International Conference on Pervasive Systems and Computing (PSC), June 27-30, 2005.

System-level Support for Macroprogramming of Networked Sensing Applications

Amol Bakshi, Animesh Pathak, and Viktor K. Prasanna
Department of Electrical Engineering,
University of Southern California,
Los Angeles, CA 90089-2562 USA
Email: {amol, animesh, prasanna}@usc.edu

Abstract

Wireless sensor networks (WSNs) provide low-cost, embedded sense-and-respond capability, and are therefore an integral part of the vision of pervasive computing. Most research on WSNs to date has focused on the development of efficient protocols for infrastructure establishment. Application development for WSNs is still very daunting for the non-expert. This paper proposes hierarchical layers of abstractions to categorize the functionality of various WSN protocols from a programming perspective. We then address the issue of system-level support for programming models in sensor networks, and describe the design of the DART runtime system, which supports a macroprogramming model called the Abstract Task Graph. DART prototypes for two different target platforms have been implemented. The modular structure of the runtime is inspired by our proposed layers of programming abstraction. The design of DART simplifies software synthesis of ATaG programs, and enables near plug-and-play integration of different protocols and services at the lower layers with minimal impact on the higher layers.

Keywords: Sensor networks, programming model, data driven computing, software architecture

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. Most of WSN research to date has focused on system-level concerns in establishing the infrastructure required for enabling context-aware processing in WSNs. A host of protocols have been proposed for localization, time synchronization, routing, in-network storage, etc., with a view to extending network lifetime through efficient use of the limited and, in many cases non-replenishable, energy resources available to a node [1]. WSNs provide embedded sense-and-respond capability and context awareness and are therefore an integral part of the vision of pervasive computing.

Currently, sensor networks are programmed using node-centric methodologies. The programmer is expected to manually translate a given global behavior specification into a set of local actions for each node in the system. A separate program is then written for each node. This program includes not just the application level functionality, but also all the details of system level control and coordination - including the management of infrastructure protocols and services that are not directly relevant to the application level. Programming a sensor network using this approach requires expertise in the application domain, distributed computing, and wireless networking. Defining methodologies to ease application development for large scale networked sensing is the next step in WSN research.

Most programming models and frameworks from traditional distributed computing are not directly applicable to WSNs. New programming models, representations, compilers, and software synthesis frameworks are required for two main reasons. First, the nature of applications is significantly different. Actions of the multiple components in the system have to be coordinated to accomplish a single objective. There are multiple data sources in the network, the spatio-temporal origin of data dictates the operations performed on it, data items could be prioritized, and some data items expire if not consumed within a specific period. In addition, data sources can be added or removed, and existing sources could be mobile. It is also desirable to process data in-network, and as close to its origin as possible for sake of energy efficiency. Finally, applications are expressed in phenomenon-centric terms, e.g., “trigger an alarm if the temperature gradient in any portion of the terrain exceeds 5 degrees/m”.

Second, the characteristics of the network deployment are very different compared to traditional parallel and distributed systems. The system can be heterogeneous with some nodes connected to the Internet through wired links, others using high bandwidth, reliable wireless communication infrastructures, and the edge of the network comprised of resource constrained nodes with low computation, communication, and storage capacities. The network topology could be dynamic as sensor nodes fail and/or new sensor nodes are added. Different components of the system may run different operating systems and have different sensing interfaces, but are still expected to be abstracted in essentially the same manner – as a distributed substrate for sensing, computation, and actuation.

The non-expert end user will likely be uninterested in the system-level infrastructure issues of networked sensing. As long as a convenient and easy to use mechanism is available

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

to concisely express the overall sense-and-respond behavior, the end user will not care how it is translated into node-level behavior, what the capabilities of each node are, what operating systems and language support is available, how failures are managed, which routing protocol is used, etc. Most of the existing research in the system level aspects of WSNs has taken a far from holistic view of the concerns of the end user, instead modeling only on those application characteristics that are directly relevant to the problem being solved. As a result, application-level interfaces to many of the existing protocols for sensor networks are ill-defined and incompatible with other protocols. There is also no end-to-end methodology which will allow the developer of a programming framework to use many of these research results in a practical context.

The role of an *application development framework (ADF)* for WSNs is twofold. The first is to package and abstract the system-level protocols and services and export them in a form that is understandable to the end user. The second is to translate the application behavior specified by the user of the framework (programmer) into a form that is understandable to the low level system components - such as configuration files, function parameters, data structures, etc. An ADF consists of: (i) a programming model (abstract syntax and semantics) for expressing the collaborative sensing, actuation, computation, and communication behaviors in the system, (ii) a program representation (concrete syntax) that is used to input the description, (iii) a template of the runtime system that will actually manage the various tasks on a node and the interactions between the nodes, and (iv) a compiler that analyzes the high level application representation and accordingly configures the runtime system template on each node of the target network. If the ADF is well designed, developers of system-level infrastructure services such as networking protocols, communication libraries, middleware, etc., can focus only on providing the interface mandated by the ADF and not worry about how the services will interact with other components of the system. Also, programming models and languages can be designed based on the assumption that all the underlying complexity will be available as a set of well-defined interfaces managed by some underlying runtime system.

In the first part of this paper (Section II), we propose a categorization of different protocols and services for WSNs into a set of *layers of abstraction* from the programming perspective. The subsequent sections deal with system-level support for application development using a macroprogramming model called the Abstract Task Graph (ATaG). We use 'system-level support' to refer to the underlying software infrastructure that manages the control and coordination in the network, and supports the high-level programming model used for application development. Specifically, we present the design of the ATaG runtime system (DART). Section III briefly describes the *key concepts of the ATaG programming model* and uses a simple example to illustrate how an ATaG program is compiled into node-level behaviors for a particular network deployment. Section IV discusses the *design of DART* and describes the functionality of each of its components in detail. We conclude in Section V by discussing the *implementation status* of DART.

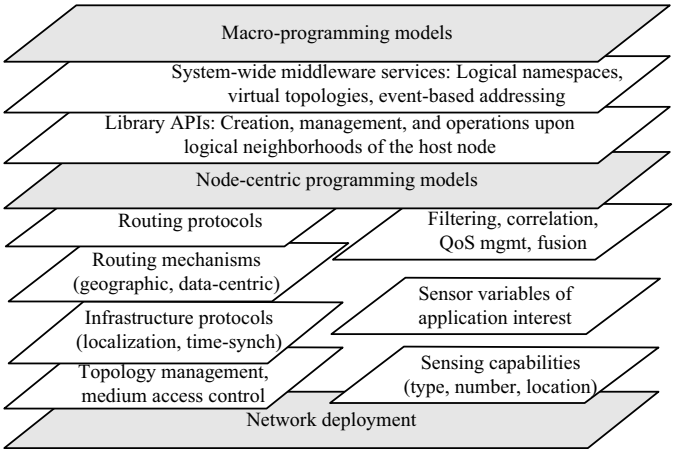

Fig. 1. Layers of abstraction for application development on WSNs

II. A Layered Approach to Sensor Network Programming

WSN research in the past few years has led to an in-depth understanding of the issues in hardware design of sensor nodes and for infrastructure establishment in ad-hoc, unreliable, and resource-constrained deployments. There is now an increasing interest in designing higher level services and abstractions for sensor networks with the objective of making the computing substrate accessible to the non-expert. In this section, we take a holistic view of the functionalities offered by different protocols and services for WSNs from the non-expert programmers' perspective. Figure 1 depicts our proposed categorization of these functionalities.

The bottom layer is the network of possibly heterogeneous sensor nodes. Each node is assumed to support at least one operating system and a native compiler. Depending on the computation and storage capability of the node, the OS support could range from TinyOS [2], μC/OS-II [3], or Contiki [4] at the lower end to Linux and WindowsCE at the higher end.

The set of infrastructure protocols can be roughly classified into two mostly orthogonal stacks - the *sensing stack* and the *processing stack* - as depicted in the figure. The sensing stack is concerned with the interpretation of readings at one or more sensing interfaces in terms of the physical phenomena they represent. The objective of protocols in the sensing stack is to perform calibration of sensors, detect possible sensor malfunctions, and to abstract the physical sensing interface in terms of variables that are interesting and meaningful to the application. For example, a sensor network application for personal health monitoring could be interested in knowing the stress level of the person [5]. This application-level variable could be calculated by protocols in the sensing stack using information from a variety of sensors for, say, the pulse rate, blood pressure, blood oxygen level, etc. A further set of services could be defined to exploit spatio-temporal correlation of sensor readings to compress the event information, perform application-directed filtering of sensor data, provide quality of service (QoS) guarantees, etc.

The processing stack comprises of the protocols that provide the infrastructure for distributed computation and communica-

tion of raw and aggregated sensor readings in the network. The lower layers of the processing stack correspond to the basic networking protocols at the physical and medium access layers, which are sufficient to provide the basic support for topology management in the network. These protocols provide the capability of sending messages between neighboring nodes. Protocols for localization [6] and time synchronization [7] use this basic communication capability to provide context awareness to the individual sensor node in terms of its spatio-temporal location in the network. Localization is especially important in sensor networks because the mere knowledge of an event's occurrence is rarely useful without knowing its physical location and time of occurrence. Also, a majority of other node-level services are based on the assumption that the node knows its own position in a real or virtual coordinate system imposed on the deployment. Localization information also supports the geographic routing mechanism, which is the basis for most of the common routing protocols [8] for various communication patterns in a WSN.

The protocols in the sensing and processing stacks 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 [9], galsC [10], 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. While node-centric programming allows manual cross-layer optimizations and thereby leads to efficient implementations, the required expertise and effort makes this approach insufficient for developing sophisticated application behaviors for large-scale sensor networks.

The concept of a logical neighborhood – defined in terms of distance, hops, or other attributes – is common in node-centric programming. Common operations upon the logical neighborhood include gathering data from all neighbors, disseminating data to all neighbors, applying a computational transform to specific values stored in the neighbors, etc. The usefulness and ubiquity of neighborhood creation and maintenance has motivated the design of *node-level libraries* [11], [12] that handle the low level details of control and coordination and provide a neighborhood API to the programmer.

The next layer of abstraction is provided by a range of *middleware* services [5], [13], [14]. We classify as middleware those protocols that provide system-wide, high level services and phenomenon-centric abstractions. Middleware services could create virtual topologies such as meshes and trees in the network, allow the program to address other nodes in terms of logical, dynamic relationships such as leader-follower or parent-child, support state-centric programming models [15], etc. The middleware protocols themselves will typically be implemented using node-centric programming models, and could possibly but not necessarily use communication libraries as part of their implementation.

At the next higher level of abstraction, and the focus of much recent research interest [16], [17] are *macro-programming models* and languages for sensor networks that specify aggregate behaviors and are automatically compiled into node-centric programs for the individual node. Macro-programming languages are typically supported by an underlying runtime system that manages control and coordination at the node level. The structure of the runtime system will depend on the structure and semantics of the macroprogramming model. Ultimately, an even higher level of abstraction will be defined and individual applications – expressed as macroprograms or node-centric programs – will be modeled as services. The end user's interaction with a networked sensor system is likely to be through a purely declarative domain-specific interface that will be compiled into macroprograms (or node-centric programs) based on the desired set of services [18].

In the following sections, we describe the design of the DART runtime system. DART provides system-level support for macroprogramming with the ATaG model, and has a modular architecture that is designed to accomodate protocols and services at the various layers of abstraction described in the preceding paragraphs, while simultaneously facilitate plug-and-play integration of different functionalities.

III. Macroprogramming with the Abstract Task Graph

The Abstract Task Graph (ATaG) [19] is a data driven *macroprogramming* model for architecture-independent development of networked sensing applications. Architecture independence allows development of the application to proceed prior to decisions being made about the final configuration of the nodes and network, and also allows the same application to be automatically synthesized for different network deployments.

The term macroprogramming broadly refers to programming of sensor networks as a whole as opposed to configuring individual node behaviors [16], [17]. For ATaG, we define (and support) two types of macroprogramming. Macro-ness at the application level means that the programmer can define and manipulate information at the desired level of abstraction without worrying about how the information is created. At the architecture level, ATaG allows concise specification of common patterns of in-network distributed processing such as neighbor-to-neighbor, tree-based, etc.

We now briefly present the key concepts of ATaG, provide an example to illustrate how global application behavior is concisely expressed in an ATaG program, and then discuss the compilation of an ATaG program for a given network deployment. The ATaG model itself is not the focus of this paper, but a brief overview of ATaG programming and compilation is necessary to place in context the design of its underlying runtime system.

A. Key concepts

ATaG employs a *data driven programming model* and *mixed imperative-declarative program specification* for separation of concerns. Tasks are defined in terms of their input and output data objects. An underlying runtime system manages

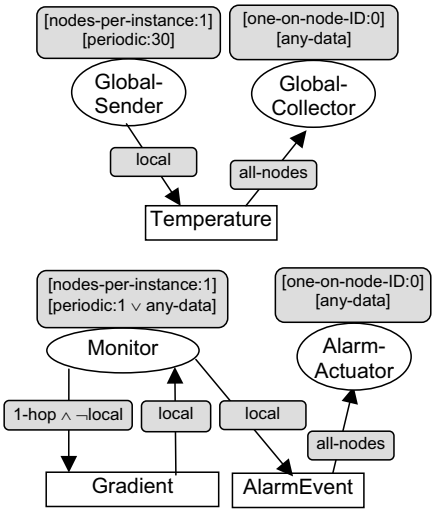

Fig. 2. An ATaG program for temperature monitoring

task scheduling and inter-task communication. Availability of operands triggers task execution, subject to firing rules. This model is attractive for computing in distributed systems for programming convenience, and the modularity and extensibility of the programs [20]. Also, a sensor network can be viewed as a system for domain-specific transformation of sensor data and many applications can be naturally expressed as a set of transformations on raw and processed sensor readings.

The mixed imperative-declarative specification separates the 'when and where' of processing from the 'what'. The same program can be compiled for a different network size and topology by interpreting the declarative ('when and where') part in the context of that network architecture, while the imperative ('what') part remains unchanged. The ATaG programmer, who writes only the task implementations, is free to focus on application-level design without being concerned about low level details of the sensor node platform and the specifics of a particular deployment.

B. Illustrative example

To help the reader get a high level understanding of the structure and expressiveness of an ATaG program, this section presents a complete ATaG program for a very simple temperature monitoring application. Much more complex behaviors such as object tracking and hierarchical data aggregation can be modeled with ATaG, and the simplistic nature of this example is meant to quickly illustrate key concepts and not as a reflection on the limits of ATaG's expressiveness.

Consider a network of sensor nodes, each equipped with one temperature sensor. 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 2 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). In addition, each task and channel is annotated (shaded rectangles). The annotations indicate that *Global-Sender* runs with a period of 30 minutes and reads the current temperature whenever executed. The *Monitor* runs on each node with a period of 1 minute, samples and transmits the current temperature reading to its neighboring nodes, and calculates the gradients when readings from neighboring nodes are received. *Global-Collector* runs on a designated root node, is executed whenever a reading from a Global-Sender is received and displays it on the screen. The *Alarm-Actuator* also runs on the root node and executes whenever an alarm notification sent by a Monitor task is received.

The annotations determine how many instances of these tasks are created in a given network deployment, where they are placed, when they are invoked, and how the data objects are to be communicated between task instances on the same node and across nodes. Task annotations shown in the figure relate to density of instantiation and firing rules. For example, Global-Sender and Monitor are to be instantiated on every node in the network, while Global-Collector and Alarm-Actuator are to be created only on a single node. The firing rule for Monitor is 'periodic∨any-data', which means that the task should be scheduled for execution when either (i) a reading from one of the neighbors is received, or (ii) the periodic timer expires. We now illustrate how this program is compiled onto a specific architecture, and then discuss in detail the design of the runtime system on each node that is responsible for the actual control and coordination.

C. Compilation of an ATaG program

Compiling an ATaG program onto a specified network means translating the annotations in the context of that specific deployment, and generating a 'configuration' for each node of the target network. The configuration of a node consists of:

- The set of tasks assigned to that node
- The firing rule of each of the assigned tasks
- The set of data items that are expected to be added to the node's data pool, either produced by a locally hosted task, or sent by another node
- The set of destination nodes for each data item produced on that node. This set of destinations could be a list of node IDs known a priori if the network topology is expected to be static, or the untranslated annotations themselves, if the translation is expected to be done on demand in the runtime system for a dynamic topology.

Figure 3 illustrates the compilation of the ATaG program of Figure 2 onto a network of 5 nodes, where node 5 is designated as the root. The task and data names are abbreviated in the figure, but the program corresponds exactly to the illustrative example discussed earlier. The set of configurations output by the compiler is shown. Arrows from a data item to one or more nodes denote the final destination of the data item, and not the exact route that is followed by that data item in a multi-hop routing scenario. To reduce the visual clutter, the data items AlarmEvent (ae) and Temperature (t) are clubbed together since the destination of both items is always the designated root node.

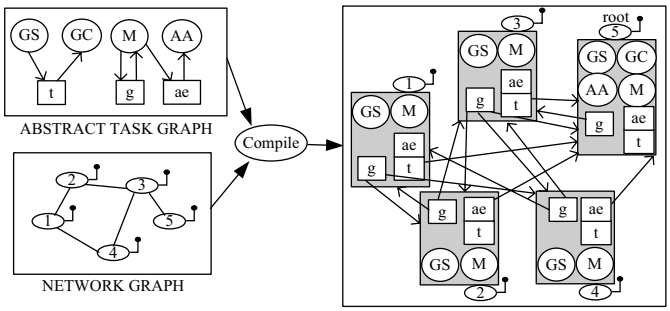

Fig. 3. Compiling an ATaG program for a particular network topology

In the following sections, we describe the design of the runtime system that interprets this configuration information, and manages the application execution in the real system.

IV. System-level support for data driven macro-programming

As mentioned in the introduction, the job of the developer of an application development framework is: (i) to understand the essential characteristics of the target class of applications in order to define suitable programming abstractions, and (ii) to understand the details of the underlying network architecture and protocols in order to synthesize a correct and efficient distributed program for the target deployment. Even if the application developer uses a macro-programming model and language, the application behavior has to be ultimately translated by the compiler into a set of node-level programs, one for each node in the network. Macro-programming does not eliminate this requirement - it merely transfers the responsibility from the programmer to the compiler. This reduces the cost of application development and, in the case of ATaG, provides other benefits such as portability and reusability of application-level code.

For a macro-programming model such as ATaG, the design of the underlying runtime system is critical for two reasons. First, a well designed runtime system can *simplify the compilation and code generation* process. As discussed in the following subsections, the data-driven ATaG runtime (DART) clearly separates the application-independent mechanisms for control and coordination from the application-specific configuration information that represents the role of that particular node in the overall system. Also, the application-specific configuration is localized within a small fraction of the overall software architecture, so that the other components of the runtime system can be integrated without modification into the final executable for that node. Second, and perhaps equally important, a well designed runtime system can allow a *plug and play integration* of the various protocols and services discussed in Section II.

A. DART: The Data Driven ATaG Runtime

Figure IV is a high level overview of the modular structure of the data driven ATaG runtime called 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. This modular structure has many advantages. First, it greatly simplifies the design by reducing interactions and dependencies among modules. Second, the implementation of a module is hidden from other modules; which means that an entirely different set of protocols can be used to provide the same functionality without affect the rest of the system. This allows for tailoring the runtime system to various deployment scenarios by selecting the suitable protocols based on the hardware and network characteristics without requiring a complete redesign. Also, as discussed in greater detail later in this section, it allows us to use 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.

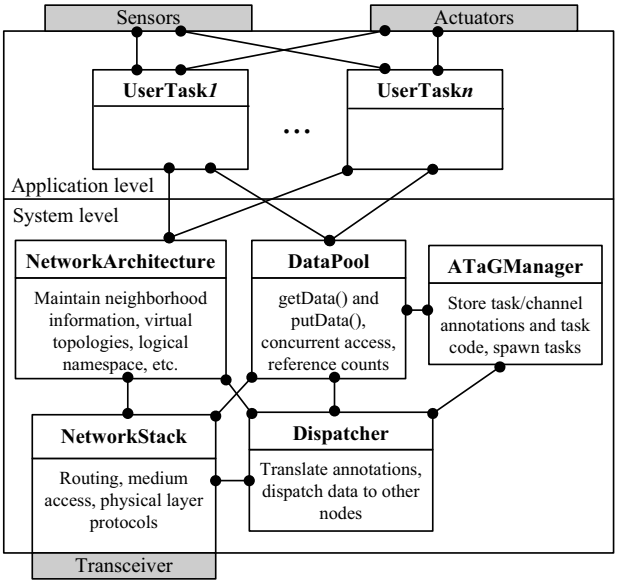

Fig. 4. The structure of the DART runtime system

Figure 4 is an overview of the DART design. We now describe each of its components in detail.

NetworkStack: This module is responsible for managing the network interface of the sensor node. The network stack provides an asynchronous `send()` and `receive()` interface. It implements *active messages* [21], i.e., tasks register their interest in specific message types which are accordingly handed to them by the receiver as and when they arrive. The network stack effectively represents the routing, medium access, and physical layer protocols of the WSN. When a functional simulation of the entire system is to be performed, the sending and receiving of messages can be performed through sockets on a single host machine. The rest of the modules remain unaffected.

Many optimizations are possible in this module. For instance, a protocol such as S-MAC [22] could be used for medium access. This protocol implements a sleep/wake schedule for the transceiver of each node, with the aim of reducing overall energy consumption. During the wake cycle, a node transmits all the outgoing data it has buffered during the sleep cycle. Since the `send()` is asynchronous, the NetworkStack

is free to buffer and consolidate the outgoing data items while the transceiver is asleep, and transmit them in a batch mode when it is awake. The MAC protocol can even be replaced by an entirely different protocol to optimize other performance metrics, without affecting the rest of the system.

NetworkArchitecture: This module maintains the topology related information for the sensor node. It runs the topology construction and maintenance protocols and provides a logical neighborhood abstraction to the other modules. It is responsible for translating channel annotations such as 'all nodes within 5 m' into a list of node IDs. The neighborhood information is made available to user level tasks to provide context awareness. For each task that is mapped onto the node, the compiler determines the union of the neighborhoods specified for each of the input channels, and passes that to the NetworkArchitecture module. The intuition is that if a task requires a particular data item from, say, all nodes within 4 hops, it is likely to require information about how many such nodes actually exist, their IDs, and their locations and nothing more.

ATaGManager: This module maintains the entire ATaG representation, including the task code, the task and channel annotations, and the I/O dependencies between tasks and data items. When new data items are added to the data pool, this module is responsible for determining which of the assigned tasks are ready to run, based on their firing rule and the availability of other data items that could be input to the tasks. Storing the channel annotations as part of the runtime system is especially important because it allows for dynamic, on demand disambiguation of the annotations. This allows the runtime system to adapt to a changing network topology while still preserving the high level intent of the ATaG program.

DataPool: This module is responsible for maintaining the local data pool and implementing the `getData()` and `putData()` function calls. Data pool management involves handling concurrent accesses by more than one user level or system level task, maintaining reference counts for each instance of a data item in order to determine if a particular instance is active (i.e., still waiting to be consumed by one or more tasks that are scheduled for execution) or inactive (i.e., it can be overwritten when a new instance of the same type of data item is produced by a local task or received by the NetworkStack from another node). The `getData()` function returns a copy of the requested data item to the caller and decrements the reference count of the associated item by one. `putData()` adds an instance of a particular abstract data item to the data pool, unless the existing instance is active, in which case it returns without changing the data pool. When a new data item is added to the data pool, the ATaG manager is notified, resulting in the possible scheduling of one or more tasks for execution The Dispatcher module is also notified, in case the data item is to be disseminated to other nodes.

Dispatcher: The task of this module is to handle the dissemination of data items produced on the node to other parts of the network. The Dispatcher supports a `notify()` function that is invoked by the DataPool whenever a new data item is produced locally. The Dispatcher then contacts the ATaGManager and determines the annotations of the output channel associated with that data item. The annotations have to be translated into a list of node IDs, based on the current state of the network topology, which is the domain of the NetworkArchitecture module. When the translation is performed, the Dispatcher then hands off the data item with the list of destination IDs to the NetworkStack.

UserTask: This represents an application-level task, which is an instance of the 'abstract task' in the ATaG model. There is one instance of the UserTask module per abstract task assigned to the node. The programmer (and hence the UserTask) has access to the `getData()` and `putData()` interfaces of the DataPool, and the interface to the NetworkArchitecture that is responsible for translating a logical neighborhood into a list of node IDs (or locations) that constitute the neighborhood at the time of invocation. The UserTask also has access to the sensor and actuator interfaces of the node, which are not explicitly modeled in the current version of DART.

B. Control flow

The control flow can be divided into two parts. The first part is the set of activities that occur at node *initialization*. The second part is the actions that are triggered during the course of *application execution* on that node.

Each module of DART is expected to implement a `start()` function that performs the basic initialization (if any) required for that module. The initialization might involve memory allocation, initialization of variables, spawning of new threads for different protocols and services, etc. As mentioned earlier in this section, the *Startup* module is the first to run when the node is turned on, and invokes the `start()` functions of the other modules in the following order. First, the Datapool is started, which mainly involves allocating memory for each entry of the data pool corresponding to the different data items in the ATAG, and then marking the entries of the datapool as empty by suitably initializing the reference counts. Next, the NetworkStack is started, which spawns the listener thread to accept incoming connections, and a transmitter thread to handle outgoing messages. The initialization, if any, needed by the MAC and routing protocols, and also the localization and time synchronization protocols, is performed before the `start()` of the NetworkStack returns control to Startup. Next, the NetworkArchitecture module is started. Since the NetworkStack is already initialized and the send/receive capability is available, NetworkArchitecture can spawn the protocols required for neighbor discovery, virtual topology construction, middleware services, etc. The startup of this module will be deemed complete when some minimum node state has accumulated; e.g., all the information about the neighborhood is available. Finally, the ATaGManager is started. This module traverses the list of user-level tasks assigned to that node, and spawns all the tasks that are marked 'run at initialization' by the programmer. These will typically be the tasks that (periodically) produce the set of sensor readings that will then drive the rest of in-network processing.

During the normal course of application execution, there are three main events that can occur: (i) a `getData()`

invocation by a user task, (ii) a `putData()` invocation by a user task, or (iii) a `putData()` invocation by the receiver thread when a data item arrives from another node. When a `getData()` invocation occurs, the DataPool merely decrements the reference count of the data item in question. When a local task invokes a `putData()`, the DataPool first checks if the corresponding data item is inactive before adding the newly produced data instance to the pool. If the addition is successful, the DataPool informs ATaGManager about the production of the data. ATaGManager determines the list of tasks that depend on this data item, checks their firing rules, and schedules the eligible tasks for execution. The DataPool then notifies the Dispatcher and finally returns control to the user task. The Dispatcher interacts with ATaGManager, NetworkArchitecture, and NetworkStack to send the data item to other nodes as indicated by the ATaG program. When the third type of event - an invocation of `putData()` by the receiver thread of the NetworkStack - occurs, it is handled in much the same way as a local invocation, except that the Dispatcher is not part of the loop.

V. Concluding Remarks

The design of DART and the control flow described above can be implemented on any operating system kernel that provides: (a) support for multithreaded execution, (b) a preemptive, priority-based scheduler, and (c) mutual exclusion semaphores, message queues, and similar mechanisms to handle concurrent accesses to critical sections, and to coordinate interactions between different threads. Most traditional operating system kernels provide these facilities. A prototype version of DART has been implemented in Java, and is designed to run on relatively heavy duty sensor nodes, although Java Virtual Machines for resource-constrained architectures are also available [23]. DART is also being implemented on the μC/OS-II real-time OS kernel [3] which has been ported to a vast number of devices. A proof of concept programming and synthesis environment based on ATaG and DART has also been designed.

DART reflects the categorization of concerns in Section II in the sense that there is a clear division of functionality among the different modules in the system, and the implementation of each module can be completely altered without affecting other components, as long as the interface remains the same. Protocols and services in various layers that were shown in Figure 1 can be designed in the context of this architecture template. For instance, the designer of a new routing protocol does not need to worry about its interaction with other middleware services, routing protocols, or the application level. The new protocol can be plugged into the NetworkStack module of DART, which can select it to send a subset of the data items produced on that node, based on performance-related annotations specified in the ATaG program. The performance of DART is likely to compare unfavorably with hand-optimized runtime systems where the different functionalities are tightly integrated into an inflexible, monolithic structure and many cross-layer optimizations are incorporated into the design. However, the tradeoff between usability and flexibility on one hand, and hand-optimized performance on the other is common in all methodologies that seek to automate the design of complex systems. A greater level of experience with implementing different applications on a real DART-based system will guide our future design choices for the ATaG runtime.

References

[1] D. Ganesan, A. Cerpa, Y. Yu, W. Ye, J. Zhao, and D. Estrin, "Networking issues in sensor networks," *Journal of Parallel and Distributed Computing (JPDC), Special issue on Frontiers in Distributed Sensor Networks*, 2004.

[2] 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.

[3] uC/OS-II RTOS, http://www.ucos-ii.com/,.

[4] 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.

[5] W. Heinzelman, A. Murphy, H. Carvalho, and M. Perillo, "Middleware to support sensor network applications," *IEEE Network*, January 2004.

[6] A. Savvides, C.-C. Han, and M. B. Srivastava, "Dynamic fine-grain localization in ad-hoc networks of sensors," in *Proc. 7th Intl. Conf. on Mobile Computing and Networking*, 2001.

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

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

[9] 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.

[10] E. Cheong and J. Liu, "galsC: A language for event-driven embedded systems," in *Proc. Design, Automation and Test in Europe*, 2005.

[11] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *2nd Intl. Conf. on Mobile systems, applications, and services*, 2004.

[12] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[13] T. Liu and M. Martonosi, "Impala: A middleware system for managing autonomic, parallel sensor systems," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[14] Y. Yu, B. Krishnamachari, and V. K. Prasanna, "Issues in designing middleware for wireless sensor networks," *IEEE Network*, 18(1), 2004.

[15] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao, "State-centric programming for sensor-actuator network systems," *IEEE Pervasive Computing*, 2003.

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

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

[18] 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.

[19] 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.

[20] V. D. Tran, L. Hluchy, and G. T. Nguyen, "Data driven graph: A parallel program model for scheduling," in *Proc. 12th Intl. Workshop on languages and Compilers for Parallel Computing*, 1999.

[21] T. Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: A mechanism for integrated communication and computation," in *19th Intl. Symposium on Computer Architecture*, 1992.

[22] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks," USC/ISI, Tech. Rep. ISI-TR-543, 2001.

[23] Real Time Specification for Java, http://www.rtj.org/,.