Virtual Sensors: Abstracting Data from Physical Sensors

Sanem Kabadayi, Adam Pridgen, and Christine Julien
The Center for Excellence in Distributed Global Environments
The Department of Electrical and Computer Engineering
The University of Texas at Austin
{*s.kabadayi, atpridgen, c.julien*}*@mail.utexas.edu*

Abstract

Sensor networks are becoming increasingly pervasive. Existing methods of aggregation in sensor networks offer mostly standard mathematical operators over homogeneous data types. In this paper, we instead focus on supporting emerging scenarios in which applications will need to extract abstracted measurements from diverse sets of sensor network nodes. This paper introduces the virtual sensors abstraction *that enables an application developer to programmatically specify an application's high-level data requirements. This paper reports on our initial work with the virtual sensors and the results of our prototype implementation.*

1. Introduction

In existing deployments of sensor networks, data collection schemes commonly require sensors to relay raw data to sink nodes to perform further processing. This is not very efficient considering the resource constraints (e.g., battery and bandwidth) of sensor networks. Furthermore, the throughput at each node decreases as the network scales, due to the broadcasting of redundant data. Sensor network aggregation mechanisms [4, 9, 12, 14] offer in-network data processing algorithms that are successful in limiting resource usage. However, these approaches support only standard mathematical operators (e.g., MIN, COUNT, and AVG) over homogeneous data types. The sensor networks of tomorrow will need to support localized cooperation of sensor nodes to perform complicated tasks and in-network data processing to transform raw data into high-level domain-dependent information.

Another challenge facing sensor networks is reusability. Current efforts create application-specific solutions, but the future will see multipurpose nets deployed to support numerous applications. The cost of physically visiting each sensor to reprogram it is prohibitive, and therefore the ability remotely reprogram sensor networks to tailor them to particular applications will be essential. This paper tackles exactly this challenge through the introduction of *virtual sensors*.

A virtual sensor is a software sensor as opposed to a physical or hardware sensor. Virtual sensors provide indirect measurements of abstract conditions (that, by themselves, are not physically measurable) by combining sensed data from a group of heterogeneous physical sensors. For example, on an intelligent construction site, users may desire the cranes to have safe load indicators that determine if a crane is exceeding its capacity. Such a virtual sensor would take measurements from physical sensors that monitor boom angle, load, telescoping length, two-block conditions, wind speed, etc. [10]. Signals from these individual sensors can be used in calculations within a virtual sensor to determine if the crane has exceeded its safe working load.

The middleware described in this paper implements a programming interface that enables applications to define tailored aggregation through virtual sensors. The power of virtual sensors lies in the fact that the physical sensors used by the virtual sensor may be heterogeneous (in the case of our example, angle and wind speed), and the virtual sensor can combine these different types of data to compute an abstract measurement. Another benefit of the virtual sensor is that it can be used to mask the *explicit* data sources (sensors) that provide data. A simple example would be a virtual position sensor that uses GPS when available on the local device but can switch to providing a position estimate based on the relative positions of other nearby (physical) location sensors if GPS is unavailable (e.g., inside a building).

The specific novel contributions of this work are twofold. First, we describe a new virtual sensor model designed to abstract data from physical sensors. This abstraction allows a developer to precisely specify the operations that are to be performed in the network over a set of data from different data sources. Second, we describe a prototype implementation of this middleware that includes the creation of virtual sensors enabling adaptive and efficient in-network process-

Proceedings of the 2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'06)
0-7695-2593-8/06 $20.00 © 2006 **IEEE**

ing that dynamically responds to an application's needs.

Section 2 of this paper examines related work. Then, Section 3 describes our virtual sensor model. In Section 4, we provide a detailed description of the middleware supporting this model. In Section 5, we discuss an example of specifying a virtual sensor. Section 6 concludes.

2. Related Work

Several recent research efforts have focused on simple in-network data aggregation techniques. Projects targeted directly for sensor networks have often explored representing the sensor network as a database. Two demonstrative examples are TinyDB [9] and Cougar [14]. Generally these approaches enable applications with data requests that flow out from a central point (i.e., a base station) and create routing trees to funnel replies back to this root. These approaches focus on performing intelligent in-network aggregation and routing to reduce the overall energy cost while still keeping the semantic value of data high. In both approaches, data aggregation is specified using an SQL-like language. Queries cannot be used to merge different data types, i.e. only homogeneous data aggregation is possible. In contrast, the virtual sensors approach offers simple programming interface, supports multiple access points, and offers raw and heterogeneous in-network data processing.

Compared to these approaches, directed diffusion [6] is a less centralized data-centric system. It is based on attribute-based naming and filtering to access sensor network data from multiple points inside or outside the network. Directed diffusion uses filtering for in-network data aggregation. A disadvantage of directed diffusion is that it requires gradients to be set up from all the sources (that will be used in an aggregation) to the sink that requests this information.

TinyLIME [2] offers the option for continuous, periodic sampling of data over which an aggregation can be computed, in addition to on-demand access to data. However, TinyLIME provides only single-hop connections to sensors and assumes that the sensors do not communicate among themselves. This effectively places all of the burden of aggregation on the shoulders of an application running on a more powerful device (e.g., a laptop) that is immersed in the sensor network.

The Sensor Model Language (SensorML) [11] provides an XML schema for defining the geometric, dynamic, and observational characteristics of a sensor. The language introduces a "sensor group" composed of multiple sensors that together provide a collective observation. However, SensorML is targeted for satellite-based high-power sensor systems. Data processing is not defined very rigorously, and it is not very flexible with respect to accessing heterogeneous devices.

A more lightweight implementation designed specifically for wireless sensor networks is TinyML [13]. It follows some of the SensorML ideas that are built on XML and has the important concept of virtualizing physical components. TinyML places an interface on a gateway (a more powerful and standard system) to the outside that translates XML to and from the application specific-sensor network format. The sensor field proxy is implemented on this external interface, which is chosen to be TinyDB.

Our virtual sensor approach differs from those in SensorML and TinyML, in that it offers raw, heterogeneous in-network data processing. Our virtual sensor focuses on forming a generic device to convert heterogeneous data. Furthermore, the virtual sensors do not rely entirely on an infrastructure or on a powerful gateway at which to perform aggregation.

3. Virtual Sensor Model

A client application runs with the support of the *virtual sensor* abstraction. In this section, we first describe how a developer defines the application's data requests using the virtual sensors. We then detail how programs dynamically interact with data from virtual sensors through an intuitive programming interface.

Figure 1 depicts the connection to an application-defined virtual sensor (represented by the dashed blue ellipse) on a tower crane. This virtual sensor uses data from three physical sensors (represented by blue dots). The virtual sensor aggregates the information from these sources into a higher-level reading that represents the effective load on the crane.

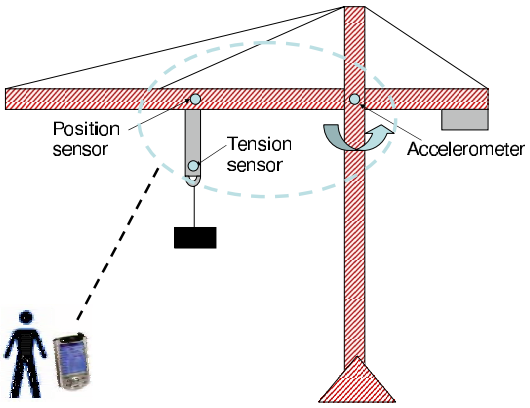

Figure 1. Connection from an application running on the user's device to the virtual sensor on a tower crane

The software implementing this virtual sensor may run on the user's handheld device or, to reduce network com-

Proceedings of the 2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'06)
0-7695-2593-8/06 $20.00 © 2006 IEEE

Table 1. `VirtualSensor` **API operations**

Operation	Description
`public VirtualSensor(DataType[] inputs, Aggregator a, DataType result, int aggfreq)`	Constructor for the `VirtualSensor`. The parameter `aggfreq` impacts how up-to-date the readings are.
`void query(ResultListener r)`	Sends a one-time query to the `VirtualSensor`. The result listener receives results from this query.
`int register(ResultListener r, int reqfreq)`	Registers a persistent query on the `VirtualSensor`. The request frequency, `reqfreq` indicates how often the application demands the data value from the virtual sensor. The method returns a receipt that can be used to cancel the registration when desired.
`void deregister(int receipt)`	Stops the registered query referenced by the receipt.

munication, be deployed to one of the sensors depicted. The mechanics of the infrastructure supporting this deployment are discussed in Section 4. First, we describe how programmers create virtual sensors through our middleware.

3.1. Declarative specifications of virtual sensors

In our model, several sensors required to supply the desired application-level data are encapsulated in an abstraction called a *virtual sensor*. This offers generality and flexibility and provides a higher level of abstraction to the application developer, in comparison to programming in nesC directly. A virtual sensor's declarative specification allows a programmer to describe the behavior he wants to create, without requiring him to specify the underlying details of how it should be constructed. This is especially important considering the fact that in an instrumented sensor network, a user's operational context is highly dynamic. While the actual data sources change over time based on the user's location and movement, the application's data needs do not change as much, and one of the benefits of a virtual sensor is that it hides the changes in data sources from the application. Our approach assumes applications and sensors share knowledge of a naming scheme for the low-level data types the sensor nodes can provide (e.g., "location," "temperature," etc.). These data types are determined by the types of sensors deployed in a network. The programmer, then, only needs to specify the following four parameters for the virtual sensor:

- Input data types: Physical (low-level) data types required to compute the desired abstract measurement.
- Aggregator: A generic function defined to operate over the specific (possibly heterogeneous) input data types to calculate the desired measurement.
- Resulting data type: The abstract measurement type that is a result of the aggregation.
- Aggregation frequency: The frequency with which this aggregation should be made. This frequency determines how consistent the aggregated value is with actual conditions (i.e., more frequently updated aggregations reflect the environment more accurately but generate more communication overhead.).

To present the dynamic virtual sensor construct to the application developer, we build a simple API that includes built-in general-purpose data types (e.g., temperature, location, angle, etc.) and provides a straightforward mechanism for developers to insert additional data types.

By providing these virtual sensor specifications, an application delegates sensor discovery to the virtual sensor (and to the framework that supports the virtual sensor). Therefore, if the data sources supporting the virtual sensor change over time, the virtual sensor adapts, but the application does not notice.

3.2. Interacting with virtual sensors

The types of queries enabled on a virtual sensor can be classified into *one-time queries* (which return a single result from the virtual sensor) and *persistent queries* (which return periodic results from the virtual sensor). To support these types, we provide two different methods for posing queries: `query()` and `register()`, respectively.

These operations and brief descriptions of their behavior are shown in Table 1. To provide the functionality described in the API, the `VirtualSensor` object keeps a list of live queries and a list of listeners as its private members. This allows a single virtual sensor to support multiple applications, in the same way that a single physical sensor can provide data for multiple applications. A virtual sensor is deployed only when there are active queries, and the information from the virtual sensor is accessed on-demand.

The virtual sensor query model also introduces a `ResultListener`, which is registered to receive the results of each query. When the result is ready, the middle-

Proceedings of the 2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'06)
0-7695-2593-8/06 $20.00 © 2006 IEEE

ware calls the `resultReceived(QueryResult)` method for the result listeners to forward the results to the application. This achieves an asynchronous and nonblocking implementation, so that the application can perform other tasks in case no response comes back to the query or the response for the query is delayed. The query model uses the result listener for both one-time and persistent queries, the difference being that it calls the result listener only once in the former case and periodically (as dictated by the `reqfreq`) in the latter case.

4. Middleware Supporting Virtual Sensors

In this section, we describe the implementation of the infrastructure underneath the virtual sensors abstraction described above. We first give brief details of the top layers of this design, relating them to our discussion of the API in Section 3. Then, we discuss the communication infrastructure, sensor discovery, and virtual sensor deployment. Figure 2 depicts the middleware's simplified object diagram.

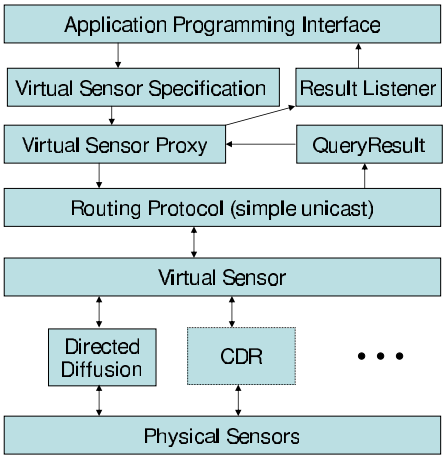

Figure 2. Simplified object diagram for the virtual sensor middleware

4.1. Application programming interface

When the application needs to query the sensor network for a data type that is not provided intrinsically by the physical sensors, the developer constructs and deploys a virtual sensor using his knowledge of the available data types. The application subsequently queries this virtual sensor directly.

If the virtual sensor happens to be running remotely, a remote handle to the listener needs to be set up. In such cases, the middleware creates a proxy for the virtual sensor on the user's device. This proxy object runs within our middleware and uses a unicast routing protocol to connect to the remote virtual sensor and collect the information desired by the application. When the query's result is ready, this proxy makes a callback to the user's result listener either once (for a one-time query) or periodically (for a persistent query).

4.2. Sensor discovery and communication with data sources

The application delegates discovery of physical sensors to a middleware that locates actual (physical) sensors based on the specified input data types. To be used by a `VirtualSensor`, a node must be able to supply at least one of the input data types specified in the creation of the virtual sensor.

Existing routing implementations such as directed diffusion [6] or CDR [7] can be used to achieve sensor discovery and communication. Different protocol implementations can be swapped into our middleware, as long as they provide the necessary data-centric communication interface. Abstractly, the communication protocol used by the middleware initially broadcasts a data type requirement across the local sensor network, and sensors that can provide that data type respond. Heuristics such as least latency, shortest path, etc., can be used to select a particular data source from many. This selection can be refreshed if the data source selected becomes unavailable. As a specific example, when the communication protocol in use is directed diffusion, the virtual sensor propagates an interest message for each of the data types it requires, creating gradients for funneling information back to the virtual sensor. A similar process is used in CDR, but, once a source is selected, unicast alone is used to communicate with it. The middleware uses the (consistency) frequency, `aggfreq`, to determine how often the sensor selections need to be to refreshed and whether or not to use CDR, directed diffusion, or some other available protocol. For one time queries (or long `aggfreq`s), the directed diffusion approach is not very efficient, since gradients have to be set up for queries which use the path only once [1]; in such cases CDR is used.

4.3. Deploying virtual sensors

As demonstrated by example in the next section, an application's high-level specification of a virtual sensor is translated into low-level code (written in nesC [3]) that provides the virtual sensor's functionality and communication and can run on TinyOS [5]. This code can then either run locally or be deployed to a resource-constrained sensor within the network. When deployed remotely, this code is dynamically received by a listener on the remote sensor and executed. While this approach does require a small amount of

Proceedings of the 2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'06)
0-7695-2593-8/06 $20.00 © 2006 IEEE

our (general-purpose) middleware to run on every sensor in the network, we believe this is a small price for dynamic reprogrammability.

The decision about where to deploy a virtual sensor is based on the expense of communicating with the physical sensors that it comprises. If all of the physical sensors are in a cluster, and that cluster is several hops away from the user's device, then it may make sense to send the virtual sensor out to the cluster. On the other hand, if each of the sensors that make up the virtual sensor is within one hop of the user, then the virtual sensor should run on the user's device. For now, our middleware enables applications to deploy virtual sensors remotely, but does not take a hand in making this decision on behalf of the application. Future work will create heuristics within the middleware for automatically determining when the virtual sensor should be deployed remotely and where it should optimally be placed. We conjecture that remotely deploying virtual sensors when appropriate will be more efficient (require less communication) than creating all of the communication flows back to the user's device. This savings is precious in sensor networks as the nodes that have to route the messages contain limited battery power.

5. Evaluating an Example Virtual Sensor

In this section we provide a simple example of the specification of a virtual sensor. As shown in Figure 2, this specification is provided to the middleware, which translates it into two components: the virtual sensor proxy and the virtual sensor. The former runs on the user's device; the latter is written in nesC and can be deployed to a sensor in the network. The virtual sensor we describe here allows the user to sense data of type `CraneDangerCircle` for nearby cranes. This circle represents the area nearby a crane where it is unsafe to walk and is centered at the base of the crane (which may move) and has a radius defined by the position of the boom (which is even more likely to move). See Figure 1 for reference. As the boom moves along the crane arm, the size of the danger circle should expand and contract accordingly. An application can use this information to maintain a map of the construction site to ensure vehicles and workers are always safe and to display warnings to a worker when he enters a danger circle.

This example has been simplified to make the explanation easier; other examples on the construction site like the one introduced in Section 1 require sophisticated physical calculations that muddle our discussion of the virtual sensor specification. In any case, the virtual sensor programmer (a domain expert) possesses the application knowledge to create a virtual sensor. Using our middleware, this programmer can use a high-level language to create tailored sensing capabilities.

Our example virtual sensor (`CraneVS`) uses two data types available in the sensor network: `BasePosition` and `BoomPosition`. This, too, is a simplification, as these data types may themselves be the result of a virtual sensor that aggregates basic `location` data with nearby identity data (e.g., from an RFID tag) to determine that a particular location sensor is located at the base of a crane. The `CraneVS` generates abstract data of the type `CraneDangerCircle` which is delivered to the application. The code the application programmer must write to construct such a sensor looks like:

```
VirtualSensor craneVS =
  new VirtualSensor({new BasePosition(),
                     new BoomPosition()},
                     new CraneAggregator(),
                     new CraneDangerCircle());
```

Within the application, `BasePosition`, `BoomPosition`, and `CraneDangerCircle` are data types that extend the `DataType` class. The application may have to create the `CraneDangerCircle`, but `BasePosition` and `BoomPosition` are likely to be common to the domain and therefore reusable across applications. In the constructor above, new instances of the classes representing the types are constructed as placeholders. The domain programmer must also specify the mechanics behind the aggregation within the `CraneAggregator`. This is accomplished by implementing the `Aggregator` interface and providing an implementation of the `aggregate()` method:

```
class CraneAggregator implements Aggregator {
  CraneDangerCircle aggregate(DataType[] inputs){
    int radius_squared = (input[0].x - input[1].x) *
                         (input[0].x - input[1].x) +
                         (input[0].y - input[1].y) *
                         (input[0].y - input[1].y);
    return new CraneDangerCircle(input[0],
                                 radius_squared)
  }
}
```

The middleware supporting virtual sensors translates this complete specification into nesC. For this reason, we require the aggregation function to be written in terms of simple arithmetic operations (i.e., no square or square root functions). Future work will include translation of more complex operations. The virtual sensor created to support this specification does two things. First, it calculates the square of the radius as specified by the function. Second, it returns (in a single message) values for anything referenced in the return statement (i.e., the calculated `radius_squared` and the (x,y) coordinates of the base of the crane, as dictated by the use of `input[0]` in the return statement). The virtual sensor proxy, running on the user's device, post-processes the message returned to encapsulate it as the object the application expects (e.g., the

Proceedings of the 2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'06)
0-7695-2593-8/06 $20.00 © 2006 IEEE

`CraneDangerCircle` in the example) and invokes the application's registered `ResultListener`. In this virtual sensor, the circle is specified by its center (the location of the crane base) and its radius. We return the square of the radius for now (due to the difficulty of using a non-primitive operation) but intend to revisit this issue either in the post-processing in the proxy or by deploying additional operations with the virtual sensor (as done in Maté [8]).

The only difference in our initial implementation of the virtual sensor from that described in the previous section is that it uses basic broadcast for data types (instead of a more sophisticated mechanism based on directed diffusion or other communication protocols). These broadcasts request data from sensors of the appropriate types, receive the information, then process it according to the virtual sensor's `Aggregator`, which has been translated to nesC as described above. The virtual sensor code translated into nesC for this example occupies 15KB in ROM when compiled for TinyOS, which is comparable with other applications written for this operating system. Future work will include further measurements about the benefit of building sophisticated virtual sensors and the performance characteristics in terms of overhead and latency with respect to alternatives.

6. Conclusion

In this paper, we have described virtual sensors that allow measurements of abstract data types. Virtual sensors abstract a set of physical sensors and the operations that are performed on them, providing a new way of extracting data from heterogeneous wireless sensors. The separation of the specification of the sensing task from the sensing behavior allows a programmer to describe the behavior of a virtual sensor, without having to specify the underlying details of how it should be constructed. Virtual sensors also offer a way to tailor a generic sensing environment to specific applications. This will be especially necessary as sensor networks become more widespread and general purpose.

Future work will include a complete network performance evaluation that measures query response times, energy usage, and overall communication overhead. Other future work will include "mobile" virtual sensors, i.e., virtual sensors that can move with an event of interest or in response to the user's movement.

References

[1] J. N. Al-Karaki and A. E. Kamal. Routing techniques in wireless sensor networks: A survey. In *IEEE Wireless Communications*, pages 6–28, Dec. 2004.

[2] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. Murphy, and G. Picco. TinyLIME: Bridging mobile and sensor networks through middleware. In *Proc. of the 3rd Int'l. Conf. on Pervasive Computing and Communications*, pages 61–72, 2005.

[3] 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*, pages 1–11, 2003.

[4] J. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Toward sophisticated sensing with queries. In *Proc. of the 2nd Int'l. Workshop on Information Processing in Sensor Networks*, pages 63–79, 2003.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[6] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heideman, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, February 2003.

[7] C. Julien and M. Venkataraman. Resource-directed discovery and routing in mobile ad hoc networks. In *Technical Report, TR UTEDGE-2005-001, Center for Excellence in Distributed Global Environments, The University of Texas at Austin*, 2005.

[8] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. of ASPLOS X*, 2002.

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

[10] L. Neitzel, S. Seixas, and K. Ren. A review of crane safety in the construction industry. *Applied Occupational and Environmental Hygiene*, 16(12):1106–1117, 2001.

[11] Sensor Model Language (SensorML). `http://vast.uah.edu/SensorML`, 2005.

[12] N. Shrivastava, C. Burgohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. In *Proc. of the 2nd Int'l. Conf. on Embedded Networked Sensor Systems*, pages 239–249, 2004.

[13] TinyML: Meta-data for wireless sensor networks. `http://dnclab.berkeley.edu/~nota/research/TinyML/TinyML2.htm`, 2005.

[14] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record*, 31(3):9–18, 2002.

Proceedings of the 2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'06)
0-7695-2593-8/06 $20.00 © 2006 **IEEE**