Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems

Chalermek Intanagonwiwat[1], Rajesh Gupta[2], and Amin Vahdat[2]

[1] *Department of Computer Engineering, Chulalongkorn University, Thailand*
[2] *Department of Computer Science and Engineering, University of California at San Diego, USA*

intanago@cp.eng.chula.ac.th, {rgupta, vahdat}@cs.ucsd.edu

Abstract

Programming Wireless Networks of Embedded Systems (WNES) is notoriously difficult and tedious. To simplify WNES programming, we propose Declarative Resource Naming *(DRN) to program WNES as a whole (*i.e., macroprogramming*) instead of several networked entities. DRN allows for a set of resources to be declaratively described by their run-time properties, and for this set to be mapped to a variable. Using DRN, resource access is simplified to only variable access that is completely network-transparent. DRN provides both sequential and parallel accesses to the desired set. Parallel, or group, access reduces the total access time and energy consumption because it enables in-network processing. Additionally, we can associate each set with tuning parameters (*e.g., *timeout, energy budget) to bound access time or to tune resource consumption.*

1 Introduction

WNES consists of a massive number of resource-constraint wireless nodes that are deployed in dynamic, hostile environments. Unlike traditional networks, WNES is property-centric. As such, nodes of interest in WNES are defined by their properties at run-time rather than by their node ids. These characteristics pose two major research challenges to the design of WNES programming.

1. How do we easily and efficiently write the WNES applications?
2. How do we reprogram the network of unattended nodes after deployment?

In this paper, we will focus on the first question of characterizing WNES applications. To simplify WNES programming, we propose *Declarative Resource Naming* (DRN) [1] to program WNES as a whole (*i.e., macroprogramming*) instead of several networked entities. DRN allows programmers to declaratively describe a set of desired resources by their run-time properties and to map this set to a variable. To access the desired resources, 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. Systems based on code migration are preferable because programs can be propagated to target nodes without human intervention. Examples of such systems include Smart Messages (SM) [3], SensorWare [4], Mate [18], and TML [20]. Even though reprogramming the network is not our focus in this paper, we have completed our preliminary implementation of a DRN run-time library using Smart Messages (SM) that runs on iPAQs equipped with 802.11 radios. SM is an appropriate choice because it supports program migration, a necessary capability for reprogramming the network. Undoubtedly, there are other reprogrammable platforms such as SensorWare, Mate, and TML. However, we select SM because the library can be implemented in Java, a well known language. Nevertheless, given network transparency, our abstraction is independent of the underlying platforms. As a result, it is possible to macroprogram other wired or wireless networks using our approach.

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)

[1] An initial design of this work appears in a UCSD technical report [15]

on a network of 20 virtual machines. When the application code is already cached or installed in the network, our tuning knob enables the DRN application to save up to 55.2% of the bytes sent without significant accuracy degradation.

2 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 approach include COUGAR [1] and TAG [19]. 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.

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 still 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 detect 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.

3 Declarative Resource Naming

To simplify the programming tasks for WNES, we propose a scheme that will program the WNES as a unit. Particularly, we consider WNES a single abstract machine. Although physically scattered, all resources are on the same machine in our model. Given this single machine model, there is no notion of networking, being remote, or local.

3.1 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 resources). 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, 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 $R->light$. Similarly, the camera can be turned off by assigning off to $X->camera$. There is no need for algorithmic detail of resource controls and operations.

3.2 Declarative Constraint

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

```
Resource R = <location == within(forest) &&
                         temperature > 30>
Resource X = <a(b,c)!=0>
```

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.

3.3 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.

Given that more than one resource can match a specified expression, a resource variable is referred to as a set of matched resources rather than a single one. Therefore, mechanisms for accessing each element in the set are required. We propose two approaches for accessing multiple matched resources: *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;
Iterator i;

foreach i in R {
printf("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.

```
Resource R;
printf("light intensity=%f", R->light);
```

  Therefore, 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 [10, 11, 13, 17, 14, 19]. 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.

3.4 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 matched resources 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 matched resources do not require attention from programmers. As a result, to conform with this strict semantic, the underlying system may need to ex-

pend 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 matched resources 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 matched resources. We propose two explicit mechanisms: the *static* resource and the *iterator*.

  Using the static resource, we can specify which resources are statically bound. The static resource 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 static resource is intended for memorizing the entire set of matched resources. 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;
```

3.5 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. [2] 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("can't access the light sensor");
}
```

[2] This problem is similar to that of TCP. Packet loss and unbound acknowledgment delay are handled using timeout.

4 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.

4.1 Goals, Metrics, and Methodology

We have implemented our object-tracking application (Section 4.2) 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 [3], our DRN code can also run on the iPAQ without any modification.)

Our goals in conducting this evaluation study are twofold: First, verify the viability of the DRN model for macroprogramming WNES. Second, 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 [23]. We study these metrics as a function of the resource binding's lifetime.

In our experiment, we study a sensor field (of 20 nodes) that is generated by randomly placing the nodes in a 20m by 40m rectangle. Each node has a radio range of 10m and a sensing range of 5m. Such ranges enable a direct communication between two nodes that detect the same object. The DRN application tracks an object that moves at a rate of 0.25m/s. The object moves clockwise along the edge of a 10m by 30m 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.

Furthermore, our experiment uses an idealistic MAC layer whereby there is no packet collision or loss. Given that we measure the application bytes sent that do not include overhead from the lower layer, this MAC layer is reasonable for our experiment. In a sense, our methodology removes the impact of MAC layers from this study.

```
1:  Space sp=UNIVERSE;
2:  Resource R1=< (within(Sp)==TRUE) & (motion>0) >;
3:  Location AverageLoc;
4:
5:  for (int i=1; i<= 25; i++) {
6:    AverageLoc = average(R1->Location);
7:    if (AverageLoc != NULL) {
8:     System.out.println("Average("+i+")="+AverageLoc);
9:     sp.updateRegion(AverageLoc, 10);
10:   } else {
11:   System.out.println("Average("+i+")=NOT FOUND");
12:   sp = UNIVERSE;
13: }
14: sleep(4000);
15: }
```

Figure 1. Pseudo-code for our object-tracking application.

4.2 Object Tracking Application

Figure 1 shows the simple DRN pseudo-code 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 begining, 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 10m 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.

We have completed preliminary implementation of a DRN run-time library using Smart Messages (SM) that can run on iPAQs communicating with 802.11 radios. SM is appropriate because it supports program migration that is necessary for reprogramming the network. Undoubtedly, there are other reprogrammable platforms such as SensorWare, Mate, and TML. However, we select SM because the library can be implemented in Java, which is a well known language. Nevertheless, given network transparency, our abstraction is independent of the underlying platforms. As a result, it is possible to macroprogram other wired or wireless networks using our approach.

The actual Java code for this application (Figure 2) is very similar to the simple DRN pseudo-code in Figure 1; it is possible to achieve a one-to-one translation from simple DRN pseudo-code 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 (*TrackingEx-*

```
1:  public class TrackingApp extends SmWrapper{
2:
3:   private final static int timeout = 24000;  // Binding lifetime
4:   private Space sp;
5:   private TrackingExpression tExp;
6:   private Resource resource;
7:   private LocationAverage agg;
8:
9:   public TrackingApp(){
10:    super("TrackingApp");
11:  }
12:
13:  public void run() {
14:    try {
15:      sp = new Space(null,-1); // sp = UNIVERSE
16:      tExp = new TrackingExpression(sp, "motion");
17:      resource = new Resource(tExp, timeout);
18:      agg = new LocationAverage();
19:      for (int i=1; i<= 25; i++) {
20:        agg = (LocationAverage)resource.access(agg, 4000);
21:        System.out.println("agg = "+agg);
22:        Location average = (Location)agg.evaluator();
23:        if (average != null) {
24:          System.out.println("Average("+i+") = "+average);
25:          sp.updateRegion(average, 10);
26:        } else {
27:          System.out.println("Average("+i+") = NOT FOUND");
28:          sp.updateRegion(null, -1); // sp = UNIVERSE
29:        }
30:        sleep(4000);
31:      }
32:    } catch(Exception e) {}
33:  }
34:
35:  public static void main(String []args) {
36:    TrackingApp trackingApp = new TrackingApp();
37:    String []types;
38:    types = new String[3];
39:    types[0] = "TrackingApp";
40:    types[1] = "TrackingExpression";
41:    types[2] = "LocationAverage";
42:    trackingApp.initSM(types, trackingApp);
43:    trackingApp.run();
44:  }
45: }
```

Figure 2. Real Java code for our object-tracking application.

pression) as a class (Figure 3). (Automatic generation of this expression class from DRN pseudo-code 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 [10, 11, 13, 14, 17, 19]. Typically, in other systems, the code for in-network aggregation cannot be dynamically installed after network deployment [19]. In some systems, an API may not be provided for writing a new in-network aggregation code. 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* (Figure 4) in DRN. To do this, we simply extend the *Aggregation* class and overload the three-mentioned functions. We use $< sum\_x, count\_x, sum\_y, count\_y >$ as our data state. Suppose the matched device is located at $(x1, y1)$. The initializer sets the data state record to $< x1, 1, y1, 1 >$. The merger combines the state $< x1, cx1, y1, cy1 >$ and

```
1:  public class TrackingExpression extends Expression{
2:
3:    private Space sp_;
4:    private String moTag_;
5:
6:    public TrackingExpression(Space sp, String moTag) throws BadSMApiUsageException {
7:        sp_=sp;
8:        moTag_=moTag;
9:    }
10:
11:   public boolean evaluate() {
12:       try{
13:           Integer moInt = (Integer)TagSpace.readTag(moTag_);
14:           GPSData gps = (GPSData)TagSpace.readTag("gps");
15:           if (!sp_.outside(new Location(gps.latitude, gps.longitude)) && (moInt.intValue()>0) ) {
16:                return true;
17:            }
18:       }catch(Exception e) {}
19:       return false;
20:   }
21: }
```

Figure 3. TrackingExpression class for matching resources.

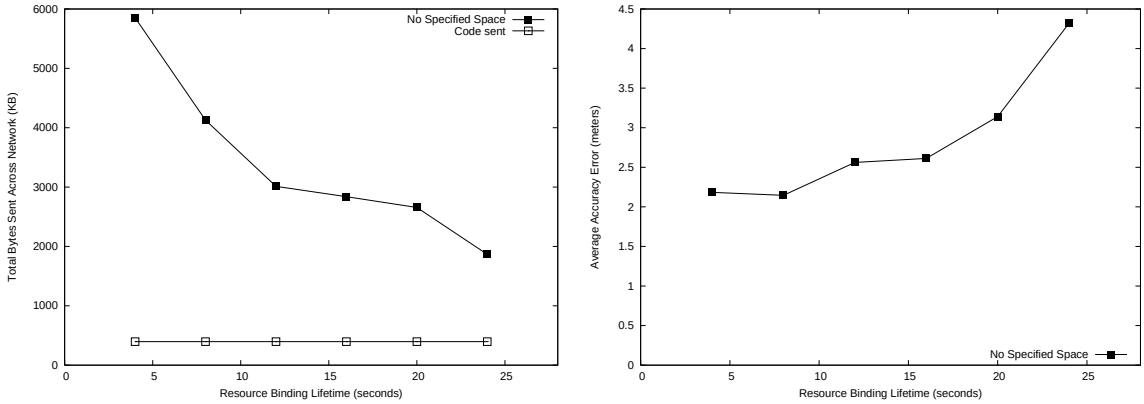
```
1:  public class LocationAverage extends Aggregate{
2:
3:    private GPSData gps;
4:    double sum_x, sum_y;
5:    int count_x, count_y;
6:
7:    public void initializer() {
8:        try {
9:            gps = (GPSData)TagSpace.readTag("gps");
10:           sum_x = gps.latitude;
11:           sum_y = gps.longitude;
12:           count_x = 1;
13:           count_y = 1;
14:       } catch (Exception e) {}
15:   }
16:
17:   public void merger(Aggregate agg) {
18:        sum_x = sum_x+agg.sum_x;
19:        sum_y = sum_y+agg.sum_y;
20:        count_x = count_x+agg.count_x;
21:        count_y = count_y+agg.count_y;
22:   }
23:
24:   public Object evaluator() {
25:        try {
26:            return new Location(sum_x/count_x, sum_y/count_y);
27:        } catch (Exception e) {
28:            return null;
29:        }
30:   }
31: }
```

Figure 4. LocationAverage class for in-network processing.

the state $< x2, cx2, y2, cy2 >$ into a single state $< x1 + x2, cx1 + cx2, y1 + y2, cy1 + cy2 >$. The evaluator returns $< sum\_x/count\_x, sum\_y/count\_y >$ as the average location.

4.3 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 matched resources 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



(a) Application bytes sent.

(b) Average distance error.

Figure 5. Impact of resource binding lifetime on our object-tracking application.

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 Figure 2 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 5(a) plots the number of bytes sent as a function of the resource binding lifetime. As expected, the number of bytes sent is reduced 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. This savings is even more significant when the application code is already cached or installed in the network. When we factor out the bytes sent for injecting the application code into the network, the savings improves to 55.2%.

The average tracking error does not significantly increase until the binding lifetime is more than 16 seconds (Figure 5(b)). The result is intuitive. If the object moves away from a bound sensor at the speed of 0.25m/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 2-3m in this experiment is considered reasonable, given our simple estimation technique, low-density network, and 5m sensing range.

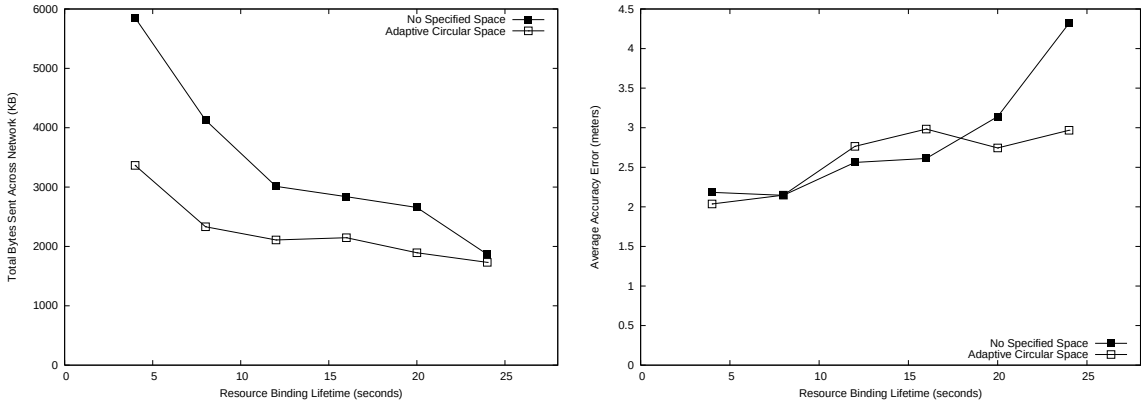
4.4 Space Scoping for Optimization

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 Figure 2 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 6(b)).

Our results indicate that we can achieve 42.5% savings



(a) Application bytes sent.

(b) Average distance error.

Figure 6. Impact of space scoping on our object-tracking application.

on the number of bytes sent when we dynamically specify the target space (Figure 6(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 matched resources. This scoped flooding incurs additional overhead and results in fewer-than-expected savings.

4.5 Impact of Space Radius

Intuitively, to minimize overhead, the specified space should be as small as possible. In the previous experiment, our space radius is 10m. The space seems unnecessarily large, given a sensing range of 5m. Understandably, one may expect that a 5m-radius space is sufficient for covering all sensors that will detect the object. However, such an expectation is rather optimistic because the object constantly moves and the estimated object location (used as the center of the space for the next search) is generally imperfect.

To study the impact of space radius on our object-tracking application, we conduct an experiment similar to the previous experiment. The difference is that we now fix the binding lifetime to 4 seconds and study the performance of this application as a function of space radius.

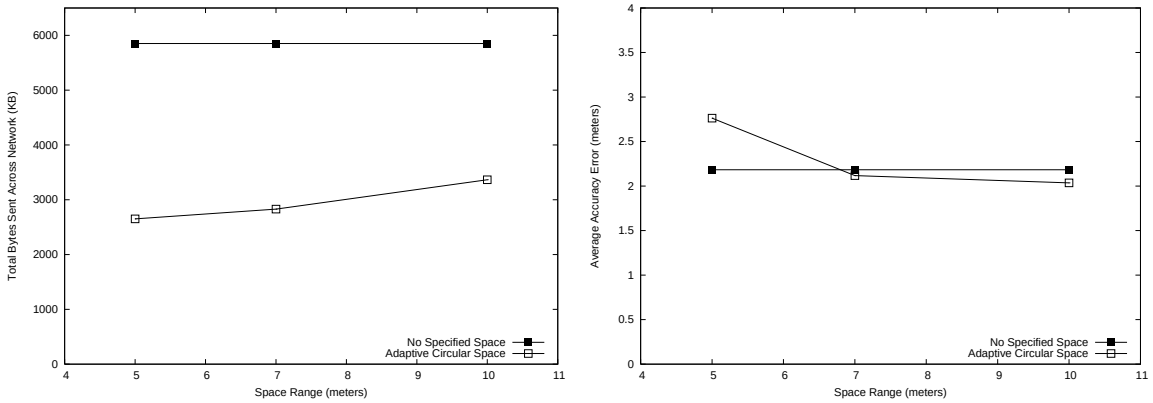
As expected, when the space radius is increased, the number of bytes sent is also increased (Figure 7(a) whereas the tracking accuray is improved (Figure 7(b)). Therefore, there exists a tradeoff between the tracking accuracy and the number of bytes sent. The result indicates that, using a space radius of 7m, we can maintain the tracking accuracy similar to searching the entire network and achieve 51.6% savings on the number of bytes sent. The use of a 7m radius (2m more than our sensing range) is intuitive, given that our accuracy error is also around 2-3m.

5 Related Work

WNES programming has begun to receive attention during the last few years. However, our work has been informed and influenced by a variety of other research efforts, which we now describe.

Our work is mostly influenced by Spatial Programming (SP) [2, 12] and Spatial View [16]. DRN, Spatial View, and SP share a vision of programming WNES as a unit, simplifying 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. Accessing resources in parallel significantly reduces the total access time and the overall energy consumption (by enabling in-network processing). Additionally, SP is purely imperative programming, but DRN is a hybrid between declarative and imperative programming. Unlike DRN binding, SP binding is, by default, static. Even though dynamic binding in SP is provided as an option, rebinding must be explicitly instructed by programmers. As opposed to SP, DRN binding is by default dynamic, whereas static binding is an option. The emphasis on dynamic bindings of DRN is also similar to that of Spatial View. Like SP, Spatial View does not provide parallel accesses and declarative abstraction.

Given that variables can be considered memory resources, mapping other resources into variables of DRN is similar to memory-mapped files. However, DRN must



(a) Application bytes sent.

(b) Average distance error.

Figure 7. Impact of space radius on our object-tracking application.

handle dynamic mappings and frequent failures; memory-mapped files do not.

The abstract region [22, 23] work focuses on a wider definition of space. Specifically, space in the abstract regions can be physical or logical. For example, the logical space can be defined by the number of hops in communication. This example indicates that, unlike our work, the abstract region does not intend to hide the networking details from programmers. In addition, the space is simply an applicable attribute (albeit a very useful one) for our declarative description of resources. Therefore, the space is hardly considered the focus of our work.

Nevertheless, our work has been influenced by directed diffusion [10, 14] and LEACH [11], and 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 [6] require declarative data description for publication and subscription, DRN and diffusion are examples of hybrid programming. Furthermore, this data-centric paradigm of diffusion effectively hides significant networking details, which is one of several DRN features. However, unlike diffusion, DRN focuses on resource naming rather than data naming.

Programming WNES as a unit has also been explored earlier by several research efforts, including TAG [19] and COUGAR [1]. While the above efforts propose programming WNES as a database, we propose programming WNES as a single abstract machine.

Other macroprogramming research efforts include Kairos [9]. Similar to Split-C [21] 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.

There exist several research efforts on a hybrid of declarative and imperative programming. Examples of such research include embedded SQL [7] and constraint-imperative programming [8]. 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.

X-Tree [5] is a recent work on hybrid programming that targets macroprogramming wide-area sensor systems. Similar to TAG, X-Tree programs the whole system as a single database. Unlike TAG and X-Tree, DRN programs the whole system as a single machine. Furthermore, X-Tree is designed for programming devices within the Internet, but DRN is designed for programming hostile dynamic WNES.

6 Conclusions and Future Work

We believe that, to efficiently develop WNES applications, appropriate programming abstractions are neccessary. DRN is one such abstraction that integrates declarative constraints with imperative constructs to form a powerful programming paradigm suitable for macroprogramming WNES. We have completed our preliminary implementation of a DRN run-time library using Smart Messages (SM) that can run on iPAQs communicating with 802.11 radios. SM is appropriate because SM supports program migration, which is necessary for reprogramming the network. Undoubtedly, there are other reprogrammable platforms such as SensorWare, Mate, and TML. However, we select SM mainly because the library can be implemented in a well known language (*i.e.,* Java). Nevertheless, given network transparency, our abstraction is independent of the underlying platforms. Therefore, our approach is applicable for macroprogramming other wired or wireless networks as well.

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) on a network of 20 virtual machines. 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.

References

[1] Philippe Bonnet, Johannes Gehrke, Tobias Mayr, and Praveen Seshadri. Query processing in a device database system. Technical Report TR99-1775, Cornell University, October 1999.

[2] Cristian Borcea, Chalermek Intanagonwiwat, Porlin Kang, Ulrich Kremer, and Liviu Iftode. Spatial programming using smart messages: Design and implementation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, March 2004.

[3] Cristian Borcea, Deepa Iyer, Porlin Kang, Akhilesh Saxena, and Liviu Iftode. Cooperative Computing for Distributed Embedded Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 227–236, July 2002.

[4] A. Boulis, C.Han, and M. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (Mobisys 2003)*, pages 187–200, San Francisco, CA, May 2003.

[5] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Database-centric programming for wide-area sensor systems. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, June 2005.

[6] Dan Coffin, Dan Van Hook, Ramesh Govindan, John Heidemann, and Fabio Silva. Network routing application programmer's interface (api) and walk through 8.0. Technical Report 01-741, USC/ISI, March 2001.

[7] Oracle Corporation. Pro*c/c++ precompiler programmer's guide release 9.2, 2002.

[8] Martin Grabmuller. *Constraint Imperative Programming*. Diploma Thesis, Technische Universitat Berlin, 2003.

[9] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairos. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, June 2005.

[10] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[11] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the Hawaii International Conference on System Sciences*, Maui, Hawaii, January 2000.

[12] Liviu Iftode, Cristian Borcea, Andrzej Kochut, Chalermek Intanagonwiwat, and Ulrich Kremer. Programming computers embedded in the physical world. In *Proceedings of the 9th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS)*, San Juan, Puerto Rico, May 2003.

[13] Chalermek Intanagonwiwat, Deborah Estrin, Ramesh Govindan, and John Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *Proceedings of the International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002. IEEE.

[14] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom'2000)*, Boston, Massachusetts, August 2000.

[15] Chalermek Intanagonwiwat, Rajesh Gupta, and Amin

Vahdat. Declarative resource naming for macroprogramming wireless networks of embedded systems. Technical Report CS2004-0800, University of California at San Diego, November 2004.

[16] Ulrich Kremer, Liviu Iftode, Jerry Hom, and Yang Ni. Spatial Views: Iterative Spatial Programming for Networks of Embedded Systems. Technical Report DCS-TR-493, Rutgers University, June 2002.

[17] Bhaskar Krishnamachari, Deborah Estrin, and Stephen B. Wicker. The impact of data aggregation in wireless sensor networks. In *DEBS'02*, pages 575–578, Vienna, Austria, July 2002.

[18] P. Levis and D. Culler. A tiny virtual machine for sensor networks. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (APLOS)*, October 2002.

[19] Samuel Madden, Michael Franklin, Joseph Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[20] Ryan Newton, Arvind, and Matt Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05)*, April 2005.

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

[22] Matt Welsh. Exposing resource tradeoffs in region-based communication abstractions for sensor networks. In *Proceedings of the 2nd ACM Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[23] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, March 2004.