Issues in Designing a Compilation Framework for Macroprogrammed Networked Sensor Systems

Animesh Pathak and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California, USA
Email: {animesh, prasanna}@usc.edu

***Abstract*— There is growing interest in the networked sensing community in the technique of *macroprogramming*, where the end-user can design a system using a high level description without worrying about the node-level details. Since the burden of customizing the code to the target architecture is moved to the compiler that translates the high-level description to generate node-level codes, research on the issues involved in compilation of such a program assumes importance. In this paper, we list some issues that need to be resolved by the designers of a compiler for such a macroprogramming framework, including the decisions to be made in the choice of an abstraction, the design of the runtime system and the generating of the code for each node. We discuss some solution techniques that we are currently exploring to solve the above problems.**

I. Introduction

Wireless sensor networks (WSNs) enable low cost, dense monitoring of the physical environment through collaborative computation and communication in a network of autonomous sensor nodes. Slowly but steadily, sensor networks are being deployed in the real world, and the network sizes, the amount of data handled, and the variety of applications is increasing. We shall soon see sensor networks being used is areas ranging from search and rescue [1] to parking garage monitoring [2].

Most of the current research in this area concerns itself with solving system level issues that would enable the maintenance of the infrastructure and perform the task assigned to the system. The main focus in developing applications on sensor networks has been *energy*, since the constituent nodes are battery powered and radio communication is energy-expensive.

Many approaches to programming sensor networks have been proposed. The majority of the current research revolves around **node-level programming**, where the programmer writes the application *bottom-up*, taking into account factors like the effect of the network topology, the radio infrastructure available and the energy capacity of nodes, among others. A sizable body of work exists currently for providing services like localization, time-synchronization, medium access, routing etc. However, the designer of a new application still has to think at the *node-level* as opposed to the *system level* during the design process. As sensor networks are fast moving from the realm of research to real-world applications, the end user is increasingly likely to neither have the expertize nor be willing to spend time learning about issues like energy-efficient topology maintenance. It is increasingly becoming necessary to develop an application development framework (ADF) for sensor networks, using which end-users with little or no networking background (geologists, biologists, petroleum engineers etc.) can achieve fast developement and deployement of wireless sensor networks.

This relatively new approach, which facilitates top-down design of sensor network applications by allowing the programmer to look at the big picture, is called *macroprogramming*. The application designer is only concerned here with providing a high level description of the system, without worrying about the low level issues discussed above. The majority of the work is done by a *compiler*, which takes the above description and generated customized code for each node depending upon the network topology and node capability.

We must mention that there is a fair amount of domain-specifity in macroprogramming techniques and one technique might not be suitable for all networked sensing applications. Also, the use of macroprogramming might be an overkill for simple applications like data-gathering. Our work focusses on active and passive monitoring applications such as structural health monitoring and target tracking.

The **contribution** of this paper is twofold. We first describe in detail the problem of compilation of programs onto sensor networks and the issues that the designer of such a compilation framework will have to face. Secondly, we illustrate our current work towards addressing these issues, namely choosing an abstraction, providing support to it in the runtime system, and the actual task of generating code for each node. We discuss our approach in solving the problems of priority management, task allocation and protocol selection that the designer of the compiler will have to solve.

In the first part of this paper (Section II), we discuss related work in the area of programming sensor networks and the protocols and services available to application designers. We state the basics of the compilation problem in Section III and focus on the issues facing the designer of the compilation framework in Section IV. The current status of our work is discussed in Section V. Section VI concludes with a description of the future directions of our work.

II. Related work

A vast array of protocols have been developed in the recent past for application development on sensor networks. This section lists the most representative of those.

This work is supported in part by the National Science Foundation, USA, under grant number CCF-0430061.

Modern sensor networks constitute of nodes ranging from coin-sized MiCA motes [3] and matchbox sized SunSPOT [4] nodes to much larger Stargates [5]. Depending on the computation and storage capability of the node, the OS support could range from TinyOS [6] or Contiki [7] at the lower end to Linux and WindowsCE at the higher end. The programmer can also utilize virtual machines like the SUN Squawk virtual machine [8] and VM* [9].

Medium access is made possible by one of several MAC protocols specially designed for energy efficiency [10]. Using these lower layer services, protocols for localization [11] and time synchronization [12] have been designed. Solutions for routing on sensor networks include GPSR [13].

The protocols above provide the basic state information and abstractions for **node-centric programming**. At this level of abstraction, the application developer has to translate the global application behavior in terms of local actions on each node, and individually program the sensor nodes using languages such as nesC [14], C/C++, or Java, depending on the node capability, operating system, and compiler support. The programmer can read the values from local sensing interfaces, maintain application level state in the local memory, send messages to other nodes addressed by node ID or location, and process incoming messages from other nodes.

Owing to the difficulties in fast and easy programming of large systems using only node-level techniques, the area of *macroprogramming* is being explored. Regiment [15] is a functional programming language based on Haskell, with support for region-based functions like filtering, aggregation and function-mapping. Kairos [16] is an imperative, control driven macroprogramming language where the application designer can write a single program in a language like Python with some extra keywords to express parallelism. ATaG [17] is a data-driven macroprogramming model that views the sensor network application as set of tasks running on the nodes, interacting via the production and consumption of data items. There are also efforts to make the sensor network's data available to the user in an interface such as SQL [18] or Excel [19].

With the developement of these languages, work on *system level support* for macroprogramming has also started. The TML intermediate language developed by Newton et. al. [20] provides an abstraction of the underlying system (e.g. TinyOS) in the form of a distributed token machine which they are using for compiling macroprograms written in Regiment. The Data Driven ATaG Runtime (DART) proposed by us [21] is a component based runtime system to support programs written in ATaG.

Finally, we would like to note that to the best of our knowledge, there is no publicly available compiler that supports macroprogramming.

III. The Compilation Problem

As is evident in the previous section, the current body of research is only beginning to understand the important of programming these large, possibly heterogeneous systems as a whole. Although compilation has been well known in both traditional computer science and VLSI design communities, its meaning is not clear in the context of sensor networks. We believe that it is relevant to clearly define the problem of compiling programs onto sensor networks in a broad sense for further work in this domain. The definition is as follows:

Starting with a clear idea of the networked sensing application, use a high-level specification of the macroprogram and topology of the network on which the application has to be deployed, and generate a node-specific code for each constituent node in the system such that the resultant system performs the task as specified in the macroprogram.

Note that by *topology* above, we mean details about the placement, mobility, power capacity and computation, sensing and communication abilities of the constituent sensor nodes in the system. It is important that **a)**The programmer is not bothered with these details and **b)** The compilation framework not only keeps in mind the constraints imposed by the specific target topology but also extracts as much performance as possible by appropriately allocating tasks keeping in mind the specifics of the given system.

In practice, it is not enough to simply generate a system that works. There are other desirable properties of such a process, the first one being *ease of use*. The compilation process should also be *fast*, allowing easy re-compilation and re-deployment of code in case of addition of new nodes to the system or a change of system goals. Since our target platform consists of nodes that are not very reliable, *fault tolerance* is a necessary attribute. Another set of desirable properties, namely *energy efficiency* and *energy balance*, arise due to the fact that these systems are mostly run on batteries and the system lifetime is severely affected by the choice of communication techniques.

We would like to clarify that this definition of compilation is drastically different from the one we are used to in course of compiling a C program. For example, the resultant code is responsible for not only the computation but also the communication performed at each node. The other difference is the fact that the process involves the appropriate configuration of the runtime system. The abstraction used should also capture the domain-specific attributes of sensor networks.

The problem can be viewed as one of **placing tasks on a graph**, variants of which have been extensively studied in the operations research, distributed computing and the VLSI design communities. While we believe that a large amount of wisdom can be applied to compiling applications on sensor networks from existing work in the above areas, care must be taken to keep in mind the differences between the systems targeted by existing research in these fields and sensor networks. We discuss some of those issues in the following section.

IV. Issues in Compilation

In this section, we discuss in detail the issues raised in the previous section related to the compilation problem.

A. Abstraction of the System

One of the first challenges in designing a macroprogramming system for sensor networks is the choice of the programming language to describe one's application. An integral part in the choice of the language is played by the abstraction of the

system provided by the compilation framework. For example, the choice of the language for programming a distributed shared memory system will be difference from that for a distributed database. We have discussed some approaches taken by researchers in the community in Section II.

Since sensor networks are essentially composed of a set of wirelessly connected compute nodes, the first impulse is to draw wisdom from the large body of work in the parallel and distributed computing area to solve the problems in this area. However, one must remember the differences between sensor networks and the system traditionally meant to run distributed applications.

The first set of differences lie in the **physical attributes** of the nodes themselves. The constituent nodes in a sensor networks are extremely constrainted. The typical RAM size of a small sensor node (TelosB) is 10KB and the Flash memory is 48KB. The nodes work on batteries and communication is much more costly than computation in terms of energy, a fact almost completely ignored in traditional distributed computing. Since the systems are expected to have lifetimes of months, if not years, energy considerations play a large role in sensor network design. To add to the complexity, there are sensor networks in place that use more than one type of node, some of which might *not* be constrained at all in terms of energy and processing power. The compilation framework should be able to address this heterogenity as well.

The second set of differences lie in way **data is generated and handled** in sensor networks. The typical sensor network application should be **spatially aware**. By spatial awareness we mean that the task should be able to utilize the fact that the data from sensor nodes that are in its close proximity is strongly correlated. Also, processing of data should be performed as close to source as possible in a sensor network to save on energy. Another special property of these applications is **reactive processing**. Since the system processes data that is produced in the network itself, there might be long periods of hibernation for most of the system while nothing "interesting" has been sensed followed by a period of intense activity when, say an intruder walks into the periphery of the system. We strongly believe that any abstraction of sensor networks should necessarily be able to export these two attributes to the programmer. For example, the programmer should be able to mention process-chains such as "when the temperature of a region of at least 10 sq. meters rises above 150 degrees F, raise an alarm".

Another desirable feature to be abstracted out is the fact that the end-user is not interested in the data at a particular *node* but in a particular *region*. A clear limitation of the previous approach comes to the fore when we take into account nodes that can focus only on a certain part of the total area they can sense. For example, asking for a camera on a certain node to point in a particular direction might cause a problem if two tasks give it conflicting instructions. This issue is sorted in a simple manner when the focus is only on the image of the area, where another camera-equipped node can provide the desired data for the second task.

Further, an important question to be answered while deciding on the level of abstraction for sensor systems is "how much work will be off-loaded to the runtime and the compilation framework?". For example, the programming abstraction can model the sensor network as a shared memory where every node has access to all the data in the network. A spatially-aware PRAM model can then be used to program the sensor system. This places an undue amount of pressure on the runtime system which will then have to maintain consistency between the data items in various nodes. On the other hand, an abstraction may expose too many low level details to be of any help to the non-expert end user. An example of this would be making the programmer worry about reliability issues in the network and write programs that will correctly work even if nodes fail. This is akin to making a C programmer responsible for page faults in a system. We believe that such responsibilities should be handed over to the compilation framework and the runtime environment.

Finally, a good abstraction will be *platform-independent* - with the programmer not worrying about the nodes used - but at the same time it should expose *knobs* using which the application designer can tune the performance of the overall system if he/she so chooses.

To cater to the need for a clean abstraction for sensor networks, we have developed a programming language that uses the abstraction of a networked sensor system based on task-graphs [17]. We believe that such a representation is extremely relevant for a networked sensing system and provides the apt amount of abstraction to the end-user so as to neither over- nor under-burden the other components of the macroprogamming framework. Details on our work in this area are discussed in Section V-A.

B. Runtime System Design

The cost of having a powerful programming paradigm is paid for by a non-trivial runtime system that provides the layer of abstraction to the programming model. This is similar to having an operating system to manage issues like memory management as opposed to handling it in the user program.

The design of a runtime system for sensor networks again differs from previous research as the designer has to now take care of the inherent attributes of sensor networks. The issue that is of utmost importance here is that of *energy efficiency*. Another issue in designing such a runtime system is the possible *heterogeneity* of the network. For example, a large sensor system may includes nodes which are communicating wirelessly using protocols like S-MAC, larger nodes using 802.11, and still larger, internet-connected nodes using ethernet. The runtime system design should not need to change much from node to node due to this heterogeneity. The runtime system should also take into account the twin properties of spatial awareness and reactive processing that we discussed earlier.

In addition to possessing these properties, we believe that a good runtime system should be **modular**. This approach has tremendous benefits with regards to both code generation and system improvement. With modular runtimes, the code-generating component of the compiler will generate mostly the same code for each node and will need to work on only a fraction of the code, thus making the code generation

process much faster. Owing to a component based approach, researchers will also be able to focus on aspects of the runtime such as task management separately from other components. Such a design also allows the use of essentially the same runtime system software for functional simulation and the actual deployment, by replacing only a subset of the modules and leaving others intact.

Finally, since we are adding a layer of abstraction to the system, there will be some loss of performance in using a runtime system as compared to using code that is written with network specifics such as the number, type and location of nodes in mind. The runtime system designer thus has to make sure to extract as much performance as possible, since he is already losing out due to the non application-specific nature of the runtime system.

Our initial work in this area has resulted in a template based runtime system [21]. DART, our runtime system, is extremely modular and consists of components which provides *plug and play* support for network protocols. We discuss it in detail in Section V-B.

C. Code Generation

Since the macroprograms are written without taking into account of the actual topology and the node operating system, code generation is an essential part of system design and implementation. *It should be understood that the compilation of macroprograms for networked sensing applications is very different from the kind of compilation performed by a standard compiler like gcc*. We call this step **code generation**, since the result of this process is not binaries, but node-level programs that need to be compiled into binaries by traditional compilers like `gcc` and `javac`. Note that we do not cover the issues in customizing the resulting code into node-specific binaries in our work. Also, although the issues discussed in this section are based on the abstract task graph model, they are also relevant in a broader perspective. A much more in-depth analysis of these issues is presented in Section V-C, since a full understanding of them needs the knowledge of our programming language and runtime system, which are discussed in the next section.

Seen from a high level, the task of compiling an ATaG program is mainly a problem of **task-allocation on a graph**, given a set of constraints on energy etc.. Such problems have been solved extensively in operations-research literature as versions of the *facility location* problem, where a certain number of facilities (police stations) need to be placed in a terrain (city) so as to achieve certain optimization goals (minimizing the maximum latency). The difference in a sensor network context is that energy is a big issue. Further, some mission critical applications (e.g., forest fire alarm system) may value latency more than energy in certain situations. The question to be answered by the compiler then is - how to place the given tasks on these nodes to achieve all these goals of coverage, latency and energy efficiency? As discussed in the following section, the firing rules of tasks further complicates the problem. Another task to be performed during compilation is that of **protocol selection**, where the best protocol for each node is selected. For example, the compiler needs to choose which routing protocol will be used to provide the end-to-end send and receive functionality, and which protocol will be used to maintain any virtual topologies (tree, 2-D mesh) needed by the applications. We show in Section V-C how this reduces to a simple maximization problem and is easily solved. Another problem that will be faced during the design of the compiler is a way of **assigning priorities** to the many tasks running on the node, since many of them will be triggered when a data item is available.

The next section provides insight into our current work towards developing solutions for them.

V. Current Status of Our Work

This section discusses the current status of our work in addressing the major challenges in the design of a macro-programming framework for a networked sensor system. We give a brief summary of our work on the Abstract Task Graph (ATaG) [17], a programming language for specifying the behavior of sensor networks and DART [21], the Data-Driven ATaG Runtime that provides system level support to systems compiled from ATaG programs. Owing to shortage of space, we only provide short summaries of these before discussing our most current ideas, which are in the area of *code-generation* for ATaG programs. The reader is referred to the works referenced above for a more in-depth understanding of ATaG and DART.

A. The Macroprogramming Language

The Abstract Task Graph (ATaG) provides a **mixed declarative-imperative** way of specifying networked sensing applications. The programmer needs to specify abstract tasks that produce or consume abstract data items. Both the tasks and channels connecting them to the data items are annotated to exercise more control over the program. By mixed declarative and imperative programming, we mean the an ATaG program in created in two steps. First the designer specifies the abstract task graph (the declarative part), and the ATaG IDE generates glue code for the final app. The designer then specifies the actual code for each task in a high level language like Java or C, this being the imperative part. These tasks will be fired as and when needed by the description of the task graph. ATaG is **architecture independent**, and the code-generation process takes care of node specific issues. ATaG is also designed to be **composable** - meaning that existing ATaG programs can easily be combined to create more complex programs.

As an example, consider an ATaG program for **temperature monitoring**. Temperature readings from the entire network are to be collected every 30 minutes at a designated *root node*. The temperature gradient between every pair of neighboring nodes is to be monitored every minute, and an alarm notification is to be raised immediately if the gradient exceeds 5 degree Celsius. Figure 1 is a complete ATaG program for this application, which shows the types of tasks (ovals), types of data items (square rectangles) and their I/O dependencies or channels (arrows). The annotations (shaded rectangles) indicate the triggering conditions of tasks, and also where they are to be instantiated. The channels are annotated with the *interest* of each task to a data item.

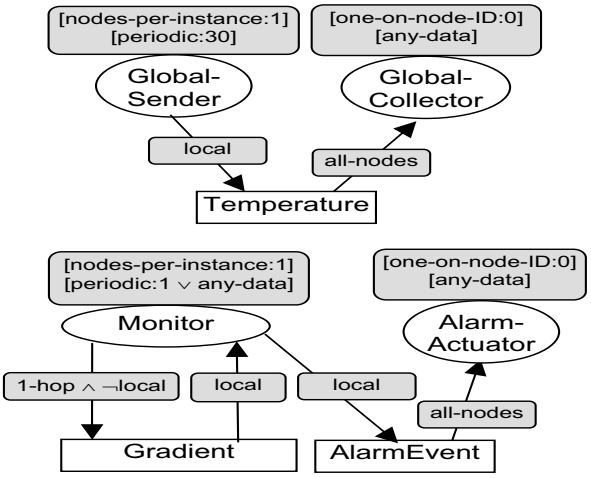

Fig. 1. An ATaG program for temperature monitoring

B. The Runtime Support

Figure 2 is a high level overview of the modular structure of the Data-driven ATaG Runtime(DART). The overall functionality is partitioned into a set of modules; where each module offers a well-defined interface to other modules in the system, and has complete ownership of the data and the protocols required to provide that functionality.

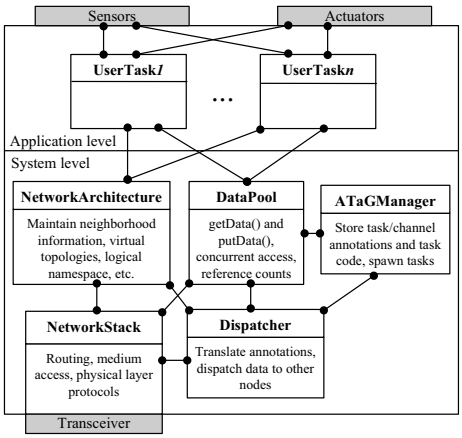

Fig. 2. The structure of the DART runtime system

The *ATaGManager* stores the information from the user-specified ATaG program that is relevant to the particular node. This includes task annotations such as firing rule and I/O dependencies, and the annotations of input and output channels associated with the data items that are produced or consumed by tasks on the node. *Datapool* is responsible for managing all instances of abstract data items that are produced or consumed at the node. *NetworkArchitecture* is responsible for maintaining all information about the real and virtual topology of the network. *NetworkStack* is in charge of communication with other nodes in the network, and manages the routing, medium access, and physical layer protocols. *Dispatcher* is responsible for disseminating data items that are produced on the node to other nodes in the network as specified in the ATaG program. In addition, a *Startup* module

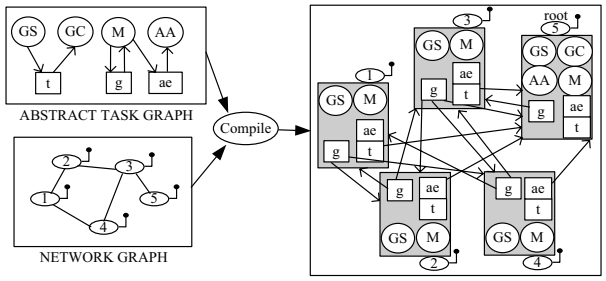

Fig. 3. Compiling an ATaG program for a particular network topology

is responsible for initializing node-level services such as the transceiver functionality, the protocols for topology discovery, etc., and then starting the initial set of application-level tasks. The remainder of the execution is driven by the results of the `getData()` and `putData()` calls made by the tasks, and the data items arriving over the network interface for addition to the data pool.

C. Code Generation and Optimizations

The **input** of the code generation process consists of **a)** The abstract task graph representation of the application and **b)** The network topology. In our proposed work, we shall use the *Generic Modelling Environment* [22] to create the task graph that will be compiled onto the target network.

For each node, the **output** of the compiler will consist of the code for **a)** The modules of the DART runtime system and **b)** The tasks to be instantiated on that particular node. We intend to use Java™ as our output language. The state of the art of the Java based sensor systems is discussed in Section II.

The actual **node-specific code generation** starts with a *syntax check*, followed by *schedulability* for tasks. Once the *task allocation* and the *protocol selection* is done, the DataPool on each node is configured and the code is ready to be compiled using the Java compiler and be loaded on individual nodes. Note that there is no deadlock detection during the compilation process. Figure 3 illustrates the compilation of the ATaG program of Figure 1 onto a network of 5 nodes, where node 5 is designated as the root.

We now discuss in detail our work on the issues faced during this stage of the compilation process.

Protocol Selection: When programs are composed into more complex programs, there might be a situation where a protocol best suited for one may be completely antagonistic to the other's goal. Formally speaking, the general version of this problem is: Given m applications $a_1, a_2, \ldots, a_m$ and n protocols $p_1, p_2, \ldots, p_n$ and a "reward" $r(a, p)$ for each application-protocol pair, find the protocol p that maximizes $\sum_{i=1}^{i=m} r(a_i, p)$. The problem is a maximum finding problem and can be solved in $\theta(mn)$ time. The important problem is assigning appropriate values to the "reward" function.

Further, considering the large memory capacity of our target platform, we may actually be agreeable to having the codes for multiple protocols in each node. The question then changes to one of switching to a certain protocol at certain times. For example, how does one make it so that when a fire breaks out,

all nodes help in routing that information to the base node, but work normally otherwise?

Scheduling and Priority Management:

One of the goals of ATaG is to make sure that a task that is fired runs to completion before another task dependent on a data item produced by it fired. We model the task graph as a tree rooted at the periodic task and assign priorities in a breadth-first manner. This approach works for relatively simple ATaG programs which have only one timer-triggered task, and where all tasks are connected. This approach will not work when ATaG programs are *composed*, by concatenating the code of two ATaG programs. We are working on techniques to model this situation in order to assign priorities for tasks of composed ATaG programs. Another possible issue with such schemes is due to the fact that the number of priorities in JVM is fixed (10 in J2SE, for example). We are working on developing a *dynamic priority assignment scheme* that reuses the "freed-up" higher priorities.

Task Allocation: Since the ATaG program does not place any hard constraints on the placements of individual tasks, it may seem to the novice compiler designer that tasks can be placed arbitrarily. While this is true and will not affect the correctness of the ATaG program, it has the potential of severely hampering the performance of the system. The reason lies in the spatial nature of all computing done on sensor networks. In the following section, we discuss some open issues in the *optimizations* possible during the task placement stage of the compilation process. Although our current work does not focus on *fault tolerance*, we are aware of work on networked sensing algorithms which incorporate fault tolerance [23]. We will utilize the techniques in our future work on compilation.

We believe that one of the important optimization goals while placing tasks is **energy-balance**, i.e. making sure that all the nodes spend nearly the same energy during the applications execution. This constraint leads to a hitherto-unexplored version of the classical *facility location problem*, where the traditional metrics have been latency and coverage. We have some early results on allocating simple tasks on binary trees and work is underway for exploring more complex graphs.

Another level of complexity is added in sensor networks by the fact that the tasks exhibit **complex dependencies** on each other. Some tasks are dependent on more than one task, and can be triggered based on the data generated by one or more tasks, so the representation is not as easy as that of a data-flow diagram. In fact, the triggering rules make it impossible to assign an optimal task placement offline. We are currently exploring ways to capture this behavior of tasks so as to design algorithms for their energy-efficieny placement.

VI. Conclusion and Future Work

In this paper, we defined the compilation problem in the context of programming large networked sensor systems and discussed the issues involved in the same. We already have a proof-of-concept system using the Java virtual machine running on a single machine. We are actively working on getting the first full version of the compiler out. At the conference we expect to present the results from a fully functional compiler which can generate code from simple ATaG programs (using a subset of the full list of primitives) to be deployed on a JVM environment. The compiler will be available for download under the GNU Public Licence.

References

[1] S. M. J.H. Huang, S. Amjad, "Cenwits: A sensor-based loosely coupled search and rescue system using witnesses," in *Proceedings of ACM SenSys 2005, San Diego, CA*, 2005.

[2] J. Liu and F. Zhao, "Towards service-oriented networked embedded computing," Microsoft Research, Tech. Rep., 2005.

[3] M. Horton, D. Culler, K. Pister, J. Hill, R. Szewczyk, and A. Woo, "MICA: The commercialization of microsensor motes," *Sensors Magazine, http://www.sensormag.com/*, April 2002.

[4] Sun™ Small Programmable Object Technology (Sun SPOT), http://research.sun.com/projects/dashboard.php?id=145.

[5] Crossbox StarGate, http://www.xbow.com/Products/XScale.html.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[7] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *1st IEEE Workshop on Embedded Networked Sensors*, 2004.

[8] D. Simon and C. Cifuentes, "The squawk virtual machine: Java™ on the bare metal," in *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM Press, 2005, pp. 150–151.

[9] J. Koshy and R. Pandey, "VM*: Synthesizing scalable runtime environments for sensor networks," in *Proceedings of ACM SenSys 2005, San Diego, CA*, 2005.

[10] P. Naik and K. M. Sivalingam, "A survey of mac protocols for sensor networks," *Wireless sensor networks*, pp. 93–107, 2004.

[11] A. Savvides, C.-C. Han, and M. B. Srivastava, "Dynamic fine-grain localization in ad-hoc networks of sensors," in *Proc. 7th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, 2001.

[12] J. Elson and D. Estrin, "Time synchronization in wireless sensor networks," in *International Parallel and Distributed Processing Symposium (IPDPS), Workshop on Parallel and Distributed Computing Issues in Wireless and Mobile Computing*, April 2001.

[13] B. Karp and H. T. Kung, "GPSR: Greedy perimeter stateless routing for wireless networks," in *Proc. ACM/IEEE MobiCom*, August 2000.

[14] 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.

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

[16] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos," in *Intl. Conf. Distributed Computing in Sensor Systems (DCOSS)*, June 2005.

[17] 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.

[18] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: An acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems (TODS)*, 2005.

[19] K. Whitehouse, F. Zhao, and J. Liu, "Semantic streams: a framework for declarative queries and automatic data interpretation," Microsoft Research, Tech. Rep. MSR-TR-2005-45, April 2005.

[20] R. Newton, Arvind, and M. 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] A. Bakshi, A. Pathak, and V. K. Prasanna, "System-level support for macroprogramming of networked sensing applications," in *Intl. Conf. on Pervasive Systems and Computing (PSC)*, 2005.

[22] The Generic Modeling Environment, http://www.isis.vanderbilt.edu/projects/gme.

[23] M. Singh, "Models and algorithms for distributed computation in wireless sensor systems," Ph.D. dissertation, University of Southern California, 2005.