Programming Wireless Sensor Networks with Logical Neighborhoods

Luca Mottola and Gian Pietro Picco
Dipartimento di Elettronica ed Informazione—Politecnico di Milano
{mottola,picco}@elet.polimi.it

***Abstract*— Wireless sensor network (WSN) architectures often feature a (single) base station in charge of coordinating the application functionality. Although this assumption simplified the path to adoption of WSN technology, researchers are now being attracted by more decentralized architectures with multiple sinks and heterogeneous nodes. These scenarios are brought to an extreme in Wireless Sensor and Actor Networks (WSANs), where sensing and acting nodes collaborate in a decentralized fashion to implement complex control loops. In these settings, new programming abstractions are required to manage complexity and heterogeneity without sacrificing efficiency.**

In this paper we propose and define a *logical neighborhood* programming abstraction. A logical neighborhood includes nearby nodes that satisfy predicates over their static (e.g., type) or dynamic (e.g., sensed values) characteristics. The span of the neighborhood and the definition of its predicates are specified declaratively, along with requirements about the cost of the communication involved. Logical neighborhoods enable the programmer to "illuminate" different areas of the network according to the application needs, effectively replacing the physical neighborhood provided by wireless broadcast with a higher-level, application-defined notion of proximity.

This paper presents the definition of a declarative language for specifying logical neighborhoods, highlighting its expressiveness, flexibility and simplicity. Moreover, although the language constructs are readily implemented using existing communication mechanisms, we briefly report about a novel routing scheme we expressly designed to support efficiently our abstractions.

I. Introduction

Wireless sensor networks (WSNs) are increasingly employed in a variety of settings to gather data from the physical world. Nevertheless, most of the WSN architectures deployed thus far are quite simple. Habitat monitoring [9], one of the most popular applications, is paradigmatic in this respect, featuring a *single* base station collecting data from a high number of *homogeneous* nodes.

More recently, researchers are investigating the use of more decentralized settings where multiple base stations are employed, different applications run on the same hardware, or heterogeneous nodes are deployed. These approaches find their extreme realization in wireless sensor and actor networks (WSANs) [1], where nodes not only observe and gather data from the environment, but are also capable of affecting it by performing a variety of actions. Applications range from localization facilities to control systems in tunnels or buildings, interactive museums and home automation [12].

These applications clearly raise new challenges in the field of networked embedded systems. Indeed, while in mainstream WSNs the application goals are mainly realized by a single task performed across the whole network (typically, that of sensing and reporting a given measure), in decentralized scenarios applications are usually composed of many collaborating tasks, each affecting only a given part of the system. For instance, a WSAN deployed to perform control and monitoring of a building needs to perform at least three main tasks, i.e., structural monitoring, in-door environment monitoring, and response to extreme events such as fire or earthquakes [2]. To realize the latter functionality, actuator nodes controlling water sprinklers need to monitor nearby temperature sensors and smoke detectors and take the appropriate countermeasures where and when it is needed. Therefore, not only devices have *heterogeneous* capabilities, but the application logic must now reside in the network: including a central base station in the control loop would inevitably degrade system performance and reliability without any sensible advantage [1].

Developing applications for this kind of systems is complex. In fact, the developer must worry not only about the implementation of the application logic, but also about which subset of the system should be involved and how to reach it. As no dedicated programming constructs and mechanisms exist for the latter task, the result is additional programming effort, increased complexity and, in absence of well-established and reusable solutions, less reliable code.

In this work we address the aforementioned issues by introducing the notion of *logical neighborhood*, an abstraction that replaces the conventional notion of physical neighborhood—i.e., the set of nodes in the communication range of a given device—with a logical notion of proximity determined by applicative information, and easily allows for data fusion and aggregation through sensor virtualization. Logical neighborhoods are specified declaratively using the SPIDEY language we designed, illustrated in the rest of the paper, conceived to be a simple extension to existing WSN programming languages (e.g., nesC [5] in the case of TinyOS [6]).

To minimize the impact on the programming task, we couple our logical neighborhood abstraction with conventional broadcast-based communication. Using our enhanced communication API, a broadcast message can now be sent to the nodes in the system that satisfy the constraints imposed by a given application-defined neighborhood, instead of the nodes within communication range. This way, application programmers still reason in terms of neighborhood relations and broadcast messages, but in addition can specify declaratively

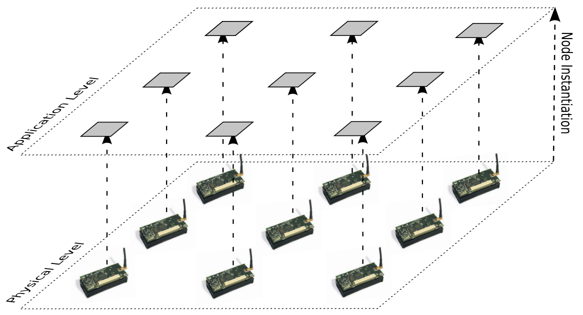

Fig. 1. A portion of a node's state and characteristics is exported at the application-level by means of (logical) node instances.

the set of nodes a given device considers as its neighbors. As such, our abstraction may foster a fresh look at existing mechanisms, algorithms, and programming models by replacing their conventional notion of physical neighborhood with our programmer-defined, logical one.

Furthermore, even if existing solutions could be used to implement the routing mechanisms needed to support communication in a logical neighborhood (e.g., [7]), we also propose a novel routing algorithm that, in our opinion, better captures the kind of *localized interactions* [3], [13] that characterize decentralized scenarios like WSANs.

The overall benefits of our proposal impact two orthogonal aspects. First, developers can concentrate on the actual application goals while relying on SPIDEY and the associated communication framework as a way to logically partition the system and interact with it. We conjecture that applications built on top of our abstraction result in cleaner, simpler, and more reusable implementations. Second, this strategy opens up opportunities for achieving a longer system lifetime and a better resource utilization, by focusing only on the nodes that actually need to be involved.

The rest of the paper is organized as follows. Section II describes the logical neighborhood abstraction and the SPIDEY language. Section III discusses solutions for supporting communication in our approach, including a novel routing approach expressly devised for the decentralized scenarios we target. Finally, Section IV surveys the related work, while Section V ends the paper with brief concluding remarks and an outlook on our future work.

II. THE LOGICAL NEIGHBORHOOD ABSTRACTION

In this section, we begin with an overview of the core concepts underlying the notion of logical neighborhood, and describe the constructs of the SPIDEY language through examples. Then, we show how the language provides also higher level abstractions by means of hierarchical composition of neighborhoods, enabling a notion of *virtual sensor*. Finally, we describe an API to support broadcast-based communication in logical neighborhoods.

```
node template Device
  static Function
  static Type
  static Location
  dynamic Reading
  dynamic BatteryPower

create node ts from Device
  Function as "sensor"
  Type as "temperature"
  Location as "room1"
  Reading as getTempReading()
  BatteryPower as getBatteryPower()
```

Fig. 2. Sample node definition and instantiation.

```
neighborhood template HighTempSensors(threshold)
  with Function = "sensor" and
       Type = "temperature" and
       Reading > threshold

create neighborhood htsn100
  from HighTempSensors(threshold : 100)
  max hops 2
  credits 30
```

Fig. 3. Sample neighborhood definition and instantiation.

A. Core Concepts

The abstraction we propose revolves around only two concepts: *nodes* and *neighborhoods*. As illustrated in Figure 1, a (logical) node is the application-level representation of a physical node, and defines which portion of a node's data and characteristics is made available by the programmer to the definition of any logical neighborhood. The definition of a node is encoded in a *node template*, which specifies the node's exported attributes. This is used to instantiate the (logical) node, by specifying the actual source of data. To make these concepts more concrete, Figure 2 shows a SPIDEY code fragment to define a node template for a generic sensor, and then instantiate a node with the structure prescribed by the template. During instantiation, each attribute in a node template is bound to an expression of the target language, e.g. a variable or a function.

The attributes in a node template can be **`static`** or **`dynamic`**. The former represent information assumed not to vary in time, e.g., the type of measurement a sensor node provides. Instead, dynamic attributes represent information that by definition changes with time, e.g., the current sensor reading. The decision about whether an attribute is static or dynamic depends on the deployment scenario. Making the distinction explicit may enable optimizations at the routing layer.

A (logical) neighborhood is the set of nodes satisfying a predicate on the nodes' attributes. As with nodes, the definition of neighborhoods is encoded in a template, which contains the predicate that essentially serves as the membership function determining whether a node belongs to the logical neighborhood. For instance, the neighborhood template `HighTempSensors` in Figure 3 is based on the `Sensor` template in Figure 2 and selects nodes that host temperature

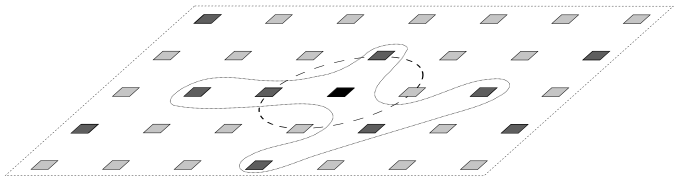

Fig. 4. A pictorial representation of the example in Figure 3. The black node is the one defining the logical neighborhood, and its physical neighborhood (i.e., nodes lying in its direct communication range) is denoted by the dashed circle. The dark nodes are those satisfying the predicate in the neighborhood template `HighTempSensors` when the threshold is set to 100^{o}C. However, the nodes included in the actual neighborhood instance `htsn100` are only those lying within 2 hops from the sending node, as specified through the **`hops`** clause during instantiation.

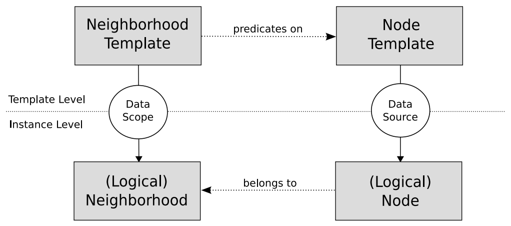

Fig. 5. Relationship between templates and their instantiation.

sensors and are currently reading a value higher than a given threshold. As exemplified in the SPIDEY code fragment of Figure 3, a neighborhood template can be parameterized, with the actual parameter values provided by expressions of the target language upon neighborhood instantiation.

Moreover, the instantiation of a neighborhood template specifies additional requirements about *where* and *how* the neighborhood is to be constructed and maintained. For instance, Figure 3 specifies that the predicate defined in the `HighTempSensors` template is evaluated only on nodes that are at a maximum of 2 hops away and by spending a maximum of 30 "credits". The latter is an application-defined measure of cost, further detailed next, which enables the programmer to retain some control over the resources being consumed during the distributed processing necessary to deliver messages to members of a logical neighborhood. A pictorial representation of the example, visualizing the logical neighborhood concept, is provided in Figure 4.

In essence, as graphically illustrated in Figure 5, templates define *what* data is relevant to the application, while the instantiation process constrains *how* this data should be made available by the underlying system. Separating the two perspectives has several beneficial effects. The same template can be "customized" through different instantiations. For instance, the very same template in Figure 3 could be used to specify a logical neighborhood with a different threshold or a different physical span. Moreover, this distinction naturally maps on an implementation that maintains a neighborhood by disseminating its template to be evaluated against the values exported by a node instance, and uses instead the additional constraints specified at instantiation time to direct the dissemination process.

Beyond the basic constructs described so far, SPIDEY gives developers further flexibility in defining the neighborhood template through a set of simple and yet expressive constructs. Traditional logical operators such as **`and`**, **`or`**, and **`not`** are provided to define complex predicates on node templates. In addition, as logical neighborhoods essentially identify sets of nodes, it becomes natural to express a neighborhood as a composition with already existing ones, using conventional set operators such as union, intersection, subtraction, and inclusion.

Our language also provides flexibility in managing the cost involved in communicating towards a (logical) neighborhood, through the **`credits`** construct. Communication cost is defined in terms of the basic operation of sending a broadcast message to physical neighbors (the node's *sending cost*), and is measured in *credits*. The mapping between the two is specified by the programmer on a per-node basis through a **`use cost`** construct, which delegates the computation of this mapping to an expression of the target language. Therefore, the programmer can define a vast array of mappings, from a straightforward one where the sending cost is fixed, to sophisticated ones where it varies dynamically to adapt to context changes (e.g., low battery power). Moreover, different nodes can have different functions, e.g., with higher costs for tiny, battery-powered sensors, and lower costs for resource rich, externally-powered nodes. The overall number of credits necessary to communicate with the members of a logical neighborhood is evaluated as the sum of the costs that each node involved in routing messages incurs in, with each node evaluating its own cost according to the function specified in the **`use cost`** declaration. Therefore, the ability to set the maximum amount of credits spent in communication in a logical neighborhood enables programmers to exploit different trade-offs between accuracy and resource consumption. Neighborhoods instantiated with a high number of credits ensure a broader coverage at the expense of a higher amount of credits and therefore resources. Instead, neighborhoods endowed with a small number of credits may not reach all the specified nodes, but are guaranteed to limit resource consumption.

B. Higher-Level Abstractions

Beyond defining and instantiating basic logical neighborhoods, SPIDEY enables, without compromising its simplicity, sensor virtualization through data aggregation and hierarchical composition of neighborhoods.

Along the lines of [2], consider an actuator node controlling a water sprinkler and monitoring the readings provided by a set of nearby (e.g., deployed in the same room) temperature sensors. When the average readings they collectively provide are beyond a given threshold, the sprinkler must be activated.

This behavior can be encoded using the SPIDEY abstraction we discussed thus far, e.g., by defining a neighborhood `nearbyTempSensors`. However, the collection of the data reported by the sensors in this neighborhood and the corre-

```
  create node vts from Device
    Function as "virtualSensor",
    Type as "temperature",
    Location as "room1"
    Reading as average(roomTempSensors) every 30
```

Fig. 6. Building a virtual sensors out of a neighborhood of lower-level real sensors: the case of average values.

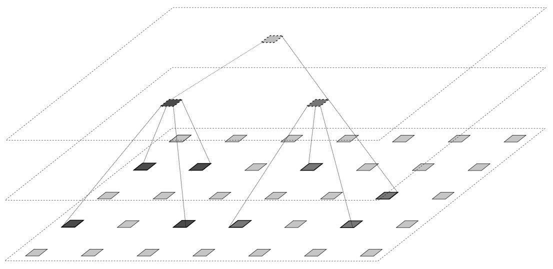

Fig. 7. A hierarchy of sensors where at the lowest level are actual sensors and at above levels are derived virtual sensors. The same color is used to represent sensors belonging to the same logical neighborhood.

sponding processing would be entirely up to the programmer. Here, we introduce instead the ability to regard the set of temperature sensors as a higher-level, *virtual sensor* able to report as its main reading directly the average temperature.

Clearly, to accomplish this, the programmer inevitably needs to encode the aggregation semantics into an appropriate function that, conceptually, depends on the values reported by the nodes holding data needed to derive the aggregated measure, i.e., the nodes in a given (logical) neighborhood, as in[1]:

```
function average(Neighborhood: nhood) {
  sum = 0; counter = 0;
  for(node in nhood) {
    sum = sum + node.Reading;
    counter = counter + 1;
  }
  return sum/counter;
}
```

However, the SPIDEY framework can spare the programmer from the burden of directly handling the communication needed to gather and aggregate data. This is achieved by instantiating a *virtual sensor* as in Figure 6, directly reporting the aggregated measure. In this definition, the key difference with what we showed in Section II-A is the use of the **`as`** clause, which binds an attribute to an aggregation function (i.e., the `average()` function above) operating on a (logical) neighborhood (i.e., the one containing temperature sensors deployed in the same room). The rate at which data should be gathered from the nodes must also be specified by the programmer using the associated **`every`** clause. Notably, we defined these virtual sensors from the same `Sensor` template we defined earlier in Figure 2: simply, at instantiation time we bind the attribute `Reading` to a different, distributed data source.

[1]Here, we use an abstract construct `in` to obtain an enumeration of the node instances in a (logical) neighborhood. However, this construct must be mapped appropriately on the target language.

Figure 7 illustrates the concept. At the bottom level is a flat space of (logical) nodes, each being a direct abstraction of the underlying hardware—temperature and water sprinklers in our example. Aggregation functions over a neighborhood enable developers to code at a higher-level of abstraction, where nodes in a logical neighborhood are perceived as a single virtual sensor. This is shown in the middle plane of Figure 7, where the virtual sensors we just defined are placed.

Once virtual sensors are created, they can be used just like any other sensor. This enables the programmer to recurse the process arbitrarily, creating hierarchies that push the level of abstraction higher, as shown by the topmost sensor in Figure 7. For instance, assume we created a virtual smoke detector reporting the average measurement obtained from real smoke detectors in a logical neighborhood. The virtual sensor in Figure 6 and this new one could then be used to define a single "virtual fire detector", operating in an appropriate neighborhood, and reporting whether a fire is detected based on the evaluation of some application-defined function over the virtual temperature and smoke sensors. Once more, the application can easily detect the presence of a fire and trigger the appropriate reaction by checking only the current reading of the topmost virtual sensor, instead of explicitly gathering and processing the raw data reported by the real sensors, thus considerably simplifying the programming task.

C. The Communication API

Neighborhoods must ultimately be used in conjunction with communication facilities, to enable interaction among the neighborhood members. On the other hand, the notion of logical neighborhood is essentially a scoping mechanism, and therefore is independent from the specific communication paradigm chosen. For instance, one could couple it with the tuple space paradigm to enable tuple sharing and access only within the realm of a logical neighborhood. However, in an effort to keep support for our abstraction as minimal as possible, we combined it with the simplest communication mechanism available for WSNs, i.e., message passing through broadcast communication.

In this sense, the API we are currently developing mimics the one traditionally provided by the bare operating system with simple `send` and `receive` operations (e.g., as in TinyOS' `GenericComm` module). The `send` operation is extended with an additional parameter representing the logical neighborhood where a message must to be delivered, i.e., the scope of that particular message. Essentially, we are replacing the broadcast facility commonly made available by the operating system with one where message recipients are not determined by the physical communication range, rather by membership in a programmer-defined logical neighborhood. In addition, an auxiliary operation that can be used to reply to a message received through a neighborhood is provided.

III. ROUTING STRATEGIES

As our logical neighborhood abstraction is essentially independent of the underlying routing layer, existing and well-established solutions to the problem of *data-centric* communication in WSNs can be directly employed. For instance, *Directed Diffusion* [7] can serve our purposes by simply redefining the interest matching function so that data is fed towards nodes satisfying a given neighborhood template. Exploiting these protocols provides a rapid development path and clearly assesses the feasibility of our approach.

At the same time, we contend that existing solutions fail in capturing the kind of localized interaction patterns that should characterize communication in decentralized, multi-sink WSNs and WSANs. For instance, actuator nodes are envisioned to communicate mainly with sensors deployed in close proximity, as typically the control loop they perform affects a small portion of the physical environment and does not involve a central base station. In this scenario, some of the characteristics of Directed Diffusion (e.g., global propagation of interests) would waste resources. Analogous considerations hold for other established approaches, e.g., the system-wide tree-shaped overlay network used in TinyDB [8]. Moreover, existing proposals do not lend themselves to an easy implementation of some of the constructs of SPIDEY (e.g., the management of credits), as they are not designed with heterogeneous devices in mind.

For these reasons, we designed a dedicated routing strategy supporting our logical neighborhood abstraction. Due to space limitations we can only sketch its behavior here. Moreover, we are currently evaluating its performance through simulation.

Our approach to routing is *structure-less* (i.e., it does not exploit overlays), is based on the notion of *local search* [10], and relies on two core mechanisms. The first mechanism builds a distributed *state space* by periodically propagating node profiles (or portions thereof) and storing at each node the cost to reach a device whose profile contains a specific $\langle$attribute,value$\rangle$ pair. This cost is evaluated in terms of the aforementioned sending cost, computed locally based on the expression given in the **`use cost`** construct and accumulated along the path to that device. However, the propagation of node profiles is constrained so that each node has enough information to reach only the *closest node* with a given attribute. Therefore, the spreading of node information is limited to small portions of the system, thus achieving better scalability, and information stored at each node can be compressed with a simple scheme to achieve less memory usage.

The second mechanism enables messages to smartly "navigate" this state space. Messages addressed to a logical neighborhood contain the neighborhood template, thus making explicit the part of the state space that must be considered. The credits specified when instantiating a neighborhood are also attached to each message and "spent" in navigating the state space. Each message is always sent in broadcast mode with the goal of exploring the state space and looking for *decreasing paths*, i.e., paths where the cost needed to reach a given $\langle$attribute,value$\rangle$ pair is decreasing. Whenever such a path is found, a *credit reservation* mechanism, exploiting the state space information, reserves enough credits to reach the corresponding node following this decreasing path. These credits are immediately removed from those in the message. The remaining credits are used to explore non-decreasing paths and are evenly divided among the physical neighbors of the sending node, in the hope to find further decreasing directions in other parts of the system. Clearly, this mechanism is able to tolerate the dynamic topology characteristic of sensor systems—induced by the low duty cycle of nodes—by providing multiple paths to a given destination.

Additionally, the reverse path set up by messages sent to members of a logical neighborhood can be exploited both to forward replies towards the original sender, as well as to better direct the search for possible members of a neighborhood. In particular, the latter consists in avoiding distributing credits towards part of the system where no member of the neighborhood has appeared so far, and in redirecting these credits towards other, still unexplored parts of the system.

As for the management of virtual nodes, several solutions are viable and we are currently investigating the tradeoffs between redundancy of information and network traffic overhead. At one extreme, neighborhood members can be requested to send data to the node defining the virtual node, where aggregation is performed. At the other extreme, we can make each member propagate its data to all the others, and perform aggregation on every node in a neighborhood. This way, each node can act as a replica of the virtual one. Furthermore, depending on the properties of the aggregation function at hand, intermediate solutions may be possible. For instance, data could be aggregated piecemeal as it flows towards the device defining the virtual node, as in TinyDB [8].

IV. RELATED WORK

Only few proposals define distributed abstractions for WSNs that support some notion of scoping. Moreover, unlike the strongly decentralized scenarios we target in this work, many assume a single data sink.

The work closer to ours is the neighborhood abstraction described in Hood [15], where each node has access to a local data structure where attributes of interest provided by neighboring nodes are cached. The current implementation considers only 1-hop neighbors and is mainly based on broadcasting all node attributes and filtering on the receiver's side. Clearly, our framework is much more flexible as it provide a much more general, application-defined neighborhood abstraction.

The work an Abstract Regions [14], instead, proposes a model reminiscent of tuple spaces to enable communication among the nodes belonging to a given *region*. The span of a region is based mainly on physical characteristics of the network (e.g., physical or hop-count distance between sensors) and each region requires a dedicated implementation. Therefore, each region is somehow separated from others, and regions cannot be combined. This results in a much lower degree of orthogonality and flexibility with respect to our approach. Moreover, the concept of *tuning interface* provides access to a region's implementation, enabling the tweaking of low-level parameters (e.g., the number of retransmissions).

Instead, our approach provides a higher-level, application-dependent notion of cost that can be used to control resource consumption.

In TinyDB [8], materialization points create views on a subset of the system and aggregation functions can be used for deriving higher level data. In this sense, common to our work is the effort in providing the application programmer with higher-level network abstractions. However, the approach is totally different, as TinyDB forces the programmer to a specific style of interaction (i.e., a data-centric model with SQL-like language) and targets scenarios where a single base station is responsible for coordinating all the application functionalities.

SpatialViews [11] is a programming language for mobile ad-hoc networks where *virtual networks* can be defined depending on the physical location of a node and the services its provides. Computation is distributed across nodes in a virtual network by migrating code from node to node. Common to our work is the notion of scoping virtual networks provides. However, SpatialViews targets much more capable devices than ours, and focuses on migrating computation instead of supporting an enhanced communication facility as we do. Still, the framework is less flexible than our proposal in defining members of a spatial view, as a virtual network is basically defined requesting a given service within some physical scope.

Finally, in [4], the authors propose a language and algorithms supporting generic role assignment in WSNs with an approach that, in a sense, is dual to ours. In fact, their work *imposes* certain roles on nodes in the system so that some specified requirements are met, while in our approach the notion of logical neighborhood *selects* nodes in the system based on their characteristics.

V. Conclusions and Future Work

In this work we introduced logical neighborhoods as a novel programming abstraction for WSNs. Logical neighborhoods capture set of nodes with functionally related characteristics and simplify the programmer's task by providing a base for node interaction that goes beyond the physical neighborhood defined by wireless broadcast.

Logical neighborhoods are defined by the application using the SPIDEY language we presented in this paper. SPIDEY constructs enable the programmer to specify neighborhoods declaratively, and yet control the trade-off between accuracy and resource consumption using an application-defined notion of cost. Data aggregation is also supported through a notion of virtual sensor that in turn relies on logical neighborhoods.

We argued that our abstractions can be readily implemented with existing routing algorithms, but also sketched a routing solution we conceived expressly to support communication in logical neighborhoods in a localized fashion.

We are currently evaluating the performance of the routing strategy we devised through simulation, as well as investigating different ways to implement virtual nodes. Future work will study an analytical model of our routing approach, to provide support for credit management, and investigate how our abstraction can be used to enhance higher-level communication abstractions (e.g. tuple spaces and event-based paradigms).

Acknowledgements. The work described in this paper is partially supported by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project, by the National Research Council (CNR) under the IS-MANET project, and by the European Union under the IST-004536 RUNES project.

References

[1] I. F. Akyildiz and I. H. Kasimoglu, “Wireless sensor and actor networks: Research challenges,” *Ad Hoc Networks Journal*, vol. 2, no. 4, pp. 351–367, October 2004.

[2] M. Dermibas, “Wireless sensor networks for monitoring of large public buildings,” 2005, tech. Report, University at Buffalo. Available at www.cse.buffalo.edu/tech-reports/2005-26.pdf.

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

[4] 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'05)*, Nov. 2005.

[5] 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 *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'03)*. ACM Press, 2003, pp. 1–11.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, “System architecture directions for networked sensors,” in *ASPLOS-IX: Proc. of the* 9^{nt} *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 2000, pp. 93–104.

[7] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, “Directed diffusion for wireless sensor networking,” *IEEE/ACM Trans. Networking*, vol. 11, no. 1, pp. 2–16, 2003.

[8] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, “TinyDB: an acquisitional query processing system for sensor networks,” *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.

[9] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, “Wireless sensor networks for habitat monitoring,” in *WSNA '02: Proc. of the* 1^{st} *ACM international workshop on Wireless sensor networks and applications*. New York, NY, USA: ACM Press, 2002, pp. 88–97.

[10] L.A. Wosley, *Integer Programming*. Wiley, 1998.

[11] Y. Ni, U. Kremer, A. Stere, and L. Iftode, “Programming ad-hoc networks of mobile and resource-constrained devices,” in *PLDI05: Proc. of the 2005 ACM SIGPLAN Conf. on Programming language design and implementation*. New York, NY, USA: ACM Press, 2005, pp. 249–260.

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

[13] H. Qi and P.T. Kuruganti, “The development of localized algorithms in wireless sensor networks,” *Sensors Journal*, vol. 2, no. 7, July 2002.

[14] 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 (NSDI04)*, March 2004.

[15] 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*. ACM Press, 2004.