Programming Computers Embedded in the Physical World [*]

Liviu Iftode[1], Cristian Borcea[2], Andrzej Kochut[1], Chalermek Intanagonwiwat[2] [†], and Ulrich Kremer[2]

[1] *Department of Computer Science, University of Maryland, College Park, MD, 20742, USA*
[2] *Department of Computer Science, Rutgers University, Piscataway, NJ, 08854, USA*
{iftode, kochut}@cs.umd.edu, {borcea, intanago, uli}@cs.rutgers.edu

Abstract

During the next decade, emerging technologies will help populating the physical space with ubiquitous networks of embedded systems (NES). Programming NES requires new abstractions and computing models since the current programming models are not designed for the scale and volatility encountered in these networks. This paper presents Spatial Programming (SP), a novel programming model for NES. The key idea in SP is to offer network-transparent access to data and services distributed on nodes spread across the physical space. SP programs access network resources using a high level abstraction, termed spatial reference, which addresses the nodes using their spatial and content properties. An underlying system takes care of mapping spatial references onto target nodes in the network. Our preliminary experience in developing SP applications suggests that SP can be a viable solution for distributed computing over NES.

1. Introduction

Recent advances in technology made it feasible to create massive networks of embedded systems (NES). Such networks will represent the infrastructure for future ubiquitous computing environments [27, 23]. For instance, sensors monitoring the environment [14, 13], robots with intelligent cameras collaborating to track a given object [17], or cars on a highway cooperating to adapt to traffic conditions [2] will become a daily reality. In the next decade, the computational power embedded in these systems will increase significantly. Thus, they will be able to run reduced versions of traditional operating systems and more complex applications. We are already witnessing this trend through the occurrence of cars, cameras, and even watches running Linux [1].

Although extremely heterogeneous, nodes in NES will have a common set of characteristics: (1) communicate with each other using short-range radio (2) are distributed across the physical space, (3) are mobile and may fail frequently, and (4) work unattended. These nodes may join or leave the network at any moment (becoming unreachable due to mobility, energy depletion, failures, or disposal) leading to dynamic and volatile network topologies. Additionally, NES will have to function completely decentralized because deploying more powerful nodes that act as base stations will not be feasible in many situations (e.g., nodes deployed over a disaster area).

All these characteristics make NES programmability a challenging task. Traditional distributed programming models, such as message passing, are not suitable for NES. Aspects of mobility, volatility, and scale make it difficult to program applications as a series of interactions with nodes identified by fixed addresses (as assumed in the message passing model). Instead, we want to gather information from, or perform actions on nodes that have certain properties, content, or location. Therefore, we are no longer interested in contacting a particular node, but any node that may be useful to perform the task. Simpler programming abstractions and more flexible programming models are needed to let the programmers design and write their applications without worrying about the underlying networking details.

In this paper, we propose Spatial Programming (SP), a novel programming model for distributed computing over massive NES. In SP, space is a first order programming concept which is exposed to applications. SP shields the complexity of programming volatile, ad hoc networks by presenting a high level, uniform abstraction, termed *spatial reference*, for naming and accessing network resources. A spatial reference addresses the nodes in the network using their location and content properties, while an underlying system takes care of mapping it onto a target node in the network. In our model, programmers write simple, sequential programs and access transparently the network resources (i.e., using

[*]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

Term of Comparison	Traditional Target Networks	Networks of Embedded Systems
Location	Indoor	Outdoor
Nodes	Functionally Homogeneous	Functionally Heterogeneous
Operation	Under User's Control	Unattended
Scale	Relatively Small	Large
Topology	Stable	Ad Hoc and Volatile
Resources	Known A Priori/Infrequent Changes	Limited A Priori Knowledge/Highly Dynamic

Table 1. Traditional Distributed Computing Networks vs. Networks of Embedded Systems

spatial references) in a similar fashion to the access to memory using variables. SP is independent of the underlying system, thus allowing for multiple implementations. Each implementation has to translate the high-level, sequential SP program into a series of actions performed by the underlying system. A first implementation of SP is currently being developed over our Smart Messages platform [8]. More details about this implementation are given in Section 4.

The rest of this papers is organized as follows. Section 2 presents the motivation and challenges for SP. Section 3 describes the SP design and shows the basic SP programming constructs. The current status and future plans are presented in Section 4. Section 5 discusses the related work, and the paper concludes in Section 6.

2. Motivation and Challenges

The trend of embedding "intelligence" everywhere in the physical world will lead to the development of massive networks of embedded systems (NES). Traditionally, the main focus of distributed computing has been on performance or availability. Instead, the focus of distributed computing over NES will be on programming the physical world (i.e., programming real world devices to sense, observe, report, or perform collaborative tasks). The two major challenges toward this goal are: (1) how to execute collaborative applications over NES, and (2) how to allow for coordinated actions among nodes of interest in a decentralized manner.

To develop distributed applications for this huge computing infrastructure, we need to understand the unique set of characteristics possessed by NES. Table 1 presents a comparison between the target networks for traditional distributed computing (e.g., message passing) and NES. Unlike traditional distributed computing which takes place "indoor" over relatively small scale networks with stable configurations, distributed computing over NES takes place "outdoor" over large scale networks with highly dynamic configurations.

As a consequence, we believe that the traditional message passing programming model cannot satisfy the requirements of the applications running in this new world. Therefore, new programming models are needed. To illustrate this claim, we present a collaborative object tracking application example, depicted in Figure 1. Two types of nodes are distributed across a given geographical region (*space1*): motion sensors and intelligent cameras. Each node is capable of determining its location (i.e., using GPS [18] or other localization methods [25, 9]). The motion sensors remain static after deployment, but the cameras can be mobile [17]. The nodes 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, a *user* starts an application (e.g., from a wireless-enabled PDA) that monitors the status of the motion sensors in the network. Each time motion is detected, the application turns on a certain number of cameras located in the proximity of that sensor (i.e., informally, *space2* is dynamically defined as the neighborhood where the motion was detected). These cameras are instructed to perform collaborative object tracking in order to identify the object that triggered the motion sensor and monitor its actions. During this process, the application accesses repeatedly the selected cameras and uses the partial results computed at each node to dynamically determine the next action. Once the object tracking completes, the active cameras are turned off.

The task stated above is difficult and tedious to program using the traditional message passing programming model.

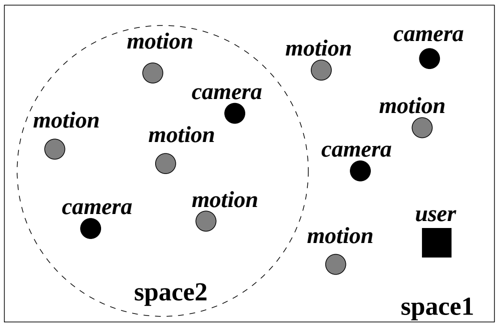

Figure 1. **Object Tracking Example**

Characteristics of message passing systems (e.g., Message Passing Interface (MPI) standard [3]) include explicit management of communication with possible deadlocks due to mismatched communication pairs, and “all or nothing” semantics (i.e., not “best effort” semantics). The programmers would also have to take care of all the details involved in reaching the area of interest and contacting the target nodes located there. This is not a trivial task in a volatile network with unknown configurations. In our example, the programmer does not know how many camera nodes are there or where exactly they are located. Additionally, the network dynamics (caused by failures, mobility, or deployment of new nodes) may cause the application to fail since fixed addressing schemes treat exceptions as failures. To summarize, a better programming model for NES needs to answer the following questions:

- How to write simple and intuitive programs for NES?
- How to refer to nodes in a network-transparent way?
- How to discover and access resources at nodes?
- How to cope with network dynamics?

Our proposed programming model, Spatial Programming (SP), is specially designed to provide solutions to the above questions. The key 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 access to memory using virtual addresses. The main benefits of SP are:

- Simple to understand programming model which allows for rapid development of NES applications.
- Network-transparent computation that uses network resources similar to normal variables.
- Possibility to program on-the-fly unattended networks for new tasks even after network deployment.
- Best-effort computation which tolerates the network dynamics while providing a certain quality of result to applications.

3. Spatial Programming Model

In this section, we detail the main SP concepts and explain how they address the unique set of challenges posed by NES programmability.

3.1. Spatial References

A spatial reference, represented as a tuple {*space:tag*}, refers to nodes in the network in the same fashion a program addresses memory locations using virtual addresses. The *space* is a geographical scope for a the node being addressed. The *tag* is a content-based name for the same node (i.e., it names certain data or service provided by that node). These can be three examples of spatial references for our application:

```
{space1:motion}, {space2:camera}, {space2:user}
```

The first spatial reference refers to a node hosting a *motion* tag in *space1*. The second one refers to a *camera* node located in *space2*. The third reference is invalid because there is no node with a *user* tag in *space2* (in such a case, an exception is raised to the application).

3.1.1. Why do we need spatial information?

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. For instance, in the motivating example, we need to activate a number of intelligent cameras to discover and track the object that triggered the motion sensor. These cameras have to be within a physical range of the trigger node because otherwise no causal relation can be established. Therefore, SP considers space as a first order programming concept and exposes it to applications through spatial references.

3.1.2. Why do we need content-based naming?

Applications running in NES will refer to resources (data, properties, or services), not individual nodes. From an 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 motion sensor, but a fixed binding between the desired property and a unique identifier for a node is inappropriate. If the destination node becomes unavailable, the application can fail even though multiple nodes providing the same property are available. Therefore, SP relies on a naming scheme based both on space and content.

3.1.3. How to define the spaces?

The application may use statically defined spaces or create dynamically new spaces. Static definitions are used to describe physical spaces that do not change over time and are commonly provided in the form of names associated with geographical regions (e.g., using topological maps). Dynamic definitions typically specify *composed spaces* (using the union, difference, or intersection operators) or *relative spaces* (defined relative to the position of a referenced node). The relative spaces are of particular importance to SP because they allow an application to "remember" and access a space where a certain event took place even after the node that produced or detected this event is no longer there. For our example, this is how we can define and use two dynamic spaces:

```
(1) diff=space1-space2
    {diff:camera}
(2) space2=rangeOf({space1:motion},Range)
    {space2:camera}
```

In the first case, we refer to a *camera* node located in *space1*, but not in *space2*. The second example shows how to define the "proximity" of a node using the *rangeOf* operator (i.e., it defines *space2* as a a circular region with the center at the position of the node referenced by {*space1:motion*} and radius equals *Range*). Similarly, *northOf, southOf, eastOf,* and *westOf* define semi-circles relative to the position of a referenced node.

3.1.4. How to define the tags?

Generally, content-based naming can involve arbitrary expressions on content names and content values. In spatial references, tags are just strings that name content or services at nodes. The tags are not globally unique since they name content or services that can be provided by multiple nodes. Although the namespace for massive NES can be extremely large, we expect that certain semantic rules will be adopted and enforced to make tag naming intuitive and manageable.

3.1.5. How to distinguish among similar nodes?

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. In our example, the references presented below represent two different nodes:

```
{space2:camera[0]}, {space2:camera[1]}
```

Commonly, a programmer is not aware of the exact number of the nodes with a certain content in a given space. For instance, an attempt to use the reference {*space2:camera[2]*} in our application will trigger an invalid reference exception.

3.2. Reference Consistency

An application that refers to a node with given spatial and content properties is guaranteed to contact the same node each time it makes subsequent successful references. This property provides the ability to perform arbitrary distributed computations over a subset of nodes of interest. For instance, in our example, the application needs to access the selected *camera* nodes multiple times, but the order and type of access are determined at runtime. Therefore, the underlying system needs to maintain bindings between spatial references and the nodes they address. These bindings are maintained per-application (similar to a page table) and are persistent during the SP program execution. When the first access to a network resource takes place, a new binding for a spatial reference is created. The data stored in a binding is implementation-specific, but it must include a unique ID for the referenced node and the location of this node (i.e., x, y coordinates). This data allows an application to visit repeatedly nodes and locations as long as the spatial reference is valid.

In some situations, reference consistency is not necessary, or it is even preferably to be avoided. 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 referenced node cannot be found in its space, the spatial reference can be rebound to a similar node rather than returning an exception for a failed access.

3.3. Accessing Resources

Spatial references allow network-transparent access to resources located on the referenced nodes. Applications use network resources in a similar fashion to normal variables. An application programmer does not perform any network-related operation to obtain the requested resources. The underlying system takes care of name resolution, access to resources, routing, communication, and security.

A node can contain multiple resources, which can be individually accessed by an SP application using the *dot* notation for spatial references. For instance, our object tracking application can access different properties on a *camera* node:

```
(1) {space2:camera[1]}.active=ON;
(2) {space2:camera[1]}.focus=location;
(3) img={space2:camera[1]}.image;
```

These examples show how network resources, named by *tags*, can be used in a program in the same way as normal variables. In the first two lines, we assign values for two properties on a *camera* node (i.e., we turn the camera on, and we focus it toward a certain point in space). In the third line, we store the image acquired by the camera into a variable.

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 these functionalities are illustrated below:

```
create({space2:camera[1]}.partialResult);
res=analyzeImage({space2:camera[1]}.image);
{space2:camera[1]}.partialResult=res;
                .........
remove(space2:camera[1]}.partialResult);
```

In this example, the application creates a new resource, *partialResult*, to store the result of analyzing a newly acquired image. An application can perform collaborative object tracking using partial results from multiple cameras. When the *partialResult* is not needed any longer, the application can remove it.

A natural question that can be raised at this point is: how are the network resources shared? For now, the resource sharing policy in SP is very simple: the resources provided by nodes are shared (i.e., the system does not guarantee that an application accessing the same resource twice gets the same result), and the resources created by applications are private. More complex policies and their interaction with the underlying system will be considered for the future design refinements.

3.4. Programming for Uncertainty

Since NES are extremely volatile, SP should make it easy for programmers 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 a certain space.

3.4.1. How long does it take to reach a node?

Unlike traditional computer systems where the access time to resources is finite and an upper bound can be determined, in a volatile and dynamic network such as 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). The operations on network resources in NES may take a substantial amount of time or may not be able to complete at all. Instead of waiting an indefinite amount of time to complete, many applications would be willing to trade the quality of result (as long as this result is semantically acceptable) for the response time. Therefore, to allow applications to make progress even in adverse network conditions, SP proposes "best effort" computing. Under this paradigm, a programmer is allowed to set time constraints on spatial references. The underlying system does not guarantee that it will be able to locate and access a resource in a certain amount of time, but it guarantees to raise a timeout violation exception to the application if the access does not succeed in that time period. In such a situation, the application decides about further actions (e.g., retry, look for a similar resource, continue without this resource). The code presented below illustrates a time constrained access to a resource on a node referenced by $\{$*space2:camera*$\}$.

```
try{
     {space2:camera,timeout}.active = ON;
}catch(Timeout e){
     // application decides further actions
}
```

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

3.4.2. What happens if a referenced node moves?

When the nodes are mobile, the semantics of spatial references requires that once the reference is bound to a node, the underlying system locates the same node in the respective space each time the reference is used again (i.e., reference consistency). If a node moves out of its space, the application receives an invalid reference exception. If the programmer has knowledge about the mobility patterns of certain nodes, however, the space for the spatial references associated with these nodes can be modified using *space casting*. In our example, if a referenced camera moves within *space2*, the reference remains semantically valid. But, if it moves outside *space2*, space casting can be used to expand its geographical scope to *space1*:

```
(1) {space2:camera}
(2) {space1:(space2:camera)}
```

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.

3.5. Putting It All Together: Program Example

Table 2 shows a comparison between traditional message passing distributed computing and SP, which emphasizes the novel concepts introduced by SP. These concepts allow for a large spectrum of NES distributed applications, ranging

Term of Comparison	**Traditional Distributed Computing**	**Spatial Programming**
Focus	Performance/Availability	Programming Physical World
Naming	Fixed Addresses	Combination of Space and Content
Communication	Managed Explicitly by Programmers	Transparent
Semantics	All or Nothing	Best Effort
Re-programmability	Difficult after Network Deployment	Easy to Execute New Applications

Table 2. Traditional Distributed Computing vs. Spatial Programming

```
 1 if ({space1:motion[i]}.detect==true){
 2  space2=rangeOf({space1:motion[i]},Range);
 3  location={space1:motion[i]}.location;
 4  try{
 5   for(j=0,k=0;j<Nc;k++)
 6    if ({space2:camera[k],timeout}.active==OFF){
 7     {space2:camera[k]}.active=ON;
 8     {space2:camera[k]}.focus=location;
 9     activeCameras[j++]={space2:camera[k]};
10    }
11  }catch(Timeout e){
12   if (j<Nc/2)
13    // restart monitoring the motion sensors
14   else
15    // else continue with the next instruction
16  }
17  result=objectTracking(activeCameras);
18  for(k=0;k<j;k++)
19   activeCameras[k].active=OFF;
20  return result;
21 }
```

Figure 2. Spatial Programming Code Example

from as simple as computing the average/maximum temperature over a given geographical region to complex collaborative applications such as distributed object tracking. Typical applications for SP are those which execute a distributed algorithm over a set of nodes selected based on their content and spatial properties.

To illustrate such an application, Figure 2 presents the source code for our object tracking example. Once the motion is detected at one of the monitored motion sensors (i.e., {*space1:motion[i]*} in the figure), a relative space is created around that sensor in order to perform object tracking within its proximity (lines 1-2). 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 *Nc* active cameras has been reached (lines 3-10). During the object tracking (line 17), the cameras may be accessed multiple times due to the reference consistency feature of SP. The actions taken at a camera node depend on the partial results computed at previously visited nodes. The application ends by turning off the set of active cameras (lines 18-19). For the sake of simplicity, we ignored the time constraints on spatial references throughout this example for most of the accesses to network resources. We show, however, what happens if a a timeout occurs in the process of constructing the set of active cameras. In such a situation, a timeout violation exception is raised (line 11), and the application has to decide what its further actions are ("best effort" computing). Our application accepts a possibly lower quality of result and goes ahead if at least half of the desired number of cameras is found. Otherwise, it restarts monitoring the motion sensors (lines 12-16). If a camera moves out of *space2* during the object tracking (line 17), the application may just ignore it (if enough active cameras), or may use space casting to re-discover it (considering the execution time and normal motion speeds, the camera should be in the proximity of *space2*).

4. Status and Future Work

In the following, we briefly outline the current status and future ideas for improving the SP design and implementation. We plan to further investigate what other features should be added to SP to make it even more flexible and easier to use in such complex environments as NES. We also plan to analyze the tradeoffs between various design choices that we face in implementing SP.

At this time, we are in the process of implementing SP using Smart Messages (SM) [8]. SMs consist of code and data sections as well as a lightweight execution state. They migrate through the network, searching for nodes of interest, and execute at each node in the path. SMs name the nodes of interest by content and self-route to them using other nodes as "stepping stones". Nodes in the network support SMs by providing a virtual machine and name-based memory, called Tag Space. The Tag Space is used for naming, inter-SM communication, synchronization, and interaction with the local host.

The use of SMs as the underlying system for SP offers several advantages. They provide the ability to program the network on-the-fly (including the possibility to dynamically install new services at nodes), while requiring only a minimal system support. 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 [7].

SP requires a set of programming constructs that can be added as extensions to any programming language or can be implemented as library calls. The decision whether to implement the SP constructs as language extensions or as library calls has to be evaluated both in terms of performance and ease of programming. We are currently working to complete an SM runtime system for SP, and an SP programming language will be developed in the near future. To investigate the practicality of SP over SM, we have designed and manually translated a simple SP application into an SM program. The application has been executed over an SM platform consisting of a modified version of Sun's Java KVM and a testbed composed of HP iPAQs running Linux and equipped with 802.11 wireless cards for communication.

For networks of resource constrained devices, such as sensor networks, a more traditional implementation may yield better performance. Therefore we plan to have an SP implementation on top of Directed Diffusion [15]. Other possible implementations might involve more traditional client-server or clustering-based approaches for relatively stable networks.

5. Related Work

Recent projects [11, 6, 4] 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.

SP is closely related to Spatial Views, an iterative spatial programming model for NES [22]. Spatial Views provide a more restrictive, higher level programming model than SP. This 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 (e.g., resources, time, quality of result). SP can serve as a target language for a Spatial Views compiler, but not vice versa since SP provides fine grained access to network resources which is not available in Spatial Views.

A research complementary to ours is TAG [24], which 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 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, while TAG offers a declarative language).

Content-based naming has been recently presented for both Internet [5, 26] and sensor networks [15]. The spatial references used in SP are similar abstractions to these content-based names, but they incorporate 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.

Although geographical [20, 21] and content-based routing [15, 12] 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 underlying implementation takes advantage of these routing algorithms (especially of those developed for ad hoc networks).

Recent work on sensor networks has focused on new communication paradigms [15], system architectures for fixed-function sensor networks [14], and energy efficient data collection for mobile sensor networks designed to support wildlife tracking [16]. 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.

The design of Smart Messages (SMs), our first SP platform, has been influenced by a variety of other research efforts, particularly mobile agents for IP-based networks [19, 10]. 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.

6. Conclusions

In this paper, we have presented the design of Spatial Programming (SP). SP is a novel programming model for networks of embedded systems (NES) deployed in the physical space. To our best knowledge, SP is the first attempt to design and implement a programming model for NES. SP provides a simple and intuitive way of programming nodes spread across the physical space without dealing with network complexity. The implementation of a simple SP application over Smart Messages and the preliminary experimental results provide us with the confidence that SP can be a viable programming model for massive NES.

References

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

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

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

[4] 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)*, pages 209–216, July 2002.

[5] 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)*, pages 186–201, 1999.

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

[7] 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)*, March 2003.

[8] 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)*, pages 227–236, July 2002.

[9] L. Girod, V. Bychkovskiy, J. Elson, and D. Estrin. 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.

[10] R. S. 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, 2001.

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

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

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

[14] 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)*, pages 93–104, November 2000.

[15] 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)*, pages 56–67, August 2000.

[16] P. Juang, H. Oki, Y. Wang, M. Martonosi, L.-S. 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)*, October 2002.

[17] 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*.

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

[19] 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)*, July 1998.

[20] 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)*, pages 243–254, August 2000.

[21] 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.

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

[23] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, August 2001.

[24] 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). To Appear.*, December 2002.

[25] N. B. Priyantha, A. K. L. 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)*, pages 1–14, 2001.

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

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