# Cooperative Computing for Distributed Embedded Systems *

Cristian Borcea, Deepa Iyer, Porlin Kang, Akhilesh Saxena, and Liviu Iftode †
Division of Computer and Information Sciences
Rutgers University
Piscataway, NJ, 08854, USA
{borcea, iyer, kangp, saxena, iftode}@cs.rutgers.edu

## Abstract

*The next generation of computing systems will be embedded, in a virtually unbounded number, and dynamically connected. The current software, network architectures, and their associated programming models are not suitable for this scenario. This paper presents a distributed computing model, Cooperative Computing, and the Smart Messages architecture for programming large networks of embedded systems. In Cooperative Computing, distributed applications are dynamic collections of migratory execution units, called Smart Messages, working to achieve a common goal. Virtually any user-defined distributed application can be implemented using our model. We present preliminary results for our prototype implementation as well as simulation results for two previously proposed applications for sensor networks, Directed Diffusion and SPIN, implemented using Smart Messages.*

## 1. Introduction

As the cost of embedding computing becomes negligible compared to the actual cost of goods, there will be a trend towards incorporating computing capabilities and wireless communication into most consumer products. The next generation of computing systems will be embedded, in a virtually unbounded number, and dynamically connected. Although these systems will penetrate every possible domain of our daily life, the expectation is that they will operate outside our normal cognizance, requiring far less attention from the human users than the desktop computers today.

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

†Current address: Department of Computer Science, University of Maryland, College Park, MD 20742

The first illustration of these systems that has received considerable interest in the last couple of years are networks of sensors [7, 5, 6]. These networks have severe resource limitations in terms of processing power, amount of available memory, network bandwidth, and energy. We envision that the next decade will bring a large class of networks of embedded systems (NES) with sufficiently large computing, communication and energy resources, although limited, to support distributed applications. For instance, there are already companies that propose computer systems embedded into cars and connected by Bluetooth [1]. For some of these networks, such as a networks of intelligent cameras performing object tracking over a large area, it might be beneficial to perform local computations and to cooperate in order to execute a global task. They may perform sophisticated filtering of data at a node that acquired an image, or even distributed object tracking rather than running a centralized algorithm at a server. The challenge that we face is how to program NES, namely, what is the appropriate computing model and what system support is necessary to execute distributed applications in these networks.

NES pose a unique set of challenges which make traditional distributed computing models difficult, if not impossible to employ in programming them. The number of devices working together to achieve a common goal will be orders of magnitude greater than those seen in distributed systems today. These systems will be heterogeneous in their hardware architectures, since each embedded system will typically be tailored to perform a specific task. Unlike the Internet, NES will typically be deployed in environments void of human attention, situations in which it is unacceptable to require a human to hit a "reset" button to recover from a failure. NES will be inherently fragile, with node and connection failures being the norm rather than the exception. The availability of nodes may vary greatly with time, with nodes becoming unreachable due to

mobility, depletion of energy resources or catastrophic failure.

Most of the nodes belonging to NES have wireless networking, thus they communicate directly only with nodes in their transmission range. In such a case, similarly to most ad hoc networks, the separation between hosts and routers disappears. Each node has to perform routing, and there is likely no common routing support for the scale and heterogeneity of NES. The applications running in NES will target specific data or properties within the network, not individual nodes. From an application point of view, nodes with the same properties are interchangeable. Fixed naming schemes, such as IP addressing, will be inappropriate in most situations. The need to target specific data or properties within the network raises the issue of a different naming scheme with dynamic bindings between a name and a node address. A naming scheme based on content or properties is more appropriate for NES than a fixed naming scheme [8].

We propose a distributed computing model, Cooperative Computing, and a system architecture for NES based on execution migration. Cooperative Computing applications consist of migratory execution units, called Smart Messages, working together to accomplish a distributed task. Smart Messages (SM) are collections of code and data that migrate through the network searching for nodes of interest and execute at each node in the path. We believe that distributed computing based on execution migration is more suitable for NES than data migration (message passing) due to volatility and dynamic binding of names to nodes specific to these networks.

Nodes in the network that support Smart Messages cooperate by providing: (1) a name-based memory (Tag Space), and (2) an architecturally independent environment (Virtual Machine) for the receipt and execution of SMs. SMs are self-routing, namely, they are responsible for determining their own paths through the network. SMs name the nodes of interest by their properties and self-route to them using other nodes as stepping stones. Applications in the Cooperative Computing model are able to adapt to the conditions encountered in the network by accepting partial results, by changing their target nodes, or by controlling the routing. SMs provide a flexible support for a wide variety of applications, ranging from data collection and dissemination to content-based routing and object tracking.

To validate the Cooperative Computing model we have developed a simulator that executes Smart Messages and allows us to evaluate both the execution and the communication time. In this simulator, we have implemented two previously proposed applications for data collection and data dissemination in sensor networks [7, 5]. The simulation results show that our model is able to provide high flexibility for user-defined distributed applications while limiting the increase in the response time to at most 15% over the traditional non-active communication implementations. We also present preliminary results for a Smart Messages prototype implementation on Compaq iPAQs over Wavelan and Bluetooth wireless networks.

The rest of this paper is organized as follows. The next section describes Cooperative Computing. Section 3 presents the system architecture that supports the model. In Section 4, we discuss the details of Smart Messages, and Sections 5 presents the SM API. Section 6 shows preliminary results for our prototype implementation. Section 7 describes the applications implemented using SMs and their simulation results are presented in Section 8. Section 9 discusses related work and Section 10 concludes the paper.

## 2. The Cooperative Computing Model

We propose a distributed computing model for large scale, ad hoc NES, called Cooperative Computing. In this model, distributed applications are defined as dynamic collections of Smart Messages (SM) that cooperate in achieving a common goal. The SM execution is described in terms of computation and migration phases. The execution performed at each step may differ based on particular properties of that node. On nodes that present interest to the current computation, the SM may read and process data. On intermediate nodes the SM executes only its routing algorithm. During migration, an SM carries its mobile data, the mobile code (when it is missing at destination), and a lightweight execution state.

Nodes in the network cooperate by providing architecturally independent programming environments for the receipt and execution of SMs (Virtual Machine) as well as a name-based memory (Tag Space). SMs along with the system support provided by nodes form the Cooperative Computing infrastructure which allows programming distributed tasks over NES.

Our model allows to program a new distributed application without a priori knowledge about the scale and the topology of the network, or the specific functionality of nodes, by injecting SMs into the network. Placing intelligence in SMs provides flexibility and obviates the issue of implementing a new application or protocol in NES, which is difficult or even impossible using current approaches [8].

SMs are resilient to network volatility. In time, cer-

tain nodes may become unavailable due to mobility or energy depletion, but SMs are able to adapt by allowing the application to control the routing. The model does not have end-to-end requirements, and therefore an SM may try to find a new route to its destination, or to discover other nodes of interest.

Moving the execution to the source of data improves the performance for certain classes of applications that need to process big amounts of data. For example, using an SM for object tracking can reduce bandwidth, energy consumption, and response time if the image analysis is performed at the node that acquired the image of the desired object instead of transferring the entire image through the network. The impact on performance of transferring code can be limited by caching code at the nodes visited by SMs during their execution.

An important issue in Cooperative Computing is security. To solve it, individual nodes should be protected against SMs, groups of nodes should be protected against malicious SMs that might consume excessive resources in the network, and SMs should be protected against nodes. Although defining a comprehensive security architecture is important, we limit the current solutions to a simple admission control and an authentication mechanism involving digital signatures.

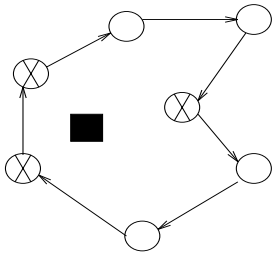

**Figure 1. Cooperative Computing Example**

Figure 1 shows an application that illustrates the novel aspects of computation and communication in Cooperative Computing. The application tracks object motion over a large area (e.g, a campus, airport, or urban highway system) using a network of mobile robots with attached cameras. In the figure, the circular nodes are robots with cameras and the square is the object of interest for an SM injected in the network to perform object tracking. The network maintains no routing infrastructure, and the SM is responsible for determining its path to the target nodes. The SM can use the direction of movement and geographical information to *chase* the objects. Once the SM arrives at a node that has a picture of a desired object (the marked circles in the figure), it generates a task to further analyze the object and its motion. The SM may migrate to neighboring nodes to obtain pictures of the object from a different angle and/or lighting condition, or to continue the tracking if the object is moving. In case of a positive identification, the SM generates a new SM that will transport the gathered information back to the node that issued the tracking request.

## 3. System Architecture

The goal of the system architecture is to keep the support required from nodes in the network to the minimum, placing intelligence in SMs rather than in individual nodes.

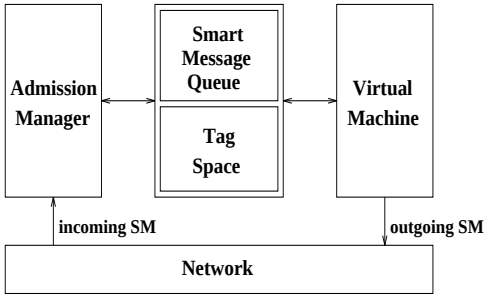


**Figure 2. System Architecture**

Figure 2 shows the common system support for Cooperative Computing provided by nodes. The Tag Space is a name-based memory region that stores data objects persistent across SM executions. The Admission Manager receives incoming messages, decides whether or not to accept them, and stores these messages into the SM Queue. The Virtual Machine (VM) acts as a hardware abstraction layer for loading, scheduling, and executing tasks generated by incoming SMs.

### 3.1. Tag Space

Each node that supports SMs manages a name-based memory region, called the Tag Space, consisting of a limited number of tags that are persistent across SM executions. The Tag Space contains two types of tags: application and I/O tags. The I/O tags define the basic hardware of the device and provide SMs with a unique interface to the local OS and I/O system. SMs are allowed to read and write both types of tags, but they can create or delete only application tags.

Figure 3 illustrates the structure of a tag. It consists of an identifier, a digital signature, lifetime information, and data. The identifier is the name of the

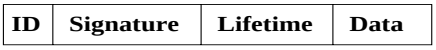


**Figure 3. Tag Structure**

tag, and it is similar to a file name in a file system. Each SM is identified by a digital signature (more details are given in Section 4). When a tag is created, the VM associates the digital signature of the current SM to this tag. The access of SMs to tags is restricted based on the digital signature. The tag lifetime specifies the time at which the tag will be reclaimed by the node from the Tag Space. The tags can be used for:

- Naming: SMs name the nodes of interest using tag identifiers.
- Data Storage: An SM can store data in the network by creating its own tags.
- Data Exchange or Data Sharing: The only way of communication among multiple SMs is by exchanging data through the Tag Space.
- Routing: SMs may create routing tags at visited nodes in the network, caching discovered route information in the data portion of these tags.
- Synchronization: An SM can block on a specific tag pending a write of the tag by another SM. Once this tag is written, all SMs blocked on it will be woken up and made ready for execution.
- Interaction with the Host System: An SM can interact with the host OS and I/O system using I/O tags.

### 3.2. Admission Manager

To prevent excessive use of its resources (energy, memory, bandwidth), a node needs to perform admission control. Each SM presents its resource requirements within a resource table. The Admission Manager is responsible for receiving incoming messages, and storing them into the SM Queue, subject to admission restrictions such as resource constraints and access rights to tags.

### 3.3. Virtual Machine

The hardware abstraction layer for the execution of SMs across different hardware platforms takes the form of a virtual machine (VM). Our current implementation uses a version of the Java virtual machine, Sun Microsystem's KVM, modified to provide the necessary functionality. An important aspect of KVM is that it was designed for constrained mobile devices and its reduced size fits well embedded systems. Using KVM allows SM programming to be done in Java, and thus enables harnessing well developed and supported Java application development tools and knowledge base.

## 4. Smart Messages

SMs are migratory execution units consisting of code and data, which migrate through network, route themselves at each node in the path and execute on nodes of interest. The SM computation is embodied in tasks. During its execution, a task may modify the data sections of the SM as well as the local tags to which it has access, may migrate, create new SMs, or may block on tags of interest. A collection of SMs cooperating towards a common goal forms a distributed application.

### 4.1. Smart Message Format

Figure 4 depicts the structure of a Smart Message. SMs are comprised of code and data sections, a lightweight execution state, and a resource table. The


**Figure 4. Smart Message Format**

SM has also a fixed size header that contains information about SM components as well as a digital signature. The digital signature identifies an SM, and it is used by nodes to protect the access to tags. The code and data sections are built from components referred to as *bricks*. Each code brick is an independent program that may be used together with the other code and data bricks to dynamically generate a new, possibly smaller SM. Code bricks are identified by statistically unique IDs computed off-line by applying a hash function on the code itself. The data bricks contain mobile data that an SM carries from node to node. The state field contains the execution context necessary for task resumption after a successful migration. The resource table consists of resource estimates: execution time, tags to be accessed or created, memory requirements, and network traffic. The resource estimates set a bound on the expected needs of the SM at a node.

### 4.2. Smart Message Life Cycle

Each SM has a well defined life cycle at a node: (1) it is subject to admission control at the destination node,

| Category | Primitives |
|---|---|
| Tag Space Operations | createTag(name, lifetime, data); deleteTag(name); readTag(name); writeTag(name, value); |
| Creation | create_SM(code_bricks, data_bricks); spawn_SM(); |
| Synchronization | block_SM(tag_name, timeout); |
| Migration | migrate_SM(tag_names, timeout); sys_migrate(next_hop); |

**Table 1. Smart Messages API**

(2) upon admission, a task is generated out of SM's code and data bricks and scheduled for execution, and (3) after completion at a node, the SM may terminate or may decide to migrate to other nodes of interest.

#### 4.2.1. Admission

To avoid unnecessary resource consumption the Admission Manager executes a three-way handshake protocol for transferring SMs between neighbor nodes. First, only the small size header and the resource table are sent to destination for admission control. If the SM admission fails, the task will be informed and the application can decide on subsequent actions.

If the SM is accepted, the Admission Manager checks, using the code bricks' IDs, whether the code bricks belonging to this SM are already cached locally. Then, it informs the source to transfer only the missing code bricks. We envision that 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 transfer cost of the code is amortized over time.

#### 4.2.2. Execution

Upon admission, an SM becomes a task which is scheduled for execution by the VM. The execution of an SM is non-preemptive, but new SMs can be admitted during execution. An executing SM can yield the VM by blocking on a tag. The VM makes sure that a task conforms to its declared resource estimates. Otherwise, the task can be forcefully removed from the system.

#### 4.2.3. Migration

If the current computation does not complete on the local node, the task may continue its execution to another node. The current execution state is captured and migrated along with the code and data bricks. Since a task accesses only mobile data and tags, we have been able to implement an efficient migration, where only a small part of the entire execution context is saved into the SM's state and transferred through the network.

### 4.3. Smart Message Self-Routing

SMs are self-routing, i.e, they are responsible for determining their own paths through the network. There is no system support required by SMs for routing, with the entire process taking place at application level. An SM names its destinations in terms of tag IDs, and executes its routing algorithm at each node in the path. SMs may create routing tags on intermediate nodes in the network to store the discovered routing information. If the routing tags are missing, an SM may spawn another SM for route discovery and block on the routing tag. A write on this tag unblocks the SM, which will resume its migration. Since tags are persistent for their lifetime, routing information, once acquired, can be used by subsequent SMs that belong to the same application, thus amortizing the route discovery effort. In this way, an SM may implement traditional routing algorithms using tags to store routing tables.

## 5. Programming Interface

The API for the Cooperative Computing model, given in Table 1, provides simple, but powerful primitives in terms of expressibility. SMs are allowed to access the Tag Space, to dynamically create new SMs, to synchronize on tags and to migrate to nodes of interest. Also, the SMs can use the uniform interface provided by the Tag Space to execute system calls on the local host (i.e, through I/O tags).

### 5.1. createTag, deleteTag, readTag, writeTag

The operations on Tag Space allow SMs to create, delete, or access existing tags. As mentioned in Section 3, the tags can be accessed subject to authentication based on digital signatures. The same interface is used to access the I/O tags: SMs can issue commands to I/O devices by writing into I/O tags, or can get I/O data by reading I/O tags.

### 5.2. create_SM and spawn_SM

An SM may use *create_SM* during the execution to assemble a new SM from a subset of its code and data bricks. An SM that needs to clone itself calls the *spawn_SM* function. Similarly to the *fork* system call, *spawn_SM* returns null in the clone, and non-null in the parent.

A new SM created by *spawn_SM* or *create_SM* is inserted into the SM Queue and will be scheduled for execution. Typically, *spawn_SM* is called when the current computation needs to migrate a copy of itself to nodes of interest while continuing the execution on the local node. A *create_SM* call is commonly used to build a new SM, for instance an SM for route discovery as a part of a routing algorithm.

### 5.3. block_SM

The update-based synchronization mechanism is implemented by the *block_SM* primitive. An SM blocks on a tag waiting for a write. If nobody writes the tag in the timeout interval, the VM returns the control to the SM. A typical example is a migrating application that creates an SM for route discovery, and blocks on a routing tag until a route is acquired.

### 5.4. migrate_SM and sys_migrate

The *migrate_SM* primitive implements a high level content-based migration, provided usually as a library function. It allows applications to name the nodes of interest by tag names and to bound the migration time. When *migrate_SM* returns normally (no timeout), the application SM reached the destination. In case of timeout, the application regains the control at one of the intermediate nodes in the path. Figure 5 presents a typical example of using *migrate_SM*. For instance, this SM can be used in the object tracking application described in Section 2. The SM migrates to nodes hosting the *tag* of interest and executes on these nodes until a certain quality of result is achieved. When this is done, the SM migrates back to the node that injected it in the network.

The *migrate_SM* function implements routing using routing tags, the low level primitive called *sys_migrate*, and possibly other SMs for route discovery. An SM can choose among multiple *migrate_SM* functions which correspond to different routing algorithms. The *sys_migrate* primitive is used to migrate SMs between neighbor nodes. The entire migration protocol of capturing the execution state and sending the SM to the next hop is implemented in *sys_migrate*.

```
1 Application_SM(tag){
2    do
3       migrate_SM(tag, timeout);
4       < do computation >
5    until(<quality of result>);
6    migrate_SM(back, timeout);
7 }
```

**Figure 5. Smart Message Example**

## 6 Prototype Implementation

This section presents very preliminary results for our SM prototype. We have implemented the SM model by modifying Sun's KVM on Compaq iPAQs running 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. We use Compaq iPAQ H3700 series, 206-MHz Intel StrongARM SA-1110 32-bit RISC Processor, 64-MB SDRAM. For wireless communication, we use Lucent Orinoco 802.11b (11 Mbs) and BrainBox BL-500 Bluetooth (1 Mbs) PCMCIA cards for communication.

Table 2 gives the time taken for Tag Space operations. *createTag* is the most expensive operation, as it involves adding a tag to the Tag Space and registering a timeout for the tag. *readTag* and *writeTag* primitives involve checking for tag expiration and accessing the tag. Their costs will affect the execution time to a greater extent, since they will be called more frequently than *createTag* or *deleteTag*. The time measured for *block_SM* is the time taken by the VM to block the SM.

| Tag Space Operation | Time ($\mu$s) |
|---|---|
| createTag | 55.8 |
| deleteTag | 30.8 |
| readTag | 25.0 |
| writeTag | 28.0 |
| block_SM | 24.6 |

**Table 2. Time for Tag Space operations**

Figure 6 gives an example of a benchmark SM which creates a tag, *waitTag*, another SM, *rttSM*, and then blocks itself on this tag. The *rttSM* migrates to a neighbor, and then migrates back to the source and writes *waitTag*, which results in waking up the original SM. The *rttSM* has one code brick of 731 bytes, one data

```
 1  mobileData{
 2    netAddress srcAddr;
 3  }
 4  simpleSM(){
 5    createTag(waitTag, 100, null);
 6    create_SM(rttSM, mobileData);
 7    block_SM(waitTag, 100);
 8  }
 9  rttSM(){
10    srcAddr = getAddress();
11    sys_migrate(any_neighbor);
12    sys_migrate(srcAddr);
13    writeTag(waitTag, null);
14  }
```

**Figure 6. Smart Message Benchmark**

brick of 20 bytes and a stack of 78 bytes. We measured the average time required for a round-trip communication and execution by estimating the time elapsed between the original SM's *block_SM* call and its resumption of execution. This is 55.2 ms using Wavelan, and 452.8 ms using Bluetooth.

## 7. Applications

To prove that virtually any protocol or application can be written using SMs, we have implemented two previously proposed applications: Directed Diffusion [7] and SPIN [5]. They present different paradigms for content-based communication and computation in sensor networks: Directed Diffusion implements data collection, and SPIN is a protocol for data dissemination.

### 7.1. Directed Diffusion using SMs

In Directed Diffusion a sink node requests data by sending "interests" for named data. Data matching an interest is then "drawn" from source nodes towards the sink node. Intermediate nodes can cache or transform data, and may direct interests based on previous cached data. At the beginning the node may receive data from multiple paths, but after a while it will reinforce the path providing the best data rate. All future data will arrive on the reinforced path only.

For the implementation of Directed Diffusion using SMs, the Tag Space of each node will host three tags: the first stores the most recent data value (*tag_data*), the second stores the best data rate available at that node (*tag_data_rate*), and the third stores the best next hop towards the source (*tag_best_route*). Directed Diffusion is initiated by injecting the SM at the sink, and it has two main phases: (1) *exploration* starts at the sink and floods the network to find data of interest, and (2) *reinforcement* chooses the best path and brings data from source to sink.

If the information of interest is not locally available (no *tag_data_rate* value), the *explore SM* spawns itself, and the "child" SM migrates to all neighbors, while the "parent" SM blocks on the *tag_data_rate*. This operation is performed recursively at every node until an SM reaches a node containing the *tag_data_rate* tag. At this point, the "child" SM migrates back to its parent carrying the discovered data rate. If the new data rate is better than the value stored in *tag_data_rate*, the SM will update *tag_data_rate* with the new value and *tag_best_route* with its source as the best node in the path to the source of data. This update unblocks the "parent" SM which, at its turn, carries the data rate one hop back. Eventually, the sink node is reached and the reinforcement phase begins.

During the reinforcement phase, a *collect SM* migrates to the best next hop starting from the sink. At each intermediate node, this SM spawns, the "child" SM migrates to the best next hop, while the "parent" SM blocks waiting for data. When the SM reaches the source, it spawns new SMs to carry the data one hop back, at the promised data rate. Recursively, a blocked SM will be awaken by the data arrival and, at its turn, will carry the data back until it reaches the sink.

### 7.2. SPIN using SMs

SPIN is a family of adaptive protocols that disseminates information among nodes in a sensor network. We present an implementation of SPIN-1 which is a three-stage handshake protocol for data dissemination. Whenever a node obtains new data, it disseminates this data in the network by sending an advertisement to its neighbors. The node receiving the advertisement checks to see if it has already received or requested that data. If not, it sends a request message back to the sender, asking for the advertised data. The initiator sends the requested data and then the process is executed recursively for the entire network.

For the implementation of SPIN using SMs, the Tag Space will host two tags: the value of the most recent data received (*tag_data*), and the timestamp associated with this data (*tag_timestamp*).

The protocol is initiated by injecting a *disseminate SM* into a node that produces the data. This SM blocks on *tag_data* waiting for new data (to be locally produced or to arrive). After an update is performed, the task spawns itself and the "child" migrates to the neighbors to advertise the new data. If the SM ad-

vertises new data at the arrival node, it creates a new SM to fetch the data, updates the *tag_timestamp*, and blocks on *tag_data* waiting for this data. Upon data arrival, the "parent" SM is woken up, recursively spawns itself, and the "child" SM migrates to its neighbors [1].

## 8. Simulation Results

For experiments we use an event-driven simulator, similar to ns-2 [11], extended with support for SM execution. To get accurate results, both the communication and the execution time have to be accounted for. The simulator is written in Java to allow rapid prototyping of applications. The simulator provides accurate measurements of the execution time by counting, at the VM level, the number of cycles per VM instruction. To correctly account for the execution time, we have simulated each node with a Java thread, and we have implemented a new mechanism for scheduling these threads inside JVM.

The communication model used in our simulator is "generic wireless", with contention solved at the message level. The nodes can communicate within an area limited by their transmission range. Before any transmission, a node "senses" the medium and backs-off if somebody else is sending.

We use the same network configuration for all experiments. The network has 256 nodes distributed uniformly over a square area, and each node has the same transmission range. The average number of neighbors per node is 4. We define the data convergence time as the time when a certain percentage of the nodes have received the data (SPIN), or the data rate (Directed Diffusion). In both cases, due to flooding, all nodes will end up receiving the data, respectively the data rate. SPIN completes after all nodes have received the data, while Directed Diffusion will start the reinforcement phase.

Figure 7 presents the data convergence time for a single Directed Diffusion SM, with the sink and source located at the diagonal corners of the square region. We plot the data convergence time for 3 different cases of the same SM plus a base case for the same application using passive communication (no SM). The top curve shows the time when code caching is not used. In the second curve, we can see an improvement of more than 4 times in performance when code caching is used during the first execution of the SM in the network. The code is cached when an SM visits a node for the first time, and it will be used by subsequent SMs during the same execution. The effects of caching are significant because the SMs visit a node multiple times in Directed Diffusion: they travel the network both forward (looking for the source) and backward (diffusion of data rate). In the third curve we can observe a 30% decrease of the completion time when the code is already cached at all nodes. The fourth curve shows data convergence time for the traditional implementation: the protocol is implemented at each node, only data is transferred through the network and the execution time is not counted. The degradation of performance is only 5%. We believe that this is a reasonable price for the flexibility to program any user-defined distributed application in NES.

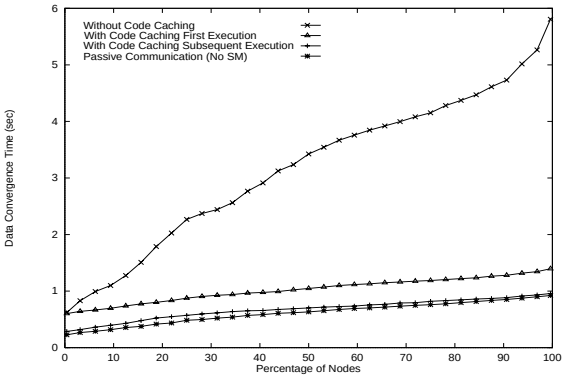


**Figure 7. Directed Diffusion using SM**

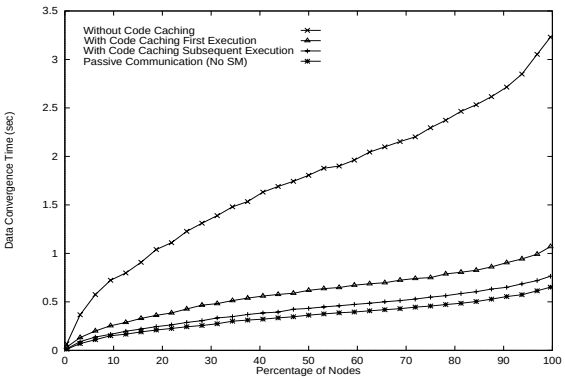


**Figure 8. SPIN using SM**

Figure 8 plots the same curves for a single SPIN SM launched in the network at a node located in a corner of the square area. During the first execution, flooding and the three-stage design of the protocol lead to a 3 times improvement in performance when code caching is used. The third curve shows a 30% decrease of the completion time (similar to Directed Diffusion)

[1] The SM code for Directed Diffusion and SPIN is provided in the companion Rutgers University Technical Report DCS-TR-464

when the code is already cached at all nodes. The completion time increases from 10% to 15% compared to the traditional implementation.

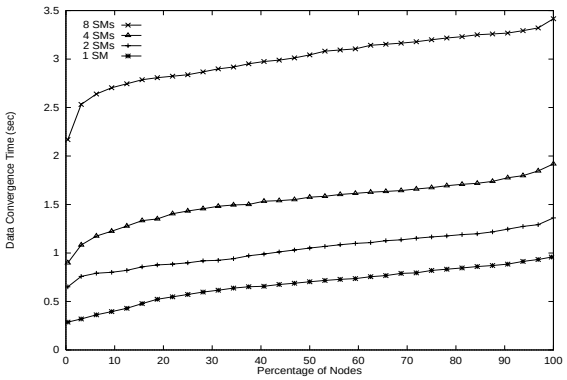


**Figure 9. Directed Diffusion - Multiple SMs**

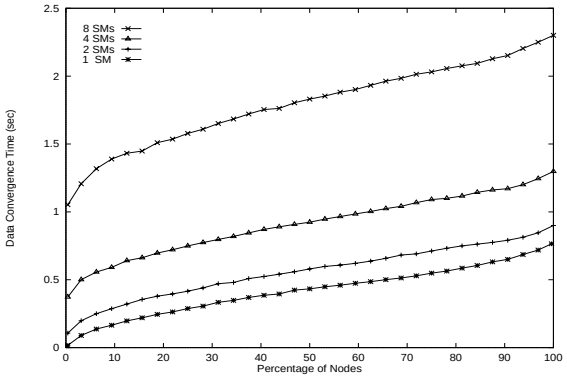


**Figure 10. SPIN - Multiple SMs**

Figures 9 and 10 show the data convergence time for both Directed Diffusion and SPIN, when multiple SMs run simultaneously through the network and the code is already cached at nodes. For these experiments data convergence time is the time when a certain percentage of nodes have received the data or the data rate for all the SMs running in parallel. The nodes at which the SMs start are distributed uniformly in the network. In all cases SPIN completes faster (i.e, 2.3s compared to 3.4s for the top curves in the figures) because it floods only the neighbors and then brings the data to them, while Directed Diffusion needs to flood the entire network until it finds the source and then the diffusion brings the data rate back to all nodes. In the initial phase Directed Diffusion generates more messages in the network leading to a higher contention, but its performance will increase as soon as the reinforcement phase begins.

## 9. Related Work

Smart Messages bear some similarity to Active Messages [15], active networks [3, 14, 13], and mobile agents [4, 12]. Although SMs borrow implementation solutions from all of them, the concept is significantly different in its goal (to support distributed computing in NES).

Like Active Messages [15], the arrival of an SM at a node leads to the execution of a task on the node. However, while Active Messages point to a handler at the destination, SMs carry code with them. Beyond the superficial similarity between the Smart Messages and Active Messages, the two models address two completely different problems. Active Messages target fast communication in system-area networks and therefore, the handler execution is short and triggered as soon as the active message arrives. On the other hand, SMs target remote programmability of massive networks of embedded devices.

The ANTS [3] capsule model of programmability allows forwarding code to be carried and safely executed inside the network by a Java VM. A first difference is that this model does not migrate the execution state from node to node. It just caches and transfers code that always starts and finishes on the same node. A second difference is that ANTS targets IP networks, while SMs does not require any routing support. The main difference in terms of programmability is that, unlike ANTS, SMs define a distributed computing model where applications cooperate and synchronize each other.

The Smart Packets [14] architecture provides a flexible means of network management through the use of mobile code. Smart Packets are implemented over IP, using the IP options header. They are routed just like other data traffic in the network and only execute on arrival at a specific location. Unlike Smart Packets, SMs are executed at each hop in the network and their execution determines the next hop in the route. Also, SMs migrate the execution context.

Programmable Packets [13] are centered around a low-level packet language that adds flexibility over IP. We share some of the design goals, like safety and flexibility, that allow for in-network processing of application specific code, but SMs define a general distributed computing model that provides more expressibility for user-defined applications.

Smart Messages are similar to mobile agents, which also use migration of code in the network. A mobile agent may be viewed as a task that explicitly migrates from node to node assuming that the underlying network assures its transport between them. Unlike mo-

bile agents, SMs are defined to be responsible for their own routing in a network. The SM architecture further defines the infrastructure that nodes in a network supporting SMs must implement, which makes self-routing of SMs possible.

Research in mobile ad hoc networking [9, 2, 10] has resulted in numerous routing protocols for peer-to-peer multi-hop networking in infrastructures without base stations. These protocols have generally been designed for IP-based networks, and have primarily targeted traditional mobile computing applications such as mobile personal computers and PDAs. These protocols can be leveraged and implemented over the SM architecture.

Recent work on large networks of embedded systems has focused on network protocols for sensor networks [7, 5], and system architectures for fixed-function sensor networks [6]. This research is complementary to the SM architecture. We prove in this paper that our model provides enough flexibility to enable the implementation of these models over the SM architecture.

## 10. Conclusions

This paper has described a programming model for large scale distributed embedded systems, in which distributed applications are implemented as collections of Smart Messages. The model overcomes the scale, heterogeneity, and connectivity issues by placing the intelligence in migratory execution units. The nodes in the network cooperate by providing a common minimal system support for the receipt and execution of Smart Messages. The implementation of two applications for sensor networks shows that Cooperative Computing represents a flexible, yet simple solution for programming large networks of embedded systems.

## Acknowledgments

The authors would like to thank Philip Stanley-Marbell for his participation in the initial design of Smart Messages, and Ulrich Kremer for his contribution to this work.

## References

[1] Sensoria corporation. http://www.sensoria.com.

[2] Charles E. Perkins, Elizabeth Royer and Samir R. Das. Ad hoc on demand Distance Vector(AODV)routing. In *2nd IEEE Workshop on Mobile Computing Systems and Applications* , pages 90–100, February 1999.

[3] David Wetheral. Active network vision reality: lessons from a capsule-based system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 64–79, 1999.

[4] 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.

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

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, November 2000.

[7] C. Intanagonwiwat, R. Govindan, and D. Estrin. 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*, pages 56–67, 2000.

[8] John Heideman, Fabio Silva, Chalermek Intanagonwiwat,Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *18th ACM Symposium on Operating Systems Principles*, pages 146–159, October 2001.

[9] D. B. Johnson and D. A. Maltz. *Dynamic Source Routing in Ad Hoc Wireless Networks*. T. Imielinski and H. Korth, (Eds.). Kluwer Academic Publishers, 1996.

[10] J. Li, J. Jannotti, D. S. J. D. Couto, D. R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking*, pages 120–130, Boston, Massachusetts, August 2000.

[11] S. McCanne and S. Floyd. ns Network Simulator. http://www.isi.edu/nsnam/ns/.

[12] D. Milojicic, W. LaForge, and D. Chauhan. Mobile objects and agents. In *USENIX Conference on Object-oriented Technologies and Systems*, pages 1–14, 1998.

[13] J. T. Moore, M. Hicks, and S. Nettles. Practical programmable packets. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies*, April 2001.

[14] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart Packets for Active Networks. *ACM Transactions on Computer Systems*, pages 397–413, February 2000.

[15] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer architecture*, pages 256–266, May 1992.