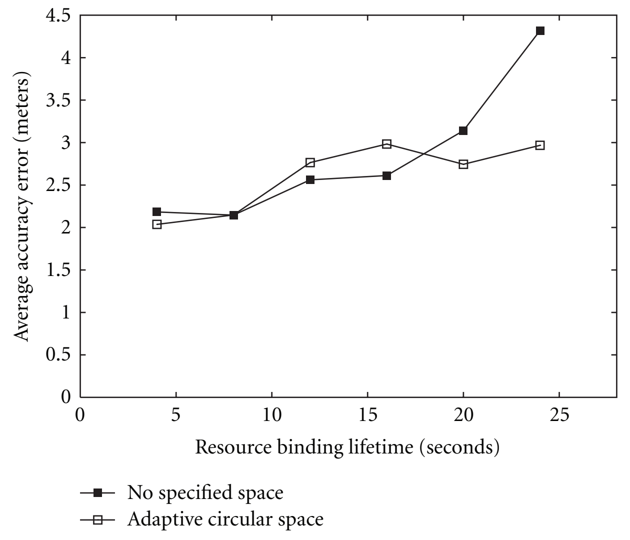


(b) Average distance error.

Figure 1: Impact of resource binding lifetime on our object-tracking application.



(a) Application bytes sent.



(b) Average distance error.

Figure 2: Impact of space scoping on our object-tracking application.

7. Discussion

In Section 6, our tracking example is intentionally simple. Undoubtedly, one can easily write a more sophisticated tracking application using DRN. One possible improvement is to estimate the object speed using exponential weighted average. This estimated speed can be used to predict the object location or the center of the space for the next access.

To simplify our evaluation, we have used fixed sensing and radio ranges. In practices, these ranges are not fixed. With realistic ranges, we can obtain more realistic accuracy results. However, these ranges have no impact on the viability of our abstraction.

While DRN works well for tracking, some may wonder if it applies well to other applications, especially data-collection applications. It is necessary to emphasize that DRN is not only useful for retrieving data but also for pushing data to declaratively named nodes. We can install a new function to all nodes in the network. This new function then reads and pushes data periodically or reactively toward the user node. Therefore, data-collection applications should be easily implemented by using DRN in this manner.

In this paper, we propose only two tuning knobs: binding lifetime and access timeout. There are still several other tuning knobs that should be exposed to programmers. For example, one might want to dynamically change the number of nodes that participate in aggregation based on energy goals. Such a tuning knob would be useful for limiting energy consumption during resource access. We plan to explore other tuning knobs in our future work.

Furthermore, one may wonder what the right binding lifetime is for each application. As long as the result is still acceptable, higher binding lifetime is generally better.

A possible solution without trial-and-error is to dynamically adjust the binding lifetime based on the quality of the result and the resource consumption.

8. Conclusions

We believe that, to efficiently develop WNES applications, appropriate programming abstractions are necessary. DRN is one such abstraction that integrates declarative constraints with imperative constructs to form a powerful programming paradigm suitable for macroprogramming WNES.

We have implemented DRN on two platforms: SM and Maté. SM can run on iPAQs with 802.11 radios whereas Maté can run on motes with 802.15.4 radios. Furthermore, given network transparency, our approach should be applicable for macroprogramming over wired or wireless networks as well. However, this network transparency feature of DRN implies that DRN is not for low-level programming or implementing a protocol that requires a distinct notion between being remote and local.

In addition, we have implemented an object-tracking application using our DRN runtime library to show the model viability. We have also evaluated our DRN runtime library and its tuning knob (i.e., resource binding lifetime). Our tuning knob enables the DRN application to save up to 55.2% of bytes sent without significant accuracy degradation when the application code is already cached or installed in the network.

In the future, we intend to further explore the design space of DRN such as other tuning knobs. Additionally, we plan to implement other applications using DRN and to conduct more extensive evaluation in order to better realize DRN's full potential.

Acknowledgments

This research has been supported by the Thailand Research Fund (TRF) under Grant no.MRG5080449. The authors would like to thank Napat Ratanasirintrawoot and Phuri Chalermkiatsakul on their work on TinyDRN and also thank Prof. Rajesh Gupta, Amin Vahdat, and anonymous reviewers for valuable suggestions to improve this work.

References

[1] 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*, vol. 30, no. 1, pp. 122–173, 2005.

[2] P. Bonnet, J. Gehrke, T. Mayr, and P. Seshadri, "Query processing in a device database system," Tech. Rep. TR99-1775, Cornell University, 1999.

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

[4] S. Choochaisri, N. Pornprasitsakul, and C. Intanagonwiwat, "Logic macroprogramming for wireless sensor networks," *International Journal of Distributed Sensor Networks*, vol. 2012, Article ID 171738, 12 pages, 2012.

[5] S. Choochaisri and C. Intanagonwiwat, "A system for using wireless sensor networks as globally deductive databases," in *Proceedings of the 4th IEEE International Conference on Wireless and Mobile Computing, Networking and Communication (WiMob '08)*, pp. 649–654, IEEE Computer Society, Washington, DC, USA, October 2008.

[6] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using Kairos," in *Proceedings of the 1st IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '05)*, pp. 126–140, July 2005.

[7] The Castle project, http://www.cs.berkeley.edu/projects/parallel/castle/split-c/.

[8] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode, "Spatial programming using smart messages: design and implementation," in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS '04)*, pp. 690–699, Tokyo, Japan, March 2004.

[9] L. Iftode, C. Borcea, A. Kochut, C. Intanagonwiwat, and U. Kremer, "Programming computers embedded in the physical world," in *Proceedings of the 9th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS '03)*, San Juan, Puerto Rico, May 2003.

[10] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proceedings of the 6th International Symposium on Information Processing in Sensor Networks (IPSN '07)*, pp. 489–498, ACM, New York, NY, USA, April 2007.

[11] T. W. Hnat, T. I. Sookoor, P. Hooimeijer, W. Weimer, and K. Whitehouse, "Macrolab: a vector-based macroprogramming framework for cyber-physical systems," in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*, pp. 225–238, New York, NY, USA, 2008.

[12] Y. H. Tu, Y. C. Li, T. C. Chien, and P. H. Chou, "EcoCast: interactive, object-oriented macroprogramming for networks of ultra-compact wireless sensor nodes," in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '11)*, pp. 366–377, Chicago, Ill, USA, April 2011.

[13] C. Intanagonwiwat, R. K. Gupta, and A. Vahdat, "Declarative resource naming for macroprogramming wireless networks of embedded systems," in *ALGOSENSORS*, S. E. Nikoletseas and J. D. P. Rolim, Eds., vol. 4240 of *Lecture Notes in Computer Science*, pp. 192–199, Springer, 2006.

[14] C. Borcea, D. Iyer, P. Kang, A. Saxena, and L. Iftode, "Cooperative computing for distributed embedded systems," in *Proceedings of the 22nd International Conference on Distributed Systems (ICDCS '02)*, pp. 227–236, July 2002.

[15] A. Boulis, C. Han, and M. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," in *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (Mobisys '03)*, pp. 187–200, San Francisco, Calif, USA, May 2003.

[16] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (APLOS '02)*, pp. 85–95, October 2002.

[17] K. Whitehouse, F. Zhao, and J. Liu, "Semantic Streams: a framework for composable semantic interpretation of sensor data," *Wireless Sensor Networks*, vol. 3868, pp. 5–20, 2006.

[18] S. Choochaisri and C. Intanagonwiwat, "An Analysis of Deductive-Query Processing Approaches for Logic Macroprograms in Wireless Sensor Networks," *Engineering Journal*, vol. 16, no. 4, pp. 47–62, 2012.

[19] Oracle Corporation, *Pro∗c/c++ Precompiler Programmer's Guide Release 9.2*, 2002.

[20] M. Grabmuller, *Constraint imperative programming [Diploma thesis]*, Technische Universitat Berlin, 2003.

[21] S. Chen, P. B. Gibbons, and S. Nath, "Database-centric programming for wide-area sensor systems," in *Proceedings of the 1st IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '05)*, pp. 89–108, July 2005.

[22] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan, "Building efficient wireless sensor networks with low-level naming," in *Proceedings of the ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[23] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MOBICOM '00)*, pp. 56–67, Boston, Mass, USA, August 2000.

[24] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, "Directed diffusion for wireless sensor networking," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 2–16, 2003.

[25] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-efficient communication protocol for wireless microsensor networks," in *Proceedings of the 33rd Annual Hawaii International Conference on System Siences*, p. 223, Maui, Hawaii, January 2000.

[26] D. Coffin, D. Van Hook, R. Govindan, J. Heidemann, and F. Silva, "Network routing application programmer's interface (api) and walk through 8.0," Tech. Rep. 01-741, USC/ISI, 2001.

[27] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann, "Impact of network density on data aggregation in wireless sensor networks," in *Proceedings of the 22nd International Conference on Distributed Systems*, pp. 457–458, IEEE, Vienna, Austria, July 2002.

[28] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[29] C. Borcea, C. Intanagonwiwat, A. Saxena, and L. Iftode, "Self-routing in pervasive computing environments using smart messages," in *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom '03)*, pp. 87–96, March 2003.

[30] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.