Physicalnet: A Generic Framework for Managing and Programming Across Pervasive Computing Networks

Pascal A. Vicaire, Zhiheng Xie, Enamul Hoque and John A. Stankovic
Department of Computer Science, University of Virginia
{zx3n, eh6p, stankovic}@cs.virginia.edu

Abstract

This paper describes the design and implementation of a pervasive computing framework, named Physicalnet. Essentially, Physicalnet is a generic paradigm for managing and programming world-wide distributed heterogeneous sensor and actuator resources in a multi-user and multi-network environment. Using a four-tier light-weight service oriented architecture, Physicalnet enables global uniform access to heterogeneous resources and decouples applications from particular resources, locations and networks. Through a negotiator module, it allows a large number of applications to concurrently execute on the same resources and to span multiple physical networks and logical administrative domains. By providing a fine-grained use-based access rights control and conflict resolution mechanism, Physicalnet not only ensures owners having total control of sharing and protecting their resources, but also dramatically increases the number of applications that can concurrently execute on the devices. Furthermore, Physicalnet supports resource dynamic location-aware mobility, application run-time reconfigurability and on-the-fly access rights specification. To quantify the performance, we evaluate Physicalnet based on memory usage, the number of concurrent applications, and dynamic responsiveness. The results show Physicalnet has excellent performance, but low overheads.

1. Introduction

We are already seeing a widespread deployment of embedded devices with sensors and actuators, thereby creating pervasive computing environments. These environments consist of a wide spectrum of devices from very minimal capacity sensor nodes (motes, smart toasters, smart thermostats) to powerful machines (PDAs, PCs). In the future, we can expect that synergistically combining capabilities from previously independently deployed pervasive computing networks in a world-wide scale will create many opportunities. For example a security protection system placed independently in several homes can collaborate so that a theft occurring in one house would result in the alarms raised in another nearby house.

To achieve this potential requires new solutions that can handle a wide spectrum of device capabilities and heterogeneity, enable across network programming, support dynamic reconfiguration and mobility of devices, allow multi-user sharing and protection of their resources, and be easy to program. To date, many middleware and service oriented architecture approaches exist that solve various aspects of these problems. However, many of these solutions are heavyweight in memory and execution time and do not adequately address minimal capacity devices. Many other available solutions do address minimal pervasive system devices, but these do not address resource sharing and protection in a multi-user environment.

To address the complete set of requirements for future world-wide pervasive computing, we designed, implemented and evaluated Physicalnet. The main contributions of Physicalnet described in this paper are: 1) a fully implemented, generic and scalable framework for world-wide pervasive computing resource management and programming. Applications can easily involve resources from different owners and networks. 2) Using a four-tier lightweight SOA architecture, applications can concurrently run on the same resource and span multiple physical networks and logical administrative domains. 3) Interoperability and global accessibility of a wide range of heterogeneous resources are enabled. 4) User-based fine-grained access rights and conflicts resolution mechanisms are utilized, which not only improves resources sharing and protection, but also dramatically increases the number of concurrent applications that can use the devices. 5) Run-time resource reconfiguration and location-aware mobility are supported. 6) Both simulation and real platform performance evaluations show low memory requirements for constrained devices, excellent performance of the Physicalnet fine-grained access rights mechanism, and suitable responsiveness of the architecture design.

Note that for ease of programming we have also developed an associated programming abstraction for dynamic groups called a Bundle. While Bundles operate closely with the Physicalnet, they are not the subject of this paper. Thus, we only briefly introduce Bundles in this paper. For more details, please see [19].

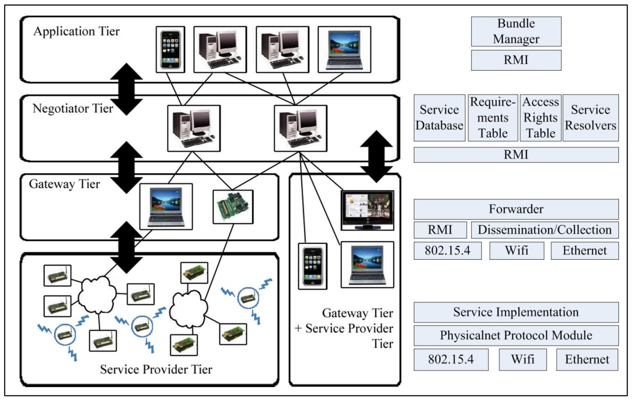


Figure 1. The High Level Network Architecture of Physicalnet

2. Overview of Physicalnet Architecture

To simplify the description of the Physicalnet framework, several terms need to be defined. 1) *Service*: an independent function provided by either hardware or software, e.g. the temperature sensor in a MicaZ mote, the display screen of a PC, or the Mediaplayer tool in a PDA. 2) *Service state*: a mode the service is currently in, e.g. the switch of a light actuator is on. 3) *Service event*: an output stream of a service which only occurs under some condition, e.g. the photo samples at a rate of 5 seconds. 4) *Service Provider*: a hardware or software entity which provides services, also called "resource" throughout this paper, e.g. a MicaZ mote or a PDA. 5) *Resource owner*: the person who owns the resource. 6) *Resource user*: the person who uses the resource.

The four-tier service oriented architecture of Physicalnet is shown in Figure 1. Each tier could be distributed and have multiple instances. The first tier contains service providers and localization anchor nodes (the nodes with surrounding flash symbols in Figure 1). The anchor nodes provide the localization service. A provider registers its services with one and only one negotiator (the third tier). It receives configuration messages (commands) from and periodically sends control messages (sample values or state reports) to its negotiator through the gateway (the second tier).

The second tier contains gateways that provide the connection and translation between service providers and negotiators. A gateway collects control messages from service providers, and then forwards them to the corresponding negotiators. Similarly, it also receives configuration messages from negotiators and forwards them to service providers. The gateway integrates different types of network interfaces in order to communicate with heterogeneous service providers. Thus service providers even without common network interfaces can communicate with each other through the gateway. Note that some devices like PDAs may contain both a service provider tier and a gateway tier.

The third tier contains negotiators. A negotiator is a registry for services, a database of service states and application requirements, and an authority center to resolve the requirement conflicts for multiple concurrent applications. Applications could discover and operate on services through the negotiator if they have enough rights. They also periodically calculate and send their requirements (e.g. the photo sensor service in provider 3002 should be turned on) to the negotiator. A negotiator allows multiple applications to access the same service concurrently and resolves possible conflicts by using a fine-grained access rights control and resolver mechanism. It sends configuration messages to and receives control messages from resources through gateways. All the states and recent event streams of registered services are stored in a database.

The fourth tier is the application tier, which contains applications that periodically generate and cancel requirements for remote services. Multiple applications can simultaneously access the same negotiator and a single application can involve multiple negotiators so as to access resources from different administrative domains.

3. Design Decision & Implementation

3.1. Resource Heterogeneity

In pervasive computing, a key concern is resource heterogeneity. Heterogeneity has three meanings: the different capabilities, the different programming platforms and the different network interfaces of different types of resources. To unify access, the typical way is to add a middleware layer into a resource. On the one hand, this middleware can provide uniform APIs for external applications; on the other hand, it can smartly manage and schedule internal services of the device. Additionally, this middleware can adaptively select the right networking protocols to communicate with other resources. Such middleware includes [5] [4] and [6]. However, the drawbacks of this approach are that the middleware is still too large and too complex for some tiny devices (e.g. Rene node, only 512 bytes RAM and 8K Flash); some of these common solutions even assume that a JVM is available on the resources; in addition, resources having no common network interfaces still cannot communicate.

Here, we argue that it is necessary to introduce a coordinator to bridge the heterogeneity. Physicalnet uses a gateway integrated with different network interfaces to translate the communication from one resource to another. It also adds a very lightweight Physicalnet protocol module into a resource. Thus, resources do not directly communicate with each other, but through the gateway. In addition, the Physicalnet protocol module is used to decode incoming configuration messages and encode outgoing control messages.

The format of the message is shown in Figure 2. For a configuration message, it includes a service provider's global ID, a timestamp value for synchronization purposes, and the data portion which includes the configuration information for each service's states and events in a service provider. For a control message, besides the global ID and timestamp fields, it has an 8 byte location field. The data portion includes services' current states or event stream values. When a service provider receives a configuration message, all its services read configuration commands from the data portion and make corresponding changes (e.g. turn on the LED, start sensing temperature). Periodically, a service provider sends out its control message which includes its current location information, services' states (e.g. LED state) and event stream values (e.g. current temperature), so that the negotiator can dynamically track its states and event streams. Therefore, Physicalnet is very suitable for remotely controlling actuators (e.g. change states) and retrieving sensory data from sensors (e.g. start event streams). But two-way synchronized request & reply communication style can also be supported by combining these two types of commands.

The benefits of this approach are: 1) that by keeping the service provider simple and generic, and placing complexity into high-level powerful PCs, more types of resources with minimal capabilities can be supported. 2) Under this scheme, resources without common network interfaces are able to communicate. 3) Integrating this scheme into the whole Physicalnet architecture, it gains even more benefits, like access rights control, across network programming ability, and mobility support which is difficult to obtain through typical middleware approaches.

3.2. Architecture Design

For the architecture design of a pervasive computing system, typically, there are three possible ways: one-tier architecture, which binds an application to a group of distributed resources forming an ad-hoc network; two-tier architecture, which separates the application tier from the service provider tier; three-tier architecture, which comprises the service provider tier, the gateway tier and the application tier. However, these three designs all have their intrinsic drawbacks for global pervasive systems. a) For one-tier architectures, the application is tied to resources. The drawbacks are: applications are hard to change, new applications cannot share the resources which are already occupied, and one application is usually limited to one location and one network. b) For two-tier architectures, by separating the application tier and the service provider tier, an application is easy to change. Additionally, if there is a

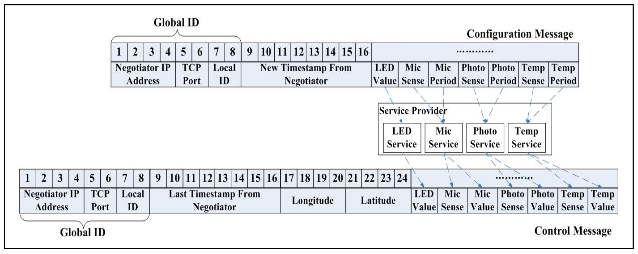


Figure 2. Physicalnet Protocol

general access rights control and conflict resolution module in the resources, it is possible to allow multiple applications share the same resources. However, the drawbacks are: the access rights control and conflict resolution module maybe too large and too complex for resource-constrained service providers, the application is still limited to one network, and as the resource numbers increase, the application tier may become a bottleneck. c) Three-tier architectures add a gateway tier between the application tier and the service provider tier. Now any complex functional modules can be integrated into the gateway tier. Therefore, applications are easy to change, multiple applications can share the same resources, and scalability is not a problem because one application can involve several networks through several gateways. However, mobility is a problem: when a service provider used by an application moves to another network whose gateway is not included by this application, how can this device be reconfigured and continuously service the original application?

By adding a negotiator tier, Physicalnet forms a four-tier lightweight SOA architecture as Figure 1 shows. This architecture not only has all benefits of the three-tier architecture, but also solves the mobility problem. In Physicalnet, each service provider is only registered to one and only one negotiator, and its global ID is *[negotiator IP]+[negotiator Port]+[service provider local ID]*. Thereby, wherever the service provider moves, the Physicalnet gateway can always infer the correct negotiator address from its global ID and help forward control messages to its registered negotiator.

More importantly, this four-tier architecture introduces the administrative domain concept into pervasive computing. An administrative domain is a logical independent system which comprises a negotiator, all registered resources and all the user accounts in the database. There is only one root administrator in one administrative domain, who has the total control power of the whole domain–he can create user accounts, specify their rights and modify service states. Other users can use legal user accounts to register their own resources to the negotiator, specify the access rights control and conflict resolvers for their resources, and use resources of others if they have enough rights. Although a negotiator is a center place to control all the resources, multiple negotiators could be used to relieve the bottleneck problem. Under this scheme, each negotiator maintains a reasonable number of resources, and these resources could spread across networks mixing with resources from other negotiators. Then the whole Physicalnet framework could be considered consisting of many independent logical administrative domains. An application can involve multiple administrative domains to use the resources it is interested in. Therefore, Physicalnet's centralized architecture is scalable. By using this four-tier architecture, Physicalnet essentially creates a paradigm to clearly and effectively organize the resources in a complex multi-user environment.

3.3. Programming Abstraction

Programming Abstraction is important for hiding low level details and easing programming. Physicalnet provides a very generic programming abstraction called *Bundle*. A bundle is a Java interface. It includes two core abstract methods: the definition of a group of services (*boolean rule (T t)*) and the specification of what these services should do (*void foreach (T t)*). There are two key features of a bundle. First, statically, the definition of a bundle can be arbitrarily complex, which could involve any number of operations and variables, and the members are not necessarily neighbors, but could be a collection of across-network services; dynamically, the membership in the bundle is updated periodically so as to respect the definition. For one application, it can involve several negotiators, and each negotiator maintains a list of services and their states and events data. Periodically, the application downloads the current differences of these states & event data comparing to the previous values from all involved negotiators. After downloading, the bundle in the application recomputes its membership based on its rule, and generates the new requirements for its new membership based on its operations. Then the application uploads its newly computed requirements to all the involved negotiators. These negotiators apply the requirements based on the application's access rights and send commands to the real services to execute.

3.4. Fine-Grained Access Rights Control And Conflicts Resolution Mechanism

Pervasive computing assumes an open multi-user environment, in which owners are willing to share their resources, but also expect them to be protected. Therefore, an access rights and conflict resolution mechanism is needed. User-based access rights is not a novel concept, however, in pervasive computing, the key points are what is the correct granularity for access rights and can the access rights mechanism support dynamic specification.

For the first question, intuitively, it seems the granularity at service level may be appropriate. However, it still has problems. For example, a group of scientists is deploying a large set of sensor nodes in a forest to measure the amount of CO_2 in the air. For energy conservation consideration, they are willing to let other research groups turn the sensing functionality on or off (event), but not willing to let them change the sampling rate (state) of the sensors. The only solution is to implement a number of services with different sensing rates, and allow others to use the particular one required.

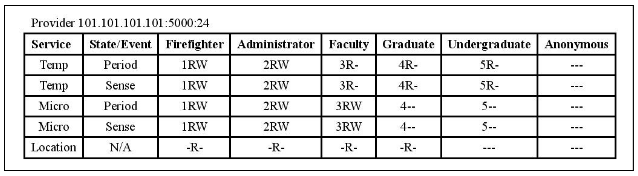
Physicalnet supports the access rights granularity at the level of individual states and events. Resource owners are allowed to specify the rights of WRITING, READING and PRIORITY for each state and each event of a service. Furthermore, when multiple users simultaneously specify contradictory requirements on the same resource, Physicalnet uses resolvers to resolve the conflicts. A resolver is a Java method in which its input is a list of requirements from different users and the output is one desired requirement. Physicalnet provides a small library of resolvers. Owners could select from it, or they can implement the resolver interface to define their own resolvers for each state & event. For example, Figure 3 shows the access rights of different users on different states and events of services. *1RW* means the user has WRITING and READING rights with PRIORITY 1 (lower value means higher priority) on the event/state. Assume Firefighter A specifies 4 second on Temp Period state, Firefighter B (using the same user account of Firefighter A) specifies 2 seconds on the same state, an administrator specifies 500 ms, and a faculty member specifies 200 ms. When a negotiator receives these requirements at the same time, it first applies access rights to them. Thus the faculty's requirement is denied. Then the negotiator orders the remaining requirements according to the priority before applying the resolver. If the resolver policy is to choose the smallest period of the highest priority requirements, then the desired requirement is 2 seconds; if the resolver policy is to choose the smallest period (the administrator's requirement), but the period should be at least 1 second, then the desired requirement is 1 second.

For the second question, Physicalnet does support dynamic access rights control specification, which means the access rights and the conflict resolvers can be dynamically changed according to the behavior of other services, the environmental phenomena, or application logic. For instance, a resource owner can specify that no application can open air conditioned vents in rooms where the window is opened. Physicalnet allows such dynamic specification of access rights by using the bundle programming abstraction. Figure 4 shows the code of the above application. The two *this.add()* in third and fourth lines involve two negotiators to retrieve the services from. *this.execute()* specifies the requirement update rate of the application. In the *for* loop, there are two bundles–one is *windows*, which includes all the open windows in one room; another is *new Ventbundle()*, which includes all the vents in the room which has windows open, and specifies no one can turn the vents on (*v.open.override(false);*). The types *WindowBundle* and *Ventbundle* are two user defined Java classes which implement the Bundle interface. The meaning of the whole application is that in two specific administrative domains, no one can turn on the vents in the rooms which has windows open.

3.5. Negotiator

In Physicalnet, the negotiator plays several important roles. First, it manages application concurrency. When a negotiator receives requirement lists from multiple applications simultaneously, it merges them into one desired requirement list by taking into consideration the access rights and resolvers, and sends configuration messages only if the current states are different from the desired states. The benefit is that it minimizes the number of configuration messages sent so that for one service, its servicing frequency is not correlated to the number of applications using it. Additionally, in one period, one resource only sends out one control message which can serve many applications. Therefore, it dramatically improves information reusability in a world-wide pervasive computing scheme. Furthermore, by keeping the resource side the same and using access rights control, applications do not interfere with each other even when they are sharing the same resources. Consequently, in Physicalnet it is easy to add new applications into previously deployed networks without disturbing the original systems.

Second, the negotiator helps applications react to undesirable conditions. For instance, if the application specifies that"all the lights should be on", perhaps only half of them are on because applications with higher priority want some lights off, the lights are currently not reachable (broken or dysfunctional radios), or the application has insufficient access rights. How can the application react to such events? The negotiator keeps track of three important variables: the *requirements* of the application, the *desired states* that result from applying access rights and conflict resolution mechanism to the requirements of all applications, and the *actual states* of the remote service provider. Then, the application can, at any time or periodically, check whether its requirements are satisfied (*requirements* = *actual states*), whether its requirements are not satisfied because of other applications or insufficient access rights (*requirements* $\neq$ *desired states*), or whether because the remote providers have not been reached (*requirements* = *desired states* && *desired states* $\neq$ *actual states*). Based on the above knowledge, the programmer is able to investigate the state of the network



Provider 101.101.101.101:5000:24

Service	State/Event	Firefighter	Administrator	Faculty	Graduate	Undergraduate	Anonymous
Temp	Period	1RW	2RW	3R-	4R-	5R-	---
Temp	Sense	1RW	2RW	3R-	4R-	5R-	---
Micro	Period	1RW	2RW	3RW	4--	5--	---
Micro	Sense	1RW	2RW	3RW	4--	5--	---
Location	N/A	-R-	-R-	-R-	-R-	---	---

Figure 3. An Example of Access Rights Specification

```
public class OnlyWhenWindowsClosed extends Application {
    public OnlyWhenWindowsClosed(){
        this.add(new Negotiator(HOST1, PORT1, USER1, PASSWORD1));
        this.add(new Negotiator(HOST2, PORT2, USER2, PASSWORD2));
        this.execute(1000/*milliseconds*/);

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

            //Creates the bundle of all the windows that are open
            // in that room.
            final WindowBundle windows=new WindowBundle(this){
                public boolean rule(MyWindow w){
                    return room.contains(w) &&
                    w.open.getActual()==true;
                }
            };

            //Creates the bundle of all the vents that are in the same
            //room as an open window.
            //These vents are closed even if other users require them
            //to be open
            new Ventbundle(this){
                public boolean rule(MyVent v){
                    return room.contains(v) &&
                    windows.size()!=0;
                }
                public void foreach(MyVent v){
                    v.open.override(false);
                }
            };

        }

    }
}
```

Figure 4. OnlyWhenWindowClosed

and modify application requirements appropriately.

Third, the negotiator plays a central role in reliably configuring service providers. Because it keeps track of the aforementioned three variables, it can retry to set the actual states of the remote services to their desired states until successful, which is a major difference from the standard RPC approach.

3.6. Synchronization Among Four Tiers

This section describes the detail synchronization that takes place among the four tiers. First is the synchronization between location anchors and service providers. A localization anchor (the nodes with flashing symbols in Figure 1) is used to localize resource constrained service providers. It is (manually, or automatically if possible) configured with its current latitude and longitude, and broadcasts this value at the owner specified rate. When a service provider receives a localization message, it assumes that it is in the same location as this anchor.

For the synchronization between service providers and negotiators, by default, a service provider sends one control message every p_{max} seconds to the gateway. However, when it is generating an event stream, the period is decreased so as to forward these messages as fast as possible. Nevertheless, the period should not be smaller than p_{min}. When the gateway receives a control message, it infers the negotiator address from the global ID and stores it in a buffer dedicated to the inferred negotiator. Periodically (configurable), the gateway forwards all the messages in the buffers to the appropriate negotiators using TCP/IP.

When the negotiator receives a batch of messages, for each one, it checks whether the service provider is registered. If it is, it updates the gateway address and the location of the service provider in the database. To extract the service data (e.g. the temperature sensor samples) from the message, the negotiator uses a provider specific decoder. By using Java reflection, the negotiator can infer the name of the decoder methods based on the name of the provider services. After that, the negotiator stores them in the database for a configurable maximum amount of time, waiting for applications to download. The timestamp in the message is for configuration consistency. If the timestamp in the received control message is not the latest, the negotiator resends the configuration message with the latest timestamp to the remote service providers.

In the other direction, when a negotiator needs to send configuration messages, it encodes the data portion (e.g. the LED on Service Provider 3001 should be on) of the configuration message by using a provider specific encoder similar to the aforementioned decoder. The message also includes the latest timestamp for synchronization purposes. Upon reception of the configuration messages, the gateway stores the configuration messages in a queue and forwards them one by one to the corresponding service providers. After receiving the configuration messages, the service provider stores the new timestamp, modifies the states of services according to the message, and initiates tasks as required by the modification of its states.

For the communication between applications and negotiators, by using the bundle programming abstraction, applications recompute and upload their requirements (a list of *[provider ID, service or event name, required value]*)) at a specified rate (the number in *execute()* method in Figure 4). At some point, a negotiator may get requirement lists from multiple applications. By looking up the access rights table and the resolvers for target states and events, the negotiator merges all these requirement lists into one desired requirement list, and then sends a batch of configuration messages to corresponding service providers. In the other direction, a negotiator always stores the latest states and event stream values of all resources in its database, and waits a configurable amount of time for applications to download.

3.7. Other Issues

3.7.1 Location-Aware Mobility

Localization anchors are used to localize the service providers. Usually, there is only one fixed anchor per room, and its radio range is adjusted to just cover the whole room. Under this scheme, one localization anchor can be mapped to one room so that an application can make use of this geographical information. For instance, the application could turn on all the temperature sensors in Room 230, 666 Fifth Avenue. Furthermore, Physicalnet allows applications to visualize the locations and states of service providers with which they are involved via Google Earth. All the information in Google Earth is dynamically updated so that we can see resources move from one room to room in Google Earth when they do so. By using this approach, whenever a service provider moves from one location to another location, or from one network to another network, Physicalnet is aware of this at run-time, and thus can take use of it (e.g. dynamic access rights specification).

3.7.2 Current Implementation State

Physicalnet fully implements all four tiers. For the service provider tier, it currently supports two types of resources, TinyOS nodes (e.g. MicaZ, TelosB) and Java nodes (e.g. PDA, Java applications on a PC). However, it could be extended to a wide range of other embedded or powerful resources. For network protocols and interfaces, the top three tiers are using the RMI method for communication. For the lowest two tiers, Physicalnet supports 802.15.4, Wifi and Ethernet network interfaces. For 802.15.4 network type, Physicalnet uses the Collection Tree Protocol (CTP) for collecting data from service providers to a gateway, and uses Dissemination Protocol or a Physicalnet unicast protocol for sending configuration messages from a gateway to service providers. Therefore, the networking protocols in Physicalnet support multi-hop. However, Physicalnet is not tied to a particular networking protocol. Any new protocols could replace the old ones.

3.8. Physicalnet Paradigm

The main contribution of Physicalnet is not any one of its parts, but the synergistic combination of them. The result is a very generic pervasive computing paradigm for organizing and programming world-wide resources. From one view point, the world-wide scale Physicalnet systems could be considered as consisting of distributed physical networks. In each network, heterogeneous resources may belong to different owners and come from different administrative domains. Multiple applications from different users may concurrently execute on them. From another view point, Physicalnet systems could also be considered as consisting of independent logical administrative domains. Any organization can set up its own domain, and there can be many administrative domains coexisting in the world. In one domain, there is a list of user accounts with different powers. Any resource owners can register their resources to the domain and specify the access rights and resolvers by using these user accounts. All the resources in one domain can be physically distributed in different networks.

For application developers, they can write an application involving several negotiators (with several user accounts) so as to retrieve distributed resources from multiple networks and multiple administrative domains. Because the application tier is decoupled from other tiers, a user can run an application on his laptop in California to remotely control the resources in Virginia and Pennsylvania. And all the resource states can be visualized in Google Earth. Furthermore, it is possible to add a negotiator registry and search engine tier above the negotiator tier so that all the negotiators can publish their information on the registry and all the applications can search for available negotiators through it.

The types of applications Physicalnet can support include streaming, event-based, user-centric and control-based (see section 4.3). Most importantly, by using Physicalnet, developers and users can write and execute new applications on already deployed systems. Furthermore, developers can write collaborative applications across previously independent systems (see our Bundle paper [19]).

4. Evaluation

Although it is difficult to evaluate an infrastructure, we use the following metrics for evaluation: memory usage to verify the Physicalnet protocol module is lightweight and can be added to resource constrained devices; the number of concurrent applications to demonstrate the effectiveness of fine-grained access rights control and conflict resolution mechanisms; and event responsiveness to evaluate the dynamic change awareness performance.

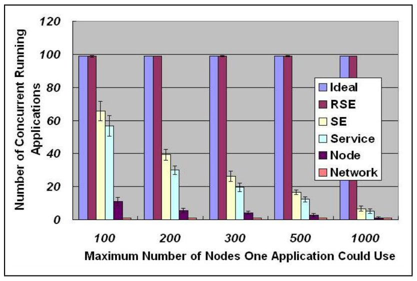
4.1. Memory Usage

The Physicalnet code is composed of 29,555 lines of Java code for the application APIs, the negotiator, the gateway, and the various tools. A MicaZ provider, equipped with a MTS310 sensor board, and implementing a temperature sensor service, a light sensor service, an accelerator service, a microphone service, a LED service, and a sounder service is programmed using 2807 lines of TinyOS code. The memory footprint of the compiled code on the MicaZ is 32182 bytes of ROM, and 2217 bytes of RAM. There are also 310 lines of TinyOS code of the Physicalnet base station, which is connected to the serial port of a gateway, thereby allowing the gateway to communicate with MicaZ motes. Note the TinyOS code here includes the implementation of all the six services. The Physicalnet protocol module (for encoding and decoding) only contains 56 lines. Comparing BASE[5] middleware's 132 KB and PCOM's [4] 120-160KB (both of them do not include the device's services code), Physicalnet can support more resource constrained embedded devices.

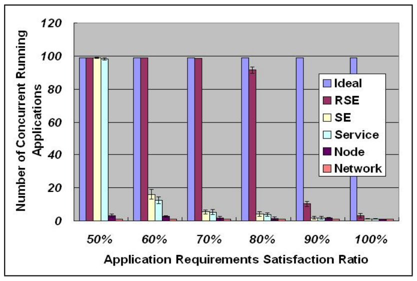
4.2. The Number of Concurrent Applications

With fine-grained access rights control and conflict resolution support, Physicalnet dramatically increases the number of applications concurrently running on the same resources. By using simulation we compared the application concurrency possible as a function of different design choices in the granularity level of access rights control. We consider the following levels of access control: the network level (e.g. EnviroTrack [3]), node level (e.g. TinyCubus [14]), service level (e.g. Melete [20], Agilla [9]), state & event level (SE), and the state & event level with conflict resolution support (RSE) all compared to the ideal situation. The term concurrency is defined as follow: assuming there are M applications attempting to run in the same network and each of them may have different requirements, we say L ($L \leq M$) applications can concurrently run only if each application's requirements are satisfied above some threshold.

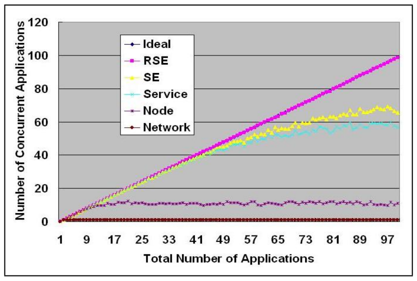
The definition of satisfaction for one application is based on a parameter *satisfaction ratio* SAT. For example, assuming $SAT = 0.6$, then for the granularity level of access rights control at the network level, applications cannot share the same network, so only one application can execute. For node level, applications can share the same network, but cannot share the same node. If an application tries to use 500 nodes, then as long as it acquires more than 300 nodes, we consider that this application is satisfied. For service level, applications can further share the same node. If one application has requirements on 10 services on one node, and it can get at least 6 of them, then we consider the application acquires this node. And only if the application acquires at least 300 nodes, then this application is considered to be satisfied. Similarly for the *SE* and *RSE* levels, they can further share the same service, but not the same state or event of a service. The difference between *SE* and *RSE* is that when two applications have conflicting requirements on the same state, *SE* can only satisfy one application, but *RSE* can satisfy both by using resolvers. For example, if one application requires the sensing period of a temperature service to be 3 seconds, but another application requires 1 second, the resolver adopts 1 second so that both of them are satisfied.



(a) Concurrency Degree on Used Nodes Number



(b) Concurrency Degree on Satisfaction Ratio



(c) 1 to 99 Applications Concurrency with Maximum 100 Nodes Used

Figure 5. Concurrency Degree for 99 Applications Based On Different Access Rights Granularity

The baseline configuration of the simulation is: 99 applications attempt to concurrently run on one network; the network consists of 1000 nodes; each node has a maximum of 5 services; each service has either one configurable state (the sensing period), or one configurable event (the switch of the service), or both; the state has 4 possible values–default (which means the application just wants to read the sample regardless of the sampling rate), low rate, medium rate and high rate; the event has 3 possible values– default, on and off; each application uses a maximum of 500 nodes of the network. The satisfaction ratio is 0.6. The required nodes for one application, the required services of each required node for one application, the services number of one node, and whether having state or event or both parameters for one service are all randomly generated. For the priority of applications, first-in-first-served policy is used. All the points of Figure 5 are the average results of 10 experiments. The error bars in the figure represent one standard deviation.

Figure 5(a) shows the number of concurrently running applications (total 99 applications) according to different values of the maximum number of nodes one application can use. As the number of maximum nodes one application can use increases from 100 to 1000, the conflicts also increase. The concurrently running application number of SE level granularity decreases from 65.8 to 6.7, service level from 56.8 to 5.1, and node level from 10.9 to 1.2. However, for RSE, the result is nearly constant 99. Figure 5(b) shows the the number of concurrently running applications according to satisfaction ratio. RSE has very good performance when the ratio is below 80%, while the number of concurrent applications for the other levels of granularity decreases dramatically after 50%. Figure 5(c) shows the actual number of concurrently running applications for 1 to 99 attempting to run when using a maximum of 100 nodes. We see that RSE is almost the same as the Ideal line, while for SE and Service level granularity, the number of concurrent applications begins to decrease after running 40 applications. For the node level, the number of concurrent applications falls after running 8 applications. Therefore, we conclude that our resolver supporting access rights control on state & event level granularity dramatically increases the number of concurrent applications in various scales of networks and under various requirements of applications.

4.3. Responsiveness

In this experiment, we study the responsiveness of Physicalnet applications to environmental stimuli (and the experiment also includes the mobility responsiveness evaluations). A responsiveness measurement is the time elapsed from the time at which an environmental stimulus is applied to the network to the time at which the predictable consequences of the environmental stimuli are observed. We use five applications and define four events for each. The five applications are:

- *PhotoAlarm*: compute the average light intensity for each room in the negotiators it connects to. If the value is above some threshold, all the sounders of that room are turned on.
- *RoomOccupancy*: turns on all the acoustic sensors of the negotiators it connects to, and infers that a room is occupied if two or more acoustic sensors are triggered in that room.
- *NeighborhoodWatch*: a collaborative surveillance application that alerts a set of neighbors if an intruder is detected in one of their houses. If the occupant (represented by a MicaZ tag node) leaves the house, all the accelerators and light sensors in that house are turned on. Any differences detected by these sensors indicate an intruder, and result in raising the alarm in another house.
- *Tracker*: a user has 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.
- *FireAlarm*: identical to PhotoAlarm except that it uses temperature sensors instead of photometric sensors.

The configuration is as follows: two PCs are used. The first PC runs only applications, while the second PC, located 1 mile away from the first PC, runs the negotiator, the

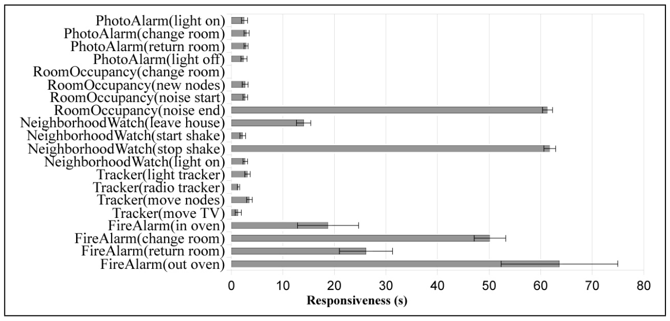


Figure 6. Responsiveness To Environmental Stimuli (CTP-UP)

gateway, and is connected to the service provider tier. We use 18 MicaZ nodes with MTS310 boards distributed over 6 rooms (3 per room) at the same site as the second PC, and 6 localization anchors (1 per room). Collection Tree Protocol (CTP) and Physicalnet unicast protocol are used in this experiment.

The results are shown in Figure 6. We observe that responsiveness is less than 4 seconds for 13 out of 20 stimuli. For FireAlarm, the responsiveness is slow because the sensor needs to adapt to changing temperatures. For StopShake, the responsiveness is about one minute because we specified in the NeighborhoodWatch application code that the sounders should ring if the accelerator sensors have been triggered during the last minute. For NoiseEnd, the responsiveness is of about one minute because we specified in the RoomOccupancy code that a room should be marked as busy if two or more acoustic sensors have been triggered in the last minute. For LeaveHouse, the responsiveness is 4.4s because all the accelerator sensors and photometric sensors must be turned on (by contrast, most other stimuli change the state of three or less sensors). Note that the responsiveness for the ChangeRoom stimulus of RoomOccupancy is 0 seconds because there is no requirement for changing the state of an acoustic sensor when it changes room during the execution of the RoomOccupancy application (the sensor must just keep generating acoustic samples). Note also that the responsiveness to the MoveTV and RadioTracker stimuli are particularly fast because these stimuli change only the state of Java service providers.

From the results, we can conclude that from responsiveness viewpoint, Physicalnet is not suitable for short deadline applications (which requires the responsiveness of less than 1 second), but is suitable for daily use applications, such as environmental surveillance, health care, and building automation.

5. Related Work

To enable uniform access to heterogeneous devices, various pervasive computing middleware is proposed. BASE [5] and PCOM [4] use a micro-broker-based middleware to solve the uniform access problem. They also decouple the application communication and underlying interoperability protocols by using an invocation broker module. Other pervasive computing systems including Aura [17] and Gaia [16] propose user-centric context-aware middleware to support applications dynamically adapting their tasks to surrounding available heterogeneous resources. However, the biggest problem with above solutions is that they do not have a clearly defined resource management mechanism which is a vital issue when a system targets an open environment with multiple users. Without access rights control, conflicts resolution mechanisms and administrative domain concepts, these systems are limited to only a small physical region and a small group of users. There also are WSN middlewares such as TinyDB [13], TinyLIME [7], Mires [18] and MiLAN [12]. However, they all suffer from one or more of the following shortcomings: they only offer relatively simple query abstractions; they do not have good support for dynamics, mobility and fault tolerance; they assume homogeneous platforms; most of them only allow a single application execution on a single node, or do not supply flexible resolution to deal with the resource conflicts.

Work is addressing the problem of making general software services accessible through the Internet. This body of work includes service oriented architectures and the technologies that enable it: RPC, RMI, CORBA, Jini [2], and Web Service (which include standards such as XMLRPC, WSDL, UDDI). Most of these architectures are too resource hungry to be used with wireless sensor and actuator networks. Tiny Web Service [15] implements web services directly in the sensor nodes in an efficient way. However, the drawbacks of this system are: the communication overhead and memory usage are still too heavyweight; as the number of concurrent applications increases, the node maybe too busy to serve all the applications; when multiple applications have conflict requirements, there is no way to resolve it; furthermore, when device failures become common in a large scale pervasive computing, programmers have to handle the exceptions one by one.

Other work focuses specifically on making resource constrained sensors available through the Internet: Agimone [11], IrisNet [10], ArchRock Primer Pack [1] and SOCRADES [8]. IrisNet and ArchRock Primer Pack do not address actuation, but only sensing. Agimone deals with both actuation and sensing, but programmers must use an assembly like language and are responsible for propagating code using one of two operators (move and clone), which have limited applicability. SOCRADES aims to integrate DPWS enabled devices into existing business systems. The idea is similar as Physicalnet's SOA architecture, but it still has the same problems as the above systems–it does not handle the problems of user access rights, node mobility, and concurrent applications.

For resource sharing, TinyCubus [14] allows the sharing of sensor network resources by partitioning the network. However, users can only access these partitions exclusively. Melete [20] enables the execution of concurrent applications on a single sensor node. However, Melete applications are very restricted in size (of the order of 50 bytes) because they run on a virtual machine located on the sensor nodes. Moreover Melete ignores the problem of conflicting application requirements. By contrast, Physicalnet not only allows users and applications to share resources at state & event level, but also has user-specific access rights control and conflict resolution mechanisms.

6. Conclusion

This paper describes the design and implementation of a very generic framework, Physicalnet, for managing and programming across pervasive computing networks. Physicalnet is based on an SOA four-tier network architecture, supports a wide range of heterogeneous devices, permits multiple applications concurrent execution in the same network, allows the programming of applications involving multiple physical networks and logical administrative domains, and provides a fine-grained access rights control and conflict resolution mechanism. Additionally, all the programming is based on a very generic programming abstraction–Bundle. Our experimental results show Physicalnet supports a large number of applications concurrent execution. The memory usage is low, and the dynamic responsiveness is adequate for practical use.

Acknowledgements

This work was supported, in part, by NSF CNS-0626616, CNS-0626632, and CSR-0720640.

References

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

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

[3] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: towards an environmental computing paradigm for distributed sensor networks. In *ICDCS'04*, pages 582–589, 2004.

[4] C. Becker, M. Handte, G. Schiele, and K. Rothermel. Pcom - a component system for pervasive computing. In *PerCom 2004*, pages 67–76, March 2004.

[5] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. Base - a micro-broker-based middleware for pervasive computing. In *PerCom 2003*, pages 443–451, March 2003.

[6] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis. The runes middleware for networked embedded systems and its application in a disaster management scenario. In *PERCOM '07*, pages 69–78, 2007.

[7] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco. Tinylime: Bridging mobile and sensor networks through middleware. In *PERCOM '05*, pages 61–72, 2005.

[8] L. M. S. de Souza, P. Spiess, D. Guinard, M. Kőhler, S. Karnouskos, and D. Savio. Socrades: A web service based shop floor integration infrastructure. In *IOT*, volume 4952 of *Lecture Notes in Computer Science*, pages 50–67. Springer, 2008.

[9] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *ICDCS'05*, pages 653–662, June 2005.

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

[11] Hackmann, Fok, Roman, and Lu. Agimone: Middleware support for seamless integration of sensor and ip networks. In *DCOSS 2006*, volume 4026, pages 101–118, 2006.

[12] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18:2004, 2004.

[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*, 30:122–173, 2005.

[14] P. Marron, A. Lachenmann, D. Minder, J. Hahner, R. Sauter, and K. Rothermel. Tinycubus: a flexible and adaptive framework sensor networks. In *EWSN'05*, pages 278–289, 2005.

[15] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *SenSys '08*, pages 253–266. ACM, 2008.

[16] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: a middleware platform for active spaces. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):65–67, 2002.

[17] J. P. Sousa and D. Garlan. Aura: An architectural framework for user mobility in ubiquitous computing environments. In *In Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, pages 29–43, 2002.

[18] E. Souto, G. Guimaraes, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner. Mires: a publish/subscribe middleware for sensor networks. *Personal Ubiquitous Comput.*, 10(1):37–44, 2005.

[19] P. A. Vicaire, E. Hoque, Z. Xie, and J. A. Stankovic. Bundle: A group based programming abstraction for cyber physical systems. *Under submission*, 2009.

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