Enabling Scope-Based Interactions in Sensor Network Macroprogramming

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

[‡]Politecnico di Milano, Italy, mottola@elet.polimi.it

[†]University of Southern California, USA, {animesh, amol, prasanna}@usc.edu

[#]University of Trento, Italy, picco@dit.unitn.it

Abstract

Wireless sensor networks are increasingly employed to develop sophisticated applications where heterogeneous *nodes are deployed, and* multiple parallel activities *must be performed. Therefore, application developers require the ability to partition the system based on the node characteristics, and specify complex* interactions *among different partitions.*

Existing programming abstractions for sensor networks tackled this problem by providing a notion of scoping*. However, this rarely emerges as a first-class programming construct, hence limiting its applicability. To address this issue, in this paper we present a flexible notion of scoping in the context of a sensor network* macroprogramming *framework. Our approach enables the specification of* complex interactions *among system partitions, thus greatly simplifying the development process. Moreover, this is not detriment to performance: our approach results reasonably close to an optimal solution computed with global system knowledge, while exhibiting a 70% gain w.r.t. baseline solutions.*

1. Introduction

Early deployments of wireless sensor networks (WSNs) focused on a single, system-wide goal, and featured fairly simple architectures. For instance, habitat monitoring [7] can be implemented using mostly *homogeneous* nodes, each running the *same* application code. In these scenarios, developers are required to describe simple patterns of interactions, e.g., that of sensing and reporting a physical reading.

Recent technological advances and the consequent advent of more powerful sensor nodes are, however, enabling the use of WSNs in increasingly sophisticated settings, from smart spaces [19] to monitoring and control in buildings [4]. These applications often involve *heterogeneous* nodes [3] equipped with actuators to influence the environment, and

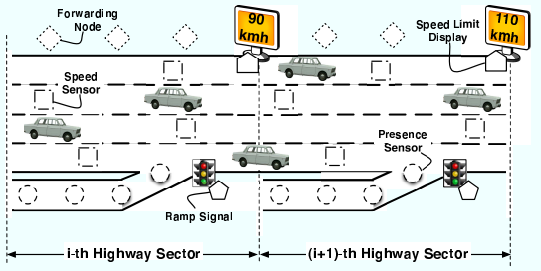


Figure 1. Traffic management scenario.

their ultimate goal is usually obtained by composing *multiple, collaborating activities*.

Reference Scenario. Consider, for instance, a *road traffic monitoring and control* application, a field where WSNs have gained increasing attention [9]. Various techniques exist to influence the vehicle movements (e.g., to minimize pollution and fuel consumption), that use solutions such as speed signaling and ramp metering. These systems are often logically divided into disjoint *sectors* [13], with each sector usually being controlled depending on the current status of the *same* and *neighboring* sectors.

A sample highway scenario is depicted in Figure 1, where a sector is identified by a single ramp leading to the highway, i.e., it spans the portion of highway from a ramp to the following. The system has five main components: *i)* **speed sensors** installed on the highway lanes to measure and report the speeds of vehicles, *ii)* **presence sensors** installed on the highway ramps to report the presence of vehicles, *iii)* **speed limit displays** installed one per highway sector to inform of the recommended speed limit, *iv)* **ramp signals** installed one per highway ramp to allow or disallow cars onto the highway, and *v)* **forwarding nodes** installed on the road side at regular intervals to enable wireless communication between the various nodes.

Figure 2 illustrates, from a high-level perspective, the various stages of data processing in the application. Data is collected from the sensing devices and processed to derive

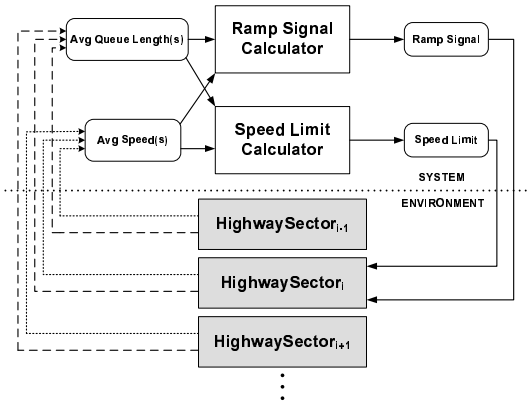


Figure 2. Data processing in traffic management.

aggregate measures —the average speed of vehicles in a highway sector or the average queue length on a ramp. This information is fed as input to an algorithm determining the *actions* to achieve the system objectives, e.g., maximize the flow of vehicles on the highway. These actions are then communicated to the ramp signals and speed limit displays. The specific algorithms employed depend on the goals and metrics of interests.

Need for Scoping. As illustrated in Figure 2, multiple concurrent activities must be performed to achieve the overall application goal, in our case, regulating the vehicles speed and access to the highway. Each of these activities can be decomposed in several, inter-dependent steps where the outputs of one step are fed as input to the following one. Since nodes have different capabilities, each such step must be ultimately mapped to a different system partition that includes only nodes with specific characteristics. As a result, each processing step can be regarded as mapping the inputs obtained from a specific *subset of nodes* to a *different subset of nodes*. Therefore, the programmer must not only identify the different *scopes* based on the application requirements, but, more importantly, express non-trivial *interactions* among them.

Scoping in the State of the Art. Most of the existing WSN programming frameworks provide little or no support for scoping. For instance, Hood [23] is a node-level programming model that provides the ability to identify subsets of nodes in a physical neighborhood using application-defined filters. The interactions are, however, limited to 1-hop communication and a many-to-one pattern. These features are insufficient for the applications we target, where the nodes in a scope may not be in range of each other. Abstract Regions [22] is another node-level solution where a subset of nodes share data using a tuple space-like communication model. However, the definition of the membership of nodes in a region is hard-wired in the region run-time layer. Therefore, a different implementation is needed for each possible region the developer may require. This poses a considerable burden on the programmer.

In [6], the authors propose a language and algorithms to support generic role assignment in WSNs. In a sense, they also identify subsets of nodes by *imposing* certain roles to nodes so that some specified requirements are met. Besides this, however, there is no support to express interactions among subsets of nodes based on the roles assigned. In EnviroSuite [11], contexts are defined with conditional statements to create a mapping between software objects and real-world elements, e.g., a moving target. Contexts determine a scope including a set of physically connected nodes with no intermediate hops outside the partition. Albeit sufficient for applications exhibiting spatial locality, such notion cannot be used to address, e.g., a set of geographically sparse actuators, as in our reference application.

Differently from traditional node-level programming, in *macroprogramming* developers reason at a high level of abstraction focusing on the system as a whole instead of single nodes. A dedicated *compiler* takes care of translating the high-level specification to node-level code. As example, TinyDB [12] offers a database interface to WSNs where users submit queries specified with a dialect of SQL. A notion of query scoping is present whereby queries are not delivered to nodes that cannot provide useful data. However, this does not emerge at the programming level, as the span of a query is ultimately dictated by the current sensor readings, and not by application-specified requirements.

The work in [17] targets shared, multi-user sensor networks, and exports a strongly-typed, functional language to express processing. Sensors are named via URI relative to the host they are connected to. Still, programmers are not provided with dedicated constructs to specify interactions among logically-defined system partitions, e.g., to direct a given output from a highway sector to the adjacent ones.

Kairos [8] proposes a macroprogramming model inspired by parallel architectures. Developers express the application behavior by writing or reading variables at nodes, iterating on the 1-hop neighbors, and addressing arbitrary nodes. Regiment [16] is a functional macroprogramming language based on the notion of region stream: a spatially distributed, time-varying collection of node states. These are taken as input to functions used to express the application processing. In both cases, no generic construct is provided to express non-trivial interactions among subsets of nodes.

In conclusion, most of the existing approaches target node-level programming where developers still handle low-level aspects, focus on specific classes of applications [11], or do not provide scoping as a first-class programming construct [12]. These characteristics drastically limit their expressive power, and therefore their applicability.

Contributions of this paper. To address the above issues, in this paper we make the following contributions:

- **Programming Constructs for Scoping in Macroprogramming.** In Section 3 we illustrate how the addition of a few constructs to an existing macroprogramming model enables the specification of complex *interactions* among *application-defined scopes*. The addition of scoping to macroprogramming provides application developers with a *logical* layer on top of the underlying physical system, abstracting away the *physical* location of data. This greatly simplifies the programming activity, thus speeding up the development process. To illustrate our ideas, we enable scope-based interactions in ATaG [1], a macroprogramming framework. For the sake of completeness, the salient features of ATaG are summarized in Section 2. Nonetheless, our techniques can, in principle, be incorporated also in alternative macroprogramming approaches.
- **Compiler and Run-time Support for Scoping.** We demonstrate the *feasibility* of our approach by developing a complete development framework in support of the resulting programming model. In this respect, Section 4 illustrates the compilation process used to map the new macroprogramming constructs to the API provided by a dedicated, node-level run-time. Next, Section 5 discusses code metrics gathered on the implementation of our reference application, as well as simulation results obtained by running the actual code resulting from the compilation process. Our results show that the ease of programming brought by our approach does not come at the cost of degraded system performance. These present significant improvements w.r.t baseline solutions, and scalability properties similar to optimal solutions computed with global system knowledge.

This paper builds upon our previous work in sensor network macroprogramming [1], briefly summarized in Section 2. Nevertheless, the integration of scoping into the ATaG language is achieved through novel programming constructs, which yield an expressive power much higher than our previous work. Furthermore, we demonstrate the practical feasibility of our approach by developing a dedicated compiler and run-time support, and by quantitatively assessing the system performance through software metrics and simulation results.

2. The ATaG Programming Model

For the sake of completeness, we briefly summarize the salient features of ATaG. For additional details, the reader is referred to [1]. The Abstract Task Graph [1] (ATaG) is a macroprogramming framework providing a

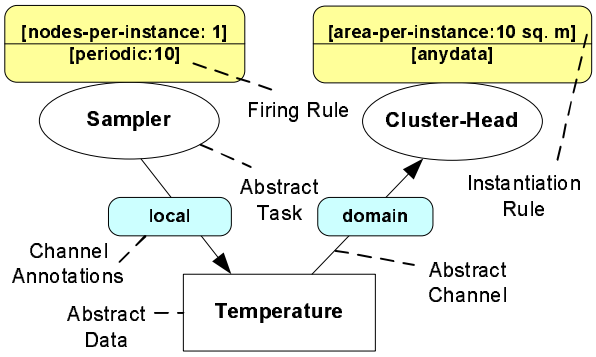


Figure 3. A sample ATaG program.

mixed *declarative-imperative* approach. The notions of *abstract task* and *abstract data item* are at the core of ATaG's programming model. A task is a logical entity encapsulating the processing of one or more data items, which represent the information. The flow of information between tasks is defined in terms of their input/output relations. To achieve this, *abstract channels* are used to connect each data item to the tasks that *produce* or *consume* it.

Figure 3 illustrates a sample ATaG program specifying a cluster-based, data gathering application. Sensors within a cluster take periodic temperature readings, which are then collected by the corresponding cluster-head. The former behavior is encoded in the *Sampler* task, while the latter is represented by *Cluster-Head*. The *Temperature* data item is connected to both tasks using a channel originating from *Sampler*, and a channel directed to *Cluster-Head*.

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 according to the **`periodic`** rule. The *Cluster-Head* fires whenever at least one data item is available on *any* of its incoming channels, in accordance with its **`any-data`** firing rule. The instantiation rules govern the placement of tasks on real nodes. The **`nodes-per-instance:`**1 construct requires the task to be instantiated once on every node. On the other hand, the **`area-per-instance`** construct used for *Cluster-Head* entails partitioning the geographical space according to the given parameter, and deploying *one* instance of the task per partition.

Abstract channels are annotated to express the *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. The *Cluster-Head* uses the **`domain`** annotation to gather data from the temperature sensors in its cluster, which binds to the system partitioning obtained by **`area-per-instance`** and connects the tasks running in the same partition.

The code within a task is the only imperative part in an

ATaG program. To express data exchange between tasks in the imperative code, programmers are provided with the abstraction of a *shared data pool*, where each task can *output* data, or be *notified* when some data of interest is available. Dedicated APIs are provided for this.

3. Scoping in a Macroprogramming Language

In this section, we bring *scoping* in macroprogramming by augmenting the ATaG programming model. We first illustrate how subsets of nodes are specified, and then discuss the novel programming constructs we introduced using an ATaG-based implementation of our reference application as example.

3.1. Determining Scopes

Subset of nodes can be determined in several ways. In this work, we take a simplistic yet general approach, and identify the nodes in a given subset as those satisfying a *membership function* $f_s(i)$, where s is a scope and i is a node. The boolean output of f returns whether i belongs to scope s or not. In turn, the actual definition of f_s is obtained as the composition of atomic *boolean predicates* on the nodes characteristics (called *node attributes* hereafter). As an example, $f_s(i) ::= isInSector(1, i) \wedge hasSpeedSensor(i)$ identifies the subset of nodes equipped with a speed sensor and deployed in the first highway sector.

The boolean predicates are automatically generated by an additional tool we developed that essentially inspects the attributes attached to nodes, and presents a list of predicates to the programmers that only need to compose them in the desired way. With this approach, it is quite natural to determine the desired scopes. In turn, node attributes can be straightforwardly generated in a variety of means, e.g., from third-party meta-data describing the characteristics of a specific hardware platform [20].

3.2. Scoping in ATaG

To enable interactions between scopes, we need to modify primarily two aspects in the ATaG programming model: *task placement* and *data exchange* between tasks. The former express the scopes *where* processing will take place, whereas the latter describe the *interactions* among scopes.

Task Placement. From the application perspective, higher expressivity in task allocation is motivated by the need of mapping a specific processing to nodes equipped with the needed sensing/acting devices, or those present in specific regions. For instance, a task designed to operate the ramp signal must be instantiated on a node having that particular device attached. However, we need only one task to compute the average speed for each highway sector, so we need to identify the different sectors uniquely. This has been achieved with revised *instantiation rules*, that give application programmers the ability to map tasks to application-defined subsets of nodes, e.g., all the nodes deployed in the same highway sector.

Data Exchange. Albeit necessary, the above additions do not yet enable the description of interactions between scopes. For instance, in our scenario the speed limit is decided based on the information sensed in three neighboring highway sectors. To achieve this, we should not only identify the speed sensors deployed in three consecutive sectors, but also *deliver* their data to a scope including the nodes where a task computing the speed limit has been instantiated. To achieve this level of expressivity, we define new *channel interests* in ATaG, so that application programmers can specify the task interests by referring to logical properties of data, regardless of their physical location.

3.3. ATaG Constructs for Scoping

The syntax and use of the scoping constructs are shown in Figure 4, where we illustrate an ATaG implementation of our reference application. All the application information is represented as ATaG data items. The actual algorithm determining the actuation part is encapsulated in two tasks: *SpeedLimitCalculator* and *RampSignalCalculator*, whose inputs are the data produced by tasks deriving the average measures. Once the actuation is determined, it is given as input to the tasks operating displays and ramp signals. As described next, only *three additional constructs* are needed to describe the interactions required in our reference application. Still, their *combination* enables the specification of complex communication patterns otherwise hard (or impossible) to describe.

Instantiating Multiple Tasks in a Scope. The *SpeedSampler* task is in charge of gathering the raw data from a speed sensor on a ramp leading to the highway. Therefore, it must run on a node equipped with the corresponding sensing device. To express this requirement, the `nodes-per-instance:1@speedSensor` construct is used, where `@speedSensor` is a placeholder for a membership function $f_{speedSensor}(i) ::= hasSpeedSensor(i)$. In our current prototype, this is specified using a simple XML file, shown in Figure 5[1]. Similar constructs are used for *RampSampler*, *SpeedLimitDisplayer*, and *RampSignalDisplayer*.

Instantiating a Single Task in a Scope. The *AvgSpeedCalculator* task takes as input the raw data coming from

[1]It is not our intention to force the programmer to write XML directly, we instead envision these specification to be auto-generated by an integrated development environment.

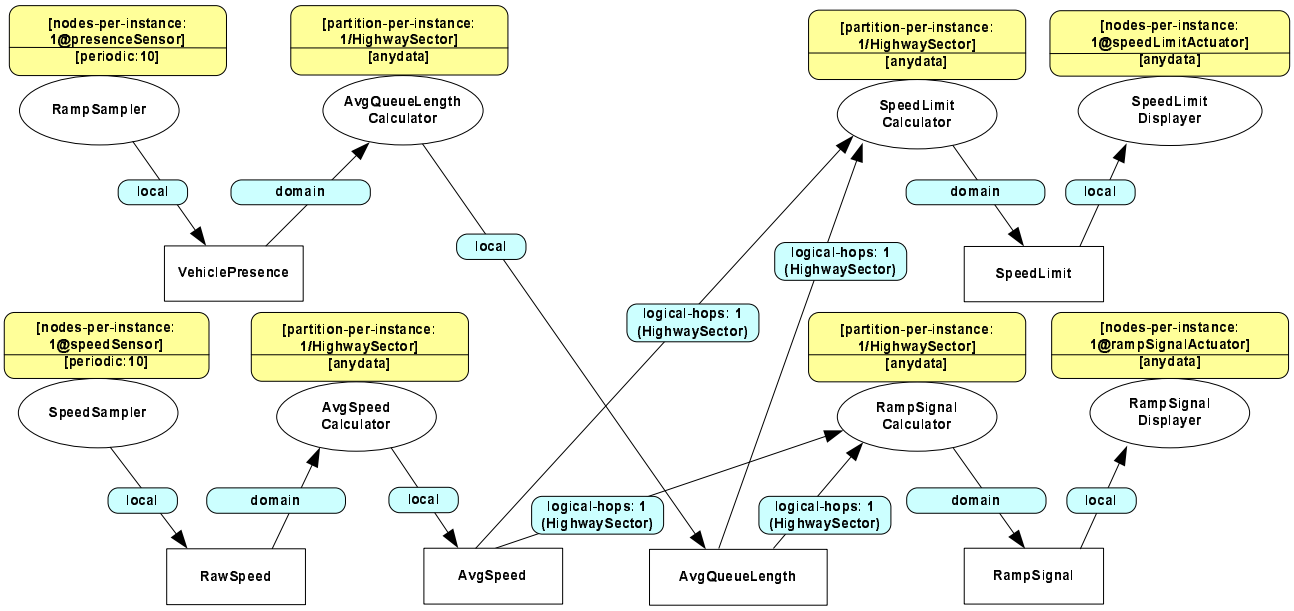


Figure 4. The ATaG program for the traffic management application.

```
<task name="SpeedSampler">
  <instantiationrule>
    <nodes-per-instance
      number="1"
      scopePredicate="hasSpeedSensor"/>
  </instantiationrule>
</task>
```

Figure 5. XML declaration for `@speedSensor` in Figure 4.

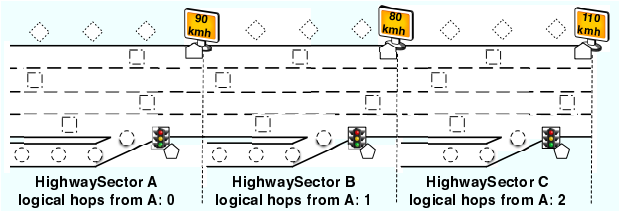


Figure 6. Logical hops over the `HighwaySector` attribute.

the speed sensors in a sector, and derives the average speed of vehicles in the same sector. Therefore, we need such a task to be instantiated once per sector. To express this, the **`partition-per- instance:1/HighwaySector`** construct is used. Again, **`HighwaySector`** is a placeholder for a membership function that identifies all the nodes in a specific sector. The compiler generates all possible values of the corresponding node attribute — that describes the sector where a node is placed in the highway— and requires the task to be instantiated on one node in each sector only.

Inter-Task Communication. To bind tasks running in the same **`HighwaySector`**, the **`domain`** annotation can still be used. However, this time it binds to the system partitioning obtained through the **`partition-per-instance`** instantiation rule. Differently from **`area-per-instance`**, this rule determines the different partitions at a logical level, by considering the node attributes instead of the geographical location.

More generally, the construct **`logical-hops:1(HighwaySector)`** connecting, e.g., the *AvgSpeedCalculator* to both the *SpeedLimitCalculator* and the *RampSignalCalculator* is used to push a data item to a different highway sector. It represents a number of hops counted not on the physical network links, but in terms of how many system partitions (derived from the attribute given in parenthesis) can be crossed. Figure 6 illustrates the concept graphically. Given the partitioning induced by the **`HighwaySector`** attribute, requiring one logical hop on that attribute means, for an *AvgSpeedCalculator* task, to push a data item to the same, immediately preceding and following highway sectors. Notice how the semantics of specifying zero hops is to not cross any partition, i.e., to push to the same partition where the data item originated. In this sense, the **`domain`** construct actually constitutes a particular case of the more general **`logical-hops`** construct.

Dynamic Scopes. In this example we define only static scopes, i.e., we use predicates over attributes that do not vary with time. However, the resulting programming model does not prevent, in principle, the definition of scopes in-

volving time-varying properties of the nodes. For instance, one may specify a predicate $isSensingCar(i)$, that holds when a presence sensor is detecting a car nearby. However, it is not clear what would be the semantics of involving such a predicate in, e.g., a task instantiation rule. Should the task be moved to another node when the condition no longer holds? If not, should the task be suspended and keep the previous state when the condition holds again, or should it just reboot? We believe supporting dynamic scopes may make the programming model unnecessarily complicated, and the final application behavior hard to predict. For this reason, we are currently investigating the application scenarios that may need such a feature, and the semantics required in each case.

4. Compiler and Run-time Support

Our prototype system leverages off the Java2ME [10] language and APIs to describe the imperative part of an ATaG program, and targets the SunSpot sensor platform [21] as underlying hardware platform. Nonetheless, any imperative language can be used instead of Java, as long as it employs a threaded execution model, e.g., the C language on top of the Contiki OS [5].

To generate the node-level code from the ATaG specifications, we implemented a dedicated compiler, whose characteristics and performance are illustrated in [18]. The compiler takes as input the ATaG program and information on the attributes attached to the nodes in the final deployment. Compilation starts by deciding the specific node where each task will be running. This is accomplished by looking at the instantiation rules specified in ATaG, and matching them against the node attributes.

When more than a choice for instantiating a task is available, as in the case of **`partition-per-instance`**, the compiler should place the tasks to minimize some metrics of interests (e.g., network traffic). This problem is orthogonal w.r.t. the support of scoping constructs, since it can be considered as an instance of a graph embedding problem. We are currently working on this aspect as an independent direction of research [18]. Here, instead, we intend to assess the performance of our run-time support to scopes in isolation, without the influence of smart compilation techniques. Therefore, we take a simplistic approach, and assign tasks to nodes randomly when these are not tied to the nodes' capabilities.

After tasks are bound to nodes, the compiler determines the program *data paths*. These are logical addresses identifying the location of tasks that should actually receive a data item once output by another task. Consider, for instance, the data exchange between *AvgSpeedCalculator* and either *SpeedLimitCalculator* or *RampSignalCalculator* in Figure 4. In this case, the data path for an *AvgSpeed* data item includes all the nodes satisfying two specific constraints: i) they are assigned *SpeedLimitCalculator* or *RampSignalCalculator*, and ii) they are deployed either in the same sector where *AvgSpeedCalculator* is running, or in one of the adjacent sectors. Notably, this can still be captured as a scope according to the specification we introduced in Section 3.1. Indeed, consider for instance an *AvgSpeedCalculator* task deployed in sector 5. The subset of nodes where the data item should be delivered can be described as:

$$\begin{aligned} f_{AvgSpeed}(i) \quad ::= \quad & (isInSector(4,i) \vee \\ & isInSector(5,j) \vee \\ & isInSector(6,j)) \wedge \\ & (isSpeedLimitCalculator(i) \vee \\ & isRampSignalCalculator(i)) \end{aligned}$$

where the former conjunct refers to an attribute describing where a node has been placed, whereas the latter conjunct predicates over the assignment of tasks to nodes.

Based on the above observation, the compiler looks at the scopes defined in the application, and generates further scope definitions to identify the data paths. Specifically, for each data item, the compiler creates the corresponding data paths by combining the channel annotations between the producer and consumer tasks with the scopes mentioned on the task instantiation rules. These are used either to determine the target system partition (as done for the highway sector in the example), or to identify the receiver node based on the task it is running.

At the run-time layer, we re-used the routing mechanisms of Logical Neighborhoods [15] to deliver data to the nodes satisfying a given scope specification. With Logical Neighborhoods, the physical neighborhood of a node is replaced by a logical notion of proximity determined by applicative information. Communication is implemented using a form of attribute-based routing where the logical properties of the nodes drive message propagation [14]. In this work, we use the node attributes involved in the definition of at least one data path as logical properties of the nodes, and the data paths themselves as neighborhood definitions[2]. To interact with the nodes in a (logical) neighborhood, the programmer is provided with a simple message-passing API, used to *broadcast* (in a logical sense) a message to all nodes member of a neighborhood. The ATaG node-level run-time leverages off this feature to distribute the data items output by tasks.

Note that our run-time layer does *not* require the data paths to be evaluated at compile-time. Conversely, every time a data item is output by a task, our run-time re-evaluates the corresponding scope definitions. Interestingly,

[2]The mapping from data paths to neighborhood definitions is straightforward, and omitted here for brevity.

this readily provides support for dynamic scopes and migrating tasks. Indeed, to support these features, our approach does not require modifications to the scope definitions output by the compiler. For instance, if the node running *SpeedLimitCalculator* changes at run-time, every scope including $isSpeedLimitCalculator(i)$ will simply evaluate to a different subset of nodes the next time a data item is output by *AvgSpeedCalculator*. As already mentioned, however, the aforementioned functionality have deep implications on the language semantics. For instance, what happens if no node is available to accept a task willing to migrate? We are actively studying how to address these issues in the programming model, leveraging off the support our run-time layer already provides.

5. Evaluation

One of the issues in devising high-level programming models for WSNs is to provide an acceptable run-time performance. Indeed, the inability to reach the lowest possible levels in the protocol stack may prevent developers from fine-tuning the final node-level code. In this section, we argue that our approach provides a reasonable trade-off between these two extremes, by first examining the development effort in our reference application, and then reporting on performance results gathered in simulations.

Evaluating the Programming Effort. Quantifying a developer's effort is a challenge per se, because of the lack of widely accepted methodologies and metrics. This is brought to an extreme in sensor network macroprogramming, where most of the existing metrics cannot even be applied given the early stages of the field. However, interesting insights can be gained by looking at the *fraction of code* developers write w.r.t. the entirety of code deployed on the real nodes. This captures the extent to which the application semantics is achieved by either leveraging off the mechanisms in the *node-level run-time*, or *automatically generating* code. In this respect, it represents the actual added value of the programming model: the *smaller* is this fraction, the *better* the abstractions provided are assisting the programmer, thus speeding up the development process.

With our solution, a total of 51 Java classes need to be compiled to deploy our reference application on the single nodes. However, only 15 of them are the direct result of developers' effort. Furthermore, considering the actual number of lines of non-commented code, only about 12% of them are hand-written by developers, whereas the rest is either part of the run-time support, or automatically generated by our dedicated compiler. We believe these results are due to the flexibility of the scoping abstraction we enabled in the programming model. Complex interactions can indeed be specified in a fully declarative manner, with the compiler taking care of automatically generating the corresponding imperative code and the inputs for the node-level run-time.

Parameter Name	**Value**
Propagation Model	Two-ray Ground
Radio Model	Additive Noise
MAC Layer	CSMA
Transmission Rate	250 Kbps
Communication Range	40 meters
Message Size	47 bytes
Simulation Time	2000 secs
Number of Repetitions	30

Figure 7. Simulation parameters.

Considering the code implementing each task, it is possible to identify a recurring pattern with only two classes needed. One represents the task itself, and contains the processing to interact with the data pool. This same class usually holds a reference to a second class containing the actual processing, e.g., to average the incoming data as in *AvgQueueLengthCalculator*. Notably, all the state variables defined in these classes relate only to the application semantics, and never refer to distribution aspects. This is achieved thanks to the way communication patterns are specified in our approach: the data recipients are always determined implicitly by the definition of scopes and the interactions among them. Therefore, the programmer does not need to care about this in the actual application code.

Simulation Settings. To verify that the above advantages do not entail a degraded run-time performance, we quantitatively characterize the behavior of our reference application in a simulated scenario. We use the SWANS/Jist simulator [2], as it is able to run *unmodified* Java code on top of a simulated network. This way, we measure the performance of the same code that can be deployed on the real nodes.

The relevant simulation parameters are reported in Figure 7. We consider the scenario in Figure 1 as network topology, with a highway sector 20 meters wide and 200 meters in length. We place the forwarding nodes 25 meters apart, and randomly distribute the speed sensors on the four lanes so that each of them is range of at least another speed sensor or a forwarding node. Similarly, the presence sensors are randomly distributed on the ramp so that each of them is in range of at least one speed sensor or another presence sensor. The node controlling the ramp signals and the speed limit displays are placed between different sectors, on opposite sides of the road. Overall, 18 nodes are deployed in each highway sector.

Note the message rate is implicitly determined by the application itself, in particular by the firing rules for tasks. For instance, a node running an instance of *RampSampler* generates one message every 10 seconds, as its firing rule is `periodic:`10. The *AvgQueueLengthCalculator* fires for any data item received, and correspondingly outputs a new data item. Therefore, if four *RampSampler*s are in its `domain`, *AvgQueueLengthCalculator* will generate a mes-

sage every 2.5 seconds, on the average.

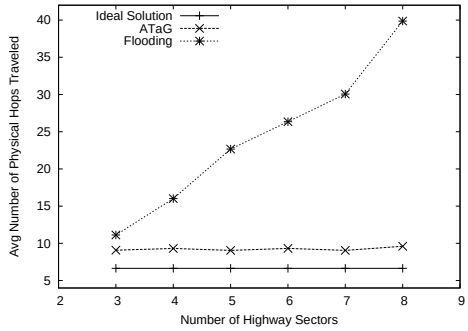
The simulation runs differ in the random seed, the location of nodes, and the assignment of tasks to nodes when the choice is not unique. As performance metrics, we consider *i)* the number of *missing actuations* on the environment, resulting from one or more *message losses* on the path from the sensing tasks to the actuation tasks, *ii)* the *network overhead*, represented as the overall number of messages sent at the physical layer, and *iii)* the average number of *physical hops* traveled by a message carrying a data item before either being discarded or delivered.

As the goal of the application developer is that of deciding *actions* based on data *sensed*, the first quantity intuitively measures the *quality of service* provided by the implemented system. Differently, as communication dominates the energy expenditures in WSNs, the second measure assesses the actual feasibility of our approach on real devices. The third measure gives insights into the trends related to communication cost, describing where communication takes place. As independent variable, we choose to vary the number of highway sectors, as this indirectly dictates the system scale.

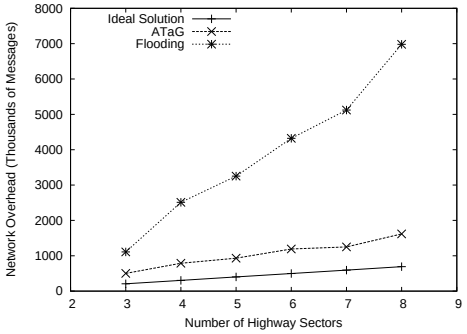
For comparison, we compute the aforementioned metrics for an *optimal solution* minimizing the network overhead, based on global knowledge of the network topology. We first identify the optimal task placement given the expected network traffic, and then the minimum cost routing tree connecting a sender to all the intended recipients. The performance obtained with a pure flooding scheme are also reported as an upper bound for further comparison.

Simulation Results. Given the message generation rates discussed earlier, our solution can provide at least 96% of the actuations that would be occurring in case there were no message losses. This illustrates how the messages carrying the application data are effectively delivered to the intended recipients. Remarkably, this metric is *not* affected by a varying number of highway sectors (and is hence not shown graphically). Such a behavior demonstrates how our scoping constructs allow the application semantics to *percolate* down to the network layers. Indeed, the application requires its processing to span at most three adjacent highway sectors, and is therefore independent of the overall number of highway sectors.

The chart in Figure 8(a) further confirms the above reasoning: as expected, the number of hops traveled by a message using flooding rapidly increases with the number of highway sectors. On the contrary, our solution keeps an almost constant performance in a range of settings, effectively ending up close to the theoretical minimum. Note how it is hard to achieve the same form of *implicit cross-layer optimization* in the absence of scoping: if the programming model does not allow interactions among application-defined scopes to be defined, it is hard to make the routing



(a) Average number of physical hops traveled.



(b) Network overhead.

Figure 8. Reference application performance.

layers aware of them.

Figure 8(b) depicts the trends in network overhead against a varying number of highway sectors. With our solution, this metric is much closer to the optimal solution than to flooding. More importantly, the trend as the number of highway sectors increases mimics that of the optimal solution, while flooding reveals a much steeper increase. We believe this performance is reasonable, also considering tasks are placed randomly when the decision is not unique. All the metrics are indeed likely to see a dramatic improvement if the compiler placed the tasks smartly using a cost model of the underlying routing scheme. This is in our immediate research agenda.

6. Conclusion and Future Work

In this paper we introduce programming constructs to enable *scope-based interactions* in sensor network *macroprogramming*. Our approach allows programmers to express complex communication patterns with a few programming constructs. The feasibility of our approach is demonstrated by a dedicated compiler we developed, and by simulation studies assessing the performance of the final implementations.

Our immediate research goals include the definition of a precise semantics associated to *dynamic scopes*, and a

full support to *migrating tasks*. As an independent direction of work, we are also exploring techniques to *optimize the placement of tasks* on the nodes during the compilation process by looking at the expected flow of information.

Acknowledgements. 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 and CNS-0627028.

References

[1] 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)*, 2005.

[2] R. Barr, Z. J. Haas, and R. van Renesse. Jist: an efficient approach to simulation using virtual machines. *Softw. Pract. Exper.*, 35(6), 2005.

[3] I. Chatzigiannakis, A. Kinalis, and S. Nikoletseas. An adaptive power conservation scheme for heterogeneous wireless sensors. In *Proc. of the* 17^{th} *Symp. on Parallelism in Algorithms and Architectures*, 2005.

[4] A. Deshpande, C. Guestrin, and S. Madden. Resource-aware wireless sensor-actuator networks. *IEEE Data Engineering*, 28(1), 2005.

[5] 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.

[6] C. Frank and K. Römer. Algorithms for generic role assignment in wireless sensor networks. In *Proc. of the* 3^{rd} *ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Nov. 2005.

[7] Habitat Monitoring on the Great Duck Island. `www.greatisland.net`.

[8] R. Gummadi, O. Gnawali, and R. Govindan. Macroprogramming wireless sensor networks using Kairos. In *Proc. of the* 1^{st} *Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[9] T. T. Hsieh. Using sensor networks for highway and traffic applications. *IEEE Potentials*, 23(2), 2004.

[10] Sun™ Java2 Micro-edition Specification, `java.sun.com/javame`.

[11] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *Trans. on Embedded Computing Sys.*, 5(3), 2006.

[12] 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)*, 30(1), 2005.

[13] C. Manzie, H. C. Watson, S. K. Halgamuge, and K. Lim. On the potential for improving fuel economy using a traffic flow sensor network. In *Proc. of the Int. Conf. on Intelligent Sensing and Information Processing*, 2005.

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

[15] 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.

[16] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proc of the* 1^{st} *Int. Workshop on Data Management for Sensor Networks (DMSN)*, 2004.

[17] M. J. Ocean, A. Bestavros, and A. J. Kfoury. snBench: Programming and virtualization framework for distributed multitasking sensor networks. In *Proc. of the* 2^{nd} *Int. Conf. on Virtual execution environments*, 2006.

[18] A. Pathak, L. Mottola, A. Bakshi, G. P. Picco, and V. K. Prasanna. A compilation framework for macroprogramming networked sensors. In *Proc. of the the* 3^{rd} *Int. Conf. on Distributed Computing on Sensor Systems (DCOSS)*, 2007.

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

[20] SensorML, `vast.uah.edu/SensorML/`.

[21] Sun™ Small Programmable Object Technology (Sun SPOT), `www.sunspotworld.com`.

[22] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc of the* 1^{st} *USENIX/ACM Symp. on Networked Systems Design and Implementation (NSDI)*, 2004.

[23] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proc. of the* 2^{nd} *Int. Conf. on Mobile systems, applications, and services (MOBISYS)*, 2004.