sMapReduce: A Programming Pattern for Wireless Sensor Networks

Vikram Gupta[†‡], Eduardo Tovar[†], Luis Miguel Pinho[†],
Junsung Kim[‡], Karthik Lakshmanan[‡], Ragunathan (Raj) Rajkumar[‡]
[†]CISTER/ISEP, Polytechnic Institute of Porto, Portugal
[‡]Electrical and Computer Engineering, Carnegie Mellon University, USA
[†]{vigu, emt, lmp}@isep.ipp.pt, [‡]{junsungk, klakshma, raj}@ece.cmu.edu

ABSTRACT

Wireless Sensor Networks (WSNs) are increasingly used in various application domains like home-automation, agriculture, industries and infrastructure monitoring. As applications tend to leverage larger geographical deployments of sensor networks, the availability of an intuitive and user-friendly programming abstraction becomes a crucial factor in enabling faster and more efficient development, and re-programming of applications. We propose a programming pattern named *sMapReduce*, inspired by the Google MapReduce framework, for mapping application behaviors on to a sensor network and enabling complex data aggregation. The proposed pattern requires a user to create a network-level application in two functions: *sMap* and *Reduce*, in order to abstract away from the low-level details without sacrificing the control to develop complex logic. Such a two-fold division of programming logic is a natural-fit to typical sensor networking operation which makes sensing and topological modalities accessible to the user.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Programming abstraction, Wireless Sensor Networks

Keywords

Wireless Sensor Networks, Macro-programming, Abstractions, Patterns

1. INTRODUCTION

Wireless Sensor Networks (WSNs) are being deployed for a multitude of applications and to further promote the development of sensing applications, the need for an efficient yet user-friendly programming support has been stressed in the past. Sensor Networks are generally deployed over a large geographical area and often in difficult terrains. Therefore, programming individual nodes can consume numerous man-hours but the network may still lack the provisions for reprogramming in most cases. Several middleware systems have been proposed in the past that provide a uniform environment for programming sensor nodes.

Abstractions provided by network-level programming systems vary from sending application-specific tiny virtual machines to individual nodes [10] to query-based schemes (like [11, 19]). A key design-decision behind a programming abstraction is the trade-off between control available to the programmer and ease-of-use. Many schemes allow programmers to specify the application needs in network-level programs, in turn providing both control and significant abstraction. However, most of such schemes are designed with abstraction as the key focus and the *patterns* behind sensor network operation are neglected. These approaches may provide high degree of abstraction but they lack the freedom for a programmer to make simple optimizations not obvious for an automated solution.

In this paper, we propose *sMapReduce*, a programming abstraction to divide the network-level user program into explicit *sMap* and *Reduce* functions. Most of the sensor networking applications can be visualised as accomplishing two important and largely disjoint functions, *namely*: *i*) Sense and compute, *ii*) Forward and Aggregate. Isolating these functions at the programming abstraction level helps a programmer not only to visualize the network operation with ease, but also helps implementing complex application logic. The *sMap* operation maps the application *behavior* to the *structure* of the network. Hence, we call our approach *sMapReduce*, which stands for *structure-Map & Reduce*. *Structure* means the network topology and the configuration of nodes, including the hardware and software capabilities. By application *behavior*, we mean the expected functionality of the structure of the network of sensor nodes. The *Reduce* function handles the responsibility of data aggregation in the network-tree topology.

Several abstraction concepts from the field of distributed computer systems can be adapted to sensor networks. Sensor networking applications are typically less data-intensive but data is highly correlated to physical location as nodes are deployed to sense the environment. In addition, gathering data efficiently from the nodes in a multihop network requires aggressive packet scheduling and aggregation to reduce the radio and computation resource-utilization. Our proposed programming pattern is inspired from the MapReduce framework [3], a popular data-processing approach in

SESENA'11, 22-MAY-2011, Waikiki, Honolulu, USA

distributed systems. MapReduce framework requires a programmer to divide the processing job into *Map* and *Reduce* functions. *Map* takes a key-value pair as input and converts it to another intermediate key-value pair; *Reduce* does the job of combining this intermediate output from *Map*. Researchers have adapted MapReduce for processing of data on large sensor networks [7] with the premise that nodes carry huge amount of data and parallel computations are required in some applications. We take this concept a step further by mapping behavior to sensor nodes based on their logical and physical topology. The reduce concept is employed to implement aggregation logic over the network tree.

The rest of the paper is organized as follows. Related work is presented in Section 2. Section 3 provides the details of *sMapReduce* pattern with illustrative examples. The design of a layered architecture to support such a programming pattern is described in Section 4. Finally, we present discussion and conclusions about our proposed pattern in Section 5.

2. RELATED WORK

Many macro-programming schemes have been proposed in the past that allow a user to program a wireless sensor network as a whole. Some of those approaches involve programming individual nodes over the air via the delivery of application-specific virtual machines [10]. On the other hand, several macro-programming systems support programming at an abstract higher level such as [5]. The programming system proposed in [9] extends 'C' language for programming at a network level. None of those schemes, however, focus on isolating sensing jobs from data-collection, making network-level programming complex while also not providing appropriate support for efficient data-aggregation.

Design Patterns are a widely accepted software engineering approach for software design. The concept of software patterns was made popular by Ward *et al.* in [1]. With the growing popularity of sensor networks and the challenges in programming sensor nodes, researchers have shown considerable interest in proposing design patterns for sensor networks. Several software design approaches for WSNs are outlined and elaborately classified in [13]. Many design patterns are proposed in [2] to support interactions between Sensor Web and sensor infrastructure through an intermediary layer. TinyOS is a popular operating system for sensor networks and its developers describe the software patterns behind its design in [4]. Various design patterns are proposed in [16] for unifying various middleware and abstractions such that users can effectively program multiple WSN's using different programming systems. None of these patterns, however, attempt to optimize the programming based on the self-evident operation of sensor network.

A macroprogramming approach comparable to ours is *Regiment* [14] that uses the concept of abstract subsets or *regions* [17, 12, 18] for selecting the nodes to be involved in computation. *Regiment* provides programming constructs to map sensor readings to data-types and other functions for convenient mapping of data to data or node subset to other subset. However, there is no explicit isolation between functionality assignment and data aggregation, thus making it hard for a programmer to understand the design, execution and coordination of the application. The complexity of programs in *Regiment* can be high because of complex operations, but the programmer still may not have much control as there is almost no freedom of explicit mapping of functionality to the nodes.

Query-based approaches [11, 19] provide convenient queries for data collection from the sensor network. They also transparently perform aggregation along the network tree to optimize the resource usage. One major drawback of such database-like approaches is that the application logic is implemented by an automated *query planner*, and the programmer does not have control over behavior mapping over structure of nodes. In the following sections, we propose *sMapReduce* programming pattern to assist in the explicit assignment of functionality to nodes while maintaining a user-friendly abstraction.

3. sMapReduce PATTERN

In this section, we describe our proposed *sMapReduce* programming pattern for developing applications on sensor networks. As has been emphasized earlier, bifurcated operation of sensor networks into behavior mapping and data-aggregation motivates corresponding split in network-level programs. The user programs each application using two key functions: *sMap* and *Reduce*. Main objective of *sMap* is to associate sensing and decision-making jobs to the sensor nodes and *Reduce* function handles collection of data through the network-tree while allowing the user to implement complex aggregation logic. *sMapReduce* is a higher-level programming pattern that maintains its expressiveness though disjoint *sMap* and *Reduce* functions.

A simplified example of *sMap* function is shown in Figure 1a. The *sMap* function takes three input arguments. `service_name` is the identifier of the application to be executed on the sensor nodes and `list_of_nodes` is a handler for data structure (or a database) containing topology and tree information of the nodes. `period` is the period in *ms* at which the application repeats itself. The information about the sensor nodes, their hardware and location is compiled and stored in a data-structure during deployment. Most sensor network deployments are done manually, hence the mapping of physical location to a node id can be obtained at this phase. Once such mapping is available, a programmer can refer to a node through its unique id, or through more abstract concept of physical location, logical location in the tree or even filtered based on sensor capability. The sensor capability can be specified by availability of certain type of sensor, computation power or available battery capacity. Even in case of dynamic topologies, the underlying routing and communication infrastructure can share the responsibility of providing frequently updated logical node location and topology information to the programming abstraction.

The functionality of nodes is decided in an *sMap* function. The user can make use of predefined library functions and programming constructs to create programs for the network. Some of the commonly used programming features have been listed in Table 1. Table 2 provides list of operators to select a subset of nodes from `list_of_nodes`. In the example shown in Figure 1, we present a simple *sMap* application for collecting temperature from all nodes in the network. Code for this application consists of a `for` loop to iterate through the list of nodes, an instruction using `get()` to read the temperature reading and then an `smap_emit()` to send the temperature reading along with the node id towards the gateway.

The *Reduce* section of the program is used to specify the aggregation scheme. A separate dedicated section in the program to perform aggregation provides more freedom and flexibility to implement data collection algorithms. The user can assign aggregation responsibilities to different nodes in

Table 1: List of programming constructs

Construct	Details
`list_of_nodes`	data structure containing the list of nodes and their properties
`smap_emit()`	Data to be returned by each node
`get()`	Function to read sensor values into integers, takes sensor name as argument
`set()`	Function to set a GPIO Pin
`clear()`	Clear a GPIO Pin
`toggle()`	Toggle a GPIO Pin

Table 2: List of operators for selecting participating nodes from among the `list_of_nodes`

Operators	Details
`LEAF.`	Nodes on the periphery of the network
`INNER.`	All nodes except the leaf nodes
`HOP(k).`	All nodes at k^{th} hop from the gateway
`HAS(t).`	All nodes that have a `t` type sensor
`BATT(c).`	All nodes having remaining battery capacity of atleast `c`
`CONN(n).`	Nodes having at least `n` neighbors

the network tree. It makes it easier to overlay complex aggregation algorithms over the tree through higher-level abstractions for node addressing. This two-fold advantage is made possible by separating the sensing operation from the data-aggregation in *sMap* and *Reduce* sections. Figure 1b shows an example of a reduce function for calculating the sum of temperature readings obtained in the *sMap* section in Figure 1a. In this example, `INNER` operator is used to select non-leaf nodes and the sum of the input temperature data is calculated over all nodes. Sum of these temperature readings can be used to calculate a more useful parameter such as average temperature at the gateway node. It is trivial to compute commutative operations like sum, maximum, minimum and count. Moreover, as a user can access the nodes according to their physical location or logical location in the network tree, more complex aggregations schemes can be implemented as well.

3.1 A Target Tracking Example

Target tracking is a common application in sensor networks and requires considerable coordination between nodes. We provide an example implementation using signal strength of beacons from a target node in order to demonstrate the advantage of using *sMapReduce*. The application logic is split into *sMap* and *Reduce* functions as shown in Figure 3. *sMap* function reads the Received Signal Strength Indicator (RSSI) values from received packets as shown in line 3 in Figure 3a. The reduce function in Figure 3b triangulates the location of the target when an intermediate node receives information packets from at least three children nodes.

In the *sMap* function each node generates four values: RSSI, corresponding time stamp, location of target and its own ID, as shown in line 5 in Figure 3a. The *Reduce* function receives these values from *sMap*, and evaluates an aggregation at all intermediate nodes. As shown in the example topology in Figure 2, only node `6` is able to collect three values required for triangulation of the target node `T` tracked by nodes `1`, `2` and `3`. The *Reduce* function in the example implements the majority of the application logic because only

```
1 smap(service_name, list_of_nodes, period){
2   for each node in list_of_nodes
3     temp_value = gets(TEMP);
4     smap_emit(temp_value, node_id);
5   end
```

(a) sMap Function

```
1 reduce(data, list_of_nodes){
2   for each node in INNER.list_of_nodes
3     sum += data.temp_value; //AGGREGATION
4   end
5   return sum;
6 }
```

(b) Reduce Function

Figure 1: A simple example for collecting sum of temperatures from a wireless sensor network

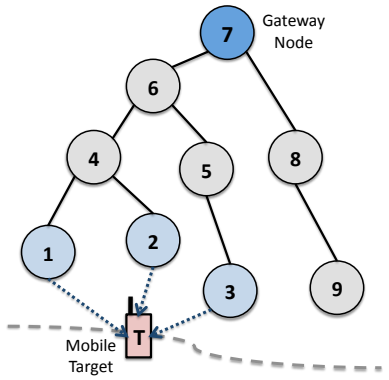

Figure 2: An example topology to demonstrate location tracking of a target node

an intermediate node can process the RSSI information to estimate the location of the target. The reduce function also ascertains temporal correlation of RSSI values from different nodes by checking whether the all time stamps are lie than a window of size `win` (line 6, Figure 3b). It is evident from this example that *sMapReduce* performs aggregation close to the leaf nodes, reducing the communication and computation overhead near the gateway node. The `triangulate()` function in line 8 calculates location of target node based on RSSI values and coordinates of infrastructure nodes. Its implementation is omitted for brevity purposes, as it does not influence the goal or design of our proposed pattern.

Approaches like TinyDB do not capture sensing or topological modalities, as the aggregation is handled by an automated query planner. The design of application logic might be simpler in TinyDB in many cases but *sMapReduce* allows a programmer more control with an implicit understanding of physical and logical location of nodes. More complex schemes like *Regiment* do not isolate the functionality from aggregation explicitly, which can complicate the application logic with sensing job being undesirably coupled to various points in the program.

3.2 Mapping Applications for Mobile Nodes

The sociometric badge [15] is an example sensor network application that targets assisted-living scenarios. The in-

```
1 smap(target_track, list_of_nodes, period){
2   for each node in list_of_nodes
3     rssi_v = get(RSSI);
4     ts = get(time);
5     smap_emit(rssi_v, ts, node_id, loc);
6   end
```

(a) sMap Function

```
1 reduce(data, list_of_nodes){
2   for each node in INNER.list_of_nodes
3     if(data.loc != NULL)
4       return data.loc;
5     else
6       if(max(ts)-min(ts)<=win
7           && size(data.rssi_v) >= 3)
8         triangulate(rssi_v, loc);
9       else
10        return data;
11      end
12    end
13  end
14 }
```

(b) Reduce Function

Figure 3: A location tracking example using RSSI values of packets received by infrastructure nodes from a mobile target.

frastructure for such an application is expensive to maintain once the nodes have been distributed and deployed. Adding additional features is likely to be impossible, and the lack of resources on specific nodes restricts the services that they can offer. The presence of mobile nodes also adds additional complexity with respect to node reprogramming and data aggregation. The proposed programming pattern *sMapReduce*, provides a flexible and extensible mechanism to develop such systems, which could consist of both mobile and static sensor nodes. In order to support such systems, *sMapReduce* introduces two new aspects: (i) multi-level mapreduce function support, and (ii) periodic map execution. This enables system designers to use *sMapReduce* on sensor network systems with mobile nodes. Figure 4a shows an example system, where a mobile node called FireFly badge [8], is used to build the above-mentioned assisted living infrastructure. The FireFly badge could be hosting two location-based applications: (i) emergency alarm that needs to be loaded when the user is in a bathroom, and (ii) a schedule reminder that needs to be loaded when the user is in a living room. The `smap_location` function is executed periodically, and it tracks the location of the FireFly badge so that `smap_location` can map the corresponding application to the badge. Then, `smap_service`, the second level map function, will map `schedule_reminder` to the badge if the user is in the living room and `emergency_alarm` if the users is in the bathroom. Therefore, depending on the user location, a different application can be dynamically mapped on to the mobile node, and this enables programming of context-sensitive map/reduce operations. This example thus illustrates a simple scenario where the multi-level mapping and periodic map execution features of *sMapReduce* can enable its use in networks with mobile nodes, where platforms

```
1 target_service = smap_location(
      FireFly_Badge, list_of_nodes, period );
2 data = smap_service( target_service.name,
      target_service.nodes, target_service.
      period );
3 result = reduce(data, list_of_nodes);
```

(a) An example code for supporting mobile nodes

```
1 smap_location(service, list_of_nodes,
      period) {
2   for each node in INNER.list_of_nodes
3     if(node.location == bathroom)
4       smap_emit(emergency_alarm);
5     else if(node.location == livingroom)
6       smap_emit(schedule_reminder);
7     end
8   end
9 }
```

(b) Implementation of the first level sMap function

Figure 4: Example of mobile node support

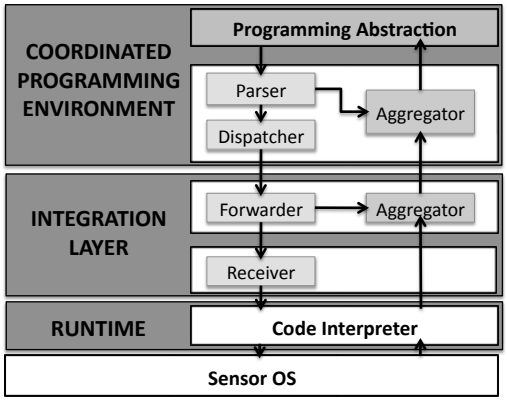

Figure 5: *sMapReduce* system architecture showing three major layers to support a network-level programming abstraction

such as [14, 10, 11] cannot be easily applied.

3.3 Features

The design of the *sMapReduce* programming pattern is based on the principle that typical sensor network operation consists of two relatively disjoint functions. One associates a behavior to sensor nodes and other executes data aggregation over the distributed network. Hence, dividing the user program in explicit *sMap* and *Reduce* sections is a natural-fit to sensor network operation. We provide below some features of the pattern to emphasize on the design decisions behind the *sMapReduce*.

Two-fold operation Typical sensor network operation consists of programming of the nodes and collection of data. These two are handled independently at different layers in the network. Further details of this operation are provided in Section 4.

Data correlation A sensor network is a distributed system where data of interest is the physical environment

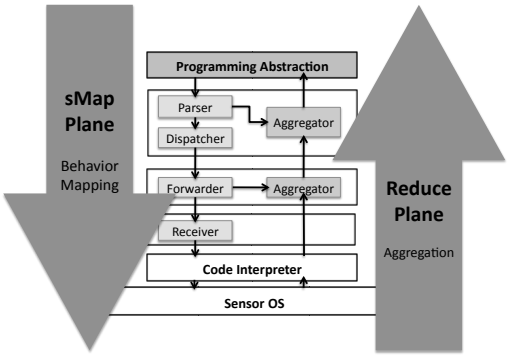

Figure 6: Operation of *sMap* and *Reduce* planes. *sMap* operation involves top-down mapping of behavior to each node from gateway to leaf nodes, and *Reduce* handles data aggregation from leaf-nodes upwards

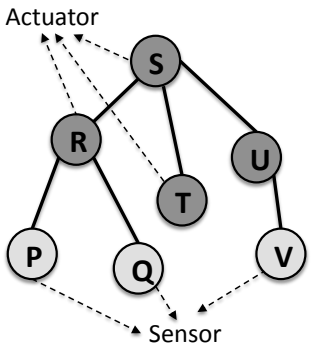

Figure 7: A network with both sensors and actuators to demonstrate that local decision to actuate may not always be isolated from the *Reduce* logic

itself. Therefore, any computation on data should be conducted in the close neighborhood of the sensor node.

Programmer Support Explicit division of programs into *sMap* and *Reduce* sections allow the programmers to easily isolate the key functions, thus helping in easy inference and debugging of applications.

Balanced abstraction and control *sMapReduce* provides easy to use libraries and abstractions to deploy large scale applications in addition to the ability to address individual nodes for fine-grained control to the user.

Expressiveness *sMapReduce* is a pattern derived from the operation of a sensor network, and it allows the programmer to conveniently map behavior of sensing and aggregation to network structure. The programmer can leverage subtle optimizations without much complexity in the application logic.

4. SYSTEM DESIGN

As previously stated, a typical operation of a sensor network involves two major components; *one* handles the programming of and coordination among nodes and *another*, governs aggregation of data over the multi-hop network tree. We can conceptualize this two-fold operation as two independent planes that we call *sMap* plane and *Reduce* plane. Based on this concept, *sMapReduce* facilitates a programmer to distribute functionality in two separate sections. Architecture of a typical framework to support proposed abstraction consists of three major layers as shown in Figure 5.

More details of different layers are provided in [6], with the design and implementation of a macro-programming framework to support multiple applications. As shown in Figure 5, the proposed abstraction needs to be supported by three similar layers, but various components of the architecture are designed to help integrate the *sMap* and *Reduce* operations into the coordinated programming environment. In the following subsections, we briefly explain the operation of the system from the context of *sMapReduce*.

4.1 sMap Plane

In the *sMap* operation of the system, the primary function is to assign specific behavior to each of the nodes in the network. *Behavior* in this case means all tasks executing on the node, along with communication handling and participating in data forwarding and aggregation. The layered structure along with *sMap* and *Reduce* operations is shown in Figure 6. The top-layer of this architecture is the programming abstraction for the user to create network level programs in *sMap* and *Reduce* sections. The user-written program is compiled and converted into byte-codes to be executed on individual nodes. Byte-code execution implements the node *behavior* with the support from the sensor operating system. Byte-code is sent via the wireless network to each of the nodes, which is handled by a data-handler in the Integration Layer. The data-handler connects all sections of sensor networking infrastructure spread over various layers, from the user-end PC at the top to the gateway node and intermediate nodes in the middle and to leaf nodes at the bottom. Once the byte-codes are delivered to each node according to their function, a code-interpreter converts them to sensor networking OS instructions. The byte-codes contain both the program to be executed and the aggregation scheme to be followed at each intermediate node. The main job of the *sMap* plane is to provide network abstraction and assign jobs to nodes while maintaining coordination among multiple applications and network hops.

4.2 Reduce Operation

Once the nodes receive the byte-code specifying their functionality/behavior, the nodes start the execution of the new application. The role of each node in the *Reduce* plane is also included in the byte-code where the intermediate nodes in the network tree help in aggregation of data. The aggregation of data is specified by the user in the Reduce section of the program. The role of aggregation can be different for different nodes, depending on both the physical and logical location of the node in the multi-hop network. A leaf node should only forward locally sensed and computed information, and intermediate nodes may combine the data from their respective children nodes. In addition to aggregation, the Reduce plane should align, merge and schedule packets to reduce the overhead in communication. The Reduce plane is implemented through an aggregator module at every node and the Integration Layer supports the communication of data among different aggregator modules. Figure 6 shows how the Reduce plane overlaps over the right half of the system architecture. This split in the operation of sensor network justifies having explicit *sMap* and *Reduce* sections.

5. DISCUSSION AND CONCLUSIONS

Most applications of sensor networks are usually designed under the "Sense and Send" functional motive, implying that the sensor nodes typically sense the physical environment, and send the data to a gateway node through a tree rooted at the gateway. Data can be aggregated along the tree to save the communication involved. Almost any application which can be classified into such model can benefit from the *sMapReduce* programming pattern. Applications with a more mesh-like topology nodes however, may not benefit from this pattern as explicit mapping and aggregation may not be possible or required. It should be noted that if there is a requirement of collecting data at a node in a mesh structure, the underlying graph can be reduced to a tree and *sMapReduce* can then be employed with ease. In case of applications that require localized decision making instead of at the root node, it may not be possible to separate the *Reduce* logic from *sMap* logic in a straightforward fashion. For example in the topology shown in Figure 7, some nodes (dark shade) have actuation capabilities in addition to the sensing nodes (light shade). If an application logic involves actuation at node `R` based on sensing from nodes `P` and `Q`, the reduce logic will invariably involve actuation logic. In this case, behavior mapping is not limited to *sMap* section and explicit distinction is no longer possible.

sMapReduce is a programming pattern that can enable most common sensor networking applications including simple data-collection ones to target tracking applications with complex logic and aggregation. As future work, we would like to explore similar patterns that can allow programmers to express applications for even more generic topologies with desirable ease.

6. REFERENCES

[1] Beck, K., and Cunningham, W. Using pattern languages for object-oriented program. In *OOPSLA '87 workshop on Specification and Design for Object-Oriented Programming* (1987).

[2] Broring, A., Foerster, T., and Jirka, S. Interaction patterns for bridging the gap between sensor networks and the sensor web. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on* (2010), pp. 732 –737.

[3] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *OSDI* (2004), 13.

[4] Gay, D., Levis, P., and Culler, D. Software design patterns for tinyos. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (Chicago, Illinois, USA, 2005), LCTES '05, ACM, pp. 40–49.

[5] Gummadi, R., Kothari, N., Govindan, R., and Millstein, T. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (Brighton, United Kingdom, 2005), ACM, pp. 1–2.

[6] Gupta, V., Kim, J., Pandya, A., Lakshmanan, K., Rajkumar, R., and Tovar, E. Nano-cf: A coordination framework for macro-programming in wireless sensor networks. *CISTER/ISEP Technical Report, HURRAY-TR-110110*.

[7] Jardak, C., Riihijärvi, J., Oldewurtel, F., and Mähönen, P. Parallel processing of data from very large-scale wireless sensor networks. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (Chicago, Illinois, 2010), HPDC '10, ACM, pp. 787–794.

[8] Kandhalu, A., Lakshmanan, K., and Rajkumar, R. U-connect: a low-latency energy-efficient asynchronous neighbor discovery protocol. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks* (2010), ACM, pp. 350–361.

[9] Kothari, N., Gummadi, R., Millstein, T., and Govindan, R. Reliable and efficient programming abstractions for wireless sensor networks. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (San Diego, California, USA, 2007), PLDI '07, ACM, pp. 200–210.

[10] Levis, P., and Culler, D. Matè: a tiny virtual machine for sensor networks. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (San Jose, California, 2002), ASPLOS-X, ACM, pp. 85–95.

[11] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. 30*, 1 (2005), 122–173.

[12] Mottola, L., and Picco, G. P. Logical neighborhoods: A programming abstraction for wireless sensor networks. In *Lecture Notes in Computer Science : Distributed Computing in Sensor Systems*, vol. 4026/2006, Springer Berlin / Heidelberg, pp. 150–168.

[13] Mottola, L., and Picco, G. P. Programming wireless sensor networks: Fundamental concepts and state-of-the-art. *ACM Computing Surveys* (2010).

[14] Newton, R., Morrisett, G., and Welsh, M. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks* (Cambridge, Massachusetts, USA, 2007), IPSN '07, ACM, pp. 489–498.

[15] Olguín, O., et al. Sociometric badges: Wearable technology for measuring human behavior.

[16] Tei, K., Fukazawa, Y., and Honiden, S. Applying design patterns to wireless sensor network programming. In *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on* (2007), pp. 1099 –1104.

[17] Welsh, M., and Mainland, G. Programming sensor networks using abstract regions. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (San Francisco, California, 2004), USENIX Association, pp. 3–3.

[18] Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services* (Boston, MA, USA, 2004), ACM, pp. 99–110.

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