Expressing Sensor Network Interaction Patterns using Data-Driven Macroprogramming *

Animesh Pathak[‡], Luca Mottola[#], Amol Bakshi[‡], Viktor K. Prasanna[‡], and Gian Pietro Picco[†]

[‡]University of Southern California [#]Politecnico di Milano [†]University of Trento

{animesh,amol,prasanna}@usc.edu mottola@elet.polimi.it picco@dit.unitn.it

Abstract

Wireless Sensor Networks (WSNs) are increasingly being employed as a key building block of pervasive computing infrastructures, owing to their ability to be embedded within the real world. So far, pervasive applications for WSNs have been developed in an ad-hoc manner using node-centric programming models, focusing on the behavior of single nodes. Instead, macro-programming models provide much higher levels of abstractions, allowing developers to reason on the sensor network as a whole.

In this paper, we demonstrate how a wide range of interaction patterns *commonly found in pervasive, embedded applications can be expressed using ATaG, a data-driven macro-programming language. To support this, we showcase* real-world applications *developed in ATaG, and consider both homogeneous, sense-only scenarios, and heterogeneous settings involving actuation on the environment under control.*

1. Introduction

Recent technological advances have made wireless sensor networks (WSNs) a viable solution for embedded sensing and actuation [1]. The WSN devices can be easily embedded within the physical world, thus realizing the vision of "disappearing" computing [16]. However, there are several important issues still to be resolved before achieving that vision. Among them, *ease of programming* is a key challenge.

Existing solutions in the field of programming WSNs can be broadly classified as either *node-centric*- or *macro-programming*. In the former approach, the programmer must translate a global behavior into local actions on a node, e.g., as message exchanges, and express these actions using low-level programming languages like nesC [7]. Although this allows cross-layer optimizations, the required expertise makes this approach unsuited to developing sophisticated applications in large-scale sensor networks. Conversely, the objective of *macro-programming* is to allow the programmer to reason on the sensor network as a whole, and write a distributed application without explicitly managing control, coordination, and state maintenance. The global behaviors specified using macro-programming are then automatically synthesized into software for each node in the target environment. Existing proposals in the field of macro-programming include the Kairos [8] system and the Regiment [13] functional language.

In this paper, we demonstrate how different *interaction patterns* taken from pervasive, embedded applications can be expressed concisely using ATaG (Abstract Task Graph) [3], a data-driven macro-programming language described in Section 2. To showcase the expressivity of ATaG in describing a range of different coordinated behaviors, in Section 3 we examine interaction patterns characterized by:

- **Hierarchical Data Collection:** one of the most common behaviors found in mainstream WSN applications, where mostly *homogeneous* nodes are usually employed [10].
- **Localized Interactions:** relevant to applications like target tracking, and characterized by *asynchronous triggering* of operations [6] when specific conditions are met.
- **Actuation Driven by Sensing:** requiring a concise description of control loops in *heterogeneous* systems [4] composed of sensors and actuators with different capabilities.

To make our illustration more concrete, we refer to the development of a relevant set of *real-world* applications as examples. The *contribution* of this paper is to demonstrate that data-driven abstractions and constructs of ATaG are able to specify the aforementioned interaction patterns effectively, while easing the programming task.

*This work is partially supported by the European Union under the IST-004536 RUNES project and by the National Science Foundation, USA, under grant number CCF-0430061.

Proceedings of the Fifth Annual IEEE International Conference
on Pervasive Computing and Communications Workshops(PerComW'07)
0-7695-2788-4/07 $20.00 © 2007 IEEE

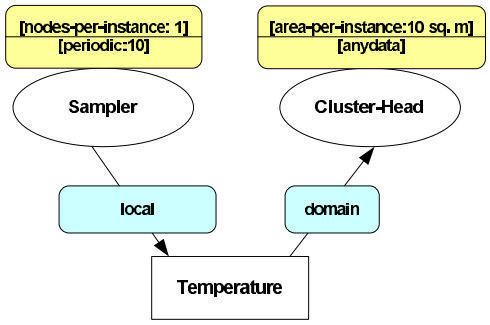

Figure 1. A sample ATaG program.

To complement our discussion, Section 4 briefly sketch the process of compiling ATaG programs, and the kind of distributed run-time support the compiler targets. Finally, Section 5 concludes with our plans on finalizing the development of the ATaG framework.

2. The ATaG Macroprogramming Language

The Abstract Task Graph (ATaG) model and language provide a mixed *declarative-imperative* approach to the development of networked embedded applications. At its core are the notions of *abstract task* and *abstract data item*. The former is a logical entity encapsulating the processing of one or more data items, which represent the information itself. The flow of information between tasks is defined in terms of their input/output relations. To achieve this, *abstract channels* are used to connect a task to a data item when the task *produces* that item, or vice versa when the task *consumes* it. The programmer then specifies the processing of data items within tasks using an imperative programming language such as Java. To express the interactions between tasks, programmers are provided the abstraction of a *shared data pool*, to which each single task either *outputs* data, or is *notified* when some data of interest is available. A dedicated API is provided in support of these operations.

Figure 1 illustrates an example ATaG program, specifying a simple cluster-based communication pattern. Sensors within a cluster take periodic temperature readings, which are collected by the corresponding cluster-head. The former aspect is encoded in the *Sampler* task, while the latter is represented by the *Cluster-Head* task. The *Temperature* data item is connected to both tasks using a channel originating from the *Sampler* task, and a channel directed to the *Cluster-Head*, as required by the application semantics.

Tasks are annotated with *firing* and *instantiation* rules. The former specify when the processing in a task must be triggered. In our example, the *Sampler* task is triggered every 10 seconds. The `any-data` firing rule requires the invocation of *Cluster-Head* whenever at least one data item is ready to be consumed on *any* of its incoming channels. A further firing rule named `all-data` can be used to defer the task firing until at least one data item is available on *all* the incoming channels. The placement of tasks on real nodes is governed using instantiation rules. The `nodes-per-instance:`q construct used in this example requires the task to be installed once every q nodes in the system. As $q = 1$ in the example, the *Sampler* task is instantiated on every node. Instead, the `area-per-instance` construct is used for the *Cluster-Head* to partition the geographical space, and deploy *one* instance of the task for each partition.

In pervasive, sensing/acting scenarios, the processing in a task may depend on the node capabilities, e.g., when different sensors and actuators are both present in the system. To account for this, nodes export a list of *attributes* describing their characteristics, for instance, the sensors they are equipped with. Developers are then given the ability to control the placement of tasks on nodes by expressing boolean conditions over these attributes, e.g., requiring the presence of a specific sensing device. Sample use of these constructs will be presented in Section 3.

Abstract channels are annotated to express the specific *interest* of a task in a data item. In our example, the *Sampler* task generates data items of type *Temperature* kept `local` to the node where they have been generated. Conversely, the *Cluster-Head* uses the `domain` annotation to gather data from the temperature sensors in its cluster. This construct binds to some system partitioning, e.g., that obtained by `area-per-instance`, and associates the tasks running in the same domain. Alternative constructs allow to express interests within a specific neighborhood, ranging from a maximum number of network hops (or maximum geographical distance) to all the tasks in the system (the latter being specified with the `all-nodes` annotation).

Instantiation rules for tasks and channels annotations jointly define *logical scopes*, i.e., set of tasks sharing common characteristics and interests, and communicating with each other. The scopes can be computed by combining the task and channel annotations in the ATaG program. The instantiation rules select the nodes in the system where the tasks are to be assigned depending on their characteristics. On the other hand, the channel annotations bind these different tasks according to their interests in terms of communication. In our example, the `domain` annotation lets temperature sensors communicate with the *Cluster-Head* obtained from `area-per-instance`. Remarkably, all the interaction patterns in the present version of ATaG can be modeled as communication within logical scopes. This enables an easy implementation of the underlying distribution aspects using middleware-level abstractions exporting this same notion. More details on this will be given in Section 4.

Proceedings of the Fifth Annual IEEE International Conference
on Pervasive Computing and Communications Workshops(PerComW'07)
0-7695-2788-4/07 $20.00 © 2007 IEEE

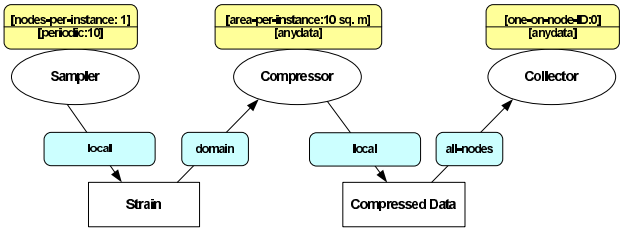

Figure 2. An ATaG program for landslide detection.

3. Application Case Studies

In this section, we put forth the contribution of this paper by focusing on a set of interaction patterns commonly seen in pervasive, embedded applications (introduced in Section 1), and demonstrate the flexibility by which ATaG can express these by means of application case studies.

3.1. Hierarchical Data Collection

The largest fraction of currently ongoing projects in sensor networks perform data gathering of some kind. In the general case, data generated by sensors is compressed in-network before being sent out to a base station. The `domain` channel annotation, combined with the `area-per-instance` instantiation rule, provide a very concise way of representing this behavior. The following application illustrates this fact.

Sample Application: Landslide Detection. In [15], the authors have proposed a novel application of WSNs to detect landslides. They deploy strain detecting sensors all over the landscape, which detect strain patterns in the ground. The communication pattern follows a traditional cluster-like paradigm. The data is first routed to a close cluster-head where it is compressed, and then ultimately routed to a base station.

ATaG Program. The ATaG program for landslide detection, shown in Figure 2, consists of three tasks. The communication of the *Strain* data item from the *Sampler* to *Compressor* task is similar to our example in Section 2. The *Compressor* converts the strain readings of all the nodes in its domain to a *CompressedData* abstract data item. Finally, the *Collector* task, located only at the base station (node 0 here, as per the corresponding instantiation rule), collects the compressed data for generating alerts using the `all-nodes`.

3.2. Localized Interactions

Nodes in a WSN are often required to interact with other nodes in their vicinity before making decisions. These *localized interactions* [6] are one of the factors differentiating WSNs from traditional distributed systems, where the geographical location of the processors is immaterial. Such local interactions can be represented in ATaG using the `neighborhood-hops` and `neighborhood-distance` channel annotations. These interactions are useful in a wide variety of scenarios, e.g., tracking a moving object or finding the contour of an oil spillage. The following target-tracking example illustrates our point.

Sample Application: Target Tracking. Target tracking is a well-studied problem in the sensor network domain. As an example, we look at an application similar to the work being done on VigilNet [9]. Our sample application consists of magnetometer-equipped sensors deployed in a battlefield. The main aim is to detect the presence and location of targets and send the information to the base station. In our case, only the perimeter sensors sample their surroundings initially, while the internal ones do not. When the perimeter sensors detect a target, they alert the nearby sensors who begin sampling their surroundings. The nodes sensing a moving object execute a leader election protocol to identify a specific node declaring to *own* the target. This will be responsible for logging the data related to that target, and report to the base station.

ATaG Program. Our ATaG program for target tracking is shown in Figure 3. The *Perimeter Sampler* is in charge of the initial monitoring on the perimeter of the controlled area. Therefore, it is the only task that should be running at system start-up. To specify this, we use `start@init` as firing rule. Furthermore, to require its instantiation only on perimeter nodes, we use `nodes-per-instance:1@perimeterNodes` as instantiation rule. In this case, `perimeterNodes` is a placeholder for a boolean predicate defined over the node attributes. We are indeed assuming each node has an associated boolean attribute `isPerimeterNode`, and the aforementioned placeholder is defined as $perimeterNodes ::= isPerimeterNode = TRUE$. The whole construct defines a *scope* in the system, where only *Perimeter Sampler* tasks should be installed. When such a task detects a target within range, it reports its readings as a *Target Entered* data item in a 10 meter radius. The latter aspect is specified in the `neighborhood-distance:10` channel annotation, again defining a (different) scope where the data should be disseminated.

The *Inner Sampler* task is similar to the *Perimeter Sampler*, except that it is instantiated on non-perimeter nodes. When it receives a *Target Confirmed* data item, it starts sampling the surroundings periodically and produces a *Target Detected* data item to be consumed by its neighbors in a 10 meter radius.

Proceedings of the Fifth Annual IEEE International Conference
on Pervasive Computing and Communications Workshops(PerComW'07)
0-7695-2788-4/07 $20.00 © 2007 IEEE

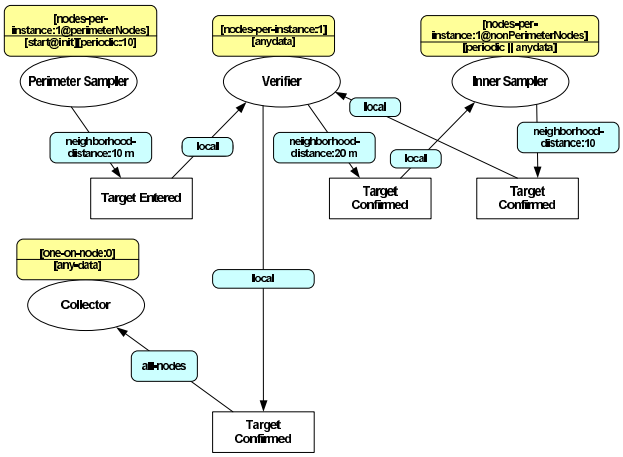

Figure 3. An ATaG program for target tracking.

The *Verifier* task runs on each node, and implements the leader election protocol. Its decision is based on the *Target Entered* and *Target Detected* data items, which convey the magnetometer readings at the nodes where they are produced. Specifically, the *Verifier* matches the reading on a node with that of others in a 10 meter radius, and decides whether the node is indeed the leader, i.e., it *owns* the target, depending on the location of the strongest reading. When a *Verifier* knows that it *owns* a target, it produces two data items. The *Target Confirmed* data item serves to induce periodic sensing in nodes that are in a 50 meter radius (the scope of this data item can be tuned according to the expected speed of the object being tracked). The *Target Info* data item is sent to the base station for logging purposes.

The *Collector* task runs only on a single node (node 0) and is responsible for collecting the target position from the sensor field using the *Target Info* data item.

3.3. Actuation Driven by Sensing

The WSN community is rapidly exploring the use of sense-and-act systems and the issues involving the closure of the control loop. A commonly seen construct in these applications is that data collected from the sensors in a region is used to make a decision about actions to be taken in a possibly different region. Examples of such systems include traffic control and fire-fighting in a building. The above mentioned behaviors can easily be specified with different sensing and actuating tasks, connected by processing tasks. We elaborate our ideas by the following example.

Sample Application: Building Management. One of the oft-cited examples of pervasive computing is the so-called *smart building* [14], which can manage itself. Part of this task is the management of the HVAC (heat, air-conditioning and ventilation) system. WSNs have been proposed as a solution to this problem recently [5]. We consider a set of nodes spread across a building, with each node possibly attached to a temperature sensor, a humidity sensor and an actuator that can control the temperature and humidity of a region.

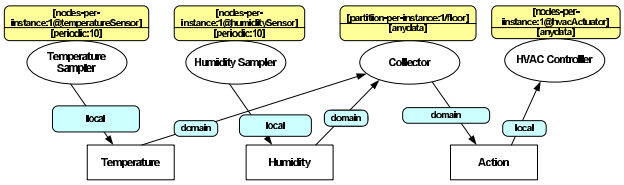

Figure 4. An ATaG program for building environment management.

ATaG Program. Figure 4 graphically describes our ATaG program. To handle heterogeneity, we make use of the same, extended instantiation rules described in the previous application. To that end, we assume the nodes in the system declaring, among their attributes, the sensors or actuators they are equipped with. The placeholders *temperatureSensor*, *humiditySensor*, and *hvacActuator* are defined according to these attributes. For instance, the former is defined as $temperatureSensor ::= temperature \in equippedSensors$, where *equippedSensors* is a node attribute describing the set of sensing devices attached to a node. The *Temperature Sampler* and *Humidity Sampler*—simply instantiated on the corresponding node type—sample their surroundings and generate *Temperature* and *Humidity* data items.

In addition, we want a single *Collector* task to be the guardian of each section of the total deployment area, thus being in charge of the local control loop in that section. In this example, we have chosen to instantiate one *Collector* on each floor. To express this, we further assume the nodes also declare the floor where they are deployed as an attribute. Based on this, we can achieve a per-floor partitioning of the system. The instantiation rule `partition-per-instance:1/floor` relies on this partitioning, and express the requirement of instantiating one *Collector* task on each floor. Similarly to the `area-per-instance` construct, this partitioning can be used as a domain. In this case, it binds the *Collector* task to the humidity and temperature sensors on the same floor, so that the former can collect data from the latter. Again, the `domain` construct defines logical scopes composed of the *Collector* task on a floor and either *Temperature Sampler* or *Humidity Sampler* tasks on the same floor.

After processing the data, the *Collector* produces a command for the actuating tasks in the form of an *Action* data item. The *HVAC Controller* task is placed on all nodes of type `hvacActuator` and responds to the *Action* data item

Proceedings of the Fifth Annual IEEE International Conference
on Pervasive Computing and Communications Workshops(PerComW'07)
0-7695-2788-4/07 $20.00 © 2007 IEEE

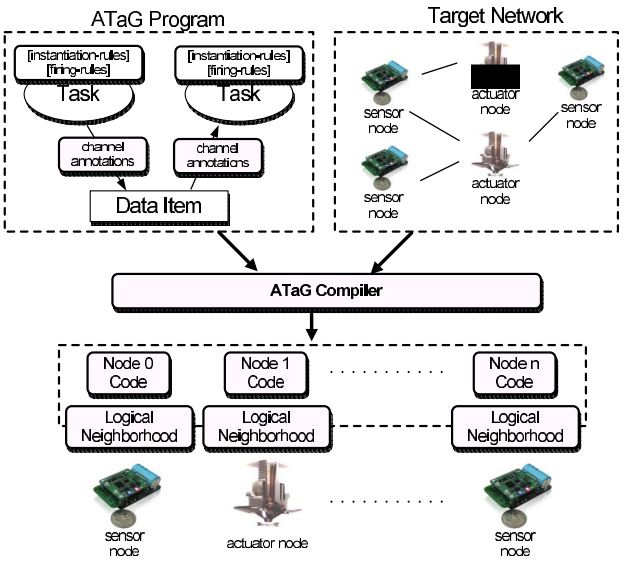

Figure 5. The compilation process.

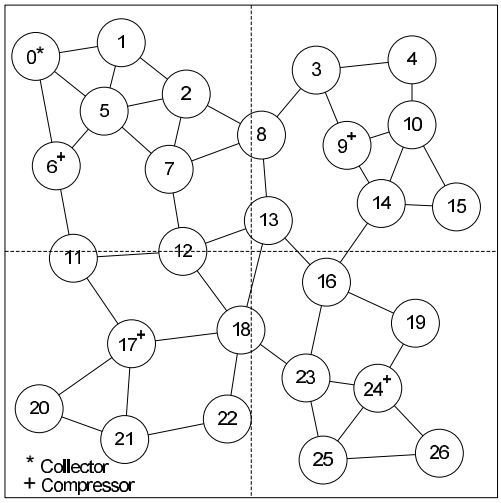

Figure 6. Task assignment for landslide detection.

by adjusting the temperature/humidity controls.

4. The ATaG Compilation Framework

The ATaG Compiler is in charge of converting the high-level description of the application to the node level code, targeting a specific API made available by an underlying run-time system, as shown in Figure 5. This section describes the compilation process using the application described in Section 3.1 as an example.

Input. The input to the compiler consists of **a)** the Abstract Task Graph, consisting of a set of abstract tasks, a set of abstract data items and a set of abstract channels; **b)** the imperative part of each abstract task, and **c)** the description of the target network. In this respect, the minimum information required by the compiler is the *number of nodes* in the system. Other information such as location of each node may be required if the program uses spatial task-instantiation primitives such as `area-per-instance`. Node attributes are needed if the programmer defines predicates over them to handle heterogeneity, as we described in Section 3.

Output. In Section 2 we mentioned how communication between ATaG tasks can be modeled in terms of logical scopes. In accordance with this, we decided to use *Logical Neighborhoods* [12, 11] as the target API of the compilation process, and as the underlying support to manage communication among the nodes. With Logical Neighborhoods, the physical neighborhood of a node is replaced with a logical notion of proximity, where one can determine the (logical) neighbors of a node using applicative information. This is achieved by expressing a boolean predicate over node attributes, that acts as a membership function for the set of nodes included in a logical neighborhood. To interact with the nodes in a neighborhood, a message-passing API is provided, supported by an efficient routing mechanism at the network level [11]. Scoping in ATaG can be naturally mapped to logical neighborhoods, since the concept of node attribute in ATaG is essentially the same as in Logical Neighborhoods, and the instantiation rules can be easily converted to boolean predicates. This way, the job of the ATaG compiler is simplified, as it targets a node-level abstraction already providing a logical layer on top of the physical topology.

Besides the calls to the Logical Neighborhoods API, the output of the compiler also includes the list of tasks to be run on each node, their firing rules, the data items to be handled at each node.

Compilation Process. For our landslide detection example, Figures 2 and 6 illustrate the input program and network respectively. The target network is a 27 node system spread over a region of 40 sq. m. The compilation process takes place in a stepwise manner. First, the abstract tasks in the ATaG program are instantiated according to the annotations used by the programmer. For example, in our application, the `nodes-per-instance:1` instantiation rule of the *Sampler* task results in 27 instantiations of the task, one for each node. For primitives such as `area-per-instance:10 sq. m.`, four instances of the *Compressor* task are generated, one for each region bordered by the dotted lines. Note that the number of instantiated tasks generated depends on the number of nodes in the target network in the former and the total deployment area of the WSN in the latter case.

The second step in compilation is the assignment of the tasks instantiated above to nodes. Since the location of tasks

Proceedings of the Fifth Annual IEEE International Conference
on Pervasive Computing and Communications Workshops(PerComW'07)
0-7695-2788-4/07 $20.00 © 2007 IEEE

will govern communication patterns and therefore, the energy dissipated by the system, we believe that there is a rich space of optimizations possible in this step.

Figure 6 illustrates a task allocation generated by the above mentioned compilation process for the application described in Section 3.1 on a particular network. In this case, the *Sampler* task is instantiated on all nodes and it not shown. The *Compressor* task is instantiated on nodes 6, 9, 17 and 24, and the *Collector* task is assigned to node 0. The presence of the tasks on the nodes is indicated by the symbols + and * next to the node IDs. Note that this is not the only assignment possible for this ATaG program. For example, the *Compressor* task on node 6 can be assigned to any node in the top-left square (node 1, for example), and still the system generated will be functionally compliant to the ATaG program. This fact is the basis of our future direction of research, where we will explore the effects of various task assignment techniques and develop optimizations based on them.

In our project on data-driven macroprogramming [2], we are developing many such applications and measuring the amount of effort involved from the application designer's side.

5. Conclusion and Future Work

In this work, we demonstrated how common interactions patterns coming from pervasive, embedded applications can be described using the ATaG language. We carried out this exercise in both homogeneous and heterogeneous scenarios. The latter are of great importance for the upcoming sense-and-actuate systems, where we claim data-driven macroprogramming well fits. We also gave a brief overview of the compilation process that converts macro-programs into node-level code, and of the middleware-level support we intend to use to support communication.

Our future work is in two directions: firstly, we are working on a full end-to-end application development framework for macro-programming sensor networks, and secondly, we are looking at the possible optimizations in the compilation process.

References

[1] I. F. Akyildiz and I. H. Kasimoglu. Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks Journal*, 2(4):351–367, October 2004.

[2] ATaG: A Toolkit for Architecture Independent Programming and Synthesis for Networked Sensing Applications, http://indus.usc.edu/ATAG/.

[3] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner. The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. In *Workshop on End-to-end Sense-and-respond Systems (EESR)*, June 2005.

[4] I. Chatzigiannakis, A. Kinalis, and S. Nikoletseas. An adaptive power conservation scheme for heterogeneous wireless sensors. In *Proceedings of the Seventeenth (17th) Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2005)*, pages 96–105, 2005.

[5] M. Dermibas. Wireless sensor networks for monitoring of large public buildings. Technical report, University at Buffalo, 2005.

[6] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: scalable coordination in sensor networks. In *Proc. of the* 5^{th} *Int. Conf. on Mobile computing and networking (MobiCom)*, 1999.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2003.

[8] R. Gummadi, O. Gnawali, and R. Govindan. Macroprogramming wireless sensor networks using Kairos. In *Intl. Conf. Distributed Computing in Sensor Systems (DCOSS)*, June 2005.

[9] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic, T. F. Abdelzaher, J. Hui, and B. Krogh. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. Sen. Netw.*, 2(1):1–38, 2006.

[10] S. Madden, M. J. Frankiln, J. M. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *Proc. of the* 1^{st} *Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 126–137, 2003.

[11] L. Mottola and G. P. Picco. Logical Neighborhoods: A programming abstraction for wireless sensor networks. In *Proc. of the* 2^{nd} *Int. Conf. on Distributed Computing on Sensor Systems (DCOSS)*, 2006.

[12] L. Mottola and G. P. Picco. Programming wireless sensor networks with logical neighborhoods. In *Proc. of the* 1^{st} *Int. Conf. on Integrated Internet Ad hoc and Sensor Networks (InterSense)*, 2006.

[13] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *1st Intl. Workshop on Data Management for Sensor Networks (DMSN)*, 2004.

[14] E. Petriu, N. Georganas, D. Petriu, D. Makrakis, and V. Groza. Sensor-based information appliances. *IEEE Instrumentation and Measurement Mag.*, 3:31–35, 2000.

[15] A. Sheth, K. Tejaswi, P. Mehta, C. Parekh, R. Bansal, S. Merchant, T. Singh, U. B. Desai, C. A. Thekkath, and K. Toyama. Senslide: a sensor network based landslide prediction system. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 280–281, New York, NY, USA, 2005. ACM Press.

[16] N. Streitz and P. Nixon. The disappearing computer. *Commun. ACM*, 48(3), 2005.

Proceedings of the Fifth Annual IEEE International Conference
on Pervasive Computing and Communications Workshops(PerComW'07)
0-7695-2788-4/07 $20.00 © 2007 IEEE