# Building Virtual Sensors and Actuators over Logical Neighborhoods

Pietro Ciciriello, Luca Mottola and Gian Pietro Picco
Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
{ciciriello,mottola,picco}@elet.polimi.it

## ABSTRACT

Recent trends in wireless sensor network (WSN) applications exhibit increasing degrees of decentralization. This is particularly true of scenarios where the data reported by *sensors* is used to control *actuators* affecting the environment. Implementing this control loop in a decentralized fashion is much more complex than in mainstream, single-sink, sense-only applications.

In this paper we describe *virtual nodes*, a programming abstraction simplifying the development of decentralized WSN applications. The data acquired by a set of sensors can be collected, processed according to an application-provided aggregation function, and then perceived as the reading of a single *virtual sensor*. Dually, a *virtual actuator* provides a single entry point for distributing commands to a set of real actuator nodes. The set of physical nodes to be abstracted into a virtual one is specified using *logical neighborhoods* [11, 12]. Using virtual nodes, the programmer focuses on the application logic, rather than on low-level implementation details. We present the programming language constructs supporting virtual nodes, exemplify their use, and show that they can be implemented by making efficient use of communication resources.

## Categories and Subject Descriptors

C.2.2 [**Network Protocols**]; D.2.11 [**Software Architectures**]

## General Terms

Languages

## Keywords

Wireless Sensor Networks, Abstractions, Middleware

## 1. INTRODUCTION

Wireless sensor networks (WSNs) are now becoming a cost-effective solution to monitor the physical environment. Mainstream WSN architectures are characterized by a single sink collecting data reported by sensors. However, there is a growing interest in scenarios where the WSN also affects the environment through *actuator* nodes [1]. These applications focus on a control loop where inputs are the *data sensed* in a given area of the system, and outputs are the *actions to be executed* in a possibly different area. For instance, in a building fire control system [3], illustrated in Figure 1, sensors and actuators collaborate to perform the following:

***Task 1*** *Actuator nodes controlling water sprinklers monitor the values of temperature and smoke sensors deployed nearby (e.g., in the same room).*

***Task 2*** *When temperature and smoke sensors collectively report values above a safety threshold, actuator nodes* i) *operate the attached water sprinklers in the same room as the sensors, and* ii) *trigger the actuators operating the emergency signals on the same floor to indicate a safe exit.*

Implementing this kind of control loop in a centralized fashion is generally impractical [1]. Therefore, these systems employ a decentralized coordination of the *sensing* and *acting* activities. This improves performance but increases complexity. The mainstream programming frameworks are too low-level and force the programmer to deal with the details of data gathering, bookkeeping, and communication, instead of focusing on the application logic. Higher-level programming abstractions are needed to deal with the complexity of decentralized WSNs.

In this paper we address this issue by introducing *virtual nodes*. Our programming constructs enable one to access the data sensed by a given set of sensors as the reading of a single *virtual sensor*, whose value is determined by an application-specified aggregation function. Dually, one can distribute commands to multiple actuator nodes from a single entry point, constituted by a *virtual actuator*. Virtual nodes allow the programmer to focus on the application logic by treating multiple nodes as one, therefore hiding the complexity of communication and data management.

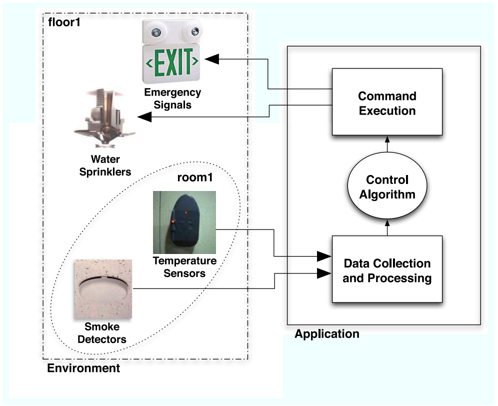


**Figure 1: A building fire control application. The control algorithm maps sensed inputs to output commands. The sensing and acting tasks insist on different parts of the system.**

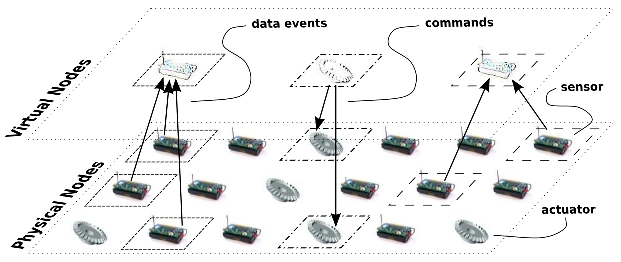


**Figure 2: The flow of information between real devices and virtual sensors or actuators. Nodes belonging to the same logical neighborhood are identified with the same dashed lines.**

The set of nodes being virtualized is specified by the programmer, using the notion of *logical neighborhood* we developed in our previous work [11, 12] and concisely summarized in Section 2. Logical neighborhoods replace the physical neighborhood provided by wireless broadcast with a higher-level, application-defined notion of proximity. The span of a logical neighborhood is specified *declaratively* based on the characteristics of nodes, along with requirements about communication costs. The interplay of virtual nodes and logical neighborhoods is illustrated in Figure 2. Virtual sensors enable data collection *from* the sensors belonging to a given logical neighborhood (e.g., temperature sensors in a given room), while virtual actuators provide a facility for sending commands *to* the nodes in a logical neighborhood (e.g., emergency signals on a given floor). Therefore, the expressive power of virtual nodes is further enhanced by the ability to limit their scope to an arbitrary subsets of nodes with desired characteristics. The key features of the virtual node programming abstraction are illustrated in Section 3.

We built an implementation of virtual nodes using TinyOS [6]. Our programming language constructs are translated in nesC code, which is then executed natively. As discussed in Section 4, this is achieved by extending the translator for the SPIDEY language [12] supporting logical neighborhoods. From a communication standpoint, our current implementation assumes that the node requesting the instantiation of a virtual node (i.e., its *anchor*) is also the one managing communication and data collection. The message routing supporting virtual nodes could rely directly on the routing protocol for logical neighborhoods we described in [11]. However, in Section 5 we show how this strategy can be improved for supporting virtual nodes, and substantiate our claims about the overhead improvement with preliminary simulation results.

As mentioned above, logical neighborhoods provide the conceptual and implementation foundation for the abstractions proposed in this paper. Therefore, they are described next.

## 2. LOGICAL NEIGHBORHOODS

The transmission range determines the devices a node can communicate with, i.e., its *physical* neighborhood. Instead, the nodes included in a *logical neighborhood* are specified by the programmer based on their characteristics. Programmers can therefore still reason in terms of neighboring relations, but retain control over how these are established, independently of the number of physical hops between the nodes. Logical neighborhoods are defined using a declarative language we designed, called SPIDEY. This is conceived to be an extension (not a replacement) of existing programming languages, and is not bound to any specific programming framework. Our current language and run-time implementation is based on TinyOS [6].

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

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

**Figure 3: SPIDEY: 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
```

**Figure 4: SPIDEY: neighborhood definition and instantiation.**

### 2.1 Abstraction

The definition of logical neighborhoods is based on two concepts: *nodes* and *neighborhoods*. Nodes represent the portion of a real node's features made available to the definition of any logical neighborhood. The definition of such a (logical) node is encoded in a *node template*, which specifies a node's exported attributes. This template is then used to derive actual instances of logical nodes, by specifying the actual source of data. Figure 3 reports a fragment of SPIDEY code that defines a template for a generic sensor and instantiates one logical node by binding attributes to constant values or functions of the target language.

A logical neighborhood can be defined using predicates on node templates. Analogously to nodes, a neighborhood is first defined in a template that basically encodes the corresponding membership function, and then instantiated by specifying *where* and *how* the neighborhood is to be constructed and maintained. For instance, Figure 4 illustrates the definition of a neighborhood including temperature sensors whose readings are above a given threshold. This template is then instantiated so that it evaluates the corresponding predicates only on nodes that are at a maximum of 2 (physical) hops away from the node defining the neighborhood, and by spending a maximum of 30 “credits”. The latter is an application-defined measure of communication costs, which exposes the trade-off between accuracy and resource consumption up to the application. The more credits are attached to a logical neighborhood, the higher is the *coverage* of the system as well as the *resources* spent to achieve that coverage. Other features include the use of boolean and set operators to compose predicates and templates. More details in [12].

Sending messages to a logical neighborhood is accomplished through a modified version of the native broadcast communication primitive, as in `send(Message m,Neighborhood n)`, supported by a dedicated routing protocol concisely described next.

### 2.2 Routing

Our routing scheme for logical neighborhood is structure-less (i.e., there are no overlays explicitly built) and works by periodically advertising *node profiles* containing node templates or portions thereof. These build a distributed *state space* that provides each node with enough information to reach the closest node with

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

create node vss from Sensor
  Function as "virtualSensor",
  Type as "smoke",
  Reading as average(roomSmokeSensors) every 60

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

**Figure 5: Definitions of virtual sensors.**

given attributes. Advertisements are limited to small area of the system by exploiting the redundancy among similar node profiles.

Messages addressed to a logical neighborhood contain the neighborhood template, thus making explicit the part of the state space that must be considered. This way, messages “navigate” towards potential neighborhood members by following paths along which the cost associated to that portion of the state space is decreasing. Local minima in the state space are avoided by propagating messages also in non-decreasing directions, in the hope of finding new decreasing paths towards different neighborhood members. The credits specified when instantiating a neighborhood are attached to each message and “spent” in navigating the state space. The routing protocol and its performance are illustrated in detail in [11].

## 3. VIRTUAL SENSORS AND ACTUATORS

We now describe the programming abstractions that constitute the main topic of this paper.

### 3.1 Virtual Sensors

Let us recall the fire control example of Section 1. The data collection necessary to monitoring (Task 1) can be encoded using the SPIDEY constructs we discussed thus far by defining two neighborhoods `roomTempSensors` and `roomSmokeSensors` including temperature sensors and smoke detectors in a room, respectively. However, the burden of collecting and processing data coming from sensors in these neighborhoods involves non-trivial, error-prone messaging and bookkeeping code, whose development still rests on the programmer’s shoulders.

We simplify the programmer’s chore by enabling her to instantiate two *virtual sensors*, as in Figure 5. Each of these virtual sensors behaves as a normal one, but reports as its reading an aggregation of the actual sensor readings in the specified neighborhood. Note how we defined these virtual sensors from the very same `Sensor` template we defined earlier in Figure 3. Simply, in the **as** clause we bind the attribute `Reading` to a different, distributed data source. The latter is represented by an aggregation function—`average()` in our case—operating over a logical neighborhood passed as a parameter, e.g., either of the aforementioned temperature or smoke neighborhoods. The programmer can also specify the rate at which data should be gathered from the nodes, using the **every** clause.

Clearly, to use virtual sensors the programmer inevitably needs to encode the aggregation semantics into an appropriate function that embodies the processing to derive the aggregated measure, as we have shown in `average`. However, this is done at a much higher level of abstraction, where the programmer does not deal explicitly with distribution aspects and keeping track of collected data.

```
node template Actuator
  static attribute Function
  static attribute Type
  static attribute Location
  dynamic attribute BatteryPower
  operation Activate(int tuning)
  operation Deactivate()

create node ws from Actuator
  Function as "actuator"
  Type as "waterSprinkler"
  Location as "room1"
  BatteryPower as getBatteryPower()
  Deactivate() as tuneSprinklerFlow(0)
  Activate(int tuning) as tuneSprinklerFlow(tuning)
```

**Figure 6: Definition and instantiation of an actuator node.**

```
neighborhood template RoomSprinklers()
  with Function = "actuator" and
       Type = "waterSprinkler" and
       Location = "room1" and
       provides(Activate(int tuning) and
       provides(Deactivate())

create neighborhood rs from RoomSprinklers()
```

**Figure 7: A neighborhood including water sprinklers that can be activated.**

### 3.2 Virtual Actuators

In our example, when Task 1 reports values beyond a safety threshold, Task 2 activates water sprinklers (in the same room) and emergency signals (on the same floor). Again, these two sets of nodes can be addressed by defining two neighborhoods `roomSprinklers` and `floorEmergencySignals` including the desired devices. However, the programmer is required to explicitly implement the distributed processing to control these nodes, e.g., by broadcasting to the neighborhood messages containing opcode and parameters for the operation to be activated.

*Virtual actuators* simplify the control of multiple nodes, as shown in Figure 6. When used for actuators, node templates also define the *operations* a node exports to applications. Similar to attributes, operations (e.g., `Activate`) are bound to a function of the target language. A remote invocation on the exported operation triggers a local invocation of the bound function, with the same parameters. Accordingly, neighborhood templates may now also include operations: the operator **provides** yields true when evaluated on a node that exports the operation given as a parameter. This enables the definition of neighborhoods based not only on the state of target nodes, but also on the operations they provide. An example with a neighborhood containing actuators controlling water sprinklers is given in Figure 7.

### 3.3 Hierarchies of Virtual Nodes

Once instantiated, virtual nodes can be used like normal ones. This enables the programmer to recur the process and create hierarchies of nodes, as shown in Figure 8. In our example, the two virtual sensors of Figure 5 could be used to define a single “virtual fire detector”, as shown in Figure 9. Here, we define a new neighborhood for temperature or smoke, containing only virtual sensors. If needed, the latter can be distinguished by the value of their `Function` attribute. Then, we create a virtual fire detector node `vfd` whose reading is a boolean stating whether a fire is detected, based on the evaluation of the aggregation function `checkFire` over the previously defined neighborhood. Using hierarchies, an application can detect the presence of a fire or trigger the appro-

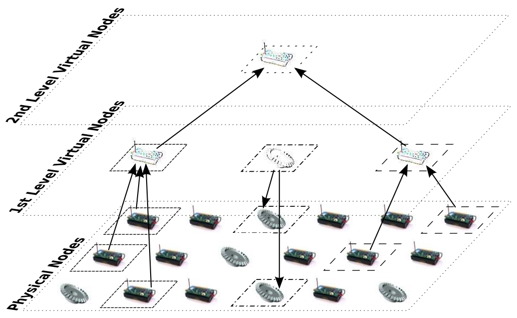


**Figure 8: A hierarchy of virtual sensors. Physical sensors are at the bottom.**

```
neighborhood template VirtualTempSmokeSensors(room)
  with Function = "virtualSensor" and
       Location = room and
       (Type = "temperature" or Type = "smoke")

create neighborhood vtss
  from VirtualTempSmokeSensors(room : "room1")

create node vfd from Sensor
  Function as "sensor",
  Type as "fireDetector",
  Reading as checkFire(vtss) every 60
```

**Figure 9: Hierarchies of virtual sensors: a fire detector example. (`checkFire` is a function that yields `true` if at least a temperature sensor and a smoke sensor report a reading above a safety threshold).**

priate reaction by interacting only with the topmost virtual node. An application component may not even be aware that it is actually communicating with a virtual node. Programming is therefore simplified, as a lot of the details concerned with communication, data collection, and operation invocation are dealt with automatically by SPIDEY and the associated run-time, described next.

## 4. LANGUAGE SUPPORT

Language support for virtual nodes builds upon the SPIDEY language developed for logical neighborhoods. We extended the SPIDEY-to-nesC translator to support the constructs described in this paper.

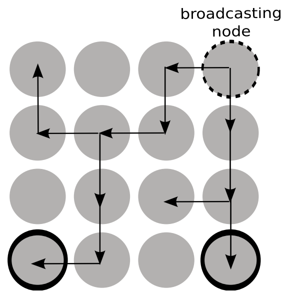
Our modified SPIDEY translator automatically creates a programming entity (e.g., a nesC component in TinyOS) whose interface provides a way to access the aggregated attributes defined in the virtual sensor. For instance, in the case of the virtual smoke detector `vss` we defined in Figure 5, our translator generates a component *providing* (in the nesC sense) the following interface:

```
interface vssInterface {
  event result_t Reading(int value);
}
```

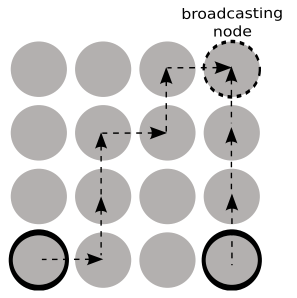
The implementation, automatically generated, signals the `Reading` event containing the average temperature every 60 seconds (as specified in Figure 5). Note how this approach is generally feasible in common programming languages for WSNs (e.g., [4]), as they all provide some notion of triggered event or periodic behavior.

A similar translation is performed for neighborhoods involving predicates over operations, used with actuators. For instance, our translator generates for the virtual actuator `rs` in Figure 7 a component providing:

```
interface rsInterface {
  command result_t Activate(int regulation);
  command result_t Deactivate();
}
```



(a) A message broadcast to the logical neighborhood propagates towards its members (in bold).



(b) A tree is built on the reverse path.

**Figure 10: Building trees from neighborhood members to a given node.**

An application gains access to the desired features of virtual nodes by listing the corresponding components in a nesC configuration, and using (in the TinyOS sense) their interfaces—as with any other component. However, this configuration is generated automatically by our translator, based on SPIDEY declarations.

Finally, note how the duality between virtual sensors and actuators is mirrored in the relationships between nesC events and commands. Events allow data to flow from the hardware up to the application, whereas commands enable the application to control the hardware. Virtual nodes build upon this duality in a distributed setting. Virtual sensors enable data to be gathered from physical sensors and be aggregated while it flows upward. These data is therefore made available as periodic *events*, as in `vssInterface`. Conversely, virtual actuators allow control to flow downward towards the physical hardware facilities, by sending *commands* to the physical actuators, as in `rsInterface`.

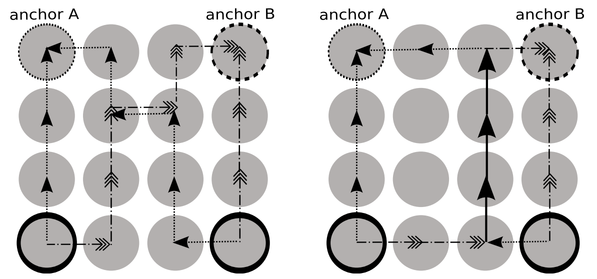
## 5. ROUTING FOR VIRTUAL NODES

In this section we discuss how to implement efficiently our abstraction from a communication standpoint. In principle, virtual nodes can be implemented in many ways, (e.g., using replication or probabilistic methods). Here, we assume that the node (called *anchor*) creating the virtual node is also the one managing the related communication and data collection. In the case of virtual sensors, this does not allow in-network aggregation. However, not every aggregation function can be computed in a distributed fashion. The approach proposed here is independent of the aggregation semantics, and can therefore accommodate complex data processing mechanisms regardless of their peculiar properties. Devising dedicated routing mechanisms to enable in-network processing is in our immediate research agenda, and is discussed further in Section 7.

### 5.1 Leveraging off Logical Neighborhoods

Logical Neighborhoods provide a first viable solution to the implementation of virtual nodes. In addition to the `send` operation, the logical neighborhood run-time provides a `reply` operation a neighborhood member can use to send a message back to the broadcasting node. Figure 10(a) illustrates this. During message propagation, whose details are in [11], each node routing a message stores a pointer to the physical neighbor the message came from. Each of these reverse paths leads to the sender; collectively, they constitute a tree with the broadcasting node as root and the neighborhood members as leaves, as shown in Figure 10(b).

This base routing for logical neighborhoods is enough to support



(a) Initial situation: two trees corresponding to two virtual sensors are built independently.

(b) Final situation: our adaptive algorithm changed the trees, increasing their overlapping.

**Figure 11: A sample execution of our adaptive algorithm that mutates multiple trees to increase the number of overlapping links. (Neighborhood members are circled in bold).**

virtual nodes. An operation on a virtual actuator generates a message with the appropriate opcode and parameters, broadcast to the logical neighborhood using `send`. Conversely, in virtual sensors data flows backwards from neighborhood members to the anchor. Thus, `send` is used only to activate the data collection process. After receiving this control message, neighborhood members periodically report their readings to anchors using `reply`. The code for packing the messages on the sender, and handle them on the receiver by invoking the appropriate operations, is generated automatically by our translator.

The aforementioned design provides an immediate implementation path for our abstraction. In the case of virtual sensors, however, a more efficient solution can be devised. This is described next.

### 5.2 More Efficient Data Collection

In the presence of multiple virtual sensors, it is likely that their associated neighborhoods overlap and therefore data sources must report the same reading to multiple anchors. With the aforementioned use of `reply`, each message from a data source would travel independently, wasting a sensible amount of network bandwidth. We devised a simple adaptive scheme that periodically "mutates" the branches of the trees to increase the number of overlapping paths—and therefore reduce overhead. Our protocol overhears messages sent by neighboring nodes, and performs local adjustments if a better route is found, based on a predefined cost metric. Many metrics can be used. Here we consider simply the number of overlapping tree branches.

The problem and its solution are illustrated in Figure 11. Figure 11(a) shows a scenario before adaptation, where the trees used for reporting data to anchors are largely disjoint. As messages flow from the neighborhood members to anchors, carrying data and metric information, each node along the path seeks opportunities for changing the tree and minimize the metric. As illustrated in Figure 11(b), the nodes in the middle detect that an increase in the number of overlapping paths can be obtained by merging two parallel tree branches, and change their tree parent accordingly.

This strategy enables the nodes on the shared path to pack in a single message readings from multiple sources, reducing the communication costs. Indeed, in WSNs this cost is determined mostly by the activation and setup of the wireless channel [14]. By packing multiple readings in a single message, reducing the overall number of messages, the impact of this cost is reduced. Moreover, the message header is also "wasted" only once for multiple readings.

### 5.3 Simulation Results

To evaluate the effectiveness of the above approach, we performed an initial evaluation using TOSSIM [8]. We defined a set of synthetic scenarios[1] with a variable number of nodes placed 35 ft apart and with a communication range of 50 ft. Four virtual sensors (anchors) are placed randomly in the system, each gathering data from the same neighborhood. Each neighborhood includes 10% of the nodes in the system, and its members are configured to sense and send a reading to each anchor every 60 s. Each simulation lasted 2000 s, and was repeated 5 times.

The difference in message delivery (i.e., number of sensor readings received by anchors) between the two approaches is negligible, with the adaptive routing performing a little better. Nevertheless, as shown in Figure 12(a), the adaptive routing significantly reduces the cost in terms of messages forwarded at the network layer, with higher improvements as the scale increases. Indeed, the base solution is often forced to send readings in separate messages routed independently. Instead, the adaptive one leverages the greater overlapping among trees, and can exploit the same physical link to deliver readings to multiple anchors using less messages. A quantification of this ability is shown in Figure 12(b), where we compare the overall number of physical links exploited by the two approaches. The adaptive routing relies on only half of the physical links used by the base approach. Remarkably, this measure exhibits the same trends of Figure 12(a), confirming that the overhead reduction is indeed enabled by the link overlapping.

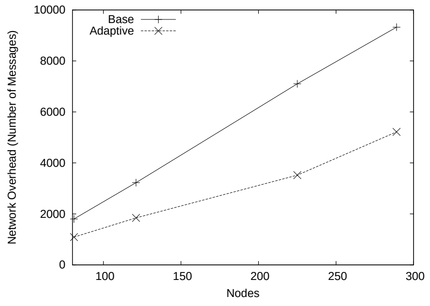
Figure 12(c) further analyzes the number $\tau$ of trees insisting on each physical link, by showing the ratio $\tau_{adaptive}/\tau_{base}$. As expected, the average number of trees insisting on the same physical link is significantly incremented using our adaptive technique. For instance, with 225 nodes the number of links shared among four trees is over 6 times more than in the base protocol. Interestingly, Figure 12(c) shows also that as the network size increases (while anchors do not) it is no longer convenient to merge different tree branches, as the trees could be too far from each other.

## 6. RELATED WORK

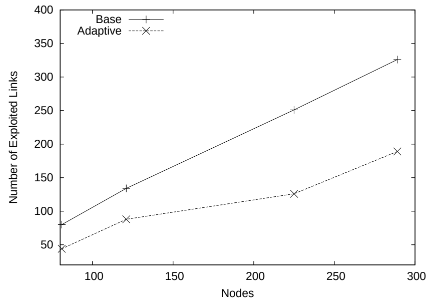
The closest relationship is with Regiment [13], a functional macro-programming framework based on Abstract Regions [16]. Programmers apply an application-defined function to data in a *region*, while regarding space and time as first-class data types combined to obtain complex streams of values. This macro-programming approach completely abstracts away the behavior of single nodes. However, it also loses fine-grained control over single nodes, e.g., to control a specific actuator. In this sense, our work strikes a balance between macro- and node-level programming, by giving the ability to perceive multiple nodes as a single virtual one, without loosing the ability to address specific real (or virtual) nodes. Moreover, our work relies on the more general notion of logical neighborhoods to partition the system, while Abstract Regions require a dedicated implementation for each region needed.

TAG [9] and TinyDB [10], along with extensions like [15], provide data aggregation through a SQL-like interface. The programming model is inherently data-centric and focuses on data aggregation, entirely lacking constructs to deal with actuators. These proposals enable in-network processing by assuming an a-priori deployment of all aggregation operators on every node. Our work does not require a system-wide deployment of aggregation functions, making the addition of new operators easier, although it may miss routing optimizations as discussed in Section 7.

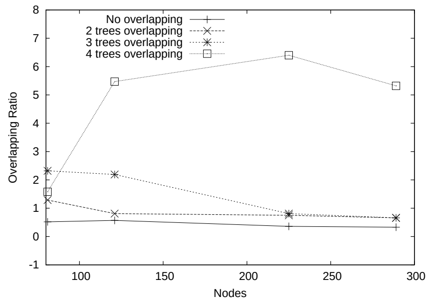
[1]We used the TinyOS' `LossyBuilder` to generate topology files with transmission error probabilities taken from real testbeds.



(a) Network overhead (forwarded messages).



(b) Number of links exploited.



(c) Ratio of overlapping tree branches.

**Figure 12: Comparing the performance of routing for virtual sensors: base (logical neighborhood) vs. adaptive routing.**

At the routing layer, efficient data aggregation is a topic of intense investigation. The work in [2] studied the placement of aggregation operators to minimize network traffic. Instead, the work in [5] studied, from a theoretical standpoint, the maximum rate at which sensor readings can be processed and communicated to a sink. Data aggregation in the presence of multiple, mobile sinks is investigated in [7] using an adaptive shared tree topology. The problem we in this work is different from the ones above. We do not take mobility into account, as we target systems deployed in controlled environments, e.g., buildings. Moreover, we consider scenarios with both multiple data sources (neighborhood members) and multiple sinks (anchors). However, we are evaluating if some of the techniques above can be borrowed and adapted to our goals.

## 7. CONCLUSIONS AND FUTURE WORK

We presented a set of programming abstractions that allow a programmer to interact with several nodes (specified in a declarative way) as if they were a single *virtual node*. We maintain that this approach provides the expressive power and flexibility necessary to program reactive and decentralized WSN applications. This is achieved by relieving programmers from the details of data collection, allowing them to focus on the application logic. We provided details about the implementation of our approach, and showed an initial evaluation of its performance.

Additional improvements can be obtained for virtual sensors by pushing the evaluation of aggregation functions closer to neighborhood members, as in [9]. However, this is not always feasible, and often requires intrinsic knowledge on the mathematical properties of the aggregation function. We are currently investigating ways to enable the programmer to specify the mathematical properties of aggregation functions, so that the compiler can select among the most appropriate routing protocol available in the run-time. Finally, we are investigating the trade-offs of alternative implementations that do not rely on anchor nodes.

## 8. REFERENCES

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

[2] B. J. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *Proc. of the $2^{nd}$ Int. Workshop on Information Processing in Sensor Networks (IPSN)*, 2003.

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

[4] A. Dunkels, O. Schmidt, and T. Voigt. Using protothreads for sensor node programming. In *Proc. of the Workshop on Real-World Wireless Sensor Networks (REALWSN)*, 2005.

[5] A. Giridhar and P. R. Kumar. Computing and communicating functions over sensor networks. *IEEE Journal on Selected Areas in Communications*, 23(4):755–764, 2005.

[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*, pages 93–104, 2000.

[7] K. Hwang, J. In, and D. Eom. Distributed dynamic shared tree for minimum energy data aggregation of multiple mobile sinks in wireless sensor networks. In *Proc. of $3^{rd}$ European Wkshp. on Wireless Sensor Networks (EWSN)*, 2006.

[8] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proc. of the $5^{th}$ Symp. on Operating Systems Design and Implementation (OSDI)*, pages 131–146, 2002.

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

[10] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

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

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

[13] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *Proc. of the $1^{st}$ Int. Wkshp. on Data management for sensor networks*, 2004.

[14] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, 2000.

[15] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proc. of the $2^{nd}$ Int. Conf. on Embedded networked sensor systems (SENSYS)*, 2004.

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