Bundle: A Group Based Programming Abstraction for Cyber Physical Systems

Pascal A. Vicaire
pascal.vicaire@gmail.com
University of Virginia

Enamul Hoque
eh6p@virginia.edu
University of Virginia

Zhiheng Xie
zx3n@virginia.edu
University of Virginia

John A. Stankovic
stankovic@cs.virginia.edu
University of Virginia

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. 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.

1 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 ([19], [18]) or some higher-level, application-defined notion of proximity ([14], [8]). 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. Moreover, none of them supports grouping of actuators. A Bundle extends the previous group based abstractions by addressing these limitations. It focuses on keeping the resource constrained sensors and actuators simple, i.e., application logic is shifted

from them to the centralized base station. No matter how many applications use a sensor, the logic on them remains the same. A Bundle is a Java class that provides a powerful abstraction with the following main features: a) ease of programming that facilitates writing applications for CPS concisely; b) grouping sensors and actuators from different networks and users; c) support for intra and inter network mobility of sensors and actuators belonging to a Bundle by dynamic update of their membership; and d) enabling multiple applications to use the same sensors and actuators concurrently.

The main contributions of this paper are: a) a new centralized 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, 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 and its acceptable energy overhead.

Note that the Bundle abstraction is implemented on top of the Physicalnet middleware [17]. While the Bundle is an important part of the Physicalnet framework, it is an orthogonal concept. In this current paper, we fully describe the Bundle as a group based abstraction, discuss how it is different from existing group based abstractions, and evaluate its conciseness and energy consumption. A previous paper ([17]) discusses the features of Physicalnet including the detailed description of its middleware architecture, its access right control mechanism, how it is different from other existing middleware, and an evaluation of its performance and concurrency degree. So these two papers are fundamentally different with minimal overlap.

The paper is organized as follows. Section 2 describes related work and compares Bundles with other similar abstractions. Section 3 explains the Bundle abstraction in detail. Section 4 describes its implementation details. Section 5 presents evaluation results. We conclude in section 6.

2 Related Work

Existing group based abstractions have several shortcomings that limit their applicability in cyber physical systems. Bundle has been designed to overcome these shortcomings. Table 1 summarizes some of the important differences between Bundles and other abstractions. We discuss the details in this section.

Hood [19] 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 platform. 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 [18] 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 Hoods, 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.

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 1. Comparison of Bundle with other Abstractions

Logical neighborhood [14, 2] 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 for this case. It does not support actuators or devices with 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 [8] 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 also 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 [16] 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 as well.

Spatial Views [15] and Spatial Programming [1] 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 with an emphasis on spatial node reference consistency, and reference timeout. 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.

3 Design

In this section we describe the main design principles of the Bundle abstraction, its underlying architecture and how Bundles work.

3.1 Design Principles

The main design philosophy of a Bundle is keeping the resource constrained devices minimally engaged in group management. This is why we choose a centralized approach for designing 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 in 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. The 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.

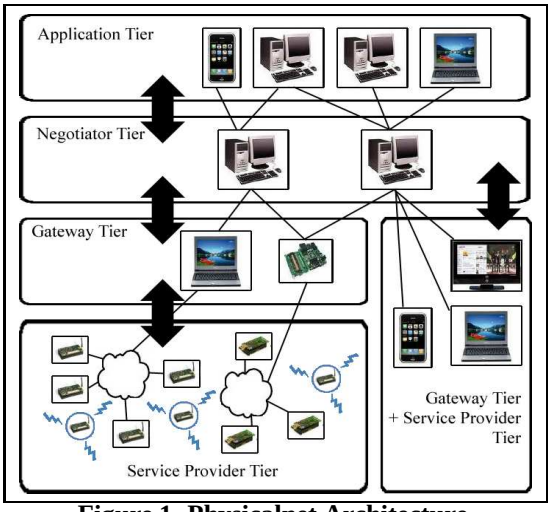

Figure 1. Physicalnet Architecture.

To solve these problems, we believe some centralized component is necessary in the architecture. The Bundle programming abstraction is centralized. 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 ([13, 12]), because only members of a Bundle send data to the base station, others send control packets only.

3.2 Architecture

The Bundle is designed as part of Physicalnet [17], a lightweight service oriented architecture middleware for wireless sensor networks. Detailed description and evaluation of Physcialnet are provided in [17]. Here, we briefly outline the implementation of Physicalnet only at the level of detailed 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 sensors 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 on remote and more powerful computers (similar to Tenet [5], Essentia [6], and Atlas [9]). 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.

3.3 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

```
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();
}
```

Figure 2. Bundle API

```
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);
                                    }
                              });
                        }
                  };
            }
      }
}
```

Figure 3. The *SamplingOneSensorPerRoom* Application Written using Bundles

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 2 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.

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 3 shows an example application *SamplingOneSensorPerRoom* that reports the temperature in each room using a single sensor per room. 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. Here, each room is a different network and the application is written using sensors from multiple networks. Similarly, inter network mobility can be supported. 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.

4 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 and finally we discuss how actuators are controlled.

4.1 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 that are discussed in detail in [17]. 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.

4.2 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*. Both *p_max* and *p_min* are specified when the Physicalnet binary is installed on the provider. 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 not registered with the negotiator, the message is dismissed. 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. Upon reception, the gateway stores the configuration message in a queue. The gateway forwards the configuration messages one after the other 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.

4.3 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.

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. Note that the state of the actuator does not change in the negotiator until the requirement is actually fulfilled. Here, our assumption is that as long as the actuators are reachable, the requests from the negotiators and the periodic updates from the providers are never lost in the communication channel. The negotiator stores this requirement as long as the application does not cancel it (or terminate). Furthermore, 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 his requirements are being satisfied or not and take appropriate action.

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 bundles sets the requirement for the state of the light to null.

```
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 5. The *Tracker* Application

5 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. We also evaluate the energy consumption of Bundles.

5.1 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 4. 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, *Tracker*, is interesting in that it demonstrates how seamlessly Java service providers and TinyOS service providers can interact. 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.

5.1.1 The Tracker Application

Figure 5 shows 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 implemented using a Java service provider that runs on a PC and that turns a mp3 player on or off. The television service is implemented using a Java service provider that runs on a PC and that turns a video player on or off.

In the Tracker code of Figure 5, 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.

```
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 6. The *NeighborhoodWatch* Application

5.1.2 The NeighborhoodWatch

Figure 6 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 intru-

sion 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.

In the *NeighborhoodWatch* code shown in Figure 6, 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 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 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 [4] and our design. The nesC code can be found in [10] and our code is shown in Figure 7. 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.

```
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;
      }
}
```

Figure 7. The *ParkingSpaceFinder* Application

5.2 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 consumptions of a target tracking application by using the Vigilnet [7] design and the Bundle design. The reason we choose Vigilnet is: Vigilnet uses TinyOS [11] to program each individual node in a distributed way. It 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 [3] and its empirical power consumption model. 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.

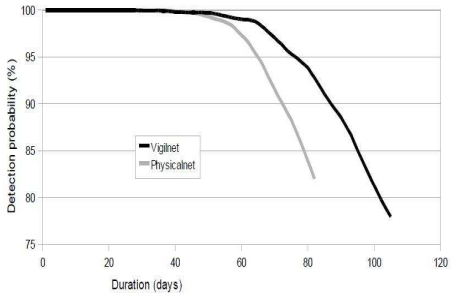

Figure 8. Average Detection Probability for Sentry Selection with Duty Cycle Scheduling

In our simulator, both the designs use 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 term 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 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 technique and in the second experiment we assume both of them employ an energy conservation technique called sentry selection [7], which is implemented in conjunction with duty cycle scheduling. In case of 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 8 presents the detection probability, as a function of the duration for which the network has been deployed. We observe that the network lifetime (defined as the number of days for which the detection probability of a target remains greater than 90%) 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, during the daily rotation, 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.

6 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. In the future, we hope to extend Bundle design to support in network aggregation and local processing within Bundles.

7 Acknowledgements

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

8 References

[1] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: Design and implementation. In *ICDCS*, pages 690–699, 2004.

[2] 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*, pages 61–72, 2005.

[3] 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*, pages 497–502, 2005.

[4] 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*, pages 1–11, 2003.

[5] 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*, pages 153–166, 2006.

[6] T. He, J. A. Stankovic, R. Stoleru, Y. Gu, and Y. Wu. Essentia: Architecting wireless sensor networks asymmetrically. In *INFOCOM*, pages 1184–1192, 2008.

[7] 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*, pages 1–12, 2006.

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

[9] 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*, pages 630–638, 2006.

[10] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. *SIGPLAN Not.*, 42(6):200–210, 2007.

[11] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*. SpringerLink, 2005.

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

[13] 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(1):122–173, 2005.

[14] L. Mottola and G. P. Picco. Logical neighborhoods: A programming abstraction for wireless sensor networks. In *DCOSS*, pages 150–168, 2006.

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

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

[17] 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.

[18] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, pages 29–42, 2004.

[19] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSYS*, pages 99–110, 2004.

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 is 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

Figure 4. 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.