Spatial Programming using Smart Messages: Design and Implementation [*]

Cristian Borcea, Chalermek Intanagonwiwat [†], Porlin Kang, Ulrich Kremer, and Liviu Iftode
Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
{borcea, intanago, kangp, uli, iftode}@cs.rutgers.edu

Abstract

Spatial Programming (SP) is a space-aware programming model for outdoor distributed embedded systems. Central to SP are the concepts of space and spatial reference, which provide applications with a virtual resource naming in networks of embedded systems. A network resource is referenced using its expected physical location and properties. Together with other SP features, such as reference consistency and access timeout, they help programmers cope with highly dynamic network configurations in a network-transparent fashion.

This paper presents the SP design and its implementation using Smart Messages, a lightweight software architecture similar to mobile agents, that we developed for networks of embedded systems. We also describe the implementation and evaluation of a simple SP application over a testbed consisting of HP iPAQs running Linux and equipped with 802.11 cards for wireless communication. The experimental results indicate that SP is a viable programming model for outdoor distributed computing.

1. Introduction

With computers moving outdoors, embedded in cars, cell phones, or buildings, outdoor computing environments populated with ubiquitous networks of embedded systems (NES) have emerged. Most of the recent research in NES area has focused on hardware, operating systems, or network protocols [14, 22, 12, 13]. We believe that a crucial challenge which has been only marginally tackled is how to program distributed applications in outdoor computing environments. The main problem that makes NES programmability difficult is the highly dynamic character of these networks. Since NES are composed of a massive number of heterogeneous systems, which may be mobile and volatile, it impossible to know the exact number or location of various network resources over time.

Given these characteristics, it is clear that traditional distributed computing models are irrelevant for programming outdoor distributed systems. An application written under these models assume functionally homogeneous nodes, stable configurations, robust networking, and ignores the physical location of nodes. A successful programming model for NES must tolerate the network volatility and heterogeneity. Additionally, it has to consider the physical location of nodes because location may dictate the role of a node in computation. To design such a programming model, two major questions have to be answered.

- How to describe distributed applications over networks of embedded systems in a simple and efficient way?
- How to deploy these applications in existing networks?

The solution to the first question requires an appropriate programming model for NES. Typically, we can choose between two programming styles: declarative or imperative. Declarative programming is goal-oriented in the sense that programmers simply specify what they want instead of how to algorithmically obtain the results. Multiple solutions for programming sensor networks illustrate this programming style in NES [5, 19]. The "database" model of programmability suits sensor networks well because these networks act primarily as large "distribute databases" for the environments where they are deployed.

Despite its simplicity, declarative programming is not a panacea for every type of task or NES. Imperative programming is more appropriate for complex tasks that go beyond data collection (especially tasks whereby algorithmic details matter). Also, we believe that networks composed of more powerful nodes (e.g., systems in cars, cell phones, intelligent cameras, mobile robots [2, 15]) cannot be programmed in a simple and effective way without having fine-grained control over individual network resources.

In this paper, we describe the design of Spatial Programming, a space-aware programming model that allows programmers to use network resources in the same way they use variables in imperative programming. Similar to the

[*] This work is supported in part by the NSF under the ITR Grant Number ANI-0121416

[†] Current Address: Department of Computer Engineering, Chulalongkorn University, Bangkok 10330, Thailand

view of the network as a database, we view the network as a single virtual address space. In SP, network resources (content or services provided by nodes) are accessed using *spatial references*. A spatial reference uses the expected location of a node and the name of a property provided by that node to define a virtual name for a network resource.

Using spatial references, programmers can write distributed applications for NES in a network-transparent fashion. Therefore, they can focus on the algorithmic details of the program rather than on the networking aspects. Similar to the mappings from virtual to physical memory in a conventional computer system, a runtime system maintains mappings between spatial references and nodes in the physical space. For every access to a spatial reference, the runtime system takes care of name resolution and binding, communication, and routing.

The second major question that a programming model for NES has to answer, as we already mentioned, is how to deploy new applications in an existing network. Given the scale of NES and the fact that most of the nodes work unattended, it is practically impossible to re-program each node individually for every new application. Therefore, an SP runtime system based on code migration is preferable for two reasons. First, a user can inject new SP programs into any NES node, and consequently the programs migrate to the target nodes without human intervention. Second, SP programs become platform-independent, given that code migration is commonly implemented on top of a virtual machine.

We present an SP implementation using Smart Messages (SM) [7]. SM is a lightweight software architecture, similar to mobile agents, that we developed for programmable NES. We also show the implementation and evaluation of an SP application over a testbed consisting of HP iPAQs running Linux and equipped with 802.11 cards for wireless communication. The experimental results indicate that SP is a viable programming model for NES and that SM can be successfully used to implement it.

The rest of this paper is organized as follows. Section 2 describes the Spatial Programming model. Section 3 presents the implementation of SP using Smart Messages. Section 4 shows the evaluation of our SP application. Section 5 discusses related work. The paper concludes in Section 6.

2. Spatial Programming

To motivate the need for a new programming model for outdoor distributed computing, let us consider a collaborative object tracking application. For this application, two types of nodes are assumed available across a given geographical region: motion sensors and intelligent cameras. Each node is capable of determining its location (i.e., using GPS or other localization methods [22]). The motion sensors remain static after deployment, but the cameras can be mobile (e.g., carried by mobile robots [15]). The application checks the status of motion sensors, and each time it encounters a sensor that detected motion, it instructs a certain number of cameras located in the proximity of that sensor to perform collaborative object tracking in order to monitor the actions of the object that triggered the sensor.

This application emphasizes the main question that any programming model for outdoor distributed computing has to answer: how to program an *unknown number of volatile embedded systems* (i.e., mobile or even disposable) to execute a user-defined application in a certain geographical area? Besides the flexibility needed to cope with a distributed system whose state evolves continuously over time, such a model must be simple and intuitive. The complex networking aspects should be hidden from the programmers to allow them to focus on the algorithmic details of applications.

Spatial Programming (SP) is a space-aware programming model designed to satisfy these requirements. The main idea of SP is to offer network-transparent, fine-grained access to data and services distributed on systems embedded in the physical space. In SP, a network of physically distributed systems is viewed as a single virtual address space, and its individual resources can be accessed by applications like normal variables. The SP model allows for a large spectrum of outdoor distributed applications, ranging from computing the average/maximum temperature over a given geographical region to collaborative applications such as distributed object tracking or coordinating military forces on the battlefield. Typical applications for SP are those which execute a distributed algorithm over a set of nodes selected based on their location and properties.

The high level view of the network as a single virtual address space is similar to the one presented by shared virtual memory systems (i.e., shared virtual memory shields the programmers from message passing communication, while offering a shared virtual address space for distributed applications). A major difference, however, is that shared virtual memory is performed over a stable and robust network, with an acceptable upper bound for memory access time, while SP must tolerate dynamic network configurations, with unknown time bounds for accessing systems embedded in the physical space. Fig. 1 illustrates this analogy and the simple abstractions defined by SP to support outdoor distributed programming: *space regions* and *spatial references*.

2.1. Space Regions

A space region is a virtual representation of a given physical space (defined as a circular region that circumscribes

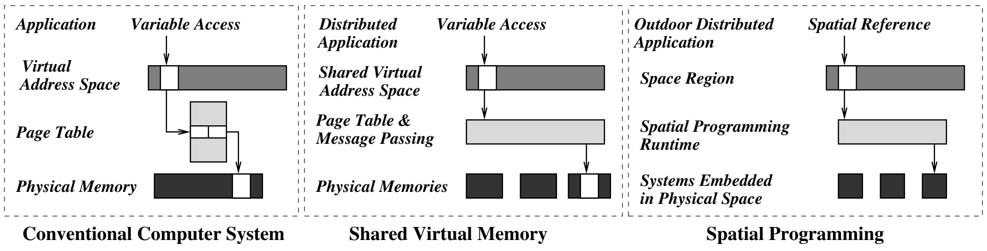

Figure 1: Analogy Between Spatial Programming and Two Traditional Programming Models

that physical space). Its role is similar to that of a virtual address space in a conventional computer system.

Most envisioned distributed applications for NES will exhibit a space-aware behavior. In order to achieve their prescribed objectives, they will need to run within certain geographical regions. For instance, the application described in this section may want to activate intelligent cameras within a physical range of the trigger node (the sensor that detected motion) since otherwise no causal relation can be established. Therefore, SP considers space as a first order programming concept and exposes space regions to applications through spatial references.

2.2. Spatial References

A spatial reference is defined as a {*space:tag*} pair which is mapped to a system embedded in the physical space. The *space* is a space region that represents the geographical scope of this system. The *tag* is the name of a property or service provided by the same system. Tags are not globally unique because they name properties or services that can be provided by multiple systems. Spatial references, like variables, are defined within applications; hence, a spatial reference has meaning only within the application that defined it.

Spatial references provide applications with a virtual resource naming in the network. Applications access network resources using spatial references in the same way they access physical memory through variables in conventional systems (or in shared virtual memory systems). Given that programmers have only limited knowledge about such dynamic networks (i.e., a programmer does not know how many resources are in a given space, what types they are, or even if they exist at all), spatial references offer a convenient method to refer to network resources using their *expected* locations and properties.

Fig. 2 presents examples of spatial references. To differentiate among systems with the same space-tag pair referenced in the same application, programmers can use indexes to refer to distinct systems. Thus, a spatial reference becomes a triplet {*space:tag[index]*}. SP guarantees that spatial references with distinct indexes (but the same space-tag pair) map to different systems. The figure shows how a programmer can use three distinct indexes to refer to distinct cameras on *Hill1*.

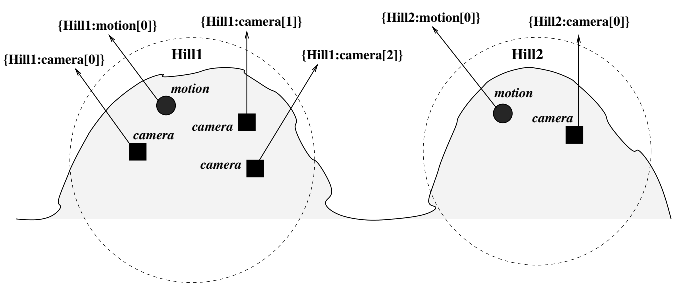

Figure 2: Example of Spatial References for Object Tracking in a Network Consisting of Motion Sensors and Intelligent Cameras Deployed over Two Hills

An SP application can name and access multiple network resources provided by a node using just one spatial reference. The construct {*space:tag[index]*}.*resource* refers to a certain *resource* located on the system referenced by {*space:tag[index]*}. In Fig. 2, {*Hill1:camera[0]*}.*active* may denote the status of the camera, while {*Hill1:camera[0]*}.*location* may represent the location of this system in space.

Spatial references relieve programmers from the burden of having to cope with all the networking details of reaching the nodes of interest and accessing data or services on those nodes. The SP runtime system takes care of name resolution, communication, and access to resources. This runtime system also takes care of reference consistency.

2.3. Reference Consistency

Conventional computer systems maintain reference consistency for variables. The operating system uses per-application page tables to guarantee that each time an allocated variable is used, it accesses the same physical memory location. Similarly, SP guarantees that each time an application uses a certain spatial reference, it accesses the same system as long as this system remains in its original space region. This property provides the ability to perform arbitrary distributed computations over a subset of nodes selected based on their location and properties.

The SP runtime system maintains mappings between spatial references and the nodes they refer to. These mappings are maintained in a *per-application mapping table* and are persistent during the SP program execution. At the time of the first access, a spatial reference is mapped to a node located in the desired space region and providing the required property. Each mapping table entry contains the location of the referenced node and a unique per-application network address for this node. The location is used for faster subsequent accesses to this node. The network address is assigned by the application (i.e., it has no global meaning) and is used to confirm the identity of the node for subsequent accesses (a referenced node may move from its recorded location, and another node may take its place). This address can also be used to locate, in the same space, referenced nodes that moved from their recorded locations.

In some situations, reference consistency is not necessary. For instance, an application that needs to contact periodically a number of temperature sensors located in a certain region and compute the average temperature may accept any sensor that provides the desired space-tag pair. In such a case, if a referenced node cannot be found in its space region, the runtime system should transparently remap the spatial reference to a similar node rather than returning an exception for a failed access. To implement this feature, SP allows an application to specify a *remap* flag for spatial references.

2.4. Spatial Reference Access Timeout

Unlike traditional computer systems where the access time to resources is finite and an upper bound for this time can be computed, in a volatile and dynamic NES, it is difficult to estimate how long it takes to access a network resource. This problem happens both for new references (no more available systems with the required space-tag pair) and for mapped references (they may become invalid because the referenced node can move from its space or simply cease to exist).

SP requires application programmers to reason about the possibility of not reaching a node by imposing a *timeout* on each spatial reference (i.e., the format of a spatial reference becomes {*space:tag[index], timeout*}). This timeout allows a programmer to limit the access time to a network resource which, given the volatility of the network, may take forever. Essentially, SP defines a “best effort” semantics that allows an application to make progress and get a semantically acceptable result even in adverse network conditions. If a node cannot be reached in the specified time interval, the SP runtime throws a timeout exception; once the application catches this exception, it can decide about further actions.

Commonly, the programmer sets each timeout based on a constraint imposed by the user on the total execution time (e.g., the total time is divided equally among all accesses, or each new access can have the entire remaining time).

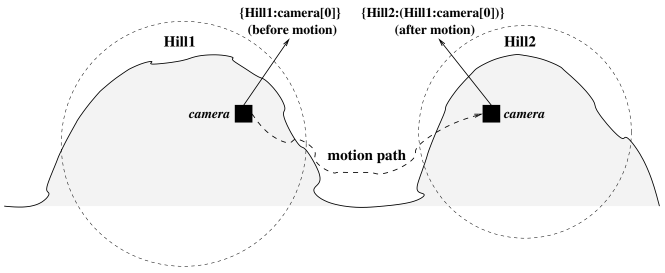

Figure 3: Space Casting: The Same System is Referenced in Different Spaces

2.5. Space Casting

The SP runtime system locates the same node each time an application uses the same spatial reference, provided that the node is still in its space region. If a node moves out of its space region, it becomes semantically unacceptable. Thus, the application receives a timeout exception (the system could not find the node during the timeout interval). However, if the programmer still wants to access this node and has knowledge about the node’s mobility patterns, the space region for the spatial reference mapped to this node can be modified using *space casting*. The construct {*space2:(space1:tag[index])*} changes the geographical scope of the spatial reference from *space1* to *space2*. Fig. 3 shows how space casting is used to reach a camera carried by a mobile robot which has moved from *Hill1* to *Hill2*. If the new space for a node is unknown, a programmer can use the *Anywhere* space constant to cast a spatial reference to any space. Note that in such a case the *timeout* ensures that the attempted access will not take forever.

2.6. Defining New Space Regions

Besides statically defined space regions, SP also supports dynamically defined space regions. *Composed* space regions can be defined using the union or intersection operators (i.e., these space regions are also defined as circles that circumscribe the actual physical space). If we consider the hills from our examples throughout this section, a spa-

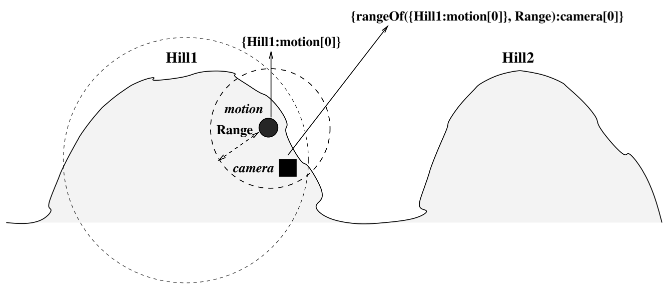

Figure 4: Dynamic Definition of a Relative Space Region

tial reference {*(Hill1 + Hill2):camera[0]*} returns a camera node located on either *Hill1* or *Hill2*.

Defining *relative* space regions based on the position of a referenced node offers two benefits for applications: access to systems located in dynamically defined space regions, and possibility to “remember” a space region where a certain event took place, even after the node that produced (or detected) this event is no longer there.

The *rangeOf* operator defines a space region in the proximity of a node referenced by a spatial reference. Fig. 4 shows how such a relative space is dynamically defined and used to refer to a camera node located in the proximity of a motion sensor.

Similar to *rangeOf*, SP defines the *northOf, southOf, eastOf,* and *westOf* operators. They create space regions relative to the position of a referenced node and the respective cardinal direction (the center of the circular region is located toward that cardinal direction at a given distance from the position of the referenced node).

2.7. Creating/Removing Network Resources

In addition to accessing resources that already exist at nodes, SP programs can also dynamically create/remove their own resources. For instance, an application may need to create new resources in order to store data in the network (i.e., similar to creating files in a file system). The primitives that offer this functionality are:

create({*space:tag[index], timeout*}*.resource)*
remove({*space:tag[index], timeout*}*.resource)*

Currently, SP provides just a limited resource sharing policy: the resources provided by nodes are shared, and the resources created by applications are private.

3. Implementation

SP requires a set of programming constructs that have to be exposed to programmers and a runtime system to support the model. The constructs can be added as extensions to any programming language or implemented as library calls. In this section, we describe the SP implementation using Smart Messages (SMs) [7], a software architecture similar to mobile agents, that we developed for NES. Under this implementation, SP applications are Java programs. The SP programming constructs can be invoked as Java methods, which are supported by our SM-based runtime system. Before describing the details of this runtime system, we present a short SM overview.

3.1. Smart Messages Overview

Smart Messages (SMs) are migratory execution units consisting of code and data sections, termed ”bricks”, and a lightweight execution state. Instead of passing data end-to-end between nodes, an SM migrates to nodes of interest named by content and executes the computation there. An SM carries routing code and routes itself at each node in the path toward a node of interest.

The nodes cooperate to support the SM execution by providing a virtual machine (VM) for execution over heterogeneous platforms, a shared memory addressable by names (tag space) for inter-SM communication and synchronization, and a code cache for storing frequently executed code. After arrival at a node, an SM generates a task which is scheduled for non-preemptive execution. The execution starts with the next instruction following a migration invocation. During execution, an SM can interact with the host or other SMs using tags. Corresponding to their functionality, there are two types of tags: application tags for “persistent” memory across SM executions (i.e., they can store application-specific data for a limited period of time), and I/O tags for interaction with the host’s operating system and I/O. The collection of all tags available at a node forms the tag space. Each tag has a name, similar to a file name in a file system, which is used for content-based naming of nodes.

SMs use a high level migration function to migrate to nodes of interest named by tags [6]. This migration is commonly provided as a library function that implements the routing (the programmers, however, are free to implement their own high level migration functions). The routing is executed at each node toward a node of interest (i.e., SMs are self-routing applications). The implementation of routing uses information stored by SMs in the tag space and a system-provided primitive for one hop migration. This primitive captures the current execution control state and migrates it to the next hop along with the code and data bricks (i.e., the only data transferred is the one incorporated explicitly by the programmer in the data bricks). A very important feature of the self-routing mechanism is the ability of SMs to use multiple routing algorithms during their lifetime and change these algorithms dynamically.

We have implemented an SM architecture by modifying

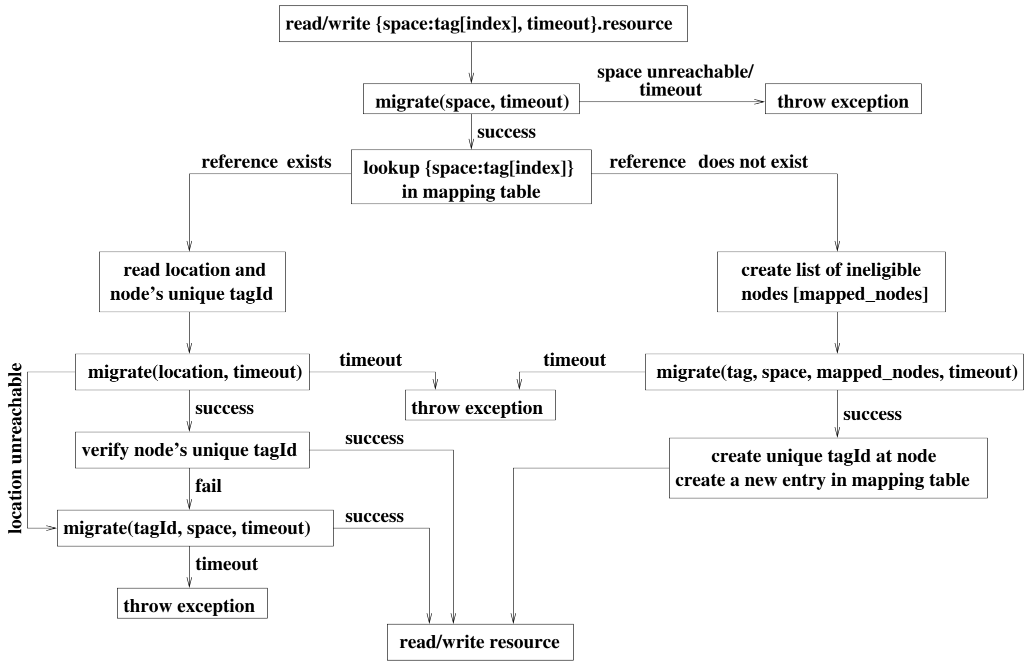
read/write {space:tag[index], timeout}.resource
migrate(space, timeout)
space unreachable/ timeout
throw exception
success
reference exists
lookup {space:tag[index]} in mapping table
reference does not exist
read location and node's unique tagId
create list of ineligible nodes [mapped_nodes]
migrate(location, timeout)
timeout
throw exception
timeout
migrate(tag, space, mapped_nodes, timeout)
location unreachable
success
verify node's unique tagId
success
success
create unique tagId at node create a new entry in mapping table
fail
migrate(tagId, space, timeout)
success
timeout
throw exception
read/write resource

Figure 5: Implementation of Spatial References with Smart Messages

Sun's K Virtual Machine (KVM). The tag space and SM operations are available to applications through a Java API. Our prototype implementation was tested on HP iPAQs running Linux and using 802.11 cards for wireless communication.

3.2. SM-based Runtime System for SP

An SM-based runtime system is suitable for SP not only because SMs provide the ability to re-program the network on-the-fly, but also because the tag space offers a simple and uniform interface for accessing data or services at nodes. Additionally, SP benefits from the SM self-routing mechanism; in reaching a node, the runtime system may use different routing algorithms and change the routing dynamically.

The main idea in our implementation is to translate high level SP programs into SMs. However, SP programs (written in Java) are not aware of the underlying SMs. To use the SM-based runtime system, they have to follow three simple rules: (1) extend an *SMWrapper* class which provides methods for the SP programming constructs, (2) initialize the SMWrapper by passing the class names for all classes that do not belong to our SM distribution (i.e., in order to be incorporated in the SM as code bricks), (3) use only class member variables (in this way, the SMWrapper knows what data needs to be transferred as data bricks). Under these rules, SP applications are just normal Java programs that access transparently network resources using spatial references.

At initialization, the SMWrapper creates the *mapping table* which maintains the mappings between spatial references and nodes. Also, the SMWrapper includes the code and data bricks for two routing algorithms: geographical routing (to reach the space of interest), and space bound content-based routing (to reach a node of interest within a given space). After the initialization is done, the SMWrapper creates and injects a new SM in the network. This SM includes the code and data for the SP program.

Essentially, the SMWrapper performs the SP-to-SM translation by transforming each access to a network resource (read/write) into an SM migration. Fig. 5 illustrates the main steps necessary to read/write a resource located on a referenced node. For both mapped and unmapped spatial references, the SM migrates to the desired space using geographical routing. We have implemented a greedy geographical routing similar to GPSR [17] (at each node, the algorithm chooses the neighbor closest to the center of the circle that represents the required space region).

When the space is reached, the SM checks if the spatial reference exists by performing a lookup in the mapping table. In the left part of the figure, we show how to reach a node referenced by an existent spatial reference (i.e., reference consistency). The right part shows how a new node of interest is found and mapped to a spatial reference.

```
public class IntruderDetection extends SMWrapper{

   public Space userSpace, monitoredSpace;
   public int i, j, count, numSensors, numCameras, timeout, threshold;
   public SpatialReference srLight, srUser, []srCamera;

   public static void main(String []args){
      IntruderDetection intruderDetection = new IntruderDetection();
      // read and store application’s parameters
      String []userClasses = {"IntruderDetection"} ;
      intruderDetection.initSMWrapper(userClasses, intruderDetection);
      intruderDetection.run();
   }

   public void run(){
      try{
         for (i=0; i<numSensors; i++){
            srLight = getSpatialReference(monitoredSpace, "Light", i, timeout);
            if (((Integer)srLight.read("Intensity")).intValue() > threshold){
               srCamera = new SpatialReference[numCameras];
               for (j=0; j<numCameras; j++){
                  srCamera[j] = getSpatialReference(monitoredSpace, "Camera", j, timeout);
                  srCamera[j].write("Active", "ON");
               }
               for(j=0,count=0; j<numCameras; j++){
                  if (((Boolean)srCamera[j].read("FaceRecognition")).booleanValue())
                     count++;
                  srCamera[j].write("Active", "OFF");
               }
               if (count > numCameras/2){
                  srUser = getSpatialReference(userSpace, "User", 0, timeout);
                  srUser.write("Message", "intruder detected!!");
                  return;
               }
            }
         }
      }catch(TimeoutException e){}
   }
}
```

Figure 6: Java Code for Intrusion Detection Application

If the reference does not exist, the SM has to discover a node of interest in the given space. Therefore, it changes dynamically its routing to a content-based on-demand routing (similar to AODV [21]) which is used to discover a node of interest. Due to its limited geographical scope, flooding does not represent a major problem for scalability. Once a matching node is found, the SM assigns a unique network address to this node by creating a unique *tagID* in this node’s tag space. Subsequently, the *tagID* and the location of the node are stored in the associated mapping table entry. In the process of mapping a new spatial reference, the mapped nodes having the same space-tag pair must be avoided (i.e., the application asked for a new node). To solve this problem, we retrieve the list of unique *tagIDs* corresponding to the mapped nodes and pass it to the routing algorithm. It is the responsibility of the routing to find an unmapped node.

To ensure reference consistency, subsequent accesses to an existent spatial reference must reach the same node. Therefore, the SM retrieves the location of the mapped node from the mapping table and migrates directly to this location. According to spatial references’ semantics, if the node is not present at that location anymore, the SM will try to reach it in the same space region using its unique *tagID*.

When the node of interest is reached, the SP program resumes its execution (i.e., it starts with the read/write operation which triggered the entire migration process). The tag space primitives are used to give the application access to local resources. If a node of interest is not found during the time interval specified by the application, or the space is unreachable, an exception is thrown to let the application decide further actions.

4. Evaluation

This section presents the implementation and evaluation of an SP application executed over our SM-based runtime system. We have evaluated this application on a testbed con-

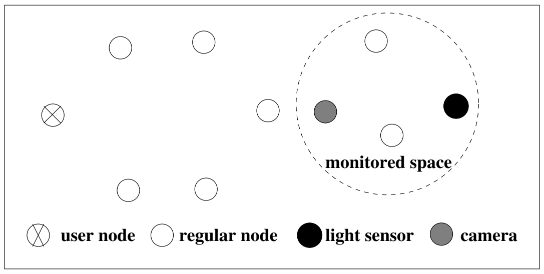

Figure 7: The Network Topology of Our Operational Testbed of Ten iPAQs

sisting of ten HP iPAQs (206-MHz Intel StrongARM Processor, 32-MB Flash ROM, 64-MB SDRAM). For wireless communication, we use Orinoco 802.11b PC cards in ad hoc mode. Each node supports the Smart Messages (SM) architecture. Our goal in conducting this evaluation study was twofold: (1) to verify the viability of the SP model in terms of ease of programming, and (2) to analyze the performance of our SM-based runtime system.

The application is similar to the object tracking application described in Section 2. Essentially, the application (injected by a user from a handheld device) performs intrusion detection over a monitored space region. It verifies the status of the motion sensors, and if one of them have detected motion, the application turns on a certain number of cameras to perform face recognition. After all these cameras have been turned on, the application returns to each of them to verify the result of the face recognition program. If at least half of the cameras have recognized a face, the application informs the user that an intruder has been detected.

For this application, some of our nodes are identified by a *Camera* tag (i.e., they have an attached video camera), while others are identified by a *Light* tag (i.e., instead of motion sensors, we use light sensors incorporated in iPAQs; we consider that motion was detected when the light intensity is above a certain threshold). The camera nodes provide also tags to activate the camera and get the result of the face recognition program.

The Java code for this application, presented in Fig. 6, demonstrates the main benefit of SP: flexibility to program complex distributed applications in outdoor computing environments in a simple, network-transparent fashion. The *run* method shows how spatial references shield the programmers from the networking details. It also demonstrates reference consistency; the runtime system guarantees that the same cameras which have been activated to perform face recognition are turned off after the operation completes. Note that the SM-based runtime system is transparent to the programmer, except in the *main* method which performs the initialization (i.e., the SMWrapper is initialized in order to

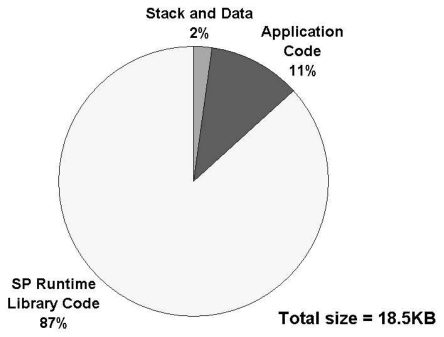

Figure 8: SM Code Breakdown for Intrusion Detection Application

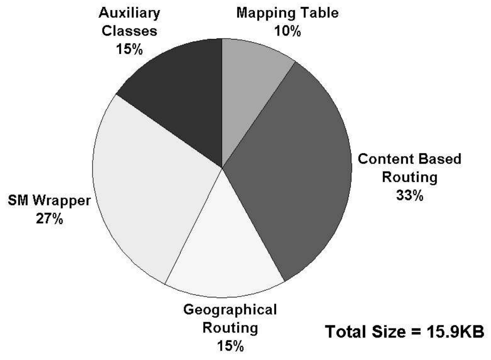

Figure 9: SP Runtime Library Code Breakdown

allow it to create the SM that will carry the SP application through the network).

For experiments, we have considered the simple network topology presented in Fig. 7. The response time is heavily influenced by the size of the payload carried by the SM "incarnation" of our SP application. Fig. 8 presents the breakdown of the SM payload (code, data, and execution state). The code consists of SP application and SM-based runtime library code (i.e., the SM needs to carry the runtime library code to those nodes where this code is not cached). We can see that the execution state is small (under 3% of the total size). The biggest contribution comes from the library code (the size of its components are shown in Fig. 9). This code, however, is cached at nodes in the common case.

In Figure 10, we present the total execution time for the application in two cases: (1) the code is not cached at any node when the application starts (but the caching is activated in the network), and (2) the SM-based runtime library code is cached at every node (i.e., only the application code is migrated through the network). In this experiment, we do

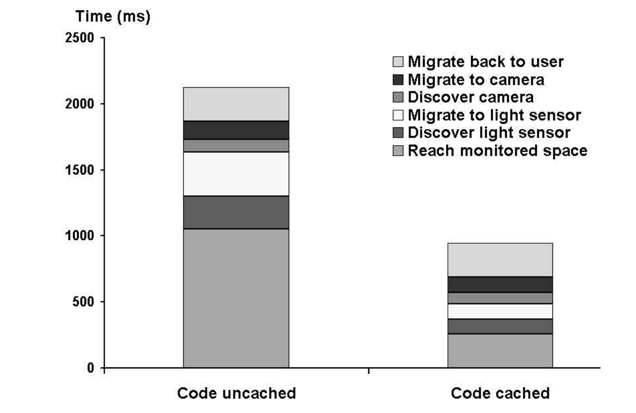

Figure 10: Execution Time for Intrusion Detection Application

not perform the face recognition because our goal is to evaluate the performance of the SP runtime system (i.e., the execution time for the face recognition is an order of magnitude greater than the rest of the application). The results indicate that our SP runtime implementation based on SMs can achieve good performance, especially when the runtime library is cached at nodes. We observe that caching leads to a 57% decrease in the overall response time. The time breakdown shows how each basic operation is affected by code caching. The time to reach the space of interest and the time to migrate to target nodes are significantly reduced (as much as 70%). The route discovery time experiences a less significant decrease due to the unavoidable contention encountered in wireless networks for flooding-based algorithms.

5. Related Work

Recent projects [11, 23, 3] have presented programming models for ubiquitous/pervasive computing. SP shares some of their goals, but its main design goal is to provide simple abstractions to program distributed applications for systems embedded in the physical space. These abstractions decouple the access to network resources from the networking details.

Although geographical routing [17, 18] and content-based naming and routing [4, 13] have been extensively studied, a simple and intuitive programming model that allows the user to express the computation in terms of physical location and content (or services) provided by nodes is still missing. SP offers such a model, and its runtime system takes advantage of these routing algorithms (especially of those developed for ad hoc networks).

The "database" model [5, 19] for programming sensor networks is a research complementary to SP. For instance, TAG [19] defines an SQL-like language for sensor networks. Both SP and TAG provide simple programming constructs that shield the programmer from the underlying network. There are two main differences between SP and this work. First, the programmer has fine-grained control over execution in SP, while TAG depends entirely on the compiler (i.e., essentially SP offers an imperative language, while TAG offers a declarative language). Second, SP focuses on flexible abstractions that support programming for uncertainty in highly dynamic networks, while TAG focuses on a set of queries executed efficiently in the network.

The design of SMs, the underlying architecture for SP, has been influenced by a variety of other research efforts, particularly mobile agents for IP-based networks [16, 10] and active networks (AN) [9, 20]. SMs leverage the general idea of code migration, but focus more on flexibility, re-programmability, and ability to perform distributed computing over unattended NES. Unlike mobile agents, SMs address nodes by content, discover the network configuration dynamically, are responsible for their own routing, and require minimal system support at nodes. SMs and AN differ in terms of their main focus. While AN targets fast communication in IP networks, SMs target programmability in NES. Another significant difference is that AN do not migrate the execution state from node to node whereas the SMs do. The migration of the execution state for SMs trades off overhead for flexibility in programming sophisticated tasks which require cooperation and synchronization among several entities.

SensorWare [8] can be an alternative solution for the SP runtime system, especially in networks composed of devices with extremely limited resources (e.g., sensor networks). SensorWare is similar to SMs in the sense that both SensorWare and SMs are systems based on code migration. Therefore, both are suitable for re-programming the network. SMs, however, offers the advantage of programming in a well known language (Java) which is supported on many embedded systems today [1]. Also, the tag space abstraction provided by the SM architecture and the SM self-routing mechanism simplify the implementation of the SP runtime system.

6. Conclusions

In this paper, we have presented the design and implementation of Spatial Programming (SP) using Smart Messages (SM). To the best of our knowledge, SP is the first attempt to design and implement a space-aware programming model for outdoor distributed computing. SP offers fine-grained, network-transparent access to systems embedded in the physical space. The main benefits of SP are the flexibility and simplicity to program user-defined distributed applications in highly volatile outdoor computing environments. The preliminary experimental results demonstrate that SP applications executed over our SM-based runtime system can achieve good performance and that caching the runtime library at nodes improves significantly the overall response time.

References

[1] Java 2 Platform, Micro Edition (J2ME). http://java.sun.com/j2me/.

[2] Linux Devices. http://www.linuxdevices.com.

[3] S. Adhikari, A. Paul, and U. Ramachandran. D-Stampede: Distributed Programming System for Ubiquitous Computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, pages 209–216, Vienna, Austria, July 2002.

[4] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP 1999)*, pages 186–201, Charleston, SC, 1999. ACM Press, New York, NY.

[5] P. Bonnet, J. E. Gehrke, and P. Seshadri. Querying the Physical World. *IEEE Personal Communications*, 7(5):10–15, October 2000.

[6] C. Borcea, C. Intanagonwiwat, A. Saxena, and L. Iftode. Self-Routing in Pervasive Computing Environments using Smart Messages. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom 2003)*, pages 87–96, Dallas-Fort Worth, TX, March 2003.

[7] C. Borcea, D. Iyer, P. Kang, A. Saxena, and L. Iftode. Cooperative Computing for Distributed Embedded Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, pages 227–236, Vienna, Austria, July 2002.

[8] A. Boulis, C. Han, and M. Srivastava. Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, pages 187–200, San Francisco, CA, May 2003.

[9] D. Wetherall. Active Network Vision Reality: Lessons from a Capsule-based System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP 1999)*, pages 64–79, Charleston, SC, December 1999. ACM Press, New York, NY.

[10] R. Gray, G. Cybenko, D. Kotz, and D. Rus. Mobile Agents: Motivations and State of the Art. In J. Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2002.

[11] R. Grimm and et al. Systems Directions for Pervasive Computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 147–151, Elmau/Oberbayern, Germany, May 2001. IEEE Computer Society, Washington, DC.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Cambridge, MA, November 2000. ACM Press, New York, NY.

[13] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the Sixth annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 56–67, Boston, MA, August 2000. ACM Press, New York, NY.

[14] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 96–107, San Jose, CA, October 2002. ACM Press, New York, NY.

[15] B. Jung and G. S. Sukhatme. Cooperative Tracking using Mobile Robots and Environment-Embedded, Networked Sensors. In *the 2001 IEEE International Symposium on Computational Intelligence in Robotics and Automation*.

[16] N. Karnik and A. Tripathi. Agent Server Architecture for the Ajanta Mobile-Agent System. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pages 66–73, Las Vegas, NV, July 1998.

[17] B. Karp and H. Kung. Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of the Sixth annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 243–254, Boston, MA, August 2000. ACM Press, New York, NY.

[18] Y.-B. Ko and N. H. Vaidya. Location-Aided Routing(LAR) in Mobile Ad Hoc Networks. In *Proceedings of the Fourth annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 66–75, October 1998.

[19] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI).*, December 2002.

[20] J. Moore, M. Hicks, and S. Nettles. Practical Programmable Packets. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, pages 41–50, Anchorage, AK, April 2001.

[21] C. Perkins and E. Royer. Ad Hoc On Demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 1999)*, pages 90–100, New Orleans, LA, February 1999.

[22] N. Priyantha, A. Miu, H. Balakrishnan, and S. Teller. The Cricket Compass for Context-Aware Mobile Applications. In *Proceedings of the 7th annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2001)*, pages 1–14. ACM Press, New York, NY, July 2001.

[23] M. Roman and R. Campbell. GAIA: Enabling Active Spaces. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 229–234, Kolding, Denmark, September 2000. ACM Press, New York, NY.