Spatial Programming using Smart Messages: Design, Implementation, and Evaluation *

Cristian Borcea, Chalermek Intanagonwiwat, Deepa Iyer, Porlin Kang, Akhilesh Saxena, Ulrich Kremer, and Liviu Iftode †

Division of Computer and Information Sciences
Rutgers University
{borcea, intanago, iyer, kangp, saxena, uli}@cs.rutgers.edu, iftode@cs.umd.edu

Abstract

Spatial Programming (SP) is a novel programming model for networks of embedded systems. The goal of Spatial Programming is to offer a simple, yet expressive way of describing distributed computations over massive, ad hoc networks of spatially distributed embedded systems. Four main design principles represent the basis for Spatial Programming: space is a first order programming concept that needs to be exposed to applications, the access to spatially distributed network resources is decoupled from networking, references to network resources are consistent throughout the program, and SP programs should tolerate the network configuration dynamics.

This paper presents the Spatial Programming design and its implementation using Smart Messages. Smart Messages are migratory execution units consisting of code and data, which migrate through the network, route themselves at each node in the path, and execute on nodes of interest. We have developed and evaluated an SM prototype over a modified version of Sun's KVM (as a Java execution environment). The SM prototype executes over Compaq's handheld iPAQs running Linux and the IEEE 802.11 MAC for wireless communication. The preliminary experimental results for one application running over our prototype show that Spatial Programming can be a viable programming model for networks of embedded systems and that Smart Messages can be successfully used to implement it.

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

†Current Affiliation: Department of Computer Science, University of Maryland

1. Introduction

During the next decade, emerging technologies will create ubiquitous computing environments [33, 29], populated with a sheer number of heterogeneous embedded systems. The processing power of these systems will increase significantly and they will be able to communicate to each other in ad-hoc manner mostly through wireless interfaces. Therefore, we envision the possibility of living in a physical world populated with large-scale networks of embedded systems (NES). For instance, sensors monitoring the environment [19, 16, 18], robots with intelligent cameras collaborating to track a given object [4], home appliances communicating to handle domestic activities [1], or cars on a highway cooperating to adapt to traffic conditions [5] will become a daily reality. In order to utilize this ubiquitous computing environment, new abstractions and programming models are needed. In fact, we argue that programming such networks, which are both large-scale and ad-hoc, cannot be done using the traditional distributed computing models.

Scale is not the only major difference between NES and conventional networks. In NES, nodes and links are volatile; they may join or leave at any moment (becoming unreachable due to mobility, energy depletion, failures, or disposal), leading to extremely dynamic network topologies. The question is how to write applications for such a fluid configuration in a way that allows the programmer to express the desired computation while ignoring most of the networking aspects. The goal is similar to the design of the shared virtual memory to hide the message passing communication from the programmer, while offering a shared memory parallel programming model for networks of computers [28]. The main difference between shared virtual memory and spatial programming is that the former is

performed over a stable and robust network while the latter must tolerate dynamic configurations deployed in the physical space with unknown time bounds for routing and node access.

Unlike traditional distributed systems where the physical location of the nodes does not matter, the spatial distribution of nodes across physical space is a key feature of massive NES. These networks will span buildings, large facilities such as campuses or airports, or even roads and forests. NES applications will prefer to express their interest for data and services in terms of locations within well-defined geographical regions, rather than by naming particular nodes in the network. Therefore, a programming model for NES must consider space as a first order programming concept to allow applications to name resources by their location.

Applications running in NES will refer resources, data, properties or services, not individual nodes. From the application point of view, nodes with the same properties, located in the same region, might be interchangeable. Fixed naming schemes, such as IP addressing, will be almost irrelevant in this case. For example, it might be desirable to reach a node that has a temperature sensor, but a fixed binding between the desired property and a unique identifier for a node is inappropriate. If the destination node becomes unavailable (due to mobility or node failures), the routing can fail even though multiple nodes providing the same property are available. However, naming consistency must be supported. After a node with the desired properties is discovered and the mapping is done, the same name should refer to the same node if desired. Therefore, a programming model for NES should rely on a naming scheme based both on content and space.

Although, both geographical routing [21, 24, 25], and content-based naming and routing [8, 15, 20] have been extensively studied, a programming model that allows the user to express the computation in terms of these abstractions is still missing. For instance, how should we write a program that performs distributed object tracking over a certain geographic region using intelligent, possibly mobile, cameras whose exact location, number and availability is unknown ? Or, how should we write a program that goes and selectively turns water sprinklers on if the field surrounding them is dry, without causing too much water overlapping ?

This paper describes Spatial Programming (SP), a novel programming model for large networks of embedded systems based on spatial and content-based naming of the network nodes. In this model, the network resources (content or services provided by nodes) are accessed transparently using *spatial references*, represented as {space:name} tuples. Similar to the mapping from virtual to physical memory in a conventional computer system, a mapping between spatial references and nodes in the physical world is maintained. For every access to a spatial reference, the underlying system will take care of name resolution and binding, communication, and routing. In this way, spatial programming is decoupled from the underlying networking implementation.

We also describe a proof-of-concept implementation of SP using Smart Messages [11]. A Smart Message (SM) is a distributed application which executes on nodes of interest, named by their properties and reached using *self-routing* at intermediate nodes. Nodes in the network cooperate to support Smart Messages by providing (1) a persistent name-based memory (Tag Space), and (2) an architecturally independent environment (Virtual Machine) for SM execution.

We have evaluated a simple SP implementation over our SM prototype running over a testbed consisting of Compaq's iPAQs running Linux. The SM prototype uses a modified version of Sun's KVM Java virtual machine as an SM execution environment. These nodes use IEEE 802.11b for wireless communication. The experimental results presented in this paper show that SP is a viable programming model for networks of embedded systems and that Smart Messages can be successfully used to implement it.

The rest of this paper is organized as follows. Section 2 describes the Spatial Programming model, its design principles, and the main programming constructs used to implement SP. Section 3 presents the implementation of SP using Smart Messages, and discusses possible alternative implementations of SP over other systems (directed diffusion, client-server architecture, and clustering-based schemes). Section 4 shows experimental results for this implementation over SM. Section 5 discusses the related work. The paper concludes in Section 6.

2. Spatial Programming

Spatial Programming (SP) is a new programming model for large networks of embedded systems. The main idea in SP is to offer network-transparent access to data and services distributed on nodes spread across the physical space in a similar fashion to memory access using virtual addresses.

SP provides the applications with a simple abstraction, called **spatial reference**, that represents a consistent resource naming in the network. A spatial reference is a reference to a node in SP and it is represented as a tuple **{space:tag}**. A **space** is a geographical

scope for a content-based naming of nodes. The **tag**s represent the properties/resources of a node.

Spaces can be defined statically or dynamically. Static definitions are used to describe physical spaces that do not change over time (e.g., a university campus or a freeway), while dynamic definitions typically specify spaces relative to a particular node or set of nodes of interest. We will discuss possible dynamic space definitions later in this paper. A proposal for static space definitions can be found in [26].

The range of applications that can benefit from the spatial programming model ranges from as simple as computing the average/maximum temperature over a given geographical region to complex collaborative applications such as distributed object tracking. Of particular interest to SP are applications that execute a distributed algorithm over a set of nodes selected based on their content and spatial properties.

Figure 1 illustrates a collaborative object tracking application which can be programmed using the SP model. In the figure, there are two spaces, *space1* and *space2*, and two types of nodes distributed across these spaces: motion sensors and intelligent cameras, marked with *motion* and *camera*, respectively. Each node is capable of determining its location (i.e., using GPS or other localization methods [23, 31, 13]) and remains static after deployment. However devices may fail, or be deployed far from other devices preventing them from participating in the computation. Since motion sensors are less expensive, their number is significantly greater than the number of cameras. Periodically, the application monitors the status of the motion sensors, and once motion is detected, the application turns on N cameras located in the neighborhood of that sensor and instructs them to perform collaborative object tracking. During this process, the application will access repeatedly these active cameras and will use partial results computed at each node to dynamically determine the next action. Once the object tracking completes, the active cameras will be turned off.

The task stated above is difficult to program using traditional message passing programming models[1]. The programmer would have to explicitly manage the communication between devices and to program all the details involved in reaching the area of interest and contacting the target nodes located there. Additionally, the network dynamics (caused by node failures or new sensors/cameras being deployed) may cause the application to fail since fixed addressing schemes treat exceptions as failures.

[1]Characteristics of message passing systems (e.g., Parallel Virtual Machine (PVM) [3], Message Passing Interface (MPI) standard [6]) include explicit management of communication (with possibility of deadlock due to mismatched communication pairs), fixed network topology with fixed addressing schemes, reliable communication channels, and unique program semantics (all or nothing semantics, not best effort semantics)

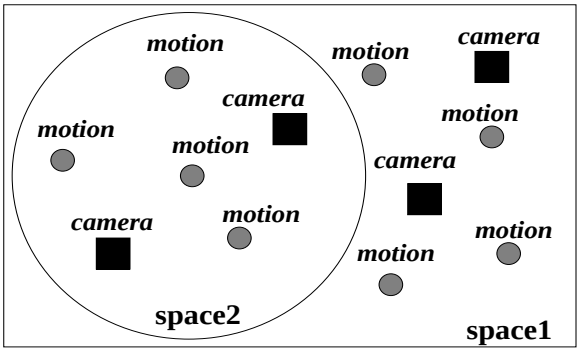

Figure 1. SP Example

```
1  for(i=0; i<N; i++){
2    {space2:camera[i]} = ON;
3    add(ActiveCameras, {space2:camera[i]});
4  }
5  objectTracking(ActiveCameras);
6  for(i=0; i<N; i++){
7    {space2:camera[i]} = OFF;
```

Figure 2. SP Application

Expressing the same task in our model is straightforward. The program that implements the desired application is presented in Figure 2. To refer to nodes of interest, the application use spatial references. For instance {*space2:camera*} is a reference to a *camera* node located in *space2* ({*space1:motion*} might be a reference to a *motion* sensor located in *space1*). The array subscript *[i]* is used to refer to different nodes with the same space-tag description. The code showed in the figure assumes that motion has been detected at a motion sensor located in *space1*, and *space2* is a subspace of *space1* determined dynamically relative to the location of this motion sensor (informally, *space2* is the neighborhood where the motion was detected). First, the SP application locates N cameras within *space2*, activates them, and includes them in the set of active cameras (lines 1-3). Second, the collaborative object tracking takes place over the set of active cameras (line 5). And third, the application turns off the active cameras once the job is done (lines 6-7). For this last step, reference consistency is assumed as the program must turn off only the cameras which it had previously turned on.

2.1. Design Principles

SP is based on four main design principles: (1) exposing space as a first order programming concept, (2) decoupling the access to network resources from networking details, (3) maintaining reference consistency, and (4) tolerance to network configuration dynamics.

1. Most envisioned distributed applications that will execute in a ubiquitous computing environment will have a localized behavior (i.e. they need to run within certain geographical regions in order to achieve their prescribed objectives). For instance, we may want to write an application that activates intelligent cameras in response to a trigger event in order to discover what caused that event. These cameras have to be within a physical range of the trigger node since otherwise no causal relation can be established. Therefore, location of nodes must be part of node naming in SP.

2. Applications should be able to access resources located on the nodes participating in NES by referring to them in the same fashion a program addresses memory locations using virtual addresses. A spatial reference should therefore include both spatial characteristics of the resource (i.e., the geographical region of the node containing the resource), and content characteristics of the resource (i.e., content that has to be present on the node). The application programmer however, should not have to perform any specific network-related operations to obtain the requested resources. An underlying system has to take care of name resolution, access to resources, routing, and communication.

3. The application that refers to a node with given spatial and content properties should be guaranteed to contact the same node each time it makes subsequent successful references using the same space-content description. This property provides the ability to perform arbitrary distributed computations over a set of nodes. For instance, in the example presented above the application needs to access multiple times the selected nodes of interest, while the order and type of access are determined at runtime. Therefore, the underlying system needs to maintain bindings between spatial references and the nodes addressed by them. These bindings are maintained per-application (similar to a page table) and are persistent during the SP program execution. A spatial reference is mapped to a particular node during the first access to the desired resource and it allows the programmer to visit repeatedly nodes and locations as long as the spatial reference is valid. An exception to this principle is reference rebinding, which is discussed in 2.2.5.

4. Since NES are extremely volatile, SP should make it easy for the programmer to deal with network configuration dynamics. This dynamics involves constant change in the location of nodes as well as intermittent network connectivity. For example, a reference made to a node may become invalid if this node has moved away from the area of interest or it simply ceased to exist. Additionally, the programmer does not even know if or how many resources of interest exist within the specified space. Therefore, a lookup operation for a spatial reference may take a substantial amount of time or it may not be able to complete at all. To cope with these situations, the SP model should allow programmers to specify time constraints for spatial references when incomplete computations are semantically acceptable (quality of result less than 100%). To bound the execution time, each spatial reference has an associated timeout value after which the attempt to access it is aborted and the application is informed. Thus, the application will be able to decide about its further actions.

2.2. Programming Constructs

Spatial Programming requires a set of programming constructs that can be added as extensions to any programming language or can be implemented as library calls. In the following, we present a set of SP programming constructs that express the most important features of our programming model (this set is not exhaustive since new constructs may prove to be necessary in the future). The main SP construct provides transparent access to network resources as well as an exception mechanism that allows applications to adapt to network configuration dynamics. The other constructs offer the possibility to create/remove network resources during execution, to define relative spaces dynamically, to change the space for, or to rebind a given spatial reference.

2.2.1. Accessing Network Resources

In SP, network resources can be accessed using a quintuple {*space:tag[instance], timeout*}.*resource* that specifies: (1) the spatial information for a node of interest, (2) the content-based name for this node, (3) the instance of a particular node with these spatial and

content-based properties, (4) an upper bound on the access time for the desired resource, and (5) the resource name at the node.

Spatial References. A *spatial reference* is a {*space:tag*} tuple that refers to a node located in *space* and named by content using *tag*. The *space* is a geographical scope for the content-based name of the node being addressed. Spatial references allow networking-transparent access to resources located on the referenced nodes. In the example presented in Figure 1 we have two spaces, *space1* and *space2*, and each node hosts either a *motion* or a *camera* tag. The reference {*space2:camera*} represents one of the the intelligent cameras that are located within *space2*.

Spatial Reference Instances. To make it possible to refer to more than one node with the same spatial and content properties we introduce the notion of *reference instance*. The reference instance is denoted by a particular index of a spatial reference. For example, in Figure 1, {*space2:camera[0]*} and {*space2:camera[1]*} denote two different camera nodes located in *space2*.

Access Timeout. Unlike traditional computer systems where the access time to resources is finite and can be upper bounded, in a volatile and dynamic NES it is difficult to estimate how long it takes to access a network resource (even the existence of a certain resource in a given space is unknown). To allow applications to complete even with a *quality of result* less than 100%, the SP programmer can specify a timeout for looking up a spatial reference (i.e., the maximum time that may be spent for that operation). If the target node is not reached, the underlying system raises a timeout violation exception and the application will decide about further actions. Commonly, the programmer sets each timeout based on the constraint imposed by the user on the total execution time (for instance, each new access can have the entire remaining time).

Resource Naming. A node referenced by a spatial reference can contain multiple resources which the SP application can individually name and access using the *dot* notation. The construct {*space:tag*}.*resource* refers the *resource* located on the node named by {*space:tag*}. In the example from Figure 1, {*space2:camera[i]*}.*active* may represent the status of this camera. Similarly, {*space2:camera[i]*}.*location* may represent the location of this node in space.

2.2.2. Creating/Removing Network Resources

Besides accessing resources that already exist at nodes, a program can dynamically create/remove its 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), or it may even create new services on nodes of interest (e.g., an image recognition service on a *camera* node). The primitives that offer this functionality are: *create({space:tag[i], timeout}.resource)* and *remove({space:tag[i], timeout}.resource)*.

2.2.3. Defining New Spaces

To allow for flexible specifications of spaces, SP supports basic operations on spaces. The application may use statically defined spaces (such as *campus1*) or create new *composed spaces* using union, difference or intersection operators. In our example from Figure 1, the reference {*(space1 - space2):camera*} returns a camera node located in *space1*, but not in *space2*.

Defining relative spaces based on the position of a node referenced by a spatial reference can be useful for many applications. For example, *rangeOf({space:tag[i]}, range)* defines a circular space with the center at the position of the node referenced by {space:tag[i]} (this position along with a unique identifier of the node are maintained in the associated binding) and the radius equals *range*. Similarly, *northOf, southOf, eastOf,* and *westOf* define semi-circles relative to the position of the node stored in the binding associated with the given spatial reference. In Figure 1, *space2* could have been dynamically defined using *rangeOf* as a function of the location of the sensor that detected motion in *space1*.

2.2.4. Space Casting

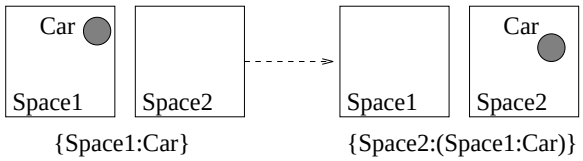

Figure 3. Space Casting Example

The semantics of spatial reference consistency requires that once the reference is bound to a node, the underlying system will locate the same node in the respective space each time the reference is used again. However, when nodes are mobile and the user has knowledge about their mobility patterns, the space for a given spatial reference can be modified using space casting. The construct {*space2:(space1:tag)*} changes the geographical scope for the given spatial reference from *space1* into *space2*. Figure 3 shows how to use space casting to reach in *Space2* a car which was previously bound to a spatial reference in *Space1*. For the case when the

```
 1  for(i=0; i<NumSensors; i++)
 2     if ({space1:motion[i]}.detect == true){
 3        space2 = rangeOf({space1:motion[i]}.location, Range);
 4        for(j=0; j<N; j++)
 5           if (space2:camera[j].active == OFF){
 6              {space2:camera[j]}.active = ON;
 7              {space2:camera[j]}.focus = {space1:motion[i]}.location;
 8              add(ActiveCameras, {space:camera[j]});
 9           }
10        objectTracking(ActiveCameras);
11        for(j=0; j<N; j++)
12           {space2:camera[j]}.active = OFF;
13     }
```

Figure 4. SP Code Example

destination space is not known SP provides the *Anywhere* space constant to cast a spatial reference to any space.

2.2.5. Spatial Reference Rebinding

In some cases, reference consistency is not necessarily or even preferably to avoid. For instance, an application that needs to contact periodically N temperature sensors located in a certain region and compute the average temperature may accept any sensor that provides the desired combination of space and content. In such a case, if a bound spatial reference cannot be found in its space, the spatial reference will be rebound to another node with the same space-tag properties rather than returning an exception for a failed access.

2.3. SP Program Example

To illustrate some of the novel concepts introduced by SP as well as the programming constructs described throughout this section, Figure 4 presents the source code for a more complex implementation of the collaborative object tracking application depicted in Figure 1. Once the motion is detected at one of the monitored motion sensors, a new space is created in order to perform object tracking over it (lines 1-3). Any *camera* node located in *space2* that is not active (i.e., working for other applications) is turned on, focused to the location of motion and added to the set of active cameras until the desired number of N active cameras has been reached (lines 4-8). During the object tracking, the cameras may be accessed multiple times and the action to be taken at a node depends on the partial results computed at previously visited nodes. The application ends by turning off the set of active cameras.

For the sake of simplicity, the access timeouts have been ignored. During the execution, there is a significant probability that some accesses will fail with a *timeout violation* exception. In such a case the application can decide to expand the geographical scope for the lookup operation, can rebind the spatial reference, or can even accept a lower quality of result (e.g., in our example, the application may accept *N/2* active cameras).

This source code shows the convenience that SP provides to the programmer to describe an application over NES using spatial references to access network resources in a networking-transparent way.

3. Implementation

SP can be implemented in different ways. In this paper, we show how SP can be implemented using Smart Messages (SM) [11]. In the following, we present an SM overview, the proof-of-concept implementation of SP over SM, and a discussion of alternative implementations for SP.

3.1. Smart Messages

Smart Messages (SM) are migratory execution units consisting of dynamically assembled code and data sections, termed "bricks", and a lightweight execution state. Each code brick is an independent program that may be used together with the other code and data bricks to generate a new, possibly smaller SM. The data bricks contain the mobile data carried by SMs during migrations. SMs migrate through the network, searching for nodes of interest, and execute at each node in the path. The SM execution is embodied in tasks described in terms of migration and computation phases.

3.1.1. SM Design

Since nodes in NES are resource constrained and have limited functionality, the goal of SM is to keep the support required from nodes in the network to the minimum, placing intelligence in SMs rather than in individual nodes. Placing intelligence in SMs provides flexibility and obviates the issue of implementing a new application or protocol in NES, which is difficult or almost impossible using current solutions [16].

Figure 5 presents the common system support provided by nodes to SMs: a name-based memory region, called the Tag Space, a Virtual Machine (VM), and an Admission Manager. The Admission Manager receives incoming SMs, decides whether or not to accept them, and stores these messages in an SM ready queue. The VM acts as a hardware abstraction layer for loading, scheduling, and executing tasks generated by accepted SMs. The Tag Space offers a name-based memory, persistent across SM executions and a uniform interface to the host OS and I/O system.

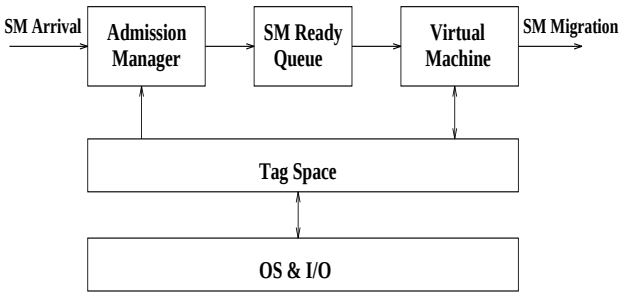

Figure 5. Node Architecture

Each SM has a three-stage life cycle: (1) creation, (2) migration to a target node and (3) execution upon acceptance at destination. After completion at a node, the SM may terminate or may decide to migrate to other nodes of interest, thus repeatedly alternating stage two and three.

Creation. Initially, an SM is created outside the network. Once it is injected in the network, new SMs (possibly smaller) can be dynamically created out of the existing SM's code and data bricks. Some of the code bricks are carried with the single purpose of creating new SMs during the life cycle of the current SM.

Migration. If the current computation does not complete at the local node, the SM may migrate to another node. Since SM tasks do not hold any resources at nodes (e.g., files, sockets) and there is no direct sharing among SMs, it is possible to implement a lightweight migration. The current execution control state is captured and migrated along with the code and data bricks (i.e., the only data transferred is the one incorporated explicitly by programmer in the data bricks). To successfully migrate, the SM must be admitted on the remote node. To prevent excessive use of resources (processor cycles, memory, energy, network bandwidth), the Admission Manager performs admission control at nodes based on the estimated resource requirements presented by SMs. The VM will make sure after admission that a task conforms to its declared requirements.

To reduce the cost of transferring the code, the code bricks are cached by the nodes. Generally, the applications in NES will have a localized behavior, exhibiting spatial and temporal locality. Thus, in the common case, the code bricks are cached in the network and the initial cost of transferring the code is amortized in time.

Execution. Upon admission, an SM generates a task which is scheduled for non-preemptive execution (other SMs can be accepted but not executed until the current SM terminates its execution). The execution starts from where it was left before a migration or from the beginning for new SMs. A task can terminate its local computation normally (exit or migrate), or it can be forcefully terminated by VM in case it does not conform to its declared resource requirements.

3.1.2. Tag Space

The Tag Space is a name-based memory which is maintained by each node and is persistent across SM executions. Essentially, each tag consists of a pair *(identifier, data)*. The identifier field is the name of the tag, and it is similar to a file name in a file system. The data field is application defined. The Tag Space contains two types of tags: application and I/O tags. The application tags are created dynamically by SMs. The SMs can read/write or delete them. The I/O tags are pre-defined on nodes and they provide SMs with a unique interface to the local OS and I/O system. The SMs cannot create/delete I/O tags, but they can read/write these tags. For example, to read the value provided by a temperature sensor or to get the image acquired by a video camera, an application has to read an I/O tag.

The tags can be used for naming, routing, synchronization, data exchange or data sharing among applications. SMs may create routing tags at visited nodes in the network, caching discovered route information in the data portion of these tags. Routing tags play the role of the routing table which can be used by subsequent SMs with similar interests, thus amortizing the route discovery effort. An application may require syn-

```
 1  Typical_SM(tag){
 2    do
 3       migrate_SM(tag, timeout);
 4       < do computation >
 5    until(<quality of result>);
 6    migrate_SM(back, timeout);
 7  }
 8  migrate_SM(tag, timeout){
 9    do
10       <decide next hop>
11       sys_migrate(next_hop, timeout);
12    until(readTag(tag));
13  }
```

Figure 6. Typical SM Code

chronization during its execution. A task can block on a specific tag pending a write of the tag by another SM. While the task is blocked, the VM may execute other tasks (i.e. the execution is non-preemptive, but a task may decide to yield the processor and block on a tag). When a task writes a tag, it wakes up all tasks blocked on the tag making them ready for execution.

3.1.3. SM Example

Figure 6 presents a typical SM example. The SM migrates to nodes hosting the *tag* of interest using *migrate_SM* (line 3) and executes on these nodes until a certain quality of result is achieved (for instance, a certain number of nodes of interest has been visited). When this is done, the SM migrates back to the node that injected it in the network (line 6)

The *migrate_SM* primitive implements content-based migration. It allows applications to name the nodes of interest by tag names and to bound the migration time. This function returns at a destination node or after the timeout expires. As nodes do not provide routing, it is the *migrate_SM* which implements an application-based routing (self-routing) using the Tag Space to store the discovered routing information, a low level migration primitive, called *sys_migrate*, and possibly other SMs for route discovery. The *sys_migrate* primitive is used to migrate an SM to the next hop in the path. Figure 6 shows also a typical routing implementation within *migrate_SM* (lines 8-13). A detailed description of the SM self-routing mechanism is presented in [12].

3.2. SP Implementation using SM

SMs represent a suitable implementation for SP for several reasons. They provide the ability to program the network on-the-fly, while requiring only a minimal system support at nodes. The Tag Space offers a uniform view of the network resources (both in terms of naming and access to resources). SMs are resilient to network volatility, being able to adapt to network conditions. For instance, an application can change dynamically its routing algorithm or it can even reduce its requirements as long as a certain quality of result is achieved. SMs can reduce the data traffic by migrating the execution to the nodes where data is produced (e.g., an image recognition program that executes directly at the node that acquired the image, or a distributed program that executes over a set of nodes located in a far away region and takes advantage of the spatial locality).

Essentially, an SP program can be translated into an SM program by translating all spatial references into migrations. Currently we are doing this translation manually, but we plan to do it automatically during a pre-compilation phase. The main steps in such a translation are:

1. Create a *binding table* for spatial references that will be carried by SM as mobile data. For each spatial reference the binding table will contain location and routing information for the corresponding node to which is was bound. Its role is similar to the role of a page table for virtual to physical address mapping.

2. Create data bricks for SP program variables (i.e., SM transfers only data saved explicitly by the programmer in data bricks).

3. Include at least one routing brick for geographical routing and one for content-based routing (these algorithms can be chosen based on the SP program analysis as well as the target network characteristics).

4. For each new spatial reference, the SM migrates to the desired space using geographical routing. When the space is reached, the SM uses the content-based migration to discover a node having the tag of interest.

5. Once a matching node is found, the corresponding binding information is added to the binding table. To reach the same node for further accesses, the SM creates a unique tag ID on the node during the first access to a spatial reference and stores it along with the space and the *location* of the node (i.e., geographic coordinates) in the associated binding entry.

```
 1 LookUpSpRef(space, tag, i, timeout){
 2  if (binding[space, tag, i])
 3   migrateSM(binding[space, tag, i], timeout);
 4  else{
 5   migrateSM(space, tag, timeout));
 6   id = createUniqueID();
 7   createTag(id, null);
 8   binding[space, tag, i] = {id, location};
 9  }
10 }
```

Figure 7. SP to SM Translation

6. To ensure reference consistency, subsequent accesses to a bound spatial reference must reach the same node, unless otherwise specified using space casting or reference rebinding (see Section 2.2). Therefore, if the spatial reference is bound, the SM migrates directly to the location of the referenced node. If the node is not found at its previous location, the SM uses the unique tag ID stored in the associated binding to discover it using content-based routing.

7. Tag Space operations are used to create/delete or access resources at nodes.

Figure 7 shows the SM code for a lookup operation performed for a spatial reference. Each *migrateSM* is aware of the global binding table and uses it to check for nodes already visited. If a binding exists, the SM uses the binding table to find the node in space based on the location and unique tag id (lines 2-3). Otherwise, the SM uses the original tag to find one matching node in space. Once a node is found, it is marked with a unique tag id and the corresponding binding information is stored in the binding table (lines 4-9).

3.3. Alternative Implementations

We have so far described an SP implementation using SM as a proof of concept. However, SP is independent from the underlying system. Examples of alternative implementations include directed diffusion, client-server architectures, and clustering-based systems. In this section, we briefly discuss possible SP implementations over these alternative systems in addition to their suitability.

Directed diffusion. Diffusion [20] is a data-centric communication paradigm for wireless sensor networks. This paradigm incorporates data-centric routing, attribute-based naming, and in-network processing for energy-efficient data dissemination. In diffusion, data is independent from its source node. Users specify what data they want rather than which node they want data from. The diffusion API is based on a publish/subscribe paradigm which supports two operations. Sinks (i.e., user nodes) can *subscribe* to named data and sources can *publish* data. Diffusion can be an attractive solution for implementing SP, especially in resource-constraint sensor networks. However, the implementation might not be trivial or natural because of the differences between their naming principle. Specifically, SP uses a dynamic node-naming scheme whereas diffusion uses attribute-based *data-naming* scheme for its communication primitives. In a sense, the diffusion API is declarative, thus diffusion might fit in better with an SQL-like language (e.g., TAG [30]). Nonetheless, the data-centric API of diffusion is sufficiently flexible for node-centric operations. Despite the naming differences, SP translation to diffusion is still possible (by using a tag name as an attribute). However, one SP program can result in multiple diffusion programs after translation: a source program, a sink program, and several filter programs for in-network processing.

The sink program contains most of the code from the SP program. Each spatial reference of a resource is translated into a subscription to that resource. This sink program also maintains a binding table which maps a tag name into a unique tag id. In each access, the sink checks to see if the binding of a requested tag exists in the binding table. If the binding does not exist, the sink subscribes for the resource using a tag name. The subscription takes the following form:

```
subscribe(tag = tag_name,
          u_tag_id = any,
          resource = resource_name,
          space = space_name)
```

Otherwise, the sink subscribes for the resource using a unique tag id:

```
subscribe(tag = tag_name,
          u_tag_id = unique_tag_id,
          resource = resource_name,
          space = space_name)
```

Conversely, the source program contains only subscription for receiving new tasks and publication for sending data. In addition, the SP program might process data from several sources. Such code can be separated out as a filter program for in-network processing.

Given that currently there is no support for remote installation of programs and filters in diffusion, the resulting SP programs will need to be installed before deployment of the sensor network. Even though there are several mechanisms which can be developed for remote installation of diffusion programs, the subject is beyond the scope of this paper.

So far, we have described SP as a sequential program accessing spatial references. However, SP might not fully benefit from diffusion because such sequential access does not encourage in-network processing or data aggregation. There are situations when a parallel access to spatial references is possible (there are no dependencies among them) and desirable in order to increase the performance. Future additions to SP will allow the programmer to specify such parallel activities (i.e., parbegin/parend, parfor). Despite these implementation issues, diffusion can be a suitable underlying system for SP because it already provides several features required by SP. Those required features include network-transparent operations, geographic [2], and content-based routing schemes.

Client-server. This approach uses traditional networking techniques. An example of such a system is an IP ad hoc network coupled with a resource discovery technique (e.g., Jini [2]) and a geographic routing scheme (e.g., GPSR [24]). Several layers of communication services in such a system can result in multiple naming layers and the overhead imposed by these layers might become unreasonable with hundreds or thousands of resource-constraint nodes that vary in availability (due to node mobility and failures).

However, one could implement SP for more powerful embedded device using the client-server paradigm. Unlike a possible SP implementation over directed diffusion, the client-server implementation must fill in network-related programming details (e.g., setting up and maintaining network connections). Each spatial reference can be translated into a request sent to a server which has the referenced resource. A client program contains most of the code from the SP program and maintains a binding table for mapping tags to a host names. For each spatial reference, if the referenced tag is not bound, the client program performs a resource discovery for the tag. Once the tag is bound to a host, the client sends a request to that host (a server of the referenced resource) using geographical routing. Upon receiving the request, the server accesses the resource and reports back to the client. Users can remotely install, program, or configure these nodes even after network deployment (due to generality of IP-style networks) but better reprogramming techniques are needed (to manually program large scale networks is too tedious and impractical).

Clustering-based. Another possible implementation for SP is a clustering-based system, which can improve the performance for relatively stable networks. There are several clustering-based approaches in ad hoc networking literature. Of particular interest are LEACH [17] and SCOUT [27] for sensor networks. A common goal in these approaches is to create a network structure which facilitates routing and data processing through cluster heads. To implement SP over such cluster-based topologies, we can assign a cluster-head per geographical region and this node can maintain approximate information for the content present in that area (e.g., information summaries represented as Bloom filters [10] can be acquired periodically for all nodes located within a region).

However, such an implementation might have a number of problems: the overhead of maintaining the cluster structure may become significant for large scale networks, the unavailability of a cluster-head may lead to the unavailability of all nodes from a region, and the use of clustering-based routing may not be justified in the presence of geographic information (a geographic-based route is potentially shorter than a clustering-based route without more excessive overhead).

4. Evaluation

This section presents the preliminary evaluation of a simple SP application manually translated to SM and executed over our SM prototype running on a testbed of eight iPAQs. Each of them communicates with one another over a wireless channel using 802.11b PC cards. We summarize our SM prototype, describe the evaluation methodology, study the code generation (and optimization) issues, provide an insight on reprogrammability of our system, and examine the time for each operation of the generated code.

4.1. SM Prototype

We have implemented an SM platform by modifying Sun's K Virtual Machine (KVM). SM applications are written in Java. The Tag Space operations and SM operations are available to SM applications as Java libraries. They are implemented using native methods. Code bricks are Java class files, and data bricks are objects. The main components of our implementation are: (1) the modified KVM which supports SM migration, and (2) the Tag Space that provides persistent memory for SMs as well as access to host OS and I/O system.

Migration involves three main operations: capturing the execution state, sending the SM to destination, and resuming the SM once it arrives at destination. To capture the SM execution state, the mobile data has to be

[2]The current distribution of diffusion is bundled with GEAR [34], a geographic and energy-aware routing scheme. Diffusion routes an interest message using GEAR if geographic information is available.

saved into the data bricks and the task execution state has to be captured. Instance of data bricks are treated as mobile data. We have implemented a trimmed-down version of the Java object serialization standard for KVM. The values of primitive fields are written out directly to the serialized stream. The class structure is known from the code bricks at the destination, so primitive field types can be identified through introspection. For non-primitive object types, the object's dynamic field descriptor is written out followed by the data values. The VM extracts the execution state of the SM generated task from the executing thread's call stack. For each method in the call stack of the thread, the execution state includes the offset of the VM instruction pointer inside the method, the offset of the stack pointer for the current VM thread, the method name, the method signature, method type (static or virtual), and either the class name (in case of a Java static method), or the object identifier (in case of a Java virtual method). Objects associated with virtual methods in the calling stack have to be included in the data bricks for invocation of the virtual methods at the destination. A list of objects, including recursive sub-objects written out to the output stream is maintained at the sender. The object's position in the output stream is used as the identifier for the object used for method invocation.

The SM presents its resource requirements in a resource table sent to the destination. The resource table contains the number of data bricks, the number of code bricks, unique identifiers for code bricks, sizes of data bricks, code bricks, and execution state. The destination uses this information for admission control and determining the cached code bricks. If the SM is accepted, according to the admission protocol, the destination replies with an acknowledgment and specifies the code bricks which have to be sent.

After an SM is accepted, the Admission Manager may add new code bricks to the code cache. Data objects are instantiated, data information is extracted from the serialized streams, and the data objects are populated. A new VM-level thread is built, and the stack frame of this thread is created such that the task resumes the execution after the call to *sys_migrate*. For each method in the calling stack, the stack pointer and instruction pointer offsets of the current method frame are set using the state received from the sender. A new stack frame is allocated for the method. For a Java static method, the name of the class and the method name are used to look up the method. For a Java virtual method, the dynamic class of object reference determines the method to be invoked. The Admission Manager adds the new thread to the SM ready queue and signals the VM indicating that a new thread is available for scheduling.

Tag Space Operation	Time (μs)
createTag	55.8
deleteTag	30.8
readTag	25.0
writeTag	28.0
block	24.6

Table 1. Time for Tag Space operations

I/O Tag	Access Time (ms)
neighbor_list	0.18
light_sensor	0.67
location	1.93
free_memory	0.12
battery_life	56.7
image_capture (32-KB)	341

Table 2. Time for I/O Tag accesses

The Tag Space is implemented within the KVM. Table 1 gives the average time for basic tag operations. The *createTag* primitive is the most expensive operation, but the cost of *readTag* and *writeTag* primitives affect the execution time in a greater extent, since they are called more frequently than *createTag* or *deleteTag*. Table 2 presents the access time to several I/O tags that are currently implemented in the SM prototype: neighbor discovery, light sensor, GPS location query, camera image capture, and system status inquiry (amount of free memory and battery lifetime). The *neighbor_list*, *location*, and *free_memory* tags have lowest access time since they simply read data cached in the kernel. The *neighbor_list* tag is typically queried by the geographical routing to determine the current list of neighbors and their locations. We have implemented a neighbor discovery protocol in 802.11b adhoc mode. A cache of known neighbors is kept in the kernel memory. The GPS cache is updated by a user level process which reads from the GPS serial interface. The *free_memory* tag is obtained directly from Linux *sysinfo* system call. The battery life and light sensor information are obtained by reading */dev/battery* and */dev/light_sensor* character devices, respectively, which translate to Linux kernel functions. The access time of *battery_life* tag is slightly higher since it involves I/O register read and write. For the *camera* tag, the system also performs YUYV to RGB format conversion on the captured image before returning it to the tag reader.

```
1 if ({space1:lightsensor[0]}.intensity > Limit){                // check light sensor intensity
2   space2 = rangeOf({space1:lightsensor[0]}.location, Range);   // create a space around the light sensor
3   if ({space2:camera[0]}.redPercentage > Threshold)            // check the red percentage for the image
4     {space1:home[0]}.redLocation = {space2:camera[0]}.location; // store camera location at the home node
5 }
```

Figure 8. SP application used in evaluation

4.2 Goals, Metrics, and Methodology

Our goals in conducting this evaluation study were three-fold: (1) to verify the viability of the SP model, (2) to understand the code optimization issues by studying the tradeoff between code migration and data migration, and (3) to explore the influence of code caching on our unattended re-programmable system.

We choose two metrics to analyze the performance of our system: the total number of bytes sent and latency. *The total number of bytes sent* measures the total amount of traffic (generated by our system) throughout the network. This metric implies the energy and bandwidth consumed by our system. It also indicates the overall lifetime of the system. *Latency* measures the delay observed in accessing the network resource. This metric defines the responsiveness of the system.

Our testbed consists of eight Compaq iPAQ H3700 and H3800 series (206-MHz Intel StrongARM SA-1110 32-bit RISC Processor, 32-MB Flash ROM, 64-MB SDRAM). For wireless communication, we use Lucent Orinoco 802.11b PC cards set for ad hoc mode. The nodes run the SM prototype (a modified version of Sun's Java KVM) over Linux. KVM is a virtual machine designed for mobile devices with resource constraints, suitable for devices with 16/32-bit RISC/CISC microprocessors/controllers, and with as little as 160 KB of total memory available.

The SP application which we designed, implemented and evaluated for this paper, is similar to the one described in Section 2 and its code is presented in Figure 8. Instead of motion sensors, each node has a light sensor incorporated, but we have activated the corresponding I/O tag, *lightsensor*, for just one of them. Another node has a video camera attached, which is identified by a *camera* tag. At this time, we have just one GPS receiver attached to a node. Therefore, the location of nodes has been set as an application tag and these values have been fixed for the duration of execution. The application reads the light intensity of a light sensor in *space1*. If this intensity is above a certain limit, a new relative space is defined based on the location of this sensor (i.e., the neighborhood) and the application accesses the information provided by a camera node in this space (it executes a very simple program that computes the percentage of red for the image acquired by this camera). If this percentage is greater than a pre-defined threshold, the application stores the camera *location* at the *home* node (i.e., a spatial reference for the *home* node already exists).

To translate this SP program into its Smart Messages (SM) form, we have implemented Java classes for location, space, spatial reference, and binding table. In the common case, all these classes will be cached at nodes since they are used by any SP program. The associated SM includes as data bricks multiple objects that instantiate these classes. We have also designed and implemented three simple routing algorithms used by SM to discover the space, content, or node of interest. The first one performs a greedy geographical routing that migrates the SM to the desired space. To simplify the implementation, the spaces were defined as circles, and this algorithm chooses the closest neighbor node to the center of the circle. The second routing brick implements an on-demand content-based routing for a given space. This routing works as follows: once the desired space is reached, the SM creates a *RouteDiscovery* that floods that space looking for the content of interest, while the *Main* SM blocks waiting for routing information. Due to its limited geographical scope, flooding does not represent a major problem for scalability. The *RouteDiscovery* SM returns to its source and unblocks the *Main* SM after the content has been discovered. On its path back, *RouteDiscovery* creates routing entries in the Tag Space. These entries will be used by the *Main* SM to migrate to the node having the desired content. The third routing algorithm is used to reach the node bound to a spatial reference (the node is identified by its location and unique id) and it is just a combination of the first two.

The network topology used for our application is typically 3 hops across (see Figure 9). The black node represents a light sensor node whereas the grey node represents a camera node. The user node is placed at the bottom left corner of the network. We studied the chosen metrics (i.e., total number of bytes sent in the

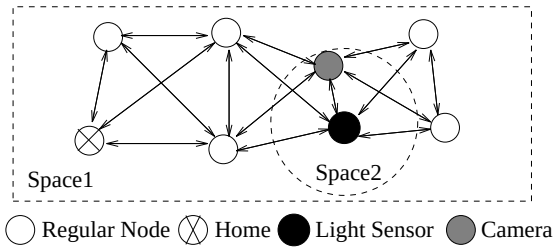

Figure 9. The network topology of our operational testbed of eight iPAQs

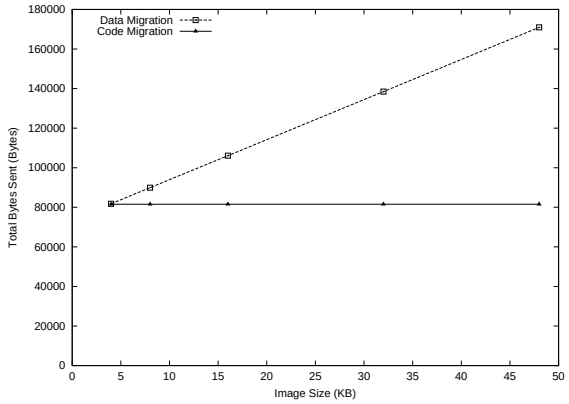

Figure 10. The total bytes sent in the network for two SP applications using SM (data migration and code migration)

network and latency) as a function of image size (4KB, 8KB, 16KB, 32KB, and 48KB). The resolution of the image is varied from 40x34 pixels to 128x128 pixels.

4.3 Code Generation and Optimization

There are several possible ways to translate an SP program to an SM program. Even though the basic translation concept is rather straightforward, the code optimization part is not. For example, in our application, one possible SM program will fetch the whole image from the camera node and process the image at the user node (i.e., data migration approach). An alternative SM program will migrate the processing code to the camera node first, process the image, and send only the result back to the user (i.e., code migration approach). The performance of these two SM programs can be significantly different. Indeed, the code migration approach has noticeably better performance (fewer total bytes sent) than the data migration approach

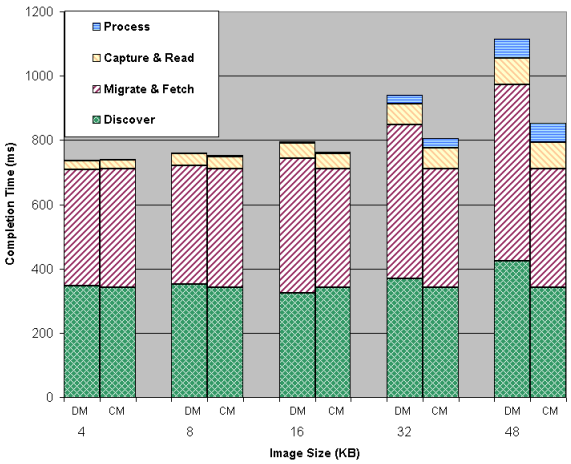

Figure 11. Latency for two SP applications using SM (data migration and code migration)

(Figure 10). In our experiments, the result (sent to the user node) is the percentage of the red color in the image. The data size of the result is constant regardless of the image size. Unsurprisingly, the total bytes sent by the code migration approach remains constant whereas the total bytes sent by the data migration approach increases with the image size. In general, if the size of the processing code is relatively small (581 bytes in our experiment), it is always better to migrate the code, given that the image size is usually larger than 4KB. However, in general, the data size is unknown during compilation. A more sophisticated code might send a small SM to check the data size first before making further decisions about code or data migration. Nonetheless, such an approach trades off latency for energy efficiency.

Why then, given the image size of 4 KB, are the total sent bytes significantly larger than 4 KB for both approaches ? At the beginning of our experiments, there was no SM program installed at each node. The SM program can be simply installed by injecting the SM code at the user node. The SM then routes itself to the nodes of interest. Therefore, our result also includes the cost for programming the network (Our SM program size is approximately 10KB. The cost in programming this 8-nodes network is about 80KB.) To factor out the installation cost, we later study the impact of code caching on this experiment.

Figure 11 plots the latency observed as a function of image size. For each image size, the first bar represents the latency for data migration whereas the second bar represents the latency for code migration. As

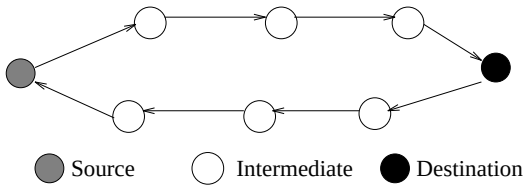

Figure 12. The network topology used for our code-caching experiments

expected, the code migration approach and the data migration approach are roughly equivalent for small image size ($\leq$ 16KB). The primary reason is that the latency is dominated by the route discovery time and the migrating-and-fetching time. The route discovery time is approximately constant regardless of the image size. The migrating-and-fetching time increases with the image size for data migration but remains approximately constant for code migration. Given that this result also includes the significant delay imposed by code installation for both approaches, the advantage of code migration becomes evident only when the image size is sufficiently large. In addition, the time for capturing, reading, and processing the image is equivalent for both approaches.

To test the geographical routing and to quantify the effects of code caching on the execution time for an SP application, we have implemented and evaluated a simple SM over the topology presented in Figure 12. The SM starts at the grey node and discovers the black node using geographical routing. Once this is done, it returns to its source on another path determined by the same geographical routing algorithm. The time taken by this execution when the code is not cached at nodes is 415.6 ms, while a second execution of the same SM (the code is cached by all nodes) takes only 126.6 ms.

5. Related Work

Recent projects [14, 9, 7] have presented programming models for ubiquitous/pervasive computing. SP shares some of their goals, but its main design goal is to define and implement a programming model that provides a simple way to program the physical spaces and to decouple the access to spatially distributed network resources from the networking details.

Spatial Views [26] provide a more restrictive, higher level programming model than SP. The Spatial Views programming model allows the specification of sets of nodes of interest, called views, together with a sequential program to be executed on each node in a view. In addition, language constructs are provided to group nodes according to their physical location, and to specify constraints that have to be satisfied by a program execution, including resources, time, and quality of result constraints. SP can serve as a possible target language for a Spatial Views compiler, but not vice versa.

A research complementary to ours is TAG [30], which defines an SQL-like language for sensor networks. Both SP and TAG provide simple programming constructions that shield the programmer from the underlying network. There are two main differences between SP and TAG. First, SP focuses on a flexible abstraction that allows programming for uncertainty in highly dynamic networks, while TAG focuses on a set of queries executed efficiently in the network. Second, the programmer has the control over execution in SP, while TAG depends entirely on the compiler (i.e., essentially SP offers an imperative language and TAG offers a declarative language).

Spatial Programming shares the idea of using spatial information with various forms of geographical routing [24, 21, 25], but it differs from them in its main goal, transparent computing over networks distributed within the physical space.

Content-based naming has been recently presented for both Internet [8, 32, 15] and sensor networks [20]. The spatial references used in SP are similar abstractions to these content-based names, but they incorporate the spatial information and present a uniform view of space and network to applications. Another difference is that SP targets ubiquitous computing environments and is independent of the underlying implementation.

Recent work on large networks of embedded systems has focused on network protocols for wired and wireless sensor networks [18, 20], system architectures for fixed-function sensor networks [19], and energy efficient data collection for mobile sensor networks designed to support wildlife tracking [22]. Sensor networks can represent a platform for SP. Since they are deployed across large geographical regions, SP provides a viable solution to alleviate the task of writing programs for them.

6. Conclusions

In this paper we have presented the design and implementation of Spatial Programming using Smart Messages. SP is a novel programming model for networks of embedded systems deployed in the physical space. To our best knowledge, Spatial Programming is the first attempt to design and implement a programming model for large networks of embedded systems. Spatial Programming provides a simple way to pro-

gram nodes spread across the physical space without dealing with network complexity. Applications written under this model use spatial references to access transparently spatially-bound network resources. Although Spatial Programming is independent of the underlying system, we have chosen Smart Messages as a suitable platform for its implementation. Smart Messages overcome the scale, heterogeneity and connectivity issues by placing the intelligence in migratory execution units, while requiring only a minimal system support.

We have designed and built two SP implementations (code migration and data migration) over our SM prototype. We have implemented this SM prototype using a modified version of KVM running over Linux on our operational testbed of eight wireless iPAQs. Our preliminary results show that the code migration approach generally performs better than the data migration approach, given that the migrated code is relatively small. We have shown that Spatial Programming is a viable idea by translating simple SP applications into SM applications and running them over this prototype.

References

[1] Electrolux Screenfridge. http://www.electrolux.se/screenfridge.

[2] Jini Network Technology. http://wwws.sun.com/software/jini.

[3] Parallel Virtual Machine. http://www.epm.ornl.gov/pvm/pvm_home.html.

[4] Sandia National Laboratories. http://www.sandia.gov/media/NewsRel/NR2000/avalanch.htm.

[5] Sensoria Corporation. http://www.sensoria.com.

[6] The Message Passing Interface (MPI) Standard. http://www-unix.mcs.anl.gov/mpi/.

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

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

[9] Banavar, G., Beck, J., Gluzberg, E., Munson, J., Sussman, J., and Zukowski, D. Challenges: An Application Model for Pervasive Computing. In *Proceedings of the Sixth annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)* (August 2000), pp. 266–274.

[10] Bloom, B. Space/time trade-offs in hash coding with allowable errors. *Communication of the ACM 13*, 7 (July 1970), 422–426.

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

[12] Borcea, C., Saxena, A., Iyer, D., Kang, P., Sarker, R., and Iftode, L. Self-Routing in Networks of Embedded Systems using Smart Messages. Tech. Rep. DCS-TR-477, Rutgers University, March 2002.

[13] Girod, L., Bychkovskiy, V., Elson, J., and Estrin, D. Locating Tiny Sensors in Time and Space: A Case Study. In *Proceedings of the International Conference on Software/Hardware Codesign (ICCD 2002). Invited Paper* (October 2002).

[14] Grimm, R., Davis, J., Hendrickson, B., Lemar, E., MacBeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., and Wetherall, D. Systems Directions for Pervasive Computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 2001.*

[15] Gritter, M., and Cheriton, D. An Architecture for Content Routing Support in the Internet. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)* (March 2001).

[16] Heideman, J., Silva, F., Intanagonwiwat, C., Govindan, R., Estrin, D., and Ganesan, D. Building Efficient Wireless Sensor Networks with Low-Level Naming. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (October 2001), pp. 146–159.

[17] Heinzelman, W., Chandrakasan, A., and Balakrishnan, H. Energy-efficient communication protocol for wireless microsensor networks. In *Proc. Hawaii Int. Conf. on System Sciences* (January 2000).

[18] Heinzelman, W. R., Kulik, J., and Balakrishnan, H. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. In *Proceedings of the Fifth annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)* (August 1999), pp. 174–185.

[19] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. System Architecture Directions for Networked Sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (November 2000), pp. 93–104.

[20] Intanagonwiwat, C., Govindan, R., and Estrin, D. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensors Networks. In *Proceedings of the Sixth annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)* (August 2000), pp. 56–67.

[21] Jinyang Li, John Janotti, Douglas De Couto, David R. Karger and Robert Morris. A Scalable Location Service for Geographic Ad Hoc Routing. In *Proceedings of the Sixth annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)* (August 2000), pp. 120–130.

[22] Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L.-S., and Rubenstein, D. 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)* (October 2002).

[23] Kaplan, E., Ed. *Understanding GPS: Principles and Applications.* Artech House, 1996.

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

[25] Ko, Y.-B., and Vaidya, N. H. 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)* (October 1998), pp. 66–75.

[26] Kremer, U., Iftode, L., Hom, J., and Ni, Y. Spatial Views: Iterative Spatial Programming for Networks of Embedded Systems. Tech. Rep. DCS-TR-493, Rutgers University, June 2002.

[27] Kumar, S., Alaettinoglu, C., and Estrin, D. SCalable Object-tracking through Unattended Techniques (SCOUT). In *Proceedings of the 8th International Conference on Network Protocols(ICNP).* (November 2000).

[28] Li, K. Shared virtual memory on loosely-coupled multiprocessors. Ph.D. Thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[29] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications* (August 2001).

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

[31] Priyantha, N. B., Miu, A. K. L., Balakrishnan, H., and Teller, S. 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), pp. 1–14.

[32] Vahdat, A., Dahlin, M., Anderson, T., and Aggarwal, A. Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS)* (October 1999), pp. 151–164.

[33] Weiser, M. The computer for the twenty-first century. *Scientific American* (September 1991).

[34] Yu, Y., Govindan, R., and Estrin, D. Geographical and Energy Aware Routing: A Recursive Data Dissemination Protocol for Wireless Sensor Networks. Tech. Rep. UCLA/CSD-TR-01-0023, UCLA, May 2001.