Bundle: A Group Based Programming Abstraction for Cyber Physical Systems

Pascal A. Vicaire, Enamul Hoque, *Member, IEEE,* Zhiheng Xie, John A. Stankovic, *Fellow, IEEE*

(Invited Paper)

***Abstract*—This paper describes a novel group based programming abstraction called a 'Bundle' for cyber physical systems (CPS). Similar to other programming abstractions, a Bundle creates logical collections of sensing devices. However, previous abstractions were focused on wireless sensor networks (WSN) and did not address key aspects of CPS. Bundles elevate the programming domain from a single WSN to complex systems of systems by allowing the programming of applications involving multiple CPSs that are controlled by different administrative domains and support mobility both within and across CPSs. Bundles can seamlessly group not only sensors, but also actuators which constitute an important part of CPS. They enable programming in a multi-user environment with fine grained access right control and conflict resolution mechanism. Bundles support heterogeneous devices, such as motes, PDAs, laptops and actuators according to the applications' requirements. They allow different applications to simultaneously use the same sensors and actuators. Bundles facilitate feedback control mechanisms by dynamic membership update and requirements reconfiguration based on feedback from the current members. The Bundle abstraction is implemented in Java which ensures ease and conciseness of programming. We present the design and implementation details of Bundles as well as a performance evaluation using 32 applications written with Bundles. This set includes across-network applications that have sophisticated sensing and actuation logic, mobile nodes that are heterogeneous, and feedback control mechanisms. Each of these applications is programmed in less than 60 lines of code.**

***Index Terms*—Programming, software, networks, actuators, transducers.**

I. Introduction

In the future, cyber physical systems (CPS) will become widespread, include heterogeneous sensing and actuation devices, support intra and inter network mobility, permit multiple applications to execute simultaneously, and be accessible and controllable via the Internet. Ubiquitously deployed wireless sensor networks (WSNs) enhanced with actuators will create a new CPS infrastructure, and along with body networks and sensor-based cell phones will create a situation with many interacting systems of systems. For this vision to become commonplace new abstractions are required that support ease of programming, grouping sensors and actuators of different kinds from different networks and administrative domains, and dynamically managing these groups in the presence of mobility and feedback control.

To better illustrate these requirements, we consider the following scenario. John and Mary are two neighbors who have separate WSNs set up in their houses for activity monitoring. They also run a collaborative surveillance application that notifies both of them if an intruder tries to steal something from any of the two houses in their absence. The notification is done by ringing sounders that are worn in their bodies. Some important features to consider for this application are: 1) The application spans multiple systems and uses heterogeneous devices. 2) It groups sensors from both houses and actuators on their bodies. 3) It supports both intra and inter network mobility. Because they move from room to room when in house and also go out for work. 4) The application only notifies them if they are not in the house and someone tries to steal something. So the actuation of sounders depends on feedback from the sensors.

Existing group based abstractions employ a distributed architecture in order to ensure energy and bandwidth efficiency. They group the nodes based on geographic location or radio connectivity ([26], [25]) or some higher-level, application-defined notion of proximity ([19], [12]). But there are certain limitations in using them for CPS. They have been designed mainly for applications that run in a single network. They cannot group sensors from different networks or sensors having inter-network mobility. New applications need to reprogram the sensors manually. Besides, writing applications with them is not straightforward. Also, they do not provide fine grained access right control and conflict resolution mechanism that are essential for multi-user environments. Moreover, none of them supports grouping of actuators. A Bundle extends the previous group based abstractions by addressing these limitations.

The main contributions of this paper are: a) a new group based abstraction called a Bundle which is an extension to existing group abstractions, but with important capabilities for across system programming, mobility, automatic dynamic updates, fine grained access right control and conflict resolution mechanism, and support for actuators; and b) an evaluation with 32 single and multi network applications to illustrate the ease and conciseness of programming with Bundles, its effectiveness of supporting mobility, its acceptable energy overhead and effectiveness in actuator configuration.

The paper is organized as follows. Section II describes related work and compares Bundles with other similar abstractions. Section III explains the Bundle abstraction in detail. Section IV describes its implementation details. Section V presents evaluation results. Section VI lists the notable limitations of our system. We conclude in Section VII.

Manuscript received October 24, 2010; revised May 3, 2011; accepted August 2, 2011.

II. Related Work

Existing group based abstractions have several shortcomings that limit their applicability in cyber physical systems. The Bundle has been designed to overcome these shortcomings. Table I summarizes some of the important differences between Bundles and other similar abstractions.

Hood [26] is a neighborhood programming abstraction that allows a given node to share data with a subset of nodes around it, specified using parameters such as the physical distance or number of wireless hops. Hood cannot group nodes that belong to different networks or that use heterogeneous communication platforms. If a mobile group member moves to another network, then it no longer belongs to that group. Additionally, all nodes must share the same code, actuators are not supported, group specification is fixed at compile time, and each instance of a Hood requires specific code compiled and deployed on the targeted nodes.

An Abstract Region [25] is an abstraction similar to a Hood: it allows the definition of a group of nodes according to geographic location or radio connectivity, and permits the sharing and reduction of neighborhood data. Abstract Regions provide tuning parameters to obtain various levels of energy consumption, bandwidth consumption, and accuracy. But, each definition of a region requires a dedicated implementation, therefore each region is somehow separated from others and cannot be combined. Like Hood, Abstract Regions also cannot group sensors from different networks, actuators, heterogeneous or mobile devices. If we need to write new applications, motes need to be reprogrammed.

Logical neighborhood ([19], [4]) is a higher level abstraction that replaces the physical neighborhood provided by wireless broadcast with a higher-level, application defined notion of proximity. It has support for heterogeneous nodes, but heterogeneity means different communication costs for different nodes. It does not support actuators or devices with a heterogeneous communication platform. Logical neighborhoods cannot cross multiple networks. With logical neighborhoods, we cannot write new applications using existing devices without reprogramming them. Also they do not support mobility.

Scopes ([12], [13]) is another abstraction that is used to structure a WSN in groups and sub-groups. Scopes use a declarative language to specify properties that have to be fulfilled by a node participating in a scope. As properties, static and dynamic values are supported just like in Bundles, but it needs more resources on a single node to fulfill its tasks. Scopes support multiple concurrent tasks that have to be installed over-the-air on the nodes. Scopes do not support actuators and spanning over different networks as supported by Bundles, however heterogeneous nodes in the WSN are possible. Mobility of nodes from one network to another is also not supported.

sChat [23] is a group communication service that allows groups of mobile entities to communicate over a WSN. Each group has a leader that needs to keep track of all the members' locations. A central registry keeps track of all the groups and their leaders. But this design does not support across-network applications and we need to reprogram the devices for new applications. It does not have support for actuators.

Abstraction	Bundle	Hood	Abstract Region	Logical Neighbors	Scope
Language Used to Write New Applications	Java	nesC	nesC	SPIDEY	C
Sensors can be Reprogrammed for New Applications Dynamically	Yes	No	No	No	No
Concurrent Applications Using Same Devices Supported	Yes	No	No	No	Yes
Span Multiple Networks	Yes	No	No	No	No
Heterogeneous Devices Supported	Yes	No	No	Partially	Yes
Inter Network Mobility Supported	Yes	No	No	No	No
Actuator Supported	Yes	No	No	No	No
Centralized Group Management	Yes	No	No	No	No

TABLE I
Comparison of Bundle with other Abstractions

A different way of grouping nodes in WSNs is the Generic Role Assignment Scheme [22]. A predefined set of roles is distributed to all the nodes, which must decide at runtime which of these roles they currently comply with. A node choosing a specific role may cause other nodes to reevaluate their role membership, leading to toggling role memberships and messaging overhead. Compared to Bundles it is not possible to steer the role membership. Also the use of new applications/roles is not possible without reprogramming all nodes.

Spatial Views [21] and Spatial Programming [2] have only been implemented on powerful PDAs and not on resource constrained sensor nodes. Spatial Views create a group of nodes defined in terms of location and service interfaces, and make it possible to iterate over the members of this group. Heterogeneity, actuators, multiple networks or mobility of nodes are not addressed in this approach. Spatial Programming provides an abstraction similar to spatial views. Applications use a mobile agent approach which is executed in a modified JavaVM. Mobility is an important paradigm, but as with Spatial Views, heterogeneity or multiple networks are not supported.

Envirosuite [16] allows the creation of virtual objects that correspond to real phenomena such as a car or an abnormally high magnetic field. Each virtual object is instantiated on a single node, or on a group of nodes that are geographically close and sense the same phenomena. Each instance of a virtual object is associated with a unique identifier; Envirosuite manages issues related to the maintenance of per object unique identifiers as the real phenomena moves or as two phenomena collide. TinyGALS [3] is another programming model for event-driven embedded systems. Software components are composed locally through synchronous method calls to form modules, and asynchronous message passing is used between modules to separate the flow of control. This programming model is structured such that all asynchronous message passing code and module triggering mechanisms can be automatically generated from a high-level specification.

Realizing the difficulties of programming wireless sensor networks, the research community investigated novel approaches. One of these approaches is macro-programming, which allows programming at a higher level of abstraction: the goal is to focus on writing central programs that specify high level network behavior rather than implementing code from the point of view of each node. Regiment [20] is a functional macro-programming language for wireless sensor networks. It provides a region stream abstraction that applies folding and mapping operations to a spatially distributed time collection of node states, thereby creating data streams from programmer-specified spatial regions. The language is side effect free: it cannot update states that are local to the nodes and, as a consequence, is problematic for programming actuators. Regiment poses numerous implementation challenges and only a very small subset of the language is implemented for resource constrained nodes.

Semantic Streams [27] is a logical macro-programming approach to wireless sensor network programming. It takes a high level query and given a particular topology of nodes with specific services, it automatically composes sensors and inference units (using a variant of the standard backward chaining algorithm) so as to respond appropriately. It supports heterogeneity and automatically optimizes simultaneous queries from multiple users by streaming sensors only once, even if multiple queries require their data. Semantic Streams assumes a fixed topology. All operators specifying how to obtain a needed output from a given input must be contained in a library: a query will be successful if and only if a chain of operators can be found that generates the desired output from sensor inputs.

Abstract task graph [1] is a macroprogramming model that builds upon the core concepts of data driven computing and incorporates novel extensions for distributed sense-and-respond applications. The types of information processing functionalities in the system are modeled as a set of abstract tasks with well-defined input/output interfaces. Macrolab [11] is another macroprogramming framework that offers a vector programming abstraction similar to Matlab for Cyber Physical Systems (CPS). In this framework, all application-specific logic are contained in a macroprogram; the user writes a single macroprogram for the entire CPS and the framework automatically decomposes it into to a set of microprograms that are loaded onto each node. MacroLab decomposes a macroprogram in the way that is most efficient for a particular deployment. Another macro-programming approach is Kairos [8] that provides three programming constructs: one for reading and writing variables at a node, one for iterating through the one hop neighbors of a node, and one for addressing arbitrary nodes. Once a program is written using Kairos python extension, Kairos generates binaries for single nodes.

III. Design

In this section we describe the underlying architecture of Bundles, the main design principles of the Bundle abstraction, how Bundles work, their access right control and conflict resolution mechanism and their semantics.

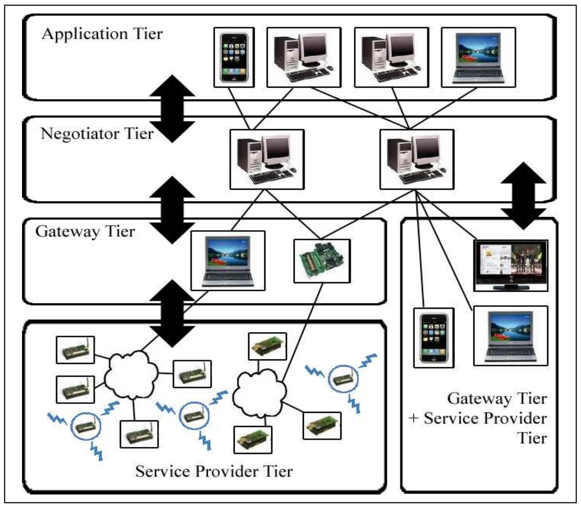


Fig. 1. Physicalnet Architecture.

A. Architecture

The Bundle abstraction is implemented on top of Physicalnet [24], a middleware for wireless sensor networks based on a lightweight service oriented architecture. Detailed description and evaluation of Physcialnet are provided in [24]. Here, we briefly outline the implementation of Physicalnet only at the level of details required to understand the implementation support for Bundles.

There are 4 tiers in Physicalnet as Figure 1 shows.

1) **Service Provider Tier:** A provider node can be running TinyOS or Java and may include several services. For example, a service can be the temperature sensor of a MICAz node, a light actuator, or the display screen of a PC. This layer also contains localization anchor nodes. A provider registers its services to one and only one negotiator, and executes the commands issued by this negotiator.

2) **Gateway Tier:** A gateway collects the control or data messages from the service providers and forwards them to the negotiators. Similarly, it forwards commands in the other direction. The communication between the gateways and the service providers is through multi-hop wireless protocols (e.g. collection and dissemination protocols), while the communication between the gateways and the negotiators is through TCP/IP. The gateway tier could consist of either Java nodes or more powerful PCs. There has to be at least one gateway per network.

3) **Negotiator Tier:** A negotiator is a repository of services, a database of service states and application requirements. A negotiator contains all the services that register on it and are available at that time. Applications can discover and operate on those services through the negotiator. A negotiator allows multiple applications to access the same service concurrently. It is important to note that negotiators are not tied to a particular WSN, and that they can manage nodes located in multiple WSNs. Each administrative domain consists of one negotiator, all the service providers registered to the negotiator and a set of users.

4) **Application Tier:** It contains applications that periodically generate and cancel requirements for remote sen-

sors and actuators by reevaluating the membership of their Bundles. Multiple applications can simultaneously access the same negotiator and a single application can involve multiple negotiators.

The main advantage of this 4-tier architecture is that the resource constrained sensor nodes have minimal functionality and most of the complexity of the applications is pushed outside the WSN onto remote and more powerful computers (similar to Tenet [7], Essentia [9], and Atlas [14]). The gateway tier ensures that heterogeneous devices can be grouped together as long as it can communicate with them using their communication protocol. They are more powerful than sensor nodes, so they are placed in a different tier. The negotiator tier communicates with different gateways and vice versa. Devices that move from one network to another, only need to communicate to the gateway of a network and the gateway communicates with the appropriate negotiator which may be in any part of the world. So having a different negotiator tier enables us to group devices from different networks and also to support inter network mobility. Having a different application tier ensures that applications can connect to the negotiators from anywhere in the world and use the services provided by them. Note that, for a particular WSN, the gateway needs to be physically at the same place as the providers. But the negotiator and application tiers can be anywhere and they can be separate from each other as well.

B. *Design Principles*

To understand the design principle of the Bundle abstraction, we must first understand the paradigm of Physicalnet (see Figure 2). The goal of Physicalnet is to facilitate programming and organizing cyber physical systems in a multi-user and multi-network environment. The key features of Physicalnet are: 1) Enabling heterogeneity through lightweight Service Oriented Architecture (SOA). Various devices, such as sensor motes, actuators, smart phones and laptops, are registered as services in a center place so that users can access them through uniform APIs. 2) Differentiating the concepts of physical networks and administrative domains: one administrative domain can span multiple networks, and one application can involve resources from multiple administrative domains. In Figure 2, the cloudshape marking represents physical networks, while the colors represent different administrative domains. Administrative Domain 2 has members in both South America and Europe, and the laptop in Australia, it runs some application involve administrative domains in Asia and in Africa. 3) Supporting fine grained access right control and conflict resolution mechanism so that device owners can share their devices. If applications concurrently using the same devices have conflicting requirements, Physicalnet can resolve it through the conflict resolution module without terminating the applications.

As the essential programming abstraction of Physicalnet paradigm, Bundle is designed to efficiently support the above features and keep the resource constrained devices minimally engaged in group management. This is why we choose a centralized approach for designing the implementation support

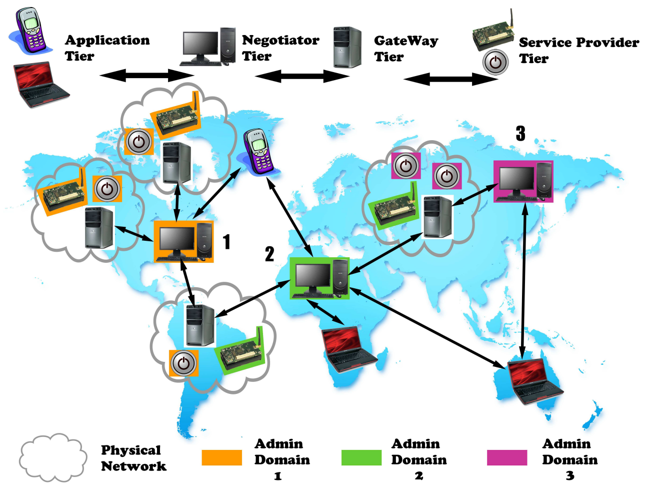


Fig. 2. Physicalnet Paradigm

of the Bundle abstraction. All the existing similar abstractions use distributed group management. Distributed design helps in reducing message transmission between nodes and the base station. It also facilitates network aggregation. But there are also some problems using a distributed scheme described as follows:

1) In cyber physical systems, it is necessary to allow the formation of dynamic sets of services provided by heterogeneous devices. Membership in a group is specified by arbitrary predicates which can involve any number of application variables. If the application variables vary over time, then the membership changes accordingly. For instance, a first group can be created to compute and update the average temperature in a building, and then a second group can be created to refer to all the ventilators that are in a room where a greater than average temperature is detected. In this case, when the average temperature changes, the selected set of ventilators changes.

Implementing such an arbitrary abstraction in a completely distributed fashion so that power consumption and latency are optimal is extremely challenging. A distributed framework has to upload binary code, bytecodes, or scripts to the resource constrained nodes. It would have to move code around when nodes move, making sure that each piece of code is transferred reliably. It would have to provide specific routing mechanisms allowing nodes to talk to one another even if they are located in different networks.

2) Memory and communication costs directly depend on number of groups a node is part of. This is because, the nodes need to store membership information and states in memory and also need to communicate this information to group members. This limits the number of groups a node may join.

3) Different sensors use different communication platforms. So it is impossible for all of them to communicate with each other directly and maintain a group. We need some powerful devices that can communicate using different protocols and thus facilitate group management. So the sensors will communicate with each other via these powerful devices.

4) It is non-trivial and costly to support multi-user access

```
public interface Bundle<T extends Service>
      extends BundleParent, Iterable<T>{

      boolean rule(T t);
      void foreach(T t);

      boolean contains(T t);
      int index(T t);
      int size();
}
```

Fig. 3. Bundle API

right control and conflict resolution. Firstly, each node needs to store all users access rights. Thus memory consumption directly depends on the number of users and rights. Secondly, for each request message, nodes need to validate whether it has the corresponding right, and if there is a conflict, nodes need to resolve it, which is a very costly process.

To solve these problems, we believe some centralized component is necessary in the architecture. The Bundle programming abstraction is supported in a centralized manner. Rather than decomposing code and shipping it to remote, unreliable, and resource constrained nodes, the Bundle brings the state of remote services, as well as the remote sensor streams to the application process. In a way, a Bundle works as a complement to the existing group based abstractions. Current implementation of Bundles is purely centralized, but it can be extended to include various distributed computing benefits. For example, a Bundle supports predicate pushdown (an efficient query processing technique for data collection in sensor networks as described in ([18], [17]), because only members of a Bundle send data to the base station, others send control packets only.

C. The Bundle Abstraction

The Bundle programming abstraction includes two parts: the definition of a group of sensors and actuators and the specification of what these devices should do. The Bundle abstraction allows the definition of a group to be arbitrarily complex, which means the definition of the Bundle memberships can involve any number of operators and application variables including, but not restricted to, constants, locations, sensor/actuator states, sensor streams, application parameters, user input, and numerical results computed by other Bundles. For instance, a Bundle can contain all the nodes that are temperature sensors, that are in the living room, that have more than half their energy remaining, that sense a temperature either greater than the average temperature in the room plus ten Fahrenheit degrees, or greater than a threshold that can be dynamically changed by the user. The second part is specification of what the members of a group should do which can depend on arbitrary operations involving complex functions that can execute only on powerful computers that can involve any application variables, including the Bundle member itself. For instance a Bundle of temperature sensors can be configured to be sensing at a rate specified by the user, and a Bundle can be configured so that the LEDs on a given node indicate the current intensity of noise sensed by the node.

Figure 3 shows the Bundle API. A Bundle is a generic set of sensors and actuators of type specified using the parameterized type T. By specifying T, programmers can for instance create a Bundle of temperature sensors, light actuators or cameras. A Bundle implements the type *BundleParent*, which means that a given Bundle can be used as a superset to define a Bundle containing a subset of its members. A Bundle implements the type *Iterable* so that the practical Java operator *for* can be used to iterate over the members. The programmer overrides the *rule (T t)* method to define the conditions of membership of a Bundle. The programmer overrides the *foreach* method to specify the state in which the members of the Bundle should be.

```
public class SamplingOneSensorPerRoom extends Application{

      public SamplingOneSensorPerRoom(){
            this.add(new Negotiator(HOST,PORT,USER,PASSWORD));
            this.execute(1000/*milliseconds*/);

            // For each room...
            for(final Zone z:this.getZones().getByType("Room")){

                  // Creates the bundle of all the temperature sensors in that room.
                  final Bundle<Temp> temps=new Bundle<Temp>(Temp.class,this){
                        public boolean rule(Temp t){
                              return z.contains(t);
                        }
                        public void foreach(Temp t){
                              return;
                        }
                  };

                  // Creates a bundle with a single temperature sensor in that room.
                  // Temperature sensed by those sensors is displayed periodically.
                  new Bundle<Temp>(Temp.class,temps){
                        public boolean rule(Temp t){
                              if(temps.index(t)==0){
                                    return true;
                              }
                              else{
                                    return false;
                              }
                        }
                        public void foreach(Temp t){
                              t.period.set(1000l/*milliseconds*/);
                              t.sense.set(true);
                              t.sense.whenNewSample(new Task<Long>(){
                                    public void run(Long l){
                                          System.out.println(z.getName()+": "+l);
                                    }
                              });
                        }
                  };
            }
      }
}
```

Fig. 4. The *SamplingOneSensorPerRoom* Application Written using Bundles

An important feature of the Bundle abstraction is its dynamic aspect. The Bundle membership is updated periodically so as to respect the membership specification. Figure 4 shows an example application *SamplingOneSensorPerRoom* that reports the temperature in each room using a single sensor per room. First, The application connects to the negotiator of temperature sensors. Then, the application sets the period of updating Bundle membership to 1 second (by the this.execute() method) i.e., in every 1 second, the application updates the list of group members of the Bundle, and also notifies the negotiator that the application is still alive and its requirements should be satisfied. For each room, first, the application creates the Bundle of all the temperature sensors in that room. Then, it creates for each room a second Bundle that contains a single temperature sensor. This sensor is configured to sense the temperature every second and the temperature samples are displayed on the standard output along with the name of the room.

Because of the periodic update of the membership, sensors that start satisfying membership rules (has to be a temperature sensor and has to be in a particular room) during application execution join the first Bundle for that room, and sensors that stop satisfying membership rules (leave the room) during application execution leave the first Bundle for that room. Note that if a temperature sensor, that leaves the first Bundle of a room, is that Bundle's first member, then the second Bundle for that room gets a new temperature sensor and that sensor is configured accordingly. If a node leaves a room, then as soon as it enters another room, it is connected to the negotiator through the new gateway and joins the first Bundle for that room. Now based on availability of other temperature sensors in that room, it may become a member of the second Bundle for that room.

D. Access Right Control and Conflict Resolution

In the future, cyber physical systems will be deployed in multi-user environments. While device owners are willing to share their resources, they also expect to protect them. Therefore, access right control mechanism is necessary in such open environments.

Physicalnet supports user-based access right control at the granularity of state and event level and provides conflict resolution mechanisms. This means the effects of a Bundle are limited by this mechanism, and the requirements can only be fulfilled when the user has enough rights and priority. Assume a user specifies a Bundle which requires turning on all the lights in a building. The results may be that only half of the lights are turned on. This is because the user may not have **WRITE** permission on some lights, or another super user who has higher priority requires some of the lights to be off, or some other user who has the same priority requires some of the lights to be off and the conflict resolution mechanism on those lights has the rule of first come first serve. In the implementation, the access right control is fulfilled by the Negotiator. When Bundle requirements of different applications arrive at the Negotiator, the Negotiator combines them into a table, then uses the access right table and conflict resolution modules to deduce the final desired requirement for each service, and finally sends these desired requirements to each service and returns the results to each Bundle.

Another key feature of a Bundle is that it enables dynamic access right specification. For example, an application may specify that no one can open the air conditioned vents in rooms where the windows are open. This can be implemented through two Bundles for each room: one is a collection of all the open windows in that room; the other is a collection of air conditioned vents in that rooms if there are one or more windows open, and the action is to specify **NO ACCESS** permission for all users. Because a Bundle periodically evaluates its membership and recomputes the requirements, it supports highly dynamic behaviors. Thus the features of dynamic access right specification are easily achieved.

E. Semantics

With the support of the Physicalnet middleware, the Bundle abstraction is able to create groups based on arbitrarily complex rules. Such rules can satisfy various application semantics. As an example, the rules can be based on node Id, network Id, domain Id, location or identity-centered. Take a simple application which aims to calculate the average temperature of a room. The interesting point is that such a simple application can have different meanings depending on different rules: 1) Node Id based. If a Bundle is created based on node Id such as nodes A and B, then the application semantics means that we intend to compute the average temperature of the rooms that Node A and Node B currently reside in. If they are in the same room, then the temperature is for that room; if they are in separate room, then the result is the average of the temperature of both rooms; if Node A and B are moved from the original room to another room, the result is for the new room instead of the old one. 2) Location based. If the Bundle is created based on a specific room, then the application semantics means computing the average temperature for that room. Therefore, if nodes A and B are in that room, the Bundle will use them to peform the computation; but if nodes A and B are removed from that room, then the Bundle dynamically discards them from the group. 3) Identity-centered. If the Bundle is created based on the rule of including all the nodes which are in the same room as a particular user with the specific identity, then the application semantics means tracking the user and computing the average temperature around him. So as the user moves, regardless of the node Id and room Id, the application always computes the average temperature of the room the user is currently in.

The discussion above is only about application logic level (AL) semantics, not the resource allocation level semantics (RA level). The resource allocation level semantics answers the question of what we should do if the application's requirements cannot be fully satisfied? Primarily, Physicalnet adopts a best effort semantics for RA. This means that if the application requests 10 temperature sensors, but currently only 7 of them are available (perhaps because other applications with higher priority require different settings on the sensors), then the application just proceeds with these 7 nodes. While this best effort RA is acceptable for many applications, the problem is that some applications have strict requirements. For example, if an application needs to measure the noise in rooms of a house to detect the occupancy, and if there are 5 rooms, but only sensors from 4 rooms are available, then we cannot completely satisfy all the requirements of this application. In other words, if we use best effort RA to satisfy this application, then the result is meaningless and these (not enough) allocated resources are wasted.

To support various application semantics, we differentiate three RA level semantics: best effort, strict, and conditional. Best effort means returning whatever resources are available; strict means returning all the resources requested by the Bundle or none; and conditional means if the the number of the resources returned by the Bundle is more than some threshold, then proceed, otherwise discard all resources and return failure. Note that when different semantic requirements from different applications arrive at a negotiator, the negotiator takes into consideration all the requirements and applications' priority, and outputs a resource allocation decision which tries to achieve the maximum resource utilization ratio without violating application priorities. The implementation and the evaluation of the resource allocation algorithm is outside the scope of this paper.

IV. Implementation

We now detail how the Bundle programming abstraction is implemented. First we describe how Bundles are managed in the application tier, then we explain the synchronization mechanism between the negotiators and the providers, which is followed by a discussion of how actuators are controlled, and finally we describe implementation of a visualization tool for Bundles that uses Google Earth.

A. Application Tier

Periodically, the application process, running on a remote PC, connects to the set of negotiators specified in the application code. Each negotiator has a global address of the form *negotiator IP address* + *TCP port*. From each negotiator, the application acquires the list of providers (e.g., motes, cameras, cell phones), the list of services for each provider (e.g., temperature, light and accelerometer sensor values for a mote), and the list of states (e.g., on/off status of a light actuator, sensing interval) for each service. The application downloads all service states when it first connects to a given negotiator. Then, it only downloads the differences from previous download.

Once all service data is downloaded, the previous application requirements for each state are transferred to a variable named *previousRequirement*. Then, the membership of all Bundles is recomputed by applying the overloaded *rule* method. After that, the new application requirements are computed by applying the overloaded *foreach* method to all the services that are member of the Bundle, and stored in a variable named *newRequirement*. Finally, *newRequirement* is uploaded to the negotiator for each state where *newRequirement* is not equal to *previousRequirement*. The cycle of download, re-computation, and upload repeats itself according to a configurable period.

Bundles can span multiple networks and administrative domains. An application can connect to several negotiators and each Bundle is a subset of all the services from all the negotiators. Each negotiator manages a set of providers pertaining to one or more users. Note that, multiple applications can use the same service provider and have conflicting requirements (e.g., one application may want the light to be on and other to be off). In that case Physicalnet uses conflict resolution mechanisms. The providers are free to move from one remote WSN to another. Whichever WSN it is currently in, the remote service provider always keeps the same global identifier of the form *negotiator IP address* + *negotiator TCP port* + *local identifier*, which allows the gateway of the current WSN to communicate with the appropriate negotiator and thus applications can uniquely identify a provider at any time.

B. Synchronization

The goal of the synchronization process is for the provider to forward its location and its data samples to the negotiator, and for the negotiator to reconfigure the state of the provider. The service provider periodically sends a control message to its gateway using a multi-hop wireless collection protocol. By default the provider sends one control message every p_max seconds. However, when a provider generates sensing samples, the period is decreased so as to forward these samples to the negotiator as fast as possible. Nevertheless, the period with which control messages are sent is not allowed to be smaller than p_min. The control message contains the global identifier of the provider, the last timestamp received from the negotiator (or 0 if no timestamp was received), the longitude and latitude of the provider, and a data section containing provider specific data samples.

When the gateway receives a control message, it reads the global identifier of the provider and infers the address of its negotiator. The gateway then stores the control message in a buffer dedicated to the inferred negotiator. Periodically (the period is configurable), the gateway forwards all the messages contained in the buffers to the appropriate negotiator using TCP/IP. When the negotiator receives a batch of messages, for each message, it queries its database to check whether the provider is registered. If the provider is registered, the negotiator updates the address of the gateway, and the location of the provider in the database. The negotiator then extracts the sensor samples from the data portion of the control message. Once the sensing samples are extracted from the control messages, they are stored in the database so that they can be later forwarded to the requesting applications.

To maintain synchronization, the negotiator reads the timestamp field of the control message, compares it with the timestamps stored in the database and thus infers whether the provider is up to date or not. If the provider is not up to date, the negotiator creates a configuration message that will configure the remote provider according to the latest application needs and send it to the appropriate gateway. This configuration message contains the global ID of the targeted provider, a new timestamp and configuration information for the provider. The gateway forwards the configuration messages to the appropriate provider using a multi-hop wireless routing protocol. Upon reception of a configuration message, the provider stores the new value for the timestamp, modifies the state of its actuators according to the negotiator desires, and initiates tasks as required by the modified values of its states.

C. Controlling the Actuators

In CPS, we often deal with unreliable actuators. This may cause major drawbacks if programmers remotely call (by Remote Procedure Call (RPC) or Remote Method Invocation (RMI) mechanism) these actuators to change their states. Consider an application that turns a light on and then desires to turn it off. Assume that when the application sends an RMI invocation to turn the light off, the actuator is unreachable. This may occur because of a temporary obstacle that significantly affects the wireless communication around it. So, the RMI call will fail and return an exception. As a consequence, a light actuator remains on even though it should be off. This problem is even more difficult to solve if the application controls the state of a large number of actuators. Another problem can arise from abnormal termination of an application due to a bug. This ungraceful termination may leave an actuator in a dirty state. Also, multiple applications may try to use the same actuators with conflicting requirements. (e.g., one application may need a light to be off, while the other may need it to be on).

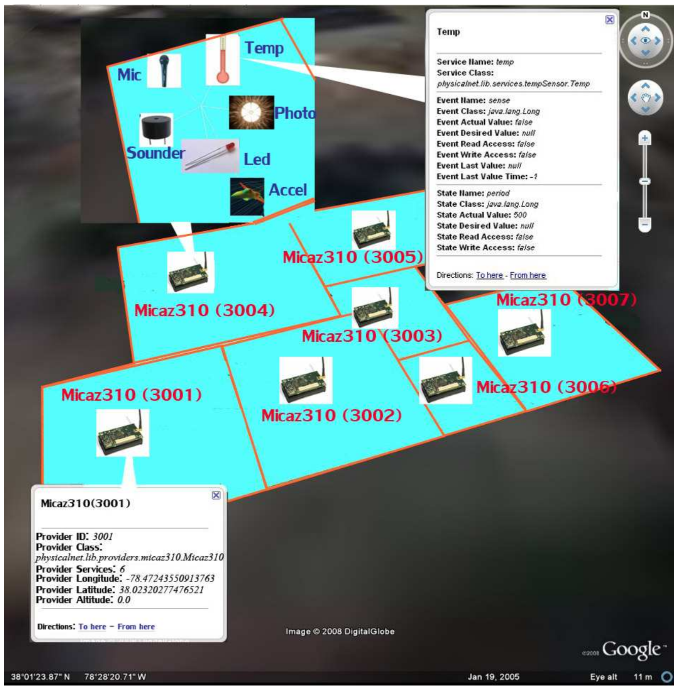
To resolve such problems, Bundles use the concept of state for each actuator. Manipulating actuators using states is very different from manipulating it using RMI. Consider the example of turning a light on. An RMI call directly connects the application to the remote light actuator and turns it on. By contrast, in our design, the application only generates a requirement for the light to be on and sends it to the negotiator by RMI. The negotiator of the light actuator then tries to fulfil this requirement by turning the remote light actuator on. If from the actuator's next periodic update, the negotiator finds that the requirement is not yet fulfilled, then it retries until being successful. It is also possible to specify a timeout interval from the application level so that the negotiator only retries until that interval.

Note that the state of the actuator does not change in the negotiator until the requirement is actually fulfilled. The negotiator stores this requirement as long as the application does not cancel it (or terminate). Also, the negotiator may store several such requirements and decide, according to rules specified by the node owner, which requirement should be satisfied. The programmers can check at any time whether their requirements are being satisfied or not and take appropriate action. Furthermore, to solve the problem of abnormal application termination, an application that specifies requirements to a negotiator also has to periodically send 'alive' messages to the negotiator. When the application aborts, the negotiator notices the death of the application (by using a timer that keeps track of when was the last 'alive' message received from that application) and automatically cancels all of its requirements. Then either the state of the actuator returns to its default value or the requirements of other applications are satisfied.

Consider the same application that turns a light on and then desires to turn it off. Now, if the light actuator is unreachable when the application desires to turn it off, the negotiator reattempts to turn the light off until it succeeds. Suppose, a programmer uses a Bundle to specify that all the light actuators in a room should be turned on. When a new light joins the Bundle, the Bundle sets the requirement for the state of the light to be turned on. When a light leaves the Bundle, the requirement for the state of the light is set to be null.

D. Visualization Tool

To support programming with Bundles various tools have been implemented. This includes utilities such as a database administration tool, an access rights control configuration tool, and a diagnosis tool. The most attractive tool is the visualization tool which uses Google Earth as its interface to display device locations and service states. As Figure 5 shows, the visualization tool renders the location (longitude, latitude and altitude) of the devices, and shows which building and in which rooms the devices currently reside. When you place the mouse arrow onto the device icon, detailed information, such as device ID, the category of the device, the owner of the device, and how many services it includes, are displayed. When you click the device icon, it shows all the services it includes (see the icons at the top of Figure 5), and the detail information, such as ID, category, access rights, current state values, and current sensing values, will be displayed if you further click on those service icons.



Temp

Service Name: *temp*
Service Class: *physicalnet.lib.services.tempSensor.Temp*

Event Name: *sense*
Event Class: *java.lang.Long*
Event Actual Value: *false*
Event Desired Value: *null*
Event Read Access: *false*
Event Write Access: *false*
Event Last Value: *null*
Event Last Value Time: *-1*

State Name: *period*
State Class: *java.lang.Long*
State Actual Value: *500*
State Desired Value: *null*
State Read Access: *false*
State Write Access: *false*

Directions: To here - From here

Micaz310(3001)

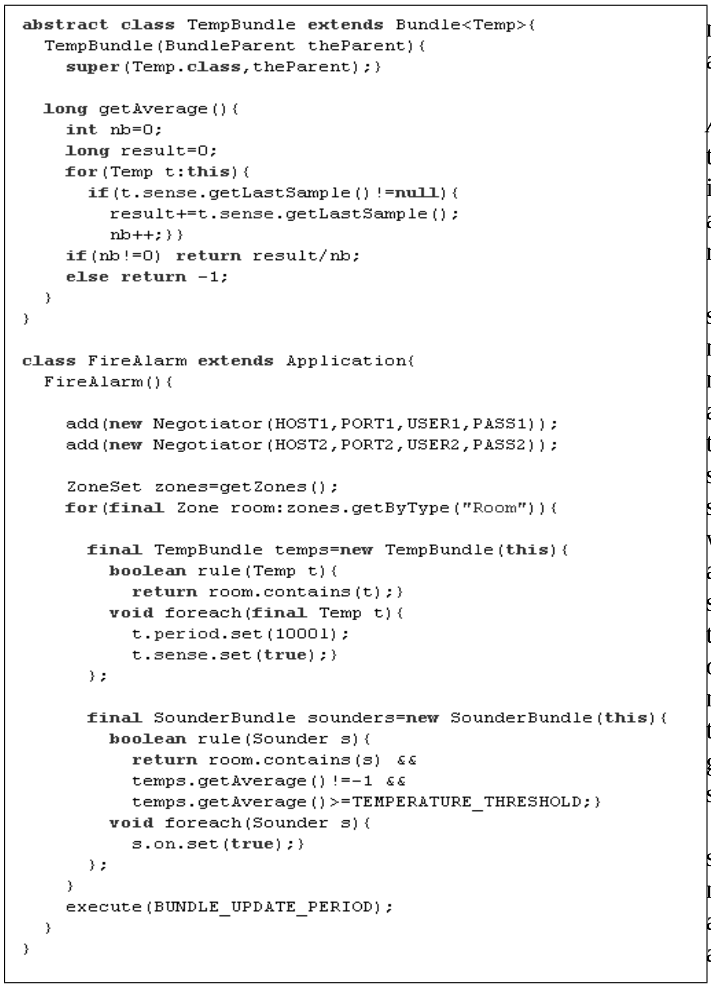
Provider ID: *3001*
Provider Class: *physicalnet.lib.providers.micaz310.Micaz310*
Provider Services: *6*
Provider Longitude: *-78.47243550913763*
Provider Latitude: *38.02320277476521*
Provider Altitude: *0.0*

Directions: To here – From here

Fig. 5. Visualization in Google Earth

To implement this tool, the application first needs to create a Bundle including the interested services. Then the Negotiator periodically collects the corresponding information such as the location, the state, and the event values. After retrieving this information from the Negotiator, the application periodically generates a KML file and feeds it into Google Earth to render these images. Therefore, all the states you see in Google Earth are in real time. If you move a device from one room to another room, you can see the change in Google Earth.

This visualization tool has several benefits: first, before you create the Bundles for your application, it is convenient to check the availability of interesting services through Google Earth (e.g., how many sensors exist and do I have the access rights?); Second, it can be used as a debug tool. You can check the actual values, desired values and the application's required value of a service state through Google Earth in order to figure out whether the current problem is caused by device itself, the network, or the application's priority.



```
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);
  }
}
```

Fig. 7. The *FireAlarm* Application

V. Evaluation

In this section, we provide an evaluation of our key research contributions. We evaluate the conciseness and mobility support of 32 applications coded using the Bundle programming abstraction. Then we evaluate the energy consumption of Bundles. We also evaluate delay in actuator configuration.

A. Conciseness and Mobility

To show programming conciseness and a wide variety of applications, many of which involve mobile nodes, we implemented 32 applications. They are summarized in Figure 6. They include environmental monitoring applications (e.g., AcousticDetector, AverageHumidity and FloodWarning), tracking applications (e.g., SpyBug, LowEnergyAlert), control automation applications (e.g., Illuminator, Tracker, TempRegulator, AutoLocks and OnlyWhen), and monitoring and alarm applications (e.g., PhotoAlarm, ParkingSpacefinder, FireAlarm, NeighborhoodWatch and AntiThiefTags). Each of them is programmed in less than 60 lines of code.

Now we provide description and Java code for 2 of the above applications. The first application, *FireAlarm*, is chosen for its simplicity. The second application, *NeighborhoodWatch*, is a more complex application involving multiple sensing modalities. Both of these applications contain actuators which are controlled based on feedback from the sensors.

1) The FireAlarm Application: Figure 7 shows the *FireAlarm* application. 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 features of this application are that: a) If sensor nodes change rooms, their temperature samples automatically start contributing to the temperature average of the new room. b) All the temperature sensors that are indoors are automatically found 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 API and the sounder API: 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 cyber physical systems that must run for a long time (months, years) in networks where sensors and actuators can be moved, removed, and/or added. Using Bundles, nodes automatically adapt to application requirements over time.

In the FireAlarm code of Figure 7, we first create the FireAlarm application class by inheriting from the 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 different areas. We then obtain the set of zones (each zone represents a room) stored in those negotiators. For each 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 mem-

Application Name	Application Description	NLC
(1) TempSensorCensus *(2) TempSampler* (M)	These applications find (1) all temperature sensors or (2) a particular temperature sensor of a remote network.	(1) 12 (2) 10
SpyBug (M C)	This application tracks the location of a list of tags and records it	20
LowEnergyAlert (M C)	This application makes sensor nodes that lack energy ring when a staff member (who wears an identifying tag) responsible for changing batteries wanders within 50 meters.	22
AcousticDetector (M)	This application stores the location of all the acoustic sensors of a network, as well as the amount of noise they sense.	18
Illuminator (A)	This application turns on all the lights of a remote network.	9
OneSensorPerRoom (C)	For each room of a building, this application shows the name of the room and the temperature sensed by a single temperature sensor in that room.	22
(1) AverageTemp *(2) AverageHumidity* *(3) AverageNoise* (M C H)	Compute and show average (1) temperature or (2) humidity or (3) noise in a remote network. For the second case, it sends an alert to the user if there are less than 10 sensors available for computing the average. For the last case it uses two types of sensors having different interfaces manipulated seamlessly by a reusable adapter.	(1) 13 (2) 14 (3) 18
BimodalOccupancy (M)	This application checks whether a remote hangar is occupied. By default it uses motion sensors but if the number of motion sensors available is less than a specified number, it uses acoustic sensors instead.	32
(1) RoomTemp1 *(2) RoomTemp2* *(3) RoomTemp3* (M C)	These applications show the temperature sensed by the nodes contained in the room where a specified tag is placed. The displayed temperatures are always (1) the ones in the same room as the tag or (2) the ones in that particular room or (3) the ones that were initially in that room. In all these cases the tag and the sensors can be mobile and new sensors can be added in the system.	(1) 18 (2) 19 (3) 20
ConsiderateSensing (M)	This application shows the temperature sensed in a remote area using only nodes that have sufficient energy reserves.	22
FloodWarning (H)	This application monitor water levels and displays alerts on nearby road message boards in case of a flood.	24
(1) OnlyWhenInRoom *(2) OnlyWhenEnergy* *(3) OnlyWhenIdle* *(4) OnlyWhenAtHome* *(5) OnlyWhenDark* *(6) OnlyWhenNoTV* *(7) OnlyWhenWClose* (M U H)	These applications can dynamically change reading rights (1) to the acoustic sensors so that they can be accessed only when in a conference room, (2) to the temperature sensors so that they can be sampled only when there is enough energy left, (3) to the accelerometers of a set of laptops so that they can only be read when idle, or (4) writing rights of the light actuators so that they can be modified only by users at home, or (5) only when the average light intensity is less than a threshold, or (6) can resolve conflicts between radio and TV so that radios within 200 meters of a TV can be turned on only if the TV is off, or (7) between air conditioners and open windows so that air conditioner vents can be opened only when window is closed.	(1) 17 (2) 19 (3) 17 (4) 17 (5) 19 (6) 16 (7) 16
(1) PhotoAlarm *(2) FireAlarm* (M A)	This application turns on (1) all the sounders in a room if the average light intensity or (2) temperature in that room exceeds corresponding threshold.	(1) 21 (2) 21
ParkingSpaceFinder (A)	This application finds a free parking space in a parking lot, and reserves that space.	20
RoomOccupancy (M)	This application uses acoustic sensors to infer whether remote rooms are occupied. A room is considered occupied if at least two acoustic sensors have been triggered in the last 10 minutes.	31
Tracker (M U H A)	This application turns on television sets, music players, and lights wherever the user of a tag goes near, it also resolves possible conflicts.	30
NeighborWatch (M U C H A)	A set of neighbors contribute to a neighborhood watch and wear tags. If not tags are in a given house, all the accelerometers, and light sensor in that house are turned on. A buzzer on all the tags is turned on to alert all the neighbors in case accelerators are moved or light intensity changes are detected.	56
AntiThiefTags (M H A)	In each room of a building, this application records the position of tagged objects when it starts. If any object is moved, all the alarms in that room are raised until the object is returned to its place.	26
AutoLocks (A)	This application automatically opens locks when any authorized users is within 1 meter of the lock and closes them when no authorized user is within one meter of the lock.	17
TempRegulator (A H)	This application automatically configures an air conditioned unit according to the current average temperature of a building. Also, it closes and open vents according to the average temperature in each room.	33

Fig. 6. Examples of Applications Written Using Bundles with Corresponding Number of Lines of Code (NLC). The meaning of the tags are defined as follows: M–mobility aware, A–includes actuators, C–across network programming, U–multiple users, and H–heterogeneous devices.

bership 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 graphical interface allowing users to dynamically change the temperature threshold.

2) The NeighborhoodWatch: Figure 8 shows 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 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.

In the *NeighborhoodWatch* code shown in Figure 8, 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 levels exceed a specified threshold. For each building, we create the Bundle of all the

```
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(1000l);
          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(1000l);
          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);
  }
}
```

Fig. 8. The *NeighborhoodWatch* Application

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. One interesting feature of the NeighborhoodWatch application is that it is easy to extend it to many neighbors and many houses.

Lack of space precludes full descriptions of all 32 applications, but from these 32 examples we see that Bundles can concisely specify the logic of a variety of applications. These applications are proof to our previous claim that Bundles can: 1) group heterogeneous types of sensors and actuators; 2) handle both intra and inter network mobility; 3) support

```
public class ParkingSpaceFinder extends Application{

      private Bundle<ParkingSpace> spaces;
      public ParkingSpaceFinder(){
            this.add(new Negotiator(HOST,PORT,USER,PASSWORD));
            this.execute(100/*milliseconds*/);

            // Creates the bundle of sensors that detect whether parking
            // spaces are free or occupied. This sensors also have a
            // state that indicates whether their space is reserved.
            spaces=new Bundle<ParkingSpace>(ParkingSpace.class,this){
                  public boolean rule(ParkingSpace p){return true;}
                  public void foreach(ParkingSpace p){
                        p.period.set(1000l/*milliseconds*/);
                        p.senseOccupied.set(true);
                  }
            };
      }

      // Get the location of the closest parking space
      // according to current location.
      synchronized public ParkingSpace getParking(Gps location){
            List<ParkingSpace> spacesCopy=spaces.copy();
            if(spacesCopy.size()==0) return null;
            ParkingSpace closest=spacesCopy.get(0);
            for(ParkingSpace p:spacesCopy){
                        if(p.senseOccupied.getLastSample()==false &&
                        p.reserved.get()==false && location.distance(closest.gps()) >
                        location.distance(p.gps())){
                        closest=p;
                  }
            }
            closest.reserved.set(true);
            return closest;
      }
}
```

Fig. 9. The *ParkingSpaceFinder* Application

applications that group sensors and actuators from multiple remote WSNs; 4) support multiple users and multiple applications to use the same sensors and actuators concurrently (because many of these applications are using the same devices and they can run concurrently).

To further illustrate ease and conciseness of programming with Bundles, we compare the code of a very simple application, *ParkingSpaceFinder* using nesC [6] and our design. The nesC code can be found in [15] and our code is shown in Figure 9. The nesC code has 42 lines of code and our code has 20 lines of code. The nesC code needs to implement explicit mechanisms to prevent one car to be reserved multiple times for the same user, and to make sure that the chosen parking space is the closest one. By contrast, Bundles relay all the necessary data in a central process which can easily check the nodes and reserve one within a synchronized method, thereby resolving the consistency issues that make the coding of application that execute in a distributed manner more difficult. As applications become more complex, the percentage improvement in code size between Bundles and nesC will grow.

B. *Energy Conservation*

While there are many benefits for Bundles, by using a centralized architecture, Bundles are expected to consume more energy than using the traditional distributed approach. To quantify the energy performance of Bundles, using simulation we compare the energy consumption of a target tracking application by using a distributed Vigilnet [10] design and a

Node state	Radio state	Processor state	Sensor state	Total power
Initialization	receive	active	off	49.449mW
SentrySleep	off	sleep	off	42μW
NonSentrySleep	LPL	sleep	off	450μW
AwakeComm	receive	active	off	49.449mW
AwakeCommSensing	receive	active	on	71.45mW
AwakeSensing	receive	active	on	70.01mW

TABLE II
POWER CONSUMPTION ACCORDING TO NODE STATE. THIS TABLE DESCRIBES THE VARIOUS SLEEP STATES AND ACTIVE STATES OF THE SENSOR NODES. WE OBTAINED THE POWER CONSUMPTION VALUES BY EMPIRICALLY MEASURING THE POWER CONSUMPTION OF XSM NODES.

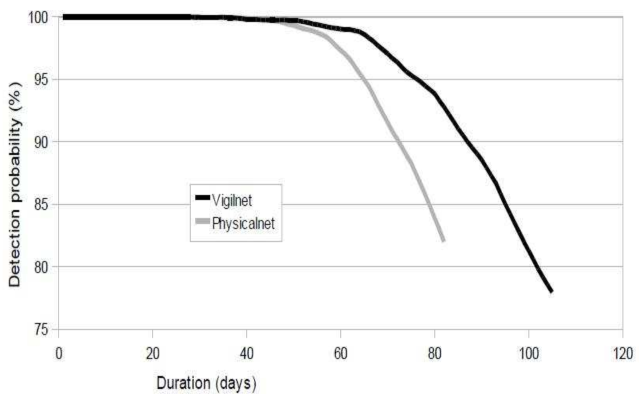


Fig. 10. Average Detection Probability for Sentry Selection with Duty Cycle Scheduling

centralized Bundle design. In addition to being distributed, Vigilnet allows in-network data aggregation and node-to-node communication, which is used to optimize the energy conservation. By contrast, Bundles use a centralized solution, in which in-network data aggregation and node-to-node communication are not allowed.

The goal is to compare the lifetime of the entire network which is defined as the number of days for which the detection probability of target, which is defined as the percentage of successful detections among all targets that enter the network area during one day, remains greater than 90%. The simulator is based on XSM platform [5] and its empirical power consumption model (shown in Table II). We suppose that a mote dies when it has used 85% of its available energy and the sensing range of sensors is 10 meters. This simulator randomly distributes 10,000 nodes within a square of edge 1000 meters. In this simulator, a target enters and exits the network area at random points on the edges of the network. The trajectory of the target is a straight line with a constant speed. There is at most one target within the sensor field at any point in time.

In our simulator, both designs use the TinyOS collection tree protocol. We assume that there is one base station (gateway) for every 100 nodes. Nodes self-organize into a collection tree rooted at their closest base station in terms of number of communication hops. The base stations are connected through TCP-IP to a central computer (which acts as both the negotiator tier and application tier) to report detections. The simulator assumes that Vigilnet uses a flooding protocol to

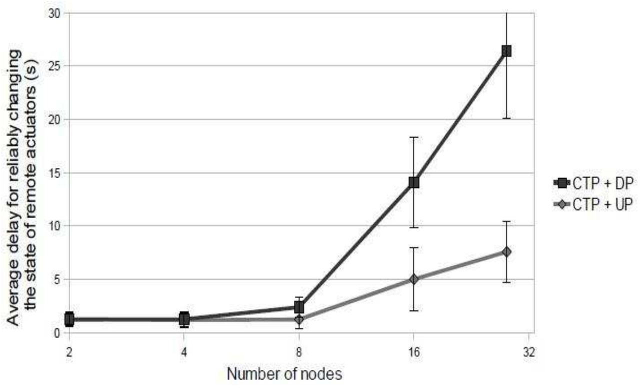


Fig. 11. Average Delay for Reconfiguring the State of Remote Actuators

reconfigure the nodes. A distributed algorithm is used to select which nodes should be awake and which node should be asleep to save power while maintaining appropriate sensing coverage of the field. On the other hand, Bundles use a unicast protocol for reconfiguration. In our first experiment, we assume none of the designs employ any energy conservation techniques and in the second experiment we assume both of them employ an energy conservation technique called sentry selection, which is implemented in conjunction with duty cycle scheduling [10]. In case of the Bundles, this technique is implemented in such a way that sentry selection is performed by the base stations and duty cycles of the nodes are configured by the base stations by control messages.

When no power management techniques are used, both the designs present the same power consumption patterns. In case of sentry selection with duty cycling, Figure 10 presents the detection probability, as a function of the duration for which the network has been deployed. We observe that the network lifetime using Bundles is only 83.9% of the lifetime using Vigilnet (the lifetime is 73 days for Bundles and 87 days for Vigilnet). The reasons that Bundles consume more energy than Vigilnet are: first, Vigilnet uses node to node communication, while in our design, all operations involving multiple nodes go through a central process; second, Vigilnet uses data aggregation, while in case of Bundles, all nodes report directly to the central process through the base station; third, Vigilnet floods a single message to initiate the sentry selection, while we must send several unicast messages to each node, one by one. Although Bundles consume more energy than the traditional distributed applications as in this example, the achieved lifetime is still acceptable.

C. Delay in Actuator Configuration

As we mentioned in Section IV-C, a negotiator remembers the requirements of applications and tries to configure the actuators of the corresponding Bundles until successful. In this section, we evaluate the average delay for reliably changing the state of remote actuators as function of the number of actuators in a Bundle.

For this experiment, we use real micaz motes. The experimental setup is as follows. We write an application that creates a Bundle of light actuators (LEDs) of the micaz motes. We vary the number of members of the Bundle from 2 to 28 actuators. The gateway, the negotiator and the application are implemented in the same machine. The application switches the light actuators of the Bundle on and off in the following way. When the actual state of a LED is equal to off, the application generates a requirement to turn it on. When the actual state of a LED becomes equal to on (and that this information reaches the application process), the application changes its requirement to turn the LED off, an so on and so forth. In the meanwhile, the application measures the time it takes from the generation of a new requirement to its satisfaction.

As the underlying routing protocol, the TinyOS collection tree protocol (CTP) is used to build a collection tree rooted at the gateway. We use CTP under two different settings. In one setting, we use the TinyOS dissemination protocol (DP) to reliably deliver configuration messages from the gateway to the actuators. In the other setting, we use unicast protocol (UP) for this purpose.

The results of this experiment are shown in Figure 11, which graphs the average delay for reliably changing the state of remote actuators, according to the the number of actuators in the Bundle. We first note that, whether DP or UP is in use, for Bundles of up to 8 actuators, the average delay for reliably changing the state of remote actuators varies from 1.24 to 2.24 seconds. Note that this delay includes the delay for the application to change its requirement on the negotiator, the delay for the negotiator to compute the new desired state of the remote actuator, the delay for the gateway to contact the negotiator about update, the delay for the negotiator to reply to the gateway with a configuration message, the delay for the gateway to forward the messages to the remote actuator, and the delay for the remote actuator to send an acknowledgment back to the gateway, which forwards it to the negotiator, which forwards it to the application. We observe that when using a Bundle larger than eight actuators, the performance starts to degrade, whether DP or UP is used; however, UP offers significant performance improvements over DP: The average delay in the case of DP increases 3.72 times as fast as the average delay in the case of UP. This can be explained as follows. In small networks with up to eight nodes, DP and UP have similar delays, because all the nodes are within one communication hop of the base station and, as a consequence, UP does not offer any advantage over DP. In the case of networks with more than eight actuators, DP suffers from a lot of contention as it tries to send each message to each node. The contention is exacerbated by the fact that DP nodes constantly try to communicate with each other to check whether they have received the last message.

VI. Limitations

Although the Bundle is a powerful and flexible programming abstraction, there are some limitations mainly due to its centralized architecture. Firstly, Bundle does not have the capability of in-network aggregation and processing. This may result in more energy consumption than the distributed programming abstraction in some specific scenarios. Secondly, all the configuration messages from the applications must be first uploaded to the negotiator. After being processed, these messages go from the negotiator to the network nodes via the gateway. The updates from the network nodes also have to go to the negotiators first, from where they eventually reach the applications. Therefore, the response time may not be not as good as the other programming abstractions. Experiments performed in [24] show that Bundles should not be used for applications having responsiveness requirement less than two seconds. Thirdly, the Bundle requires at least one base station per network (i.e., gateway). However, in some scenarios, a centralized machine may not be available (e.g., in the wild area). Finally, another limitation of Bundles is that they create a strong dependency between the negotiators and the resource constrained sensors: the sensors must be able to communicate with their negotiator in order to configure themselves properly and store the data they generate. In future, we aim to extend Bundle design to support in-network aggregation and local processing within Bundles so that energy consumption and responsiveness improve.

VII. Conclusion

In this paper, we present a group based abstraction called Bundle for cyber physical systems. Characteristics of Bundles include easy and concise across networks programming, support for both intra and inter network mobility and multiple applications using same sensors and actuators concurrently. Evaluations show that application programming is concise and energy consumption is also acceptable. Memory usage for each device is constant regardless of the number of concurrent applications.

VIII. Acknowledgements

This work was supported, in part, by NSF Grants CNS-0626632 and IIS-0931972.

References

[1] A. Bakshi and V. K. Prasanna, "The abstract task graph: a methodology for architecture-independent programming of networked sensor systems," in *EESR*, 2005.
[2] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode, "Spatial programming using smart messages: Design and implementation," in *ICDCS*, 2004, pp. 690–699.
[3] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "Tinygals: A programming model for event-driven embedded systems," in *SAC*, 2003, pp. 698–704.
[4] C. Curino, M. Gianni, M. Giorgetta, A. Curino, A. L. Murphy, and G. P. Picco, "Tinylime: Bridging mobile and sensor networks through middleware," in *PerCom*, 2005, pp. 61–72.
[5] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler, "Design of a wireless sensor network platform for detecting rare, random, and ephemeral events," in *IPSN*, 2005, pp. 497–502.
[6] D. Gay, P. Levis, R. V. Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *PLDI*, 2003, pp. 1–11.
[7] O. Gnawali, K.-Y. Jang, J. Peak, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler, "The tenet architecture for tiered sensor networks," in *SenSys*, 2006, pp. 153–166.
[8] R. Gummadi, O. Gnawali, and R. Govindan, "Macroprogramming wireless sensor networks using kairos," in *DCOSS*, 2005, pp. 126–140.
[9] T. He, J. A. Stankovic, R. Stoleru, Y. Gu, and Y. Wu, "Essentia: Architecting wireless sensor networks asymmetrically," in *INFOCOM*, 2008, pp. 1184–1192.

[10] T. He, P. Vicaire, T. Yan, Q. Cao, G. Zhou, L. Gu, L. Luo, R. Stoleru, J. A. Stankovic, and T. F. Abdelzaher, "Achieving long-term surveillance in vigilnet," in *INFOCOM*, 2006, pp. 1–12.

[11] T. W. Hnat, T. I. Sookoor, P. Hooimeijer, W. Weimer, and K. Whitehouse, "Macrolab: a vector-based macroprogramming framework for cyber-physical systems," in *SenSys*, 2008, pp. 225–238.

[12] D. Jacobi, P. E. Guerrero, I. Petrov, and A. Buchmann, "Structuring sensor networks with scopes," in *EuroSSC*, 2008.

[13] D. Jacobi, P. E. Guerrero, K. Nawaz, C. Seeger, A. Herzog, K. V. Laerhoven, and I. Petrov, *From Active Data Management to Event-Based Systems and More*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6462, ch. Towards Declarative Query Scoping in Sensor Networks, pp. 281–292.

[14] J. King, R. Bose, H.-I. Yang, S. Pickles, and A. Helal, "Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces," in *LCN*, 2006, pp. 630–638.

[15] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," *SIGPLAN Not.*, vol. 42, no. 6, pp. 200–210, 2007.

[16] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic, "Envirosuite: An environmentally immersive programming framework for sensor networks," *Trans. on Embedded Computing Sys.*, vol. 5, pp. 543–576, 2006.

[17] S. Madden, R. Szewczyk, M. J. Franklin, and D. Culler, "Supporting aggregate queries over ad-hoc wireless sensor networks," in *WMCSA*, 2002, pp. 49–58.

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

[19] L. Mottola and G. P. Picco, "Logical neighborhoods: A programming abstraction for wireless sensor networks," in *DCOSS*, 2006, pp. 150–168.

[20] 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*, 2004, pp. 78–87.

[21] Y. Ni, U. Kremer, and L. Iftode., "Spatial views: Space-aware programming for networks of embedded systems," in *LCPC*, 2003, pp. 258–272.

[22] K. Römerand, C. Frank, P. J. Marrón, and C. Becker, "Generic role assignment for wireless sensor networks," in *EW11*, 2004, pp. 7–12.

[23] F. Sun, C.-L. Fok, and G.-C. Roman, "schat: A group communication service over wireless sensor networks," in *IPSN*, 2007, pp. 543–544.

[24] P. A. Vicaire, Z. Xie, E. Hoque, and J. A. Stankovic, "Physicalnet: A generic framework for managing and programming across pervasive computing networks," in *RTAS*, 2010.

[25] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *NSDI*, 2004, pp. 29–42.

[26] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *MobiSYS*, 2004, pp. 99–110.

[27] 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.

Pascal A. Vicaire Pascal A. Vicaire graduated with a PhD in Computer Science from the University of Virginia in October 2008 and is now an Engineer for the Elastic Compute Cloud division of Amazon Web Services. His research interests include Wireless Sensor Networks and Cloud Computing.

Enamul Hoque Enamul Hoque completed his Bachelor of Science degree in Computer Science from Bangladesh University of Engineering and Technology in 2007 and obtained a Master in Computer Science (MCS) Degree from University of Virginia in 2010. He is now a Ph.D. candidate in Computer Science Department of University of Virginia supervised by Professor John A. Stankovic. His current research directions are on middleware for wireless sensor networks, applications of wireless sensor networks in behavioral monitoring for healthcare and energy conservation.

Zhiheng Xie Zhiheng Xie graduated with a Master Degree in Software Engineering from Tsinghua University China in 2007 and is now a Ph.D. candidate in Computer Science Department of University of Virginia, following advisor John A. Stankovic. Xie's current research directions are on wireless sensor networks middleware, communication reliability and localization.

John A. Stankovic Professor John A. Stankovic is the BP America Professor in the Computer Science Department at the University of Virginia. He served as Chair of the department for 8 years. He is a Fellow of both the IEEE and the ACM. He won the IEEE Real-Time Systems Technical Committee's Award for Outstanding Technical Contributions and Leadership. He also won the IEEE Technical Committee on Distributed Processing's Distinguished Achievement Award (inaugural winner). He has won four Best Paper awards, including one for ACM SenSys 2006. He has given more than 25 Keynote talks at conferences and many Distinguished Lectures at major Universities. He was the Editor-in-Chief for the IEEE Transactions on Distributed and Parallel Systems and was founder and co-editor-in-chief for the Real-Time Systems Journal. His research interests are in real-time systems, distributed computing, wireless sensor networks, and cyber physical systems. Prof. Stankovic received his PhD from Brown University.