Physicalnet: A Framework for Programming Mobility-aware, Cross-network, Concurrent Applications for Sensor and Actuator Networks

UNIVERSITY OF VIRGINIA, DEPT. OF COMPUTER SCIENCE, TECH. REPORT CS-2008-2
January 2008

Pascal A. Vicaire, John A. Stankovic
pascal@cs.virginia.edu, stankovic@cs.virginia.edu

Abstract

This paper describes the design, implementation, and evaluation of Physicalnet, a framework allowing the programming and concurrent execution of applications that span multiple sensor and actuator networks. Physicalnet uses a service oriented architecture to make node services globally accessible and interoperable. Within this architecture, dedicated processes list the services available at a given geographical site, allow the concurrent access to these services, and enforce specified user access rights. On top of this architecture, Physicalnet provides the Bundle Abstraction, which allows the concise manipulation of abstract sets of nodes with dynamic membership. Physicalnet has been especially designed with mobility in mind: nodes are globally localized and their function dynamically changes according to their location.

To evaluate the Physicalnet framework, we provide a complete implementation in TinyOS and Java. We deploy two geographically separated testbeds, each composed of PCs and MICAzs forming a multi-hop wireless network. We study 5 applications that span multiple testbeds and run them concurrently to demonstrate the utility of our solution. We conclude that programming is concise, and that execution speed meets application requirements.

1 Introduction

Early sensor network research focussed primarily on gathering sensor data both efficiently and reliably. More recent projects have identified novel directions for sensor network research. Some projects require sensors to be globally accessible [13]. Others require resource constrained sensors to interact with actuators and more powerful devices such as PCs, PDAs, and cellular phones. Some work requires sensor networks to support concurrent application execution [24]. Also, numerous research papers note that sensor networks are difficult to program and identify a need for powerful and flexible programming abstractions [19].

In this paper, we seek to satisfy these requirements simultaneously, within a unified sensor and actuator framework. Our vision is that of a world where many sites are equipped with one or more multi-hop, wireless networks of sensors and actuators. In our vision, the users of such networks want the following: 1) they want their resource-constrained wireless nodes to interact with more powerful Internet-connected devices such as PCs, PDAs, and cellular phones. For instance, in the case of a fire alarm application, if sensors detect a fire, users require an alert message to be displayed on all the PCs, PDAs, and cellular phones of the building. 2) Because user needs change as time passes, users want to be able to create and run novel applications that use the existing sensor and actuator infrastructure. 3) Users want to be able to create applications that span several independently deployed sensor networks. For instance, a police officer may want to monitor all the buildings of a university for possible intruders. 4) Users want to be able to run their applications concurrently and continuously, even if these applications use the same sensors and actuators. For instance, if user A wants to run a temperature monitoring application, user B a fire alarm application, and user C a temperature regulation application, all in the same building, the three users need an architectural framework that will alow them to use the same temperature sensors continuously and simultaneously. 5) The owners of the sensors and actuators want to be able to specify who can use their nodes and how. 6) Programmers need high level programming abstractions that allow them to manage large sets of sensors and actuators and their interactions.

Our solution for satisfying this set of requirements is Physicalnet. Physicalnet is composed of three key elements. First, Physicalnet uses a service oriented architecture to make node services globally accessible and interoperable. Second, within this architecture, dedicated processes called negotiators list the services available at a given geographical site, make possible the concurrent access to these services, allow the specification of user access rights, and enforce these access rights. Third, on top of this architecture, Physicalnet provides a programming abstraction called a bundle that allows the concise manipulation of abstract sets of nodes with dynamic membership. Physicalnet provides extensive mobility support: nodes are globally localized and, as our example applications reveal, the bundle abstractions is particularly useful to specify location aware applications that involve mobile nodes.

To evaluate Physicalnet, we created a complete implementation using Java and TinyOS. We deployed two geographically separated sensor and actuator networks composed of MICAzs, MTS310 sensor boards, and PCs. We provide code excerpts from 5 applications to demonstrate the conciseness of the Bundle programming abstraction. We provide results concerning application code size, installation time, and responsiveness to environmental stimuli. Our results show that programming is concise, and that execution time meets application requirements, even when multiple applications run concurrently.

The remainder of this paper is organized as follows. In Section 2, we give an overview of Physicalnet. In Section 3, we detail its implementation. In Section 4, we provide a qualitative and quantitative evaluation of the proposed framework. In Section 5, we discuss related work before concluding in Section 6.

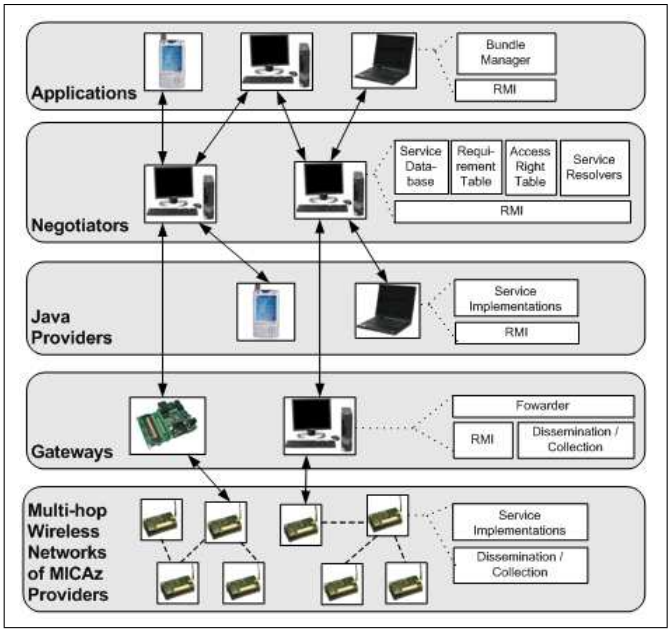

Figure 1. Physicalnet Architecture.

2 Overview of the Physicalnet Framework

Physicalnet allows two types of nodes to interact. The first type of nodes, the Java nodes, run a Java virtual machine. They have an IP address and can connect to the Internet. Examples of Java nodes include PDAs, desktop PCs, laptop PCs, Stargate gateways, and cellular phones. The second type of nodes, the TinyOS nodes, are relatively more resource constrained. They do not directly connect to the Internet but rather form a multi-hop wireless network, and use a multi-hop wireless networking protocol to communicate with a gateway, this gateway having access to the Internet. Examples of TinyOS nodes are MICAz nodes, Telos nodes, and XSM nodes. To give an idea of what we mean by "resource constrained", a MICAz node has a 7.37MHz processor, 4KB or RAM, 250kbps of bandwidth, and relies on two AA batteries for power supply. We now describe the three key elements of Physicalnet, which are represented in Figure 1.

2.1 A Service Oriented Architecture for Interoperability and Global Accessibility

The first key element of Physicalnet is its service oriented architecture that makes node services globally accessible and interoperable. The Physicalnet service architecture is composed of service providers, gateways, negotiators, and applications.

A provider process controls the services of the network node on which it runs. A service can for instance be the temperature sensor of a MICAz node, the beeper of a XSM node, a light actuator, the display screen of a PC, or the speakers of a PDA. A provider registers the services of its node with one and only one negotiator, and executes the commands issued by this negotiator.

A negotiator lists all the services that are available at a given geographical site in its service directory. It informs applications of the services that it lists, but only of those services that the application is allowed to access. A negotiator allows several application to access the same service concurrently and resolves possible conflicts. There is only one negotiator for each service, and this negotiator remotely controls the service according the needs of currently executing applications.

While Java service providers and applications communicate directly with negotiators using TCP/IP, TinyOS providers do so through a gateway. They communicate with the gateway using a multi-hop networking protocol, and the gateway relays provider packets to the the negotiator using TCP/IP.

At startup, a Physicalnet application connects to one or more negotiators. The application specifies the services it needs as well as the state these services should be in. As the application executes, these needs may vary depending on the current sensor and actuator states, and the application dynamically informs the negotiators of the changes in its requirements.

The Physicalnet architecture is different from other service oriented architectures such as Jini [3] because it is specifically targeted to networks of sensors and actuators. Recently, Arch Rock [1] undertook to expose sensor network functionality as Web Service. The Physicalnet service architecture is different in that it allows the concurrent access of sensors and actuators, and manages their access rights.

2.2 Negotiators for Resource Sharing and Access Right Enforcement

The second key element of Physicalnet are negotiators. Concurrent applications dynamically inform negotiators of the services they need, and request these services to be in a specific state. The negotiator first filters application requests by enforcing user access rights. Each application is associated with a user name. For each application, the negotiator looks up the access rights of the application user and determines whether the application has access to a given service, whether it has access to the location of the service, whether it can modify or read the states of the service, and whether it can listen to events generated by the service.

If several applications are allowed to access a service and concurrently formulate incompatible requirements, the negotiator reaction depends on a service specific resolver specified by the service owner. For instance, in the case of a temperature sensor, if application 1 requests a sampling period of 10s, and application 2 requests a sampling period of 15s, the resolver may select the minimum sampling rate so that all applications obtain a number of samples greater or equal to the number they requested. In the case of a light actuator, if application 1 requests the light to be on, and application 2 requests the light to be off, the resolver may only satisfy the highest priority request.

Some Physicalnet applications execute on a best effort basis i.e., even though they formulate specific requests for some services, they do not take action if these requests are not satisfied. For instance, an application may request samples from all the temperature sensors in a building and continue execution despite the fact that it does not have access to some of these sensors. Other Physicalnet applications may have more stringent requirements and need to verify that their needs are met at any time during execution. Consider the example of a fire alarm application that does not have access to the alarms! To verify that their needs are met, applications can dynamically inspect their access rights and check that their requests are satisfied. Applications can then take any programmatically specifiable action if they deem their access rights to be insufficient. For instance, they can terminate or display a warning message for the application user.

2.3 Concise Programming with Bundles

The third key element of Physicalnet is an abstraction called a bundle. A bundle is a set of service references. It is a generalization of previous abstractions that have been proposed in the

wireless sensor network literature. Like Abstract Regions [20] and Hoods [21], bundles can be used to refer to the nodes in a particular spatial zone. However, and this is the first originality of the bundle abstraction, the membership of a bundle can be much more complex. The membership of a bundle is described using a logical predicate written using the Java programming language. This means that the expression of the bundle membership can be arbitrarily complex, involving as many conditions as the programmer desires. Bundle membership can span multiple sites. Membership conditions can involve the service location, the service interface, the access rights to the service, and sensory data. Bundle membership can also depend on the characteristics of a service that is not part of the bundle, on another bundle, and on any application variable. For instance it is possible to create the bundle of all the light actuators that are in a room that contains either a motion sensor or an acoustic sensor that has been triggered in the last five minutes. Note however that, unlike Abstract Regions and Hoods, bundles cannot be used for low level sensor network programming. For instance, bundles cannot be used to program a networking protocol for sensor networks.

The second originality of the bundle abstraction is its dynamic aspect. The bundle membership is updated periodically so as to respect the membership specification. Programmers specify the state in which bundle members must be. For instance, a programmer can specify that all the members of a bundle must sense the temperature with a period of 2 seconds. If a service leaves the bundle, the state requirements of the application are automatically canceled. If a service joins the bundle, it automatically modifies its state according to application requirements.

3 Physicalnet Design and Implementation

In this Section, we describe the Physicalnet implementation, which is summarized in Figure 1.

3.1 Prerequisites

3.1.1 Implementation Languages

Physicalnet is entirely implemented using nesC, TinyOS, and Java. NesC and TinyOS are used to implement MICAz service providers. NesC is a language derived from C that has been designed especially for embedded programming. It allows programmers to create components and to define the relations among them. TinyOS is a tiny operating system that supports a lightweight and event driven computation model.

The negotiator code, the gateway code, the service provider code for the PC platform, and the Physicalnet programming API (including the bundle abstraction) are implemented in Java. The Physicalnet API uses many of the features of Java 1.5.0: generics, generic methods, and reflection. These language features allow us to make the API concise and practical. As the Physicalnet API is in Java, programming sensor and actuator networks using Physicalnet does not require programmers to learn a new, purpose specific language. Moreover, Java provides many features that facilitate code organization and conciseness such as object oriented programming and generics. It also allows programmers to create applications that use the Physicalnet API at the same time as some of the powerful Java libraries that are available. For instance, within the same program, it is possible to use Physicalnet to manipulate sensors, and Jini [3] to manipulate more traditional software services.

3.1.2 Networking Protocols

Negotiators, application, gateways, and PC service providers communicate using Java remote method invocation (RMI [5]) over TCP/IP. MICAz service providers form a multi-hop wireless network and use the collection and dissemination networking protocols available in the latest TinyOS 2.0 distribution to communicate with their gateway. The gateways forward the packets of MICAz providers to their negotiators, and the negotiator updates to the MICAz providers.

Let us give more details about collection and dissemination. Dissemination is an epidemic networking protocol that reliably transports data from the gateway to the MICAz providers. Collection builds a spanning tree allowing the MICAz providers to send acknowledgments, data, and events to the gateway. This spanning tree is rebuilt only when necessary. Collection and dissemination allow MICAz providers and gateways to reliably communicate without imposing a high overhead on the MICAz providers. One problem with dissemination is that it floods the whole wireless network even if a message must be sent to only one node. In the future, we plan to study in detail the kind of traffic that Physicalnet generates, and to find protocols with lower bandwidth usage that are still reliable and that still have low memory overhead.

3.2 Description of Physicalnet Operation

We summarize the interactions between Physicalnet processes in Figure 2. In this figure, each set of circles connected by light arrows represents a thread. Circles represent the states of the threads. Light arrows represent state transitions. Heavy arrows represent communications between processes. Double-circles represents the starting states of the threads. Dashed lines separate the processes (composed of several threads) of a MICAz (i.e. a TinyOS service provider), a gateway, a negotiator, and an application. Physicalnet operates as follows:

Beaconing of MICAz Provider: Service providers are registered with one (and only one) negotiator that manages their state. Providers have a global identifier of the form *Negotiator IP address:Negotiator TCP port:Local Identifier*. To signal their functioning, providers periodically send beacons to a gateway. The beacon contains the provider global identifier, the current location of the provider, an acknowledgement number for the last message received from the negotiator, and sampled values (such as temperature or photometric samples) if the provider is currently sensing.

Provider Beacon Forwarding by the Gateway: When a gateway receives a provider beacon, it reads the provider global identifier, infers the negotiator IP address and TCP port and forwards the beacon to the negotiator. The negotiator verifies that the provider is registered with it, updates the location of the provider in its database, and records the sampled values. The negotiator uses the acknowledgement numbers included in the beacons to infer whether the messages that command the provider to change state have been received.

Provider State Update Request by the Gateway: Periodically, the gateway asks the negotiators of the providers it recently heard of for state updates. The negotiators check the identifiers of the providers, read their database to infer whether provider state is inconsistent with application requirements, and returns required state updates to the gateway. The gateway forwards these state updates one by one to each provider using the dissemination sensor network protocol. Upon reception of an update, providers record the acknowledgment number that they will include in their next beacon, and launch tasks of which the execution is dictated by the modification of their state (for instance, they start sensing the temperature if the

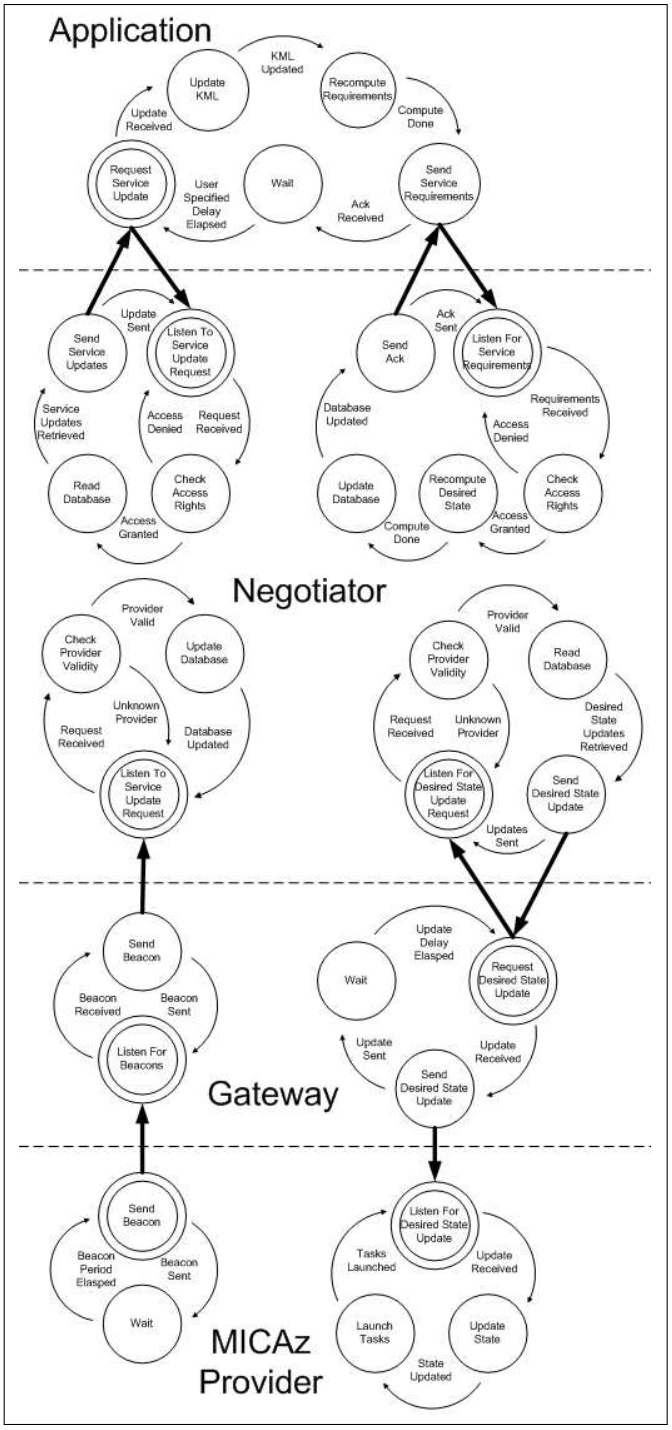

Figure 2. Physicalnet State Diagram.

sense temperature state is set to true).

Service Update Request by the Application: Periodically, applications request updates about the actual state of services from negotiators (e.g., is the sensor on? What is the location of the sounder?). With updated service information, applications update their KML (Keyhole Markup Language) output which allows the display of service information in Google Earth. Then, the application recomputes its service requirements. For instance, if the application programmer specified than the sounders in the bathroom should be on, the application redetermines which sounders are in the bathroom using the updated service locations received from the negotiator. The application then uploads its requirements to the negotiator. The negotiators verify the access rights of the application and, taking into consideration the requirements of all the applications, updates the desired service states.

Several Physicalnet operations are not represented in Figure 2. They include:

Java Provider Operations: A Java provider roughly behaves like a MICAz provider and a gateway that run in the same process. It contacts the negotiator directly to obtain desired state updates and upload samples.

Detection of Missing Providers by the Negotiator: If a negotiator does not receive provider beacons for a configurable duration, it assumes that the provider is turned off.

Garbage Collection of Applications by the Negotiator: Applications periodically send small heartbeat messages to inform the negotiator that they are still running. If the negotiator does not receive heartbeat messages for a configurable duration, it assumes that the application ungracefully terminated and removes all the application requirements from its database.

Garbage Collection of Samples by the Negotiator: Negotiators store provider samples (e.g., temperature samples) and communicate them to applications when they request service updates (which can happen very frequently such as every 200ms). Negotiators keep the samples in their memory only for a configurable amount of time after which the samples are erased.

Localization Beacons: To localize MICAz providers, we use MICAz localization beacon nodes. These localization beacons are configured with their latitude and longitude. They periodically broadcast their latitude and longitude. When a MICAz provider receives a localization beacon, it assumes that its location is equal to the location of the beacon. We set the beacon broadcasting power so that the beacon range does not exceed the dimensions of the room that contains it. With such a scheme we obtain a localization scheme that can determines in which room MICAz providers currently are located. In the future, we plan to integrate more powerful localization algorithms with Physicalnet.

To conclude, we want to emphasize two points. First, providers have a unique global identifier allowing any application to reach them through their negotiator. Second, MICAz providers can communicate with their negotiator through any gateway and are therefore not tied to a particular wireless sensor network: they can move freely from one geographical location to another.

3.3 The Physicalnet Programming API

To create a Physicalnet application, programmers inherit from the Application class (see Figure 3). The application class provides methods to 1) add a connection to a negotiator, 2) get a reference to a service by specifying a provider global identifier and a service name, 3) to launch the periodic reevaluation of application requirements according to service states and sensing samples, 4) to periodically output a KML description of the services used by the application to a file of provided URL (the KML description can then be visualized in Google Earth), 5) to get the set of all the zones defined in the negotiators to which the application is currently connected.

Programmers can create bundles of services by inheriting from the bundle class and overriding the rule method and the foreach method. The rule method specifies the membership of the bundle. The foreach method specifies the application requirements for the service states. A bundle is a dynamic abstraction. Its membership is periodically recomputed. If a service satisfies bundle membership conditions, it dynamically joins the bundle and application requirements are sent for that service to the negotiator. If a service does not satisfy the bundle membership conditions anymore, it dynamically leaves the

```
class Application implements BundleParent{
  void add(Negotiator n);
  <T extends Service> T getService(
      String host, int port,int id, String name,Class<T> type);
  void execute(long period);
  void makeKML(URL url,String name);
  ZoneSet getZones();
}
abstract class Bundle<T extends Service>
implements Iterable<T>, BundleParent{
  Bundle(Class<T> serviceClass,BundleParent superset);
  abstract boolean rule(T t);
  abstract void foreach(T t);
  boolean contains(T t);
  Iterator<T> iterator();
  int size();
}
class Service{
  Provider getProvider();
  Gps getGps();
}
class State<T extends Serializable>{
  void set(T t);
  T get();
  T getActual();
  Boolean getWritable();
  Boolean getReadable();
}
class Event<T extends Serializable>{
  void set(Boolean b);
  Boolean get();
  Boolean getActual();
  T getLastSample();
  void whenNewSample(Task<T> t);
  Boolean getReadable();
  Boolean getWritable();
}
class Temp extends Service{
  final State<Long> period=new State<Long>();
  final Event<Long> sense=new Event<Long>();
}
```

Figure 3. Physicalnet API.

bundle and the application cancels its requirements for that service with the negotiator.

The membership conditions specified by overriding the rule method can involve service position, service type, service variables, sensory data, other bundles, and any application variable. For instance, it is possible to create a bundle of all the light actuators that still have more than half of their energy remaining, that are in the same room of an acoustic sensor that has been triggered in the last five minutes, and that are in a room with low light intensity.

Note that bundle membership can span multiple geographically separated networks. The Physicalnet programming API is such that running an application over an additional sensor and actuator networks is as simple as invoking the add method of the Application class one more time in the application code.

Within the rule and foreach method of bundles, programmers can specify membership conditions and requirements on services using the general Service class and the specific service classes that inherit from it. The Service class provides a method to get the current location of the service and a method to access information on the service provider (such as its global identifier and the other services it provides). Any specific service (such as a temperature sensor) is manipulated through a class that inherits from the Service class. Such a class defines states and/or events. For instance, the Temp class (see Figure 3), which inherits from the Service class, defines a period state that is used to formulate requirements about the sampling period of the temperature sensor. The Temp class also defines a sense event. If an application sets an event to true, it means that the application requires the provider to generate samples corresponding to that event. The samples can be accesses through the Physicalnet API using the whenNewSample method. The state and event classes provide methods to 1) inquire the access rights (read and right) of the application for a specific state or event, 2) to set a requirement for a sate or event, 3) to get the value of the last requirement that the application formulated for a given state or event, 4) to get the actual value of the state or event on the remote provider. The applications requirements may differ from the actual values of the states and events on the remote provider if the application does not have sufficient access rights or has a lower priority than simultaneously executing applications.

Application	Provider	Service	S / E	Requirement
A1	3001	sounder	on	true
A2	3001	sounder	on	false
A3	3001	sounder	on	true
A4	3001	sounder	on	false
A1	3001	temp	period	10s
A2	3001	temp	period	15s
A3	3001	temp	period	20s
A4	3001	temp	period	1s
A1	3001	temp	sense	true
A2	3001	temp	sense	true
A3	3001	temp	sense	false
A4	3001	temp	sense	false

Figure 4. Example of Requirement Table.

3.4 Enforcement of Application Requirements and Conflict Resolution

During execution, applications send their service requirements to the negotiators and update these requirements periodically when they recompute bundle membership. The negotiators receive requirements from multiple applications. They store these requirements in a requirement table and remember these requirements until the application cancels them, modifies them, or terminates. Figure 4 gives an example of requirement table where four applications have specified requirements for the sounder and the temperature sensor of provider 3001. Let us take the example of the temperature sensor service called "temp" and of its state called "period". Four applications, A1, A2, A3, and A4 have specified requirements for the period of the temperature sensor which are, respectively, 10s, 15s, 20s, and 1s. The negotiator has to decide the value to which the period of the temperature sensor is going to be set. To the set of four requirements, the negotiator is first going to apply access rights. Assuming that A4 does not have writing rights for the period of the temperature sensor, its requirement will be dismissed. Then, the negotiator is going to apply access priorities. For instance if A1 and A2 have a higher priority than A3, the requirements of A3 will be dismissed. If A1 and A2 have the same priority, there is a conflict as A1 requires a period of 10s while A2 requires a period of 15s. To resolve this situation, the negotiator uses a resolver. A resolver is a Java object that takes as input a set of requirements and provides as an output a single requirement. The default resolver uses a first come first serve policy to resolve conflicts. Provider owners can set a specific resolver for any state of the services they own. For instance, if the owner of the temperature sensor sets a MIN resolver for the sensing period, the minimum period is selected (10s) to resolve the conflict. With a MIN resolver for the sensing period, conflicting users get either the number of samples they required or more. Physicalnet provides a small library of resolvers including OR, AND (for boolean requirements), MIN, MAX, FIRST_COME_FIRST_SERVE and LAST_COME_FIRST_SERVE. Provider owners can also program custom resolvers that behave exactly as they desire.

3.5 User Access Rights

Physicalnet users can use a shell to specify the access rights of the service providers they own. The specifiable access rights for a provider include visibility rights (the right to know that the provider exists) and localization rights (the right to know

Provider ID	Service	S / E	User A	User B	User C
3001	temp	period	1RW	2R-	3R-
3001	temp	sense	1RW	2RW	3–

Figure 5. Example of Access Right Table.

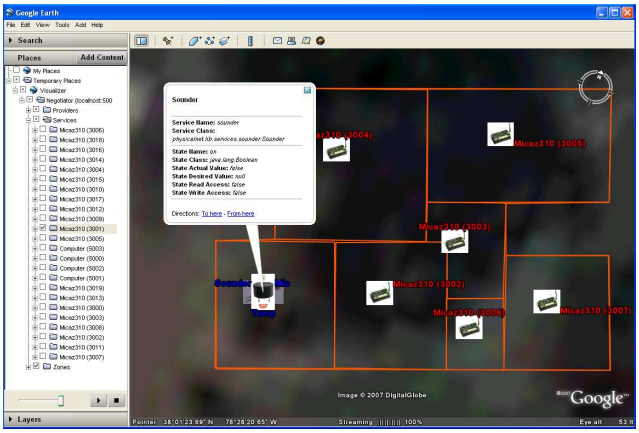

Figure 6. Resource Visualizer Output.

where a provider is). For each state and each event of each service, the owner can grant reading rights and/or writing rights. Reading rights allow users to know the current value of a state, and to obtain samples generated by an event. Writing rights allow users to modify the value of a state and to specify whether an event must generates samples. Events and states also have access priorities. When a high priority user application specifies a required value for a service state or event, the values required by lower priority user applications are ignored. If two user have the same priority and issue conflicting requirements, service specific resolvers located on the negotiator decide the resulting value of the state or event. Figure 5 provides an example of access right table for a temperature sensor that is accessible to three users. The S/E column contains a state name or an event name, W stands for writing rights, R stands for reading rights, and the number preceding these letters is the priority. As an example, we can see that user B can turn on the temperature sensor and access the generated samples. On the other hand, user B can only read the sampling period but cannot modify it. Because of the priorities, if user A requires that the temperature sensor should be off while user B specifies that the sensor should be on, the sensor will remain off.

3.6 System Tools

To facilitate network visualization and configuration, Physicalnet providers several tools briefly described in this section.

Zone Configuration Tool: The zone configuration tool allows users to upload Google Earth KML files describing zones such as rooms and building into the negotiators. It is then easy to use these zone description with the Physicalnet API to, for instance, create the bundle of all the service contained in a specific room.

Resource Visualizer: The resource visualizer allows users to connect to a set of negotiators and to obtain a KML file describing the providers, services, states, events, and zones available through those negotiators. The visualizer can periodically update the KML file, which can be loaded in Google Earth (see Figure 6).

Physicalnet Shell: The Physicalnet shell connects to a negotiator, asks for a username and password and allows users to configure service providers, specifying their interface, their resolvers, their identifier, and their access rights.

Localization Beacon Configuration Tool: The localization

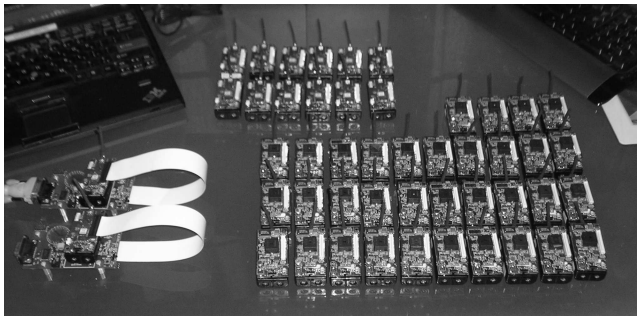

Figure 7. Hardware Used in the Evaluation.

beacon configuration tool allows users to turn a Google Earth KML file describing the location of beacons and the TinyOS identifiers of beacons into a file that, when installed on the beacons, configures them with the specified location depending on the TinyOS identifier.

4 Evaluation

In the previous section, we described both the design and implementation of Physicalnet. In this section, we evaluate Physicalnet. We describe our testbed, discuss examples of application code, and provide a quantitative evaluation.

4.1 Testbed

Sites and Hardware: To complete our experiments, we use two geographically separated sites that are approximately 1 mile apart. The hardware configuration at each site depends on the experiment and is described in corresponding sections. The total hardware available for the experiments includes a) 52 MICAz sensor nodes, which draw their energy from 2 AA batteries, which have a radio data rate of 250kbps, a 128KB programmable Flash Memory, 521KB of flash memory for data storage, 4KB of RAM, a 4MHz CPU, and 3 LEDs. b) 38 MTS310 sensor boards, which connect to the MICAzs and embed a temperature sensor, a photometric sensor, an accelerometer, a microphone, and a sounder. c) 2 desktop PCs (one at each geographical location) and a laptop. d) 2 MIB510s that connect MICAz base stations to a PC. Part of the available hardware is shown in Figure 7.

Creating a Physicalnet: Before application programmers can start programming Physicalnet applications from a computer connected to the Internet, network administrators need to create a Physicalnet. Creating a Physicalnet includes the following steps: a) The MICAzs with MTS310 sensor boards that assume the role of TinyOS service providers must run the Physicalnet TinyOS provider sofware, configured with their global identifier (which includes the IP address and TCP port of their negotiator). b) The MICAzs that assume the role of location beacons must be loaded with the Physicalnet TinyOS beacon binary properly configured to include the location (latitude and longitude) of the beacon. c) The MICAzs that assume the role of base stations must run the Physicalnet TinyOS base station sofware, must be connected to a PC with a MIB510, and that PC must run the Physicalnet Java gateway program. d) The Physicalnet Java negotiator software must run on the PCs that play the role of negotiators. Gateways do not need to know the address of the negotiators as they can infer it from the global identifier of the MICAz providers. Negotiators must be configured with the coordinates of the zones (rooms, buildings) that are to be used in applications. Negotiators must also be configured with the id and the type of the service providers they

manage. e) Computers that provide services to the Physicalnet (such as a music player, or a message board) must run the Physicalnet Java provider software (which implements the services), configured with a global identifier. At this point, application users can launch their applications. The application then connect to one or more negotiators and these negotiator update provider services according to application requirements. Note that Java providers, gateways, negotiators, and applications all run as individual processes communicating through RMI. It is, therefore, possible to install several of these components on the same PC (e.g., an application, a gateway, and a negotiator can all run on the same PC). Creating a Physicalnet is like creating a local area Internet network in that it is a one time process after which anyone connected to the Internet can use services of the sensor and actuator network.

4.2 Application Examples

In this section, we provide the description and Java code of representative applications. The first application, FireAlarm, was chosen for its simplicity to introduce Physicalnet programming. The second application, Tracker, is interesting in that it demonstrates how seamlessly Java service providers and TinyOS service providers can interact. The third application, NeighborhoodWatch, is a more complex application involving multiple sensing modalities. We slightly simplified the Java code by omitting language features such as access level modifiers and package declarations. We also replaced some numerical constants by more meaningful identifiers (e.g., 1000 is replaced by UPDATE_PERIOD).

4.2.1 The FireAlarm Application

Figure 8 shows the code for the FireAlarm application. This application has been chosen to demonstrate how to implement reusable operations for groups of nodes using the Bundle API. FireAlarm computes the average temperature in each room described (in terms of longitude and latitude) in the negotiators it connects to. If the average temperature in a room exceeds a specified threshold, all the sounders of that room must ring. Some interesting properties of the FireAlarm application are: a) If sensor nodes change rooms, their temperature samples automatically start contributing to the temperature average of the new room. b) Physicalnet finds all the temperature sensors that are indoors and the programmer does not need to know their global identifers. c) During a fire alarm, if a sounder enters a room where a fire is detected, it automatically starts ringing. Conversely, if a sounder is removed from a room where a fire is detected, it automatically stops ringing. d) The application uses all the services that have the temperature sensor interface and the sounder interface: the implementing platform is transparent to the programmer and can be a Java sensor node or a TinyOS sensor node. e) If during application execution, a new sensor node is turned on, it automatically starts sampling the temperature. f) If during application execution, the user gains access rights to a new sensor, it automatically starts sampling the temperature. These two last features are very important for sensor network applications that must run for a long time (months, years) in networks where sensors and actuators can be moved, removed, and/or added. Using Physicalnet, nodes automatically adapt to application requirements over time.

In the FireAlarm code of Figure 8, we first create a custom operation for a set of temperature sensors by creating the TempBundle class that inherits from a bundle of services that have the Temp interface. We add a method that computes the average temperature of a set of temperature sensors. We then create the FireAlarm application class by inheriting from the

```
abstract class TempBundle extends Bundle<Temp>{
  TempBundle(BundleParent theParent){
    super(Temp.class,theParent);}

  long getAverage(){
    int nb=0;
    long result=0;
    for(Temp t:this){
      if(t.sense.getLastSample()!=null){
        result+=t.sense.getLastSample();
        nb++;}}
    if(nb!=0) return result/nb;
    else return -1;
  }
}

class FireAlarm extends Application{
  FireAlarm(){

    add(new Negotiator(HOST1,PORT1,USER1,PASS1));
    add(new Negotiator(HOST2,PORT2,USER2,PASS2));

    ZoneSet zones=getZones();
    for(final Zone room:zones.getByType("Room")){

      final TempBundle temps=new TempBundle(this){
        boolean rule(Temp t){
          return room.contains(t);}
        void foreach(final Temp t){
          t.period.set(1000l);
          t.sense.set(true);}
      };

      final SounderBundle sounders=new SounderBundle(this){
        boolean rule(Sounder s){
          return room.contains(s) &&
          temps.getAverage()!=-1 &&
          temps.getAverage()>=TEMPERATURE_THRESHOLD;}
        void foreach(Sounder s){
          s.on.set(true);}
      };
    }
    execute(BUNDLE_UPDATE_PERIOD);
  }
}
```

Figure 8. The FireAlarm Application.

Application class. We connect to two negotiators by specifying the IP hostname, TCP port, user name, and passwords for these negotiators. The FireAlarm application uses the nodes of those two negotiators, which are the nodes located in two building that are one mile apart. We then obtain the set of zones stored in those negotiators. For each zone of type Room, we create the set of all temperature sensor nodes contained in that room by overriding the rule method. We specify that these sensors must sense the temperature every one second by overriding the foreach method. For each room, we then define the bundle of all the sounders in that room if the average temperature in that room is higher than a specified threshold. Note that this bundle does not contain any elements if the average temperature does not exceed the specified threshold. We then specify that the sounders that are part of the bundle must be turned on. Finally, we call the execute method of the Application superclass to set the period with which the bundle membership will be reevaluated, and with which the application requirements will be recomputed and uploaded to the negotiators. Some interesting features of this code are that: a) the TempBundle class that defines the averaging operation can be reused in any application to compute over time the average temperature over an arbitrary set of nodes with dynamic membership. b) We can easily add new negotiators to run the FireAlarm application over more buildings. c) If TEMPERATURE_THRESHOLD is a variable, the membership of the bundle of sounders is computed using the latest value of the variable, not the value at the time of the bundle definition. As a consequence, we could easily create a

```
class Tracker extends Application{
  Tracker(){
    add(new Negotiator(HOST1,PORT1,USER1,PASS1));
    add(new Negotiator(HOST2,PORT2,USER2,PASS2));
    final Led mediaTag=getService(P1_HOST,P1_PORT,
        P1_MEDIA_ID,"led",Led.class);
    final Led lightTag=getService(P1_HOST,P1_PORT,
        P1_LIGHT_ID,"led",Led.class);

    ZoneSet zones=getZones();
    for(final Zone room:zones.getByType("Room")){
      final TvBundle tvs=new TvBundle(this){
        boolean rule(MyTv tv){
          return room.contains(mediaTag) &&
            room.contains(tv);}
        void foreach(MyTv tv){
          tv.on.set(true);
          tv.channel.set(FAVORITE_CHANNEL);}
      };
      new MusicPlayerBundle(this){
        boolean rule(MyMusicPlayer p){
          return (p.getGps().distance(mediaTag.getGps())
              < DISTANCE) &&
            tvs.size()==0;}
        void foreach(MyMusicPlayer p){
          p.on.set(true);
          p.station.set(FAVORITE_PLAYLIST);
        }
      };
      new LedBundle(this){
        boolean rule(MyLed l){
          return room.contains(l) &&
            room.contains(lightTag);}
        void foreach(MyLed l){
          l.value.set(YELLOW);}
      };
    }
    execute(BUNDLE_UPDATE_PERIOD);
  }
}
```

Figure 9. The Tracker Application.

graphical interface allowing users to dynamically change the temperature threshold.

For complete understanding of the code, we consider the example of the sounder service. The "on" state of a sounder can have one of three values: TRUE, FALSE, or NULL. TRUE means that the application wants the sounder to be on. FALSE means that the application wants the sounder to be off. NULL means that the application has no requirement for the sounder. The NULL value is very different from the FALSE value in this case. If the application specifies NULL, the sounder can be used freely by other applications. If the application specifies FALSE, and if another application specifies ON, Physicalnet uses its conflict resolution mechanisms (priorities and resolvers) to resolve the conflict. If all applications specify NULL, the sounder returns to its default state (on = FALSE), which is determined by the negotiator. When services join a bundle, the application code determines what are the state requirements for those services (e.g., on = TRUE). When services leave a bundle, the state requirements are all nullified (e.g., on = NULL).

4.2.2 The Tracker Application

Figure 9 shows the code for the Tracker application. This application has been chosen to demonstrate how TinyOS service providers can seamlessly interact with Java service providers: the end-user does not need to have any knowledge about the platform that implements the services. Tracker assumes that a user moves around his home with two MICAz nodes. One is called the mediaTag, the other the lightTag. If the mediaTag is on, Tracker turns on the televisions that are in the same room as the user. If there is no television in a room, Tracker turns on all the music players that are within a specified distance of the user. If the lightTag is on, Tracker turns on all the lights that are in the same room as the user. Some interesting features of this application are that: a) the user can turn the mediaTag off to automatically turn off all televisions and music players. b) The user can turn the lightTag off to automatically turn off all lights. c) If new televisions, music players, and lights are introduced in the network, they automatically start satisfying application requirements as long as they run the Physicalnet provider platform specific software. d) If users, lights, televisions, music players move from one room to another, their state is automatically modified to satisfy application requirements i.e. state constraints based on location and distance to the mediaTag and lightTag. e) As service providers are not tied to a particular gateway and can communicate with their negotiator through the Internet, the application still works if network nodes are moved from one building to a another one, as long as each building possesses Physicalnet gateways.

Note that in our implementation, we use the yellow LED of MICAzs as room lights as we do not have real light actuators. The music player service is implenented using a Physicalnet Java service provider that runs on a PC and that turns a mp3 player on or off. The television service is implemented using a Physicalnet Java service provider that runs on a PC and that turns a video player on or off.

In the Tracker code of Figure 9, we first connect to two negotiators. The Tracker application runs over the nodes of the two buildings that report to those negotiators. We create a reference to the MICAz used as the mediaTag and to the MICAz used as the lightTag by specifying their global identifier. For each zone of type room, we create the bundle of all the televisions in that room if the room contains the mediaTag. This bundle has no members if the room does not contain the mediaTag. The televisions that are member of the bundle must be turned on and display the favorite channel of the user. For each room, we then create the bundle of all the music players that are within a specified distance of the mediaTag, if there is no television turned on in that room. This bundle has no members if the room contains a television that is on. The music players that are member of the bundle must be turned on and play the favorite playlist of the user. Finally, for each room, we create the bundle of all the lights that are in the same room as the lightTag. These lights must be on. Some interesting features of this code are that: a) the networkTag and lightTag can move from one wireless network to another and, as long as there are Phyiscalnet gateways connecting the tags to their negotiators, the music, television shows, and lights will follow the wearer of the tags wherever he/she goes. b) Several Physicalnet users can run the Tracker application simultaneously. If two users have conflicting requirements, for instance if they require a different television channel, Physicalnet applies its conflict resolution mechanisms using priorities and service specific resolvers.

4.2.3 The NeighborhoodWatch Application

Figure 10 shows the code for the NeighborhoodWatch application. This application has been chosen to demonstrate multimodal sensing. NeighborhoodWatch is a collaborative surveillance application that alerts a set of neighbors if an intruder is detected in one of their houses. In our implementation, we consider two neighbors (Mary and John) that wear MICAzs equipped with sounders. We refer to those MICAZs as the security tags. If there are no security tags in one of the houses, all the accelerators in that house are turned on. If any of those accelerators triggers, the sounders of Mary and John ring for ten

```
class NeighborWatch extends Application{
  final List<AccelBundle> accelBundles=new ArrayList<AccelBundle>();
  final List<PhotoBundle> photoBundles=new ArrayList<PhotoBundle>();

  NeighborWatch(){
    add(new Negotiator(HOST1,PORT1,USER1,PASS1));
    add(new Negotiator(HOST2,PORT2,USER2,PASS2));
    final Sounder s1=getService(P1_HOST,P1_PORT,P1_ID,
        "sounder",Sounder.class);
    final Sounder s2=getService(P2_HOST,P2_PORT,P2_ID,
        "sounder",Sounder.class);
    ZoneSet zones=getZones();
    for(final Zone building:zones.getByType("Building")){

      accelBundles.add(new AccelBundle(this){
        boolean rule(final MyAccel a) {
          return (!a.getProvider().equals(s1.getProvider()) &&
              !a.getProvider().equals(s2.getProvider()) &&
              building.contains(a)) && !building.contains(s1) &&
              !building.contains(s2);}
        void foreach(final MyAccel a) {
          a.period.set(10001);
          a.sense.set(true);
          a.sense.whenNewSample(new Task<Long>(){
            void run(Long d){
              if(d>ACCEL_THRESHOLD){a.setTriggered(true);}
              else{a.setTriggered(false);}}});}
      });

      photoBundles.add(new PhotoBundle(this){
        boolean rule(final MyPhoto p){
          return (!p.getProvider().equals(s1.getProvider()) &&
              !p.getProvider().equals(s2.getProvider()) &&
              building.contains(p)) && !building.contains(s1) &&
              !building.contains(s2);}
        void foreach(final MyPhoto p){
          p.period.set(10001);
          p.sense.set(true);
          p.sense.whenNewSample(new Task<Long>(){
            void run(Long l){
              p.samples.add(l);}});}
      });
    }
    new Timer().schedule(new TimerTask(){
      void run(){
        for(AccelBundle a:accelBundles){
          if(a.getNbTriggered(DURATION)>=1){
            s1.on.set(true);
            s2.on.set(true);
            return;}}
        for(PhotoBundle p:photoBundles){
          if(p.getNbAnomalies(DIFFERENCE)>=1){
            s1.on.set(true);
            s2.on.set(true);
            return;}}
        s1.on.set(null);
        s2.on.set(null);}
    },0,CHECK_PERIOD);
    execute(BUNDLE_UPDATE_PERIOD);
  }
}
```

Figure 10. NeighborhoodWatch.

minutes so that they are informed that an intrusion may be in progress. Accelerators can be triggered when an intruder tries to steal a television on which it is placed. Also, if there are no security tags in one of the houses, all the light sensors are turned on. If a difference in measured light intensity is detected while Mary and John are away, the sounder of Mary and John ring so that they are informed of the intrusion. A difference in measured light intensity can occur when the intruder opens a closed cupboard in which a MICAz is placed. As for other applications, the sensors and tags can be moved / turned on / turned off at any time during application execution and they will automatically update their state to conform to application requirements.

In the NeighborhoodWatch code of Figure 10, we first connect to two negotiators contained in the house of Mary and John. We create references to the sounders of the security tags of Mary and John. For each building, we create the bundle of all the accelerometers that are in that building, if neither Mary nor John sounders are in that building. This bundle does not contain any member if the sounder of either Mary or John is in the building. The accelerometers that are members of the bundle are turned on and marked as triggered if their acceleration

Application	Lines of Code
FireAlarm	72
PhotoAlarm	72
RoomOccupancy	90
NeighborhoodWatch	143
Tracker	111

Figure 11. Code Sizes for 5 Applications.

level exceed a specified threshold. For each building, we create the bundle of all the photometric sensors in that building, if neither Mary nor John sounders are in that building. This bundle does not contain any member if the sounder of either Mary or John is in the building. The photometric sensors are turned on and the samples are recorded. Periodically, we check the number of accelerometers that have been triggered within the last minute and the number of photometric sensors that have detected light anomalies. If either number is greater than 1, the sounders of Mary and John are triggered. Note that the sounders of Mary and John must be explicitly turned off when the intruder alert is over. Only the application requirements for the services that participate in a bundle are canceled automatically when they leave the bundle. One interesting feature of the NeighborhoodWatch application is that it is easy to extend it to many neighbors and many houses. A whole neighborhood can then watch each others' houses. In case sounders are triggered, neighbors can know in which house the intruder entered by launching GoogleEarth and loading the KML file that Physicalnet applications generate.

Note that we do not show all the code for NeighborhoodWatch in Figure 10. We did not include the declarations of the MyAccel, MyPhoto, AccelBundle and PhotoBundle classes. MyAccel inherits from the Accel service class and defines a settriggered operation for an accelerometer. MyPhoto inherits from the Photo service class and defines a member called "samples" of which the add method stores samples, generate a timestamp for each sample, and delete old samples. AccelBundle inherits from a bundle of accelerometers and defines a method to get the number of triggered sensors contained in the bundle. PhotoBundle inherits from a bundle of photometric sensors and defines a method to get the number of sensors that have detected light variations in the recent past.

4.2.4 PhotoAlarm and RoomOccupancy

Our performance evaluation includes two additional applications of which we do not show the code because of space constraints. The fourth application is RoomOccupancy, which turns on all the acoustic sensors of the negotiators it connects to, and infers that a room is occupied if two or more acoustic sensors are triggered in that room. RoomOccupancy periodically displays occupancy statistics in the standard output of the terminal in which it is running.

The fifth application, PhotoAlarm is identical to FireAlarm except that it uses photometric sensors instead of temperature sensors. It is interesting to compare the performance of PhotoAlarm and FireAlarm because the temperature sensor of the MICAz adapts very slowly to the ambient temperature (minutes) while the photometric sensor of the MICAz adapts promptly to the ambient light intensity (seconds).

4.3 Code Sizes

In this section, we evaluate the code size of the five application that we previously specified. The results are presented in Figure 11. The code sizes reported here are slightly greater than the code sizes that can be observed in the previous section.

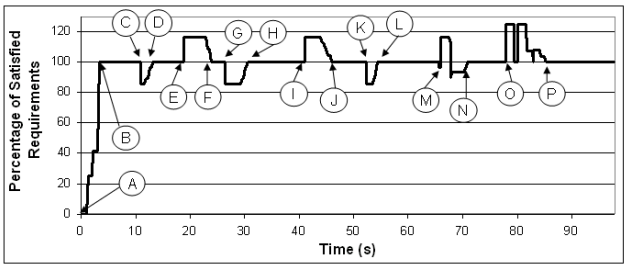

Figure 12. PhotoAlarm Execution Timeline.

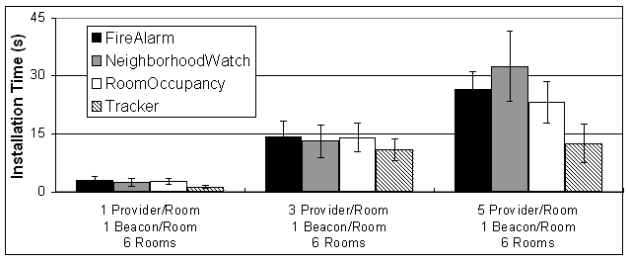

Figure 13. Installation Time as a Function of Network Size

Indeed, in the previous section, we used a more compact code representation and omitted some lines of code such as package declarations. We observe that all applications are implemented in less than 150 lines of code. This is a key result given the dynamic nature of our applications that adapt to node mobility and that span multiple, geographically separated sensor networks. Also, note that Physicalnet can let all these applications run simultaneously even if they use the same sensors and actuators. Different users can even run distinct instances of each application from any computer connected to the Internet.

4.4 Performance Evaluation

In this section, we evaluate the performance of Physicalnet applications. Our metrics are installation time and responsiveness to environmental events. We evaluate various system configurations, each configuration using either one wireless sensor and actuator network or two geographically separated sensor and actuator networks.

4.4.1 Execution Timeline

In Figure 12, we show the timeline of the execution of the PhotoAlarm application. This timeline displays the percentage of satisfied requirements over time. The percentage of satisfied requirements is defined as follows. When an application starts, it contacts the negotiators specified in the source code to ask for service information. At this point, the application decides on its initial set of requirements. For instance, FireAlarm decides that all the temperature sensors should be on and should sample with a period of 1s. The requirements are sent to the negotiators and the state of the service providers (e.g., MICAzs with MTS310) is subsequently modified. The nodes acknowledge that they modified their state to the negotiator. When the application asks for updates about the service states to the negotiator, it is then informed that its requirements are now satisfied. A percentage of satisfied requirements of 100% indicates that all the application requirements are satisfied and that the application knows about it (i.e. the acknowledgements traveled from the service providers to the application passing by the gateway and the negotiator). A percentage of satisfied requirements of less than 100% means that some of the application requirements are not satisfied yet, or that the service provider state is consistent with application requirements but that the application has not received the acknowledgements yet. A percentage of satisfied requirements of more than 100% means that an application has canceled some of its requirements (e.g., it does not want the sounders to be ringing anymore) but that this has not yet been taken into consideration by the service providers (or that the acknowledgements have not yet been received).

For this experiment, we run a single instance of PhotoAlarm on a desktop PC. The PC also runs a negotiator process and a gateway process. We use 12 MICAz service providers equipped with MTS310 sensor boards distributed in three different rooms (4 MICAzs per room), and 3 MICAz localization beacons (1 per room). PhotoAlarm connects to a single negotiator in this experiment.

At time A (see Figure 12), we start PhotoAlarm. All the initial application requirements are satisfied within 3.8 seconds for this particular run (and all acknowledgements have traveled back from the MICAzs to the application). The time elapsed between point A and point B is called the application installation time. This installation time is a metric that we use in the following sections. At points C, G, and K, we turn the light on in one of the rooms (the application then requires the sounders in that room to be turned on), while at points E and I, we turn the light off in the same room (the application then requires the sounders in that room to be turned off). We observe that after these events, it takes from 3 to 5 seconds for node states to become consistent with application requirements and for the application to be informed of the state modification. At time M, we remove two nodes from a room where the light is off to put them in a room where the light is on (they must start ringing). We observe that the system seems a little confused for a short amount of time, increasing and then decreasing the number of requirements before reaching an equilibrium. This can be explained by the fact that when we move two nodes from one room to another, It takes a couple of seconds for the new node location to be updated on the negotiator. During that time, the application uses the light measurements of the sensor as if they were coming from the previous room, and changes sounder requirement accordingly. In other words, moving nodes from a dark room to a lit room temporarily increases the computed average light in the dark room and, at the same time, reduces the computed average light in the lit room. This happens because of localization delays. After a few seconds however, the negotiator obtains correct location values and can accurately enforce application requirements. We make a similar observation at point O, when we move two nodes from one lit room to a dark room.

4.4.2 Relation Between Installation Time and Network Size

In Figure 13, we show the impact of network size on application installation time. The installation time (as defined in the previous section) is the duration between the time at which the application starts and the time at which the service providers satisfy application requirements and have successfully sent acknowledgements.

In this experiment, we run four applications (FireAlarm, NeighborhoodWatch, RoomOccupancy, and Tracker). Each run is independent from one another, i.e. applications do not run simultaneously. We run each application 10 times for each network configuration and measure the installation time. There are three network configurations. All configurations use a single PC that runs the application, a negotiator, a gateway, and a Java service provider for the Tracker application. The first network configuration uses 6 MICAzs with MTS310 boards

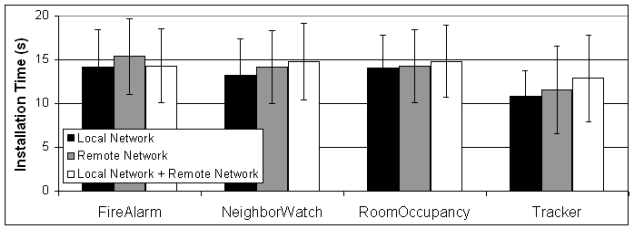

Figure 14. Comparison of Local, Remote, and Multi-Network Installation Times.

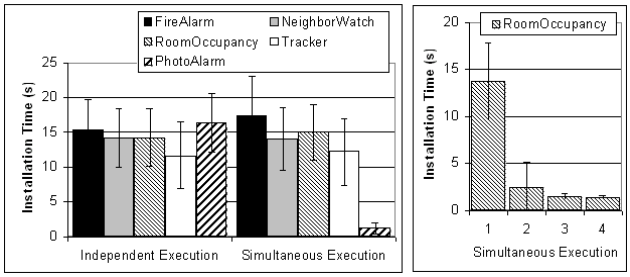

Figure 15. Installation Time and Application Concurrency.

distributed in 6 rooms (1 per room) and 6 location beacons (1 per room). The second network configuration uses 18 MICAzs with MTS310 boards distributed in 6 rooms (3 per room) and 6 location beacons (1 per room). The third network configuration uses 30 MICAzs with MTS310 boards distributed in 6 rooms (3 per room) and 6 location beacons (1 per room). In this experiment each application connects to a single negotiator.

Figure 13 shows the installation time as a function of network size along with error bars of length that is twice the standard deviation. We observe that installation time increases significantly with network size. For instance, for NeighborhoodWatch, the installation time varies from 2.4s for a network of 6 sensors, to 13.1s for a network of 18 sensors, to 32.4s for a network of 30 sensors. Such an increase could be expected as all packets must pass through a single base station connected to the PC.

We also observe that the Tracker applications performs consistently better than the others. This is due to the fact that the Tracker application needs only, at installation time, to turn on the LEDs of the nodes that are in the same room as two MICAzs specified in the application code. By comparison, FireAlarm need to turn on all the temperature sensors.

4.4.3 Remote and Multi-Network Execution

In this experiment we study the relationship between installation time and whether we run applications on a local network, a remote network, or multiple sensor and actuator networks. We run four applications (FireAlarm, NeighborhoodWatch, RoomOccupancy, and Tracker). Each run is independent from one another, i.e. applications do not run simultaneously. We run each application 10 times for each network configuration and measure the installation time. There are three network configurations. The first configuration, named the local configuration, uses one PC that runs the application, the negotiator, the gateway, and the Java service provider. It uses 18 MICAzs with MTS310 boards distributed in 6 rooms (3 per room) located at the same site as the PC, and 6 location beacons (1 per room). The second configuration, named the remote configuration, uses two PCs. The first PC runs only the application. The second PC, located 1 mile away from the first PC, runs the negotiator, the gateway, and the service provider. It uses 18 MICAzs with MTS310 boards distributed over 6 rooms (3 per room) at the same site as the second PC, and 6 location beacons (1 per room). The third configuration, named the local+remote configuration, uses two PCs. The first PC runs the application, a negotiator, a gateway, and a Java service provider. The second PC, located 1 mile away from the first PC, runs a negotiator, a gateway, and a Java service provider. Both sites have 18 MICAzs with MTS310 boards distributed over 6 rooms (3 per room), and 6 location beacons (1 per room), for a total of 48 nodes distributed across 12 rooms. In the local configuration and the remote configuration, applications connect to one negotiator, while in the local+remote configuration, applications connect to two negotiators. The PCs on both sites are connected to the Internet through a cable connection.

Figure 14 compares the installation times for the local, remote, and local+remote configurations and shows error bars of length that is twice the standard deviation. We first observe that connection to a remote sensor network rather than a local one has little impact on installation performance. For all four applications, the average installation time increases by no more than 2 seconds. This could be expected as the performance bottleneck of our system is located at the sensor network base station. Second, we observe that the local+remote configuration performs nearly as well as the local configuration. Indeed, for all four applications, the average installation time increases by no more than 3 seconds. This is a key result: the application can control twice as many nodes at little cost. This could be expected as the performance bottleneck of our system is located at the sensor network base station. These results mean that the problem observed in the previous section (the significant increase in installation time with network size) can be resolved by introducing more sensor network gateways.

4.4.4 Remote and Simultaneous Execution

In this experiment, we study the impact of simultaneous application execution on installation performance. We use the remote configuration described in the previous section. The experiment has two parts. In the first part, we execute five applications independently (no application is running when a new one is launched). In the second part, we execute all applications simultaneously. FireAlarm starts at time t=0s, NeibhorhoodWatch at time t=60s, RoomOccupancy at time t=120s, Tracker at time t=180s, and PhotoAlarm at time t=240s. When PhotoAlarm is launched, the four other applications are still running. Each experiment is repeated 10 times.

The left graph of Figure 15 shows the installation time for each application for both the independent execution and the simultaneous execution. The length of error bars is of twice the standard deviation. For the independent execution, we observe performance characteristics similar to the ones of the previous section. For the simultaneous execution, performance is similar to the independent execution, except for PhotoAlarm that has a very small execution time (1.2s). To explain the similar execution times, let's take the example of RoomOccupancy. In the independent execution case, RoomOccupancy must turn on all the acoustic sensors and set their sampling period. In the simultaneous execution case, RoomOccupancy must do the same. In the simultaneous execution, the fact that sensors already send temperature samples, accelerometer samples, and photometric samples back to the base station when RoomOccupancy starts its execution does not impact performance significantly because the samples are added to the beacon that the sensor periodically sends to its negotiator through a gateway.

Now, why is the installation of PhotoAlarm so fast when it executes while all the other applications are running? It is

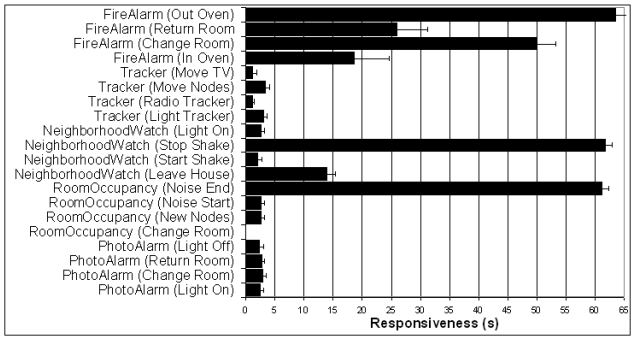

Figure 16. Responsiveness to Environmental Stimuli.

because when PhotoAlarm starts, it requires all the photometric sensors to sample the light intensity. However, these requirements are immediately satisfied when PhotoAlarm starts as NeighborhoodWatch already started the sensors: the negotiator does not need to contact any of the sensor network nodes when PhotoAlarm starts as all the requirements of this application are already satisfied.

This phenomenon can be better observed in the right graph of Figure 15. In this experiment, four different users start RoomOccupancy at t=0min, t=1min, t=2min, and t=3min. After minute 4, the four users can observe room occupancy statistics on their computer. Only the first run of RoomOccupancy takes a significant amount of time (13.7s) because the negotiator must reconfigure all the nodes of the remote network so that they start their acoustic sensor. During the subsequent executions, the negotiator does not need to contact the sensor network as all application requirements are already satisfied. The negotiator will only need to forward the acoustic samples to four instances of RoomOccupancy instead of one. In other words, the overhead of simultaneous application execution is here on the negotiator, not on the sensor network. As the sensor network is the performance bottleneck, simultaneous application execution does not significantly impact performance.

For complete understandability, we here emphasize that the acoustic and acceleration samples sent by the acoustic and accelerometer services of the MICAzs are the standard deviation of 50 actual samples. This standard deviation can be used to quantify the amount of environmental noise in the case of the acoustic sensor, and to quantify the amount of acceleration of the sensor node in the case of the accelerometer.

4.4.5 Responsiveness to Environmental Stimuli During Remote and Simultaneous Execution

In this experiment, we study the responsiveness of Phyiscalnet applications to environmental phenomena. We use the remote configuration described in Section 4.4.3. We use five applications: FireAlarm, RoomOccupancy, PhotoAlarm, NeigbhorWatch, and Tracker. For each application, we define four events. A responsiveness measurement is the time elapsed from the time at which an environmental stimulus is applied to the network to the time at which the predictable consequences of the environmental stimuli are observed. For instance, if an operator turns on the light and PhotoAlarm is running, the responsiveness is the time necessary for all the alarm sounders to be turned on. In these experiments, we measure durations using a time watch. Whenever necessary, we modify the MICAz service provider binary so that it turns on a chosen LED to reveal a state we are interested in. That way, we can for instance be sure that all sounders are turned on rather than only a couple. The five applications execute simultaneously on the remote sensor network. Measurements start when application installation for the five applications is finished. Each event is reproduced ten times, sequentially. The results are displayed in Figure 16 along with error bars that have the length of twice the standard deviation.

The environmental stimuli we study are application specific. For FireAlarm, the studied stimuli are: a) InOven: the time it takes for all the sounders of a room to be turned on when the sensors of that room are put on the open door of an oven heating at a temperature of 200 Fahrenheit degrees. b) ChangeRoom: the time it takes for all the sounders of 3 nodes to be turned off when they are moved from the oven to another room. c) ReturnRoom: the time it takes for the same sounders to be turned on when they are moved back to the oven door. d) OutOven: the time it takes for the same sounders to be turned off when they are moved out of the oven door to the floor. In this experiment, we observe that the responsiveness is very slow, ranging from 18.8s to 63.6s. However, this bad performance is not due to Phyiscalnet but rather is due to the duration the sensor needs to adapt to changing temperatures. We confirm this fact by experimenting with PhotoAlarm, which has exactly the same application logic as FireAlarm except that it uses photometric sensors instead of temperature sensors. When using PhotoAlarm, the responsiveness drops to less than 3 seconds for all four stimuli.

For the Tracker application, the studied stimuli are: a) LightTracker: the time it takes for all the yellow LEDs of a room to be turned on when the lightTag specified in the application code enters that room. b) RadioTracker: the time it takes for the music player on a laptop in a room to be turned on when the mediaTag enters that room. c) MoveNodes: the time it takes for the yellow LEDs of 3 MICAzs to be turned on when they are moved to the room where the lightTag is. d) MoveTv: the time it takes for the video player on a laptop to be turned on when the laptop is moved to the room where the mediaTag is. For this last stimulus, the laptop is connected to a MICAz that can read packets sent by location beacons and forward them to the laptop Java provider.

For the NeighborhoodWatch application, the studied stimuli are: a) LeaveHouse: the time it takes for the 18 accelerometers and the 18 photometric sensors of a house to be turned on when the neighbors leave the house (along with the sounders used to pinpoint their location). Note that in this experiment we need to add an additional beacon outdoors so that the neighbors' sounders can notify to the negotiator that they are outside the house. Note also that despite the house walls, the two MICAzs that the neighbors are wearing remain connected with the sensor network during the experiment. b) StartShake: the time it takes for the neighbors' sounders to start ringing after a MICAz sensing acceleration has been shaken. c) StopShake: the time it takes for the neighbors' sounders to stop ringing after the shaking of the MICAz stops. d) LightOn: the time it takes for the neighbors' sounders to start ringing when the light is turned on in one room of the house.

For the RoomOccupancy application, the studied stimuli are: a) ChangeRoom: the time it takes for 3 acoustic sensors to adapt their state when they are moved from one room to another. b) NewNodes: the time it takes for 3 acoustic sensors to start sensing when they are turned on in on of the rooms. c) NoiseStart: the time it takes for a room to be marked as busy on the PC display when noise is made in that room. d) NoiseEnd: the time it takes for a room to be marked as non-busy on the PC display when noise is stopped in that room.

We observe that reponsiveness is less than 4 seconds for 13 out 20 stimuli. We already explained why the responsiveness is slower for the four FireAlarm stimuli. For StopShake, the responsiveness is of about one minute because we specified in

the Tracker application code that the sounders should ring if the accelerometers have been triggered during the last minute. For NoiseEnd, the responsiveness is of about one minute because we specified in the RoomOccupancy code that a room should be marked as busy if two or more acoustic sensors have been triggered in the last minute. For LeaveHouse, the responsiveness is 14.1s because all the accelerometers and photometric sensors must be turned on (by contrast, most other stimuli change the state of three or less sensors).

Note that the responsiveness for the ChangeRoom stimulus of RoomOccupancy is 0 second because there is no requirement changing in the state of an acoustic sensor when it changes room during the execution of the RoomOccupancy application (the sensor must just keep generating acoustic samples). Note also that the responsiveness to the MoveTv and RadioTracker stimuli is particularly fast because these stimuli change only the state of Java service providers.

5 Related Work

When we researched ways to satisfy simultaneously the set of all proposed system requirements, our first step was to consider research projects that address these requirements individually. We then investigated whether the proposed solutions could be combined easily. From this investigation, we concluded that such combinations are either impossible or require significant modification of the solutions to individual requirements. A large amount of literature addresses one or more of our system requirements. We now discuss a relevant and representative subset of this literature. Given the size limit, our survey cannot be comprehensive.

Global Accessibility and Interoperability: A large amount of literature addresses the problem of making services accessible through the Internet. This body of work includes service oriented architectures and the technologies that enable it: RPC [6], ERPC [22], RMI [5], CORBA [4], Jini [3], and WebServices [2]. We believe that service oriented architectures are a practical way to make services both interoperable and globally accessible. This is why the Physicalnet framework is based on such an architecture. Note that Physicalnet does not aim at replacing existing service oriented architecture. Rather it aims to complement them. Indeed, Physicalnet is especially designed for sensors and actuators. Because the user interface of Physicalnet is entirely in Java, programmers can, within the same application, use Physicalnet to manipulate sensor and actuators, and still use state of the art solutions such as Jini or WebServices to manipulate more traditional software services. Some literature has also focussed on connecting resource constrained sensors nodes with the Internet: Agimone [14], IrisNet [13], and ArchRock Primer Pack [1]. These solutions do not address the problems of resource sharing and access rights.

Dynamic Resource Sharing: A first way to satisfy the resource needs of several application simultaneously, is to treat the network as a distributed database [16, 17]. This approach however does not resolve the issue of conflicting requirement as in the case of two users requiring different states for a light actuator. Other solutions include Agilla [12], which uses agents to efficiently implement motion tracking applications, and SensorWare [9], which focusses on platforms that have an order of magnitude more resources than the typical sensor network node. TinyCubus [18] has a different paradigm. It partitions a sensor network so that each application has exclusive access to one of the partitions. It does not support node-level concurrency. More recently, Melete [24] enables the execution of concurrent applications on a single sensor node. Melete and Physicalnet are complementary approaches. Physicalnet focuses on sharing existing node services (sensing, actuating) among a large number of users. Within this context, Melete could be used to dynamically add new services to a sensor node.

Access Rights: Few works (such as CodeBlue [15]) make mention of user access rights for sensor and actuator networks. Physicalnet access right management system is a main contribution of this paper. It allows the fine-grained specification of service accessibility.

Programming Abstractions and Languages: A lot of work has focussed on programming abstractions for sensor networks. For instance, Envirotrack [7] focusses on target tracking. Regiment [19] introduces the region stream abstraction, which derives from Abstract Regions [20]. Abstract Regions represents a spatial and temporal distribution of node states. Related abstractions include Hood [21], which allows the programming of nodes in terms of neighborhoods rather than messaging protocols. MetroSense [11] uses opportunistic tasking and sensing to leverage the unpredictable mobility patterns of sensors. Spatial Programming [8] allows the referencing of services using physical locations and/or service properties. Physicalnet bundles are more general: they allow the definition of a set of services using a logical predicate specified using the Java programming language. The set can be defined according to an arbitrary set of parameters including current sensor readings, location, contextual information, and application variables. The membership of the set may vary over time and the tasks that the nodes execute vary accordingly. The membership of the set can include nodes from multiple remote networks.

Semantic streams [23] is a framework that automatically interprets sensor data to satisfy declarative queries. For instance, a user can query the speed of a vehicle and the system decides which sensor data and which operations to use to compute the vehicle speed. In Physicalnet, programmers can use the provided abstractions to create code that compute the results of such queries. This code is then reusable from one application to another.

More recently, Chu et al. created a declarative language for sensor networks, the SNLOG language [10], and determined that a declarative approach is a good fit for sensor network programming. We agree with this conclusion. In fact, even though the Physicalnet API is implemented using an imperative language (Java), it has a declarative flavor: it allows the user to declare bundles, thereby specifying the state of sensors and actuators according to varying environmental conditions.

Note however that Physicalnet and SNLOG have completely different goals and characteristics. Physicalnet is used to program applications at a high level of abstraction, applications that can span multiple networks, applications that can use the same sensors and actuators concurrently. Physicalnet cannot be used for the low level programming of sensor networks. For instance, Physicalnet, unlike Hood, Abstract Regions, and SNLOG, cannot be used to program a sensor network routing protocol.

6 Conclusion

In this paper, we described the design, implementation, and evaluation of Physicalnet, a framework that aims to facilitate the programming of global, multi-user, dynamic networks of sensors and actuators. Physicalnet uses a service oriented architecture to make services globally accessible, and to allow resource constrained devices to interact seamlessly with Internet-enabled platforms. Through its negotiators, Physicalnet allows concurrently executing applications to dynamically share resources, while enforcing access rights and access priorities. The programming of large, dynamic networks is made practical through the use of bundles.

To evaluate the Physicalnet implementation, we use a testbed composed of MICAzs, MTS310 sensors boards, laptop PCs, and desktop PCs. The testbed is composed of two geographically separated wireless networks located about one mile from one another. During the experiments, testbed nodes are moved, turned off and turned on. We report code excerpts, number of lines of code, and performance results for five applications named FireAlarm, PhotoAlarm, NeighborhoodWatch, RoomOccupancy, and Tracker. All these applications can be deployed over one or several wireless networks. All these applications react to sensed phenomena by modifying the state of a subset of the testbed nodes. Examples of sensed phenomena include node mobility, node vibration (using accelerometers), temperature (using temperature sensors), light intensity (using photometric sensors), and noise intensity (using acoustic sensors). Performance metrics include installation time and responsiveness to operator stimulus. Our first conclusion is that application programming is concise: each application is programmed in less than 150 lines of code. Our second conclusion is that Physicalnet performance satisfies application requirements with a responsiveness of a few seconds, and with installation times varying from less than 10 seconds to 30 seconds depending on network size.

We are currently exploring whether Physicalnet satisfies the needs of additional application domains such as medical, military, environmental, and industrial. We plan to extend our framework to integrate additional hardware platforms, more advanced localization techniques, and additional networking protocols. We also plan to investigate potential security threats and to propose a secure Physicalnet implementation.

All sensor and actuator nodes that run the Physicalnet platform specific software, communicate with the Internet through a Physicalnet gateway, and register with a negotiator are part of the Physicalnet. Programmers can create applications involving any number of nodes participating in the Physicalnet. This could potentially give birth to world-wide applications such as a weather forecast application requiring the participation of a large number of environmental networks from all over the world. Alternatively, the Physicalnet could host a large set of smaller, cross-network, user specific applications that interact with each other in complex ways, turning our planet into a sensitive and responsive entity. Future research shall determine the feasibility of such a vision and the means for it to become a reality.

References

[1] Arch rock, http:// www.archrock.com/.

[2] Java web services at a glance, http:// java.sun.com/ webservices/.

[3] Jini network technology, http:// java.sun.com/ developer/ products/ jini/ index.jsp.

[4] Omg's corba website, http:// www.corba.org/.

[5] Remote method invocation home, http:// java.sun.com/ javase/ technologies/ core/ basic/ rmi/ index.jsp.

[6] Remote procedure call protocol specification version 2, http:// tools.ietf.org/ html/ rfc1831.

[7] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 582–589, 2004.

[8] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: design and implementation. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 690–699, 2004.

[9] A. Boulis, C.-C. Han, R. Shea, and M. B. Srivastava. Sensorware: Programming sensor networks beyond code update and querying. *Pervasive Mob. Comput*, 3:386–412, 2007.

[10] D. Chu, A. Tavakoli, L. Popa, and J. Hellerstein. Entirely declarative sensor network systems. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1203–1206, Seoul, Korea, 2006. VLDB Endowment.

[11] S. Eisenman, N. Lane, E. Miluzzo, R. Peterson, G.-S. Ahn, and A. Campbell. Metrosense project: People-centric sensing at scale. In *WSW 2006 at Sensys 2006*, 2006.

[12] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 653–662, June 2005.

[13] P. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: an architecture for a worldwide sensor web. *Pervasive Computing, IEEE*, 2:22–33, 2003.

[14] Hackmann, Fok, Roman, and Lu. *Agimone: Middleware Support for Seamless Integration of Sensor and IP Networks*. 2006.

[15] D. Hromin, M. Chladil, N. Vanatta, D. Naumann, S. Wetzel, F. Anjum, and R. Jain. Codeblue: a bluetooth interactive dance club system. In *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, volume 5, pages 2814–2818 vol.5, 2003.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev*, 36:131–146, 2002.

[17] 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*, 30:122–173, 2005.

[18] P. Marron, A. Lachenmann, D. Minder, J. Hahner, R. Sauter, and K. Rothermel. Tinycubus: a flexible and adaptive framework sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 278–289, 2005.

[19] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*, pages 78–87, Toronto, Canada, 2004. ACM.

[20] M. Welsh and G. Mainland. Programmin sensor networks using abstract regions. In *NSDI*, 2004.

[21] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, Boston, MA, USA, 2004. ACM.

[22] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *Proceedings of the fifth international conference on Information processing in sensor networks*, pages 416–423, Nashville, Tennessee, USA, 2006. ACM.

[23] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *Proceedings of the European Workshop on Wireless Sensor Networks*, 2006.

[24] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *Conference On Embedded Networked Sensor Systems*, pages 139–152, Boulder, Colorado, USA, 2006. ACM Press.