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 programmers to declaratively describe a set of desired resources by their run-time properties and to map this set to a variable. Using DRN, resource accesses are simplified to completely network-transparent accesses of variables. DRN provides both individual and group accesses to the desired set. Group accesses (*i.e., *parallel accesses) reduce total access time and energy consumption because of possible 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 which are likely deployed in hostile dynamic environments. Unlike traditional networks, WNES is property-centric as nodes of interest in WNES are defined by node properties at run-time rather than by node ids. These characteristics pose two major research challenges to the design of WNES programming.

1. How to reprogram the network after deployment?
2. How to efficiently and easily describe the WNES applications?

It is possible that we may need to reprogram the WNES after deployment because we may find a better algorithm to perform the task or simply because we want to assign other tasks to the network. Given that WNES may be deployed in hostile dynamic environments, we may not be able to physically reach the nodes. Therefore, it is necessary that we can remotely program these unattended nodes. In addition, given the massive number of the unattended nodes, it is practically impossible to manually configure or reprogram each node individually for every application. 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 [3], SensorWare [4], and Mate [15]. However, reprogramming the netwok is not our focus in this paper.

This paper focuses on how to efficiently and easily describe the WNES applications. 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 programmers to declaratively describe a set of desired resources by their run-time properties and to map this set to a variable. Using DRN, resource accesses are simplified to completely network-transparent accesses of variables. DRN provides both individual and group accesses to the desired set. Group accesses (*i.e.,* parallel accesses) reduce total access time and energy consumption because of possible 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.

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 somehow fill in the algorithms for programmers. Automatic generation of algorithmic details can be efficient for simple and specific tasks (*e.g.,* database) but questionable for others. Examples of such an approach include TAG [16] and COUGAR [1] Despite its simplicity,

declarative programming is not a panacea for every WNES application. Imperative programming is more appropriate for complex tasks whereby efficient algorithmic details are not obvious (or not simple to automatically generate).

Declarative and imperative programming work well in 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 parts which are unrelated to the core algorithms, common to applications, and tedious for programmers. To identify the abstractable parts, basic understandings of WNES programs are required. Typically, programs are collections of operations on variables and resources. Given that variables are more frequently accessed, programming languages provide much more simple way to access variables than that to access resources.

Unsurprisingly, traditional resource accesses are more tedious, especially in networked systems whereby there exists a distinction between local and remote resources. Resources are normally bound to nodes which are known priori. Therefore, node ids are traditionally required parts for specifying the remote resources of interest. 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 tedious because the resources of interest are specified by their property at runtime rather than node ids. For example, we may want to access sensors at the hill where temperature is more than 30. Resource discovery in WNES becomes necessary and common rather than optional. The resource property is highly dynamic because the environment is hostile and volatile. Temperature can drop below 30 at any moment. Some resource bindings (*i.e.*, mappings) may have to be invalidated because the bound resources may not match the desired property any more. Even if the resource property may not change, bound resources may not be accessible because of network dynamics (*e.g.*, node mobility). WNES programs are required to detect changes, to invalidate bindings, to discover equivalent resources, and to bind the new 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 to program the WNES as a unit. Particularly, we consider WNES a single abstract machine. All resources are on the same machine in our model, even though they are physically scattered around. Given this single machine model, there is no notion of networking, being remote, or being local.

3.1 Resource Variable

WNES programming can be simplified by making a resource access appear as simple as a variable access. We propose *resource variables* (*i.e.*, variables which 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 programmers can turn the camera off by assigning off to $X->camera$. Programmers do not have to provide any algorithmic detail of resource controls and operations.

3.2 Declarative Constraint

However, one may wonder to which exactly physical nodes (or resources) these variables (R and X) are bound. Rather than specify the node ids for binding, the programmers can declaratively specify the desired property of the target resources using a boolean expression. For example, we can specify that R will be bound to nodes within the forest with temperature greater than 30. 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.

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

3.3 Resource Access

Given that more than one resource can match a given expression, a resource variable is semantically referred to a set of matched resources rather than a single one. Therefore,

mechanisms for accessing each element (resource) in the set are required. We propose two approaches for accessing multiple matched resources: *individual* and *group*.

- **Individual Access.** Each element in a set can be referred using an iterator (similar to an iterator in C++ standard template library). The iterator enables sequential and selective accesses 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.

- **Group Access.** Conversely, in 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 does not only reduce the total access time but also provide 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) which can significantly reduce the overall energy consumption of the system [12, 8, 9, 14, 11, 16]. 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 spends energy only on delivering that max element, not on the others. This ideal delivery can be practically approximated by in-network suppressing the elements whose value are less than that of the previously seen elements of the same access. The mentioned 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, programmers do not have to write the code for maintaining the binding between the physical resources and the resource variables. Given that the resource property is constantly changing, the rebinding of the set of matched resources are frequent and tedious. 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 */
```

  Programmers simply provide the declarative expression which are associated with the resource variable to describe the resources of interest. In general, a reference to a resource variable implies a resource access. Our strong semantic of a resource access strictly enforces that the access is always performed only on the resource which matches the declarative expression at the time of access. Changes in the set of matched resources do not need attentions from programmers. However, this strict semantic could incur significant overhead and excessive energy consumption for ensuring that this reactive binding is up to date. Therefore, we propose options or *tuning knobs* for trading off strong semantics for energy savings. For example, programmers can slightly relax the semantic by allowing the access if the resource was 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 can notice that there are situations in which dynamic bindings may not be appropriate. Specifically, we may want to access the previously matched resources which are no longer matched. For example, we may have turned on cameras in the area A. However, after a period of time, we may want to turn them off but some cameras have been moved out of the area. If the area A is included in our declarative expression, those moved cameras will no longer match the expression. As a result, we may not be able to turn off the moved cameras directly using the resource variable.

  A naive 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 always describe the desired set of resource. Similarly, the underlying system could provide the function $last()$ which returns the previous set of the matched resources.

```
Resource R = <expression1>;
Resource X = last(R);
```

  However, both solutions incur excessive overhead as they require the system to maintain all changes of a set at all time.

  An alternative solution is to provide explicit instructions for memorizing the 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 match the expression.

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

  However, the 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 changed without an explicit assignment.

```
Iterator i1 = R1->first_element;
```

3.5 Access Timeout

Regardless of the binding type, there is no guarantee that every resource access in WNES will succeed. Unfortunately, resource access time in WNES is unbound and failures are common mainly because of network dynamics. In general, unbound access time and failures cannot be easily differentiated from one another, given that there is no response in both cases [1] Timeout is usually a common technique for handling such problems. Therefore, we propose associating a resource variable with an access timeout. On every access, the access time is monitored. Once the time is out, an exception is raised (similar to Java exceptions). It is necessary that programmers explicitly specify how to handle the timeout expression.

```
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");
}
```

4 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) [10, 2] and Spatial View [13]. DRN, Spatial View, and SP share a vision of programming WNES as a unit, simplifying resource accesses as variable accesses, exposing the space property to the programmers, hiding network details, and supporting imperative programming. However, SP supports only sequential resource accesses whereas DRN supports both sequential and parallel accesses. Accessing resources in parallel can significantly reduce 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 programming and imperative programming. Unlike the DRN binding, the SP binding is, by default, static. Even though dynamic binding in SP is provided as an option, rebinding

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

has to be explicitly instructed by programmers. As opposed to SP, the DRN binding is, by default, dynamic whereas the 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 does have to handle dynamic mappings and frequent access failures whereas the memory-mapped file does not.

The abstract region [17, 18] 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 [12, 8] and LEACH [9], especially the energy savings gained by processing data in the network. Despite this influence on our parallel access, DRN is also similar to diffusion in several ways. Specifically, DRN and diffusion are hybrid programming, given that diffusion APIs [5] require declarative data description for publication and subscription. Furthermore, this data-centric paradigm of diffusion effectively hides significant networking details (from programmers) 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 [16] and COUGAR [1]. While the above efforts propose programming WNES as a database, we propose programming WNES as a single abstract machine.

There exist several research efforts on a hybrid of declarative programming and imperative programming. Examples of such efforts include embedded SQL [6] and constraint imperative programming [7]. In embedded SQL, SQL is mainly used for database accessing whereas imperative programming is used for data processing. In a sense, resources in DRN is analogous to the database in embedded SQL whereby declarative accesses are appropriate. In constraint imperative programming, variables are constrained 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 whereas constraint imperative programming is designed for solution searching on traditional systems. In contrast, DRN targets resource naming on highly dynamic WNES.

5 Conclusions and Future Work

We believe that, to efficiently develop WNES applications, appropriate programming abstractions are neccessary. DRN is such an abstraction which integrates declarative constraints with imperative constructs to form a powerful programming paradigm suitable for macroprogramming WNES. In the future, we intend to further explore the design space of DRN as well as to complete our implementation of a DRN run-time library. Our target platform is Smart Messages (SM) which runs on IPAQs communicating with 802.11 radios. SM is appropriate in a sense that SM supports program migration which is necessary for reprogramming the network. Undoubtedly, there are other reprogrammable platforms such as SensorWare and Mate. 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. As a result, it is possible to macroprogram other wired or wireless networks using our approach.

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] Dan Coffin, Dan Van Hook, Ramesh Govindan, John Heidemann, and Fabio Silva. Network routing appli-

cation programmer's interface (api) and walk through 8.0. Technical Report 01-741, USC/ISI, March 2001.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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). To Appear.*, December 2002.

[17] 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.

[18] 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.