IRACST – International Journal of Computer Networks and Wireless Communications (IJCNWC), ISSN: 2250-3501
Vol. 2, No. 1, 2012

The Feasibility of Co-Operative Computing in Sensor Networks

Dr. M. Usha Rani[1], K. Sailaja[2], [1]Associate Professor; [2]Research Scholar, Dept. of Computer Science,
SPMVV, Tirupati. -517502, AP, INDIA.

Abstract : As the cost of embedding computing becomes negligible compared to the actual cost of goods, a trend toward incorporating computing and wireless communication capabilities in most of the consumer products occurs. Therefore, the next generation of computing systems will be embedded, in a virtually unbounded number, and dynamically connected. The first illustration of these systems that has received considerable interest in the last couple of years is sensor networks. During the next decade sensor networks will become part of a larger class of networks of embedded systems (NES) that have sufficient computing, communication, and energy resources to support distributed applications. In this paper, I presented distributed computing model, cooperative computing, and a software architecture for NES based on execution migration. The challenge is how to program NES, namely, to determine the appropriate computing model and the system support necessary to execute distributed applications in these networks. Cooperative computing provides flexible support for a wide variety of applications, ranging from data collection and dissemination to content based routing and object tracking. For larger scale evaluation, a simulator was developed that executes SMs and allows one to account for execution as well as communication time. In this simulator, i implemented two applications for data collection and data dissemination in sensor networks. They are directed diffusion and SPIN . The simulation results show that this model is able to provide high flexibility for user-defined distributed applications while limiting the increase in response time to, at most, 15% over traditional non active communication implementations.

Keywords:- Sensor networks, NES, SPIN, Cooperative computing, Distributed computing.

I. INTRODUCTION

Due to advances in wireless communications and electronics over the last few years, the development of networks of low-cost, low-power, multifunctional sensors has received increasing attention. These sensors are small in size and able to sense, process data, and communicate with each other, typically over an RF (radio frequency) channel. A sensor network is designed to detect events or phenomena, collect and process data, and transmit sensed information to interested users. Basic features of sensor networks are:

- Self-organizing capabilities
- Short-range broadcast communication and multihop routing
- Dense deployment and cooperative effort of sensor nodes
- Frequently changing topology due to fading and node failures
- Limitations in energy , transmit power, memory, and computing power.

These characteristics, particularly the last three, make sensor networks different from other wireless ad hoc or mesh networks.

The nodes in NES communicate through wireless network interfaces. Thus, they can communicate directly only with nodes within their transmission range. Similarly to most ad hoc networks, the separation between hosts and routers disappears (i.e., each node must perform routing). However, the scale and heterogeneity encountered in NES as well as different application requirements preclude the existence of a common routing support. Therefore, the flexibility to use multiple routing algorithms in the same network is desirable.

In cooperative computing, the applications consist of migratory execution units, called smart messages (SMs), working together to accomplish a distributed task. Distributed computing based on execution migration is more suitable for NES than data migration (message passing) due to the volatility and dynamic binding of names to nodes specific to these networks.

Nodes in the network support SMs by providing: a name-based shared memory (tag space) for inter- SM communication and access to the host system; and an architecturally independent environment (virtual machine) for SM execution. SMs are self-routing, namely, they are responsible for determining their own paths through the network. SMs name the nodes of interest by properties and self-route to them using other nodes as "stepping stones." Applications in cooperative computing are able to adapt to adverse network conditions by changing their routing dynamically.

II. THE FEASIBILITY OF COMPUTING MODEL

Feasible computing is a distributed computing model for large-scale, ad hoc NES. In this model, distributed applications are defined as dynamic collections of migratory execution units, called SMs, that cooperate in achieving a common goal. The execution of an SM is described in terms of computation and migration phases. The execution performed at each step is determined by the 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 migrations, SMs carry mobile data, the code missing at destination, and a lightweight execution state.

Nodes in the network cooperate by providing an architecturally independent programming environment (virtual machine) for SM execution and a name-based shared memory (tag space) for inter-SM communication and interaction with the host system. SMs, along with the system support provided by nodes, form the cooperative computing infrastructure, which allows programming user-defined distributed applications in NES.

In this model, a new distributed application can be developed without *a priori* knowledge about the scale and topology of the network or the specific functionality of each node. Placing intelligence in SMs provides this flexibility and also obviates the issue of implementing a new application or protocol in NES, which is difficult or even impossible using conventional approaches[10].

Moving the execution to the source of data improves the performance for applications that need to process large amounts of data. For example, instead of transferring large size images through the network for an object tracking application, an SM can perform the analysis of the images at the nodes that acquired them. Thus, it reduces the network bandwidth and energy consumption, and in the same time, it improves the user-perceived response time. The impact of transferring code on performance can be limited by caching code at the nodes.

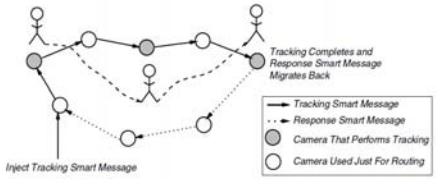

Fig 1. Distributed object tracking using cooperative computing.

Fig.1. shows a simple application that illustrates the novel aspects of computation and communication in cooperative computing. The application performs object tracking over a large area (e.g., a campus, airport, or urban highway system) using a network of mobile robots with attached cameras[17} .

III. COOPERATIVE NODE ARCHITECTURE

The goal of the SM software architecture is to keep the support required from nodes in the network to the minimum, placing intelligence in SMs rather than in individual nodes. Fig.1 shows the common system support provided by nodes for cooperative computing. The admission manager receives incoming SMs, decides whether to accept them, and stores these messages into the SM-ready queue. The code cache stores frequently used codes to reduce the amount of traffic in the network. The virtual machine (VM) acts as a hardware abstraction layer for scheduling and executing tasks generated by incoming SMs. The tag space is a name-based shared memory that stores data objects persistent across SM executions and offers a unique interface to the host's OS and I/O system.

A. 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 in the SM ready queue, subject to admission restrictions.

B. Code Cache

Commonly, the applications executing in NES have a localized behaviour, exhibiting spatial and temporal locality. Therefore, frequently used SM codes are cached in order to amortize over time the initial cost of transferring the code through the network.

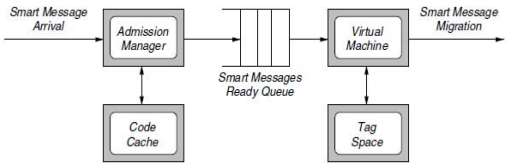

Fig.2. Cooperative node Architecture

B. Virtual Machine

The VM schedules, executes, and migrates SMs. To migrate an SM, the VM captures the execution state and sends it along with the code and data to the next hop. The VM at the destination will resume the SM from the instruction following the migration invocation. The VM also ensures that an SM conforms to its declared resource estimates; otherwise, the SM can be removed from the system.

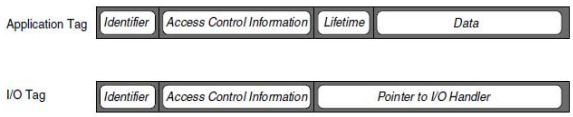

Fig. 3. Structure of application and I/O tags.

C. Tag Space

Each node that supports SMs manages a name-based shared memory, called tag space, consisting of tags that are persistent across SM executions. The tag space contains two types of tags: application tags, which are created by SMs, and I/O tags that are provided by the system. The I/O tags define the basic hardware of the node 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.

Fig.3. illustrates the structure of application and I/O tags. The identifier is the name of the tag and is similar to a file name in a file system; it is used by SMs for content-based node naming. For application tags, the VM associates the access control information carried by the SM that created the tag (i.e., the owner of the tag). For I/O tags, the owner of the device sets the access control information.

Application tags and I/O tags differ in terms of functionality and lifetime. Application tags offer persistent memory for a limited lifetime (i.e., application tags are still "alive" for a certain amount of time after the SMs that created them have finished the execution at the local node); after this time interval, the tags expire, and the node reclaims their memory. I/O tags, on the other hand, are permanent and provide a pointer to an I/O handler (i.e., a system call or an external process) capable of serving I/O requests. The list of all the possible utilizations of tags consists of:

• *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 and data sharing*
. Exchanging data through the tag space is the only communication channel among different SMs.

• *Routing*
SMs may create routing tags at visited nodes to store routing information in the data portion of these tags.

• *Synchronization*
An SM can block on a specific tag pending a write of this tag by another SM. Once the tag is written, all SMs blocked on it are waked up and made ready for execution.

• *Interaction with the host system*

An SM can issue commands to or request data from the host OS and I/O devices using I/O tags.

IV. SMART MESSAGES

SMs are execution units that migrate through the network to execute on nodes of interest and route themselves at each node in the path toward a node of interest. SMs comprise code and data sections (referred to as "bricks"), a lightweight execution state,

and a resource table. The code and data bricks can be dynamically used to assemble new, possibly smaller SMs. The ability to incorporate only the necessary code and data bricks in the new SMs can reduce their size and, consequently, the amount of traffic in the network (i.e., the code and data carried by SMs are divided into bricks solely for this purpose). The execution state contains the execution context necessary to resume the SM after a successful migration. The resource table consists of resource estimates: execution time, tags to be accessed or created, memory requirements, and network bandwidth. These resource estimates set a bound on the expected needs of an SM at a node; they are used by the admission manager to make the admission decision.

A collection of SMs cooperating toward a common goal forms a distributed application.

A. Smart Message Life Cycle

Each SM has a well-defined life cycle at a node: (1) it is subject to admission control; (2) upon admission, a task is generated out of the 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.

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 resource table is sent to the destination for admission control. If the SM admission fails, the task will be informed, and it can decide on subsequent actions. If the SM is accepted, the admission manager checks, using the code bricks' Ids (computed off-line by applying a hash function on the code), whether the code bricks belonging to this SM are cached locally. Then, it informs the source to transfer only the missing code bricks.

2) . Scheduling and Execution

Upon admission, an SM becomes a task scheduled (in FIFO order) for execution. The execution is non pre emptive; new SMs can be accepted, but they will not be dispatched for execution until the current SM terminates. An executing SM can yield the VM, however, by blocking on a tag. The execution time is bounded by the estimated running time presented during admission (i.e., the VM may terminate an SM that does not respect the admission contract).

Non pre-emptive scheduling is used for three reasons. First, the execution time of SMs is usually short (many times a node is used merely as a "stepping stone" en route to a node of interest). Thus, context switching would incur too much overhead with respect to the total execution time of the SM. Second, it is not necessary to support multiprogramming for interactive programs (unlike traditional computer systems, embedded systems commonly operate unattended). Third, the communication always terminates the current SM (i.e., the only form of communication in cooperative computing is a migration invocation) and, consequently, the idea of using multiple threads in one application to overlap communication and computation does not make sense for SM programs. On the other hand, non pre emptive scheduling makes inter-SM synchronization and sharing particularly simple to implement.

3). Migration

If the current computation does not complete at the local node, the task may continue its execution at another node. The current execution state is captured and migrated along with the code and data bricks. Because a task accesses only mobile data and tags, an efficient migration has been implemented in which only a small part of the entire execution context is saved and transferred through the network. Essentially, the instruction and stack pointers are transferred for all the stack frames corresponding to the current task. It is important to notice that migration is explicit (i.e., the programmers call a "migration" primitive when needed), and that data transferred during a migration are specified by the programmer as data bricks.

4). Smart Message Self-Routing

SMs are self-routing, i.e., they are responsible for determining their own paths through the network. SMs require no system support for routing; the entire process takes place at application level. An SM names its destinations in terms of tag identifiers and executes its routing algorithm at each node in the path. SMs may create routing tags at intermediate nodes in the network to store routing information. If routing information is not locally available, an SM may create other SMs for route discovery and block on a routing tag. A write on this tag unblocks the SM, which will resume its migration. Because tags are persistent for their lifetime, the routing information, once acquired, can be used by subsequent SMs, thus amortizing the route discovery effort.

Each SM must include at least one *routing brick* among its code bricks. A single routing algorithm, however, might not always reach a node of interest in the presence of highly dynamic network configurations. Therefore, an SM can carry multiple routing algorithms and change them during execution according to the current network conditions. For instance, an SM can use a proactive routing algorithm in a stable and relatively dense network and an on-demand algorithm in a volatile and sparse network. In this way, the SM may complete even if network conditions change significantly during its execution. Borcea and colleagues offer a complete description of the self-routing mechanism.

V. SMART MESSAGES API.

The API for the cooperative computing model is given in Table.1. It provides simple, yet powerful, primitives. SMs can access the tag space, dynamically create new SMs, synchronize on tags, and migrate to nodes of interest.

createTag, deleteTag, readTag, and writeTag. These operations allow SMs to create, delete, or access existing tags. As mentioned in section.3. these operations are subject to access control. 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.

createSMFromFiles, createSM, and spawnSM. An SM is created by injecting a program file at a node; this program calls createSMFromFiles with a list of program file names to build the new SM structure. An SM may use createSM during execution to assemble a new SM from a subset of its code and data bricks. A createSM call is commonly used to create a route discovery SM when routing information is not locally available. An SM that needs to clone itself calls spawnSM; this primitive returns true in the "parent" and false in the "child" SMs.

blockSM. This primitive implements the update-based synchronization mechanism. An SM blocks on a tag waiting for a write. To prevent deadlocks, blockSM takes a timeout as parameter. If nobody writes the tag in the timeout interval, the VM returns the control to the SM. A typical example is an SM that blocks on a routing tag while waiting for a route discovery SM to bring a new route.

Table 1. Cooperative Computing API

Category	Primitives
Tag space operations	createTag(tag_name, lifetime, data); deleteTag(tag_name); readTag(tag_name); writeTag(tag_name, value);
SM creation	createSMFromFiles(program_files); createSM(code_bricks, data_bricks); spawnSM();
SM synchronization	blockSM(tag_name, timeout);
SM migration	migrateSM(tag_names, timeout); sys_migrate(next_hop);

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

Fig .4. Code skeleton for typical smart message.

migrateSM and sys_migrate. The migrateSM 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 migrateSM returns normally (no timeout), the SM is guaranteed to resume its execution at a node of interest. In case of timeout, the SM regains control at one of the intermediate nodes in the path.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 into the network. The migrateSM 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 migrates functions that correspond to different routing algorithms. The sys_migrate primitive is used to migrate SMs between neighbour nodes. The entire migration protocol of capturing the execution state and sending the SM to the next hop is implemented in sys_migrate.

VI. PROTOTYPE IMPLEMENTATION AND EVALUATION

The authors have implemented their SM prototype in Java over Linux, thus harnessing well-developed and supported Java application development tools and knowledge base. Specifically, Sun Micro system's KVM (Kilobyte Virtual Machine)[1] has been modified because it has a small memory footprint (i.e., as little as 160 KB, which makes it suitable for resource-constrained devices) and its source code is publicly available.

The SM API is encapsulated in two Java classes: *SmartMessage and TagSpace* for efficiency, the API was implemented as Java native methods. The authors have also implemented their own serialization mechanism because KVM does not support serialization. In addition to the KVM interpreter thread, two additional threads have been introduced for admission control and local code injection. The design of the SM computing platform is not specific to any hardware or software environment. It can be implemented on any VM language, or underlying operating system.

Next, micro benchmark results for this SM prototype are reported. Specifically, the cost of one-hop migration and the cost of tag space operations have been measured. The test bed consists of HP iPAQs 3870 running Linux 2.4.18-rmk3-hh24. Each iPAQ contains an Intel StrongARM 1110 206-Mhz RISC processor, 32-MB flash memory, and 64-MB RAM memory. For communication, Lucent Orinoco 802.11b Silver PC Cards are used in ad hoc mode. To factor out the cost of Java method call overhead , the code for measuring costs has been inserted inside the native methods associated with the SM API.

A. *Cost of SM Migration*

The one-hop migration has three phases: execution capture at source, SM transfer, and execution resumption at destination. The SM is converted into a machine-independent representation to allow state capture and resumption. Because the code bricks are already in machine-independent Java class format, only the data bricks and execution state need to be converted. This conversion is done using the authors' simple object serialization mechanism. The serialization of the execution state does not have a significant impact because only the execution control state is captured and transferred, not the local variables. Therefore, the important factors that determine the cost of one-hop migration are the data brick serialization, the SM transfer, and data brick deserialization.

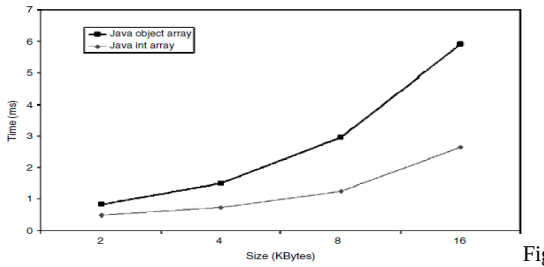

Fig.5. Cost of data brick serialization.

1). Data Brick Serialization and Deserialization.

To study the effect of data brick serialization, a fixed-size code brick (1197 bytes) has been used and the data brick size has been varied from 2 to 16 KB. The stack frames have also been kept constant (131 bytes for two activation records). The cost of serializing these two stack frames is 0.235 ms. Commonly, the data bricks in an SM consist of a mixture of objects and primitive types. Two types of data bricks have been used in this evaluation; they represent a practical lower and upper bound for typical data bricks: an array of integers and an array of objects. The object array represents an upper bound because each of its elements causes a call to the top-level VM serialization method, while the integer array represents a lower bound because there is only one call to the top level VM serialization method.

Figure.5 shows that the serialization cost is below 6 ms for data bricks as large as 16 KB. Commonly, the SMs process the data at source and therefore they carry small size data. The applications developed by the authors carry less than 2 KB, which costs less than 1 ms to serialize. Fig.6 presents the cost of deserialization for the same data bricks. Observe that this cost is as much as 30% larger than the cost of serialization — an increase caused by the memory allocation costs during object deserializations.

2). SM Transfer

To evaluate the total cost of migrating an SM (serialization, transfer, deserialization), two sets of experiments were performed. In the first, the code brick size was varied while data brick size and stack frame size were kept fixed at 53 and 131 bytes, respectively. In the second, data brick size was varied while keeping the code brick size and stack frame size fixed at 1197 and 131 bytes, respectively.

Fig.6 and Fig.7 show the results of these two experiments for two cases: when the code is not cached and when the code is cached. In Fig.7, the time to transfer the SM when the code is

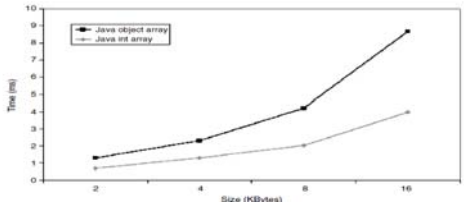

Fig. 6. Cost of data brick deserialization.

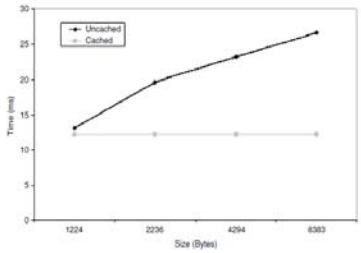

Fig.7. Effect of code brick size on single- hop migration.

cached represents, essentially, the overhead of the three-way handshake protocol because the sizes of the data bricks and stack frames are small. Fig.7 demonstrates that the data brick size contributes significantly to the total cost of migration. Thus, it is important to have a serialization mechanism with minimal space overhead.

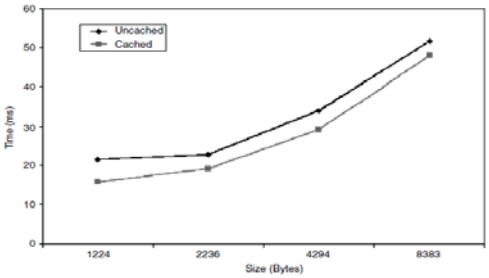

Fig.8. Effect of code brick size on single- hop migration.

B. Cost of Tag Space Operations.

Table 26.2 shows the cost of the tag space operations for application tags. The ***readTag*** primitive has the lowest cost because it performs the least number of operations. When an SM reads a tag, the VM interpreter acquires a lock, performs a lookup in the tag space, and returns the data to the SM. The ***writeTag*** operation costs are slightly higher because the interpreter must check and unblock any SMs blocked on the tag. The ***createTag*** primitive involves an additional step to register a timer for the tag lifetime, while ***blockSM*** needs to append the SM to the queue and suspend the current task. The ***deleteTag*** primitive has the highest cost because the interpreter needs to wake up all SMs blocked on the tag, remove the timer for the tag lifetime, and remove the tag structure from the tag space Table.3 presents the access time to several I/O tags that are currently implemented in our prototype: GPS location query; neighbor discovery; camera image capture; light sensor; and system status inquiry (battery lifetime, system time, and amount of free memory). The *gps_location* is updated by a user-level process that reads from the GPS serial interface. The location of the neighbours along with their identifiers can be returned by reading the ***neighbour_list*** tag, which is typically used by geographical routing algorithms carried and executed by SMs. To get the information about neighbour nodes, a neighbour discovery protocol has been implemented that maintains a cache of known neighbours. For the ***image_capture*** tag, the system also performs YUYV to RGB format conversion on the captured image before returning it to the tag reader. All the other tag values are obtained directly from Linux using system calls.

TABLE .2 Time for Tag Space Operations

Tag Space Operation	Time (μs)
createTag	43.4
deleteTag	55.9
readTag	20.8
writeTag	31.7
blockSM	45.8

TABLE .3 Cost of Reading I/O Tags

Tag Name	Time (ms)
gps_location	0.20
neighbor_list	0.34
image_capture (32-KB)	341.23
light_sensor	0.11
battery_lifetime	25.63
system_time	0.09
free_memory	0.12

Fig. 9 Prototype node with video camera and GPS receiver attached.

VII. APPLICATIONS

To prove that virtually any protocol or application can be written using SMs, two previously proposed applications — SPIN and directed diffusion[6] have been implemented. They present different paradigms for content-based communication and computation in sensor networks; SPIN is a protocol for data dissemination and directed diffusion implements data collection.

A. SPIN Using Smart Messages

SPIN is a family of adaptive protocols that disseminates information among nodes in a sensor network. The implementation of SPIN-1 is a three-stage handshake protocol for data dissemination. Each time a node obtains new data, it disseminates them in the network by sending an advertisement to its neighbours.

B. Directed Diffusion Using Smart Messages

In directed diffusion, a sink node requests data by sending "interests" for named data. Data matching an interest are then drawn from source nodes toward the sink node. Intermediate nodes can cache and aggregate data; they may also direct interests based on previously cached data. At the beginning, the sink 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.

```
1  DisseminateSM(String tag, int timeout){
2     int timestamp;
3     Data data;
4     String tagData=tag+"data";
5     String tagTimestamp=tag+"timestamp";
6     Address src, dest;
7     while(true){ // SM at source
8        TagSpace.blockSM(TagData, timeout);
9        timestamp = TagSpace.readTag(tagTimestamp);
10       if (!SmartMessage.spawnSM()){ // child SM
11          while(true){ // SM at every node
12             src = SmartMessage.getLocalAddress();
13             SmartMessage.sys_migrate(all); // migrate to all neighbors
14             if (timestamp.CompareTo((Integer)TagSpace.readTag(tagTimestamp))<=0){
15                System.exit(0); // the same or more recent data exists at this node
16             }
17             TagSpace.writeTag(tagTimestamp, timestamp);
18             dest = SmartMessage.getLocalAddress();
19             SmartMessage.sys_migrate(src); // migrate back to source
20             data = TagSpace.readTag(tagData);
21             SmartMessage.sys_migrate(dest); // bring data to destination
22             TagSpace.writeTag(tagData, data);
23          }
24       }
25    }
26 }
```

Fig. 10 SPIN with smart messages.

For the implementation of directed diffusion using SMs, the tag space at each node hosts three tags: the most recent data value (*tagData*); the best data rate available at that node (*tagDataRate*); and the best next hop toward the source (*tagBestRoute*). Directed diffusion is initiated by injecting an SM at the sink. The execution of this SM 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 tagDataRate value), the *explore SM* spawns itself; the child SM migrates to all neighbors, while the parent SM blocks on tagDataRate. This operation is performed recursively at every node until an SM reaches a node containing the tagDataRate. 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 tagDataRate, the SM updates tagDataRate with the new value and tagBestRoute with its source as the best node in the path toward the source of data. This update unblocks the parent SM, which will carry 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 is awakened by the data arrival, and it will carry the data back until it reaches the sink.

VIII. SIMULATION RESULTS

For large-scale evaluation, the authors have developed an event-driven simulator, similar to ns-2, extended with support for SM execution. The simulator is written in Java to allow rapid prototyping of applications. To get accurate results, the communication and the execution times must be accounted for. The simulator provides accurate measurements of the execution time by counting, at the VM level, the number of cycles per VM instruction. To account for the execution time, each node has been simulated with a Java thread, and a new mechanism has been implemented for scheduling these threads inside JVM. The communication model used in this simulator is "generic wireless," with contention solved at the message level. Before any transmission, a node "senses" the medium and backs off in case of contention.

The main goal in conducting the simulation experiments was to quantify the data convergence time for the authors' implementations of SPIN and directed diffusion using SMs and to compare these results with those for traditional message-passing implementations. Data convergence time is defined as the time when a certain percentage of the total number of nodes has received the data (SPIN), or the data rate (directed diffusion). In both cases, due to flooding, all nodes end up receiving the data and the data rate. SPIN completes after all nodes have received the data; directed diffusion will start the reinforcement phase after all nodes have received the data rate. The same network configuration is used 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 four.

The first set of experiments evaluated the data convergence time when only one SM is injected in the network. Fig.11. 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. The data convergence time for three different cases of the same SM and a base case that uses passive communication (no SM) are plotted. The top curve shows the time when code caching is not used. The second curve shows a more than fourfold improvement in performance when code caching is activated 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 will be used by subsequent SMs during the same execution. The effects of caching are very important in this case because the SMs visit a node multiple times in directed diffusion; they travel the network forward (looking for the source) and backward (diffusion of data rate).

In the third curve, a 30% decrease can be observed in completion time when the code is already cached at all nodes. The fourth curve shows the data convergence time for a traditional implementation: the protocol is implemented at each node; only data are transferred through the network; and the execution time is not accounted for. Observe that the degradation in performance for this implementation, when the code is cached at all nodes, compared to the traditional implementation is only 5%. This is a reasonable price for the flexibility to program any user-defined distributed application in NES.

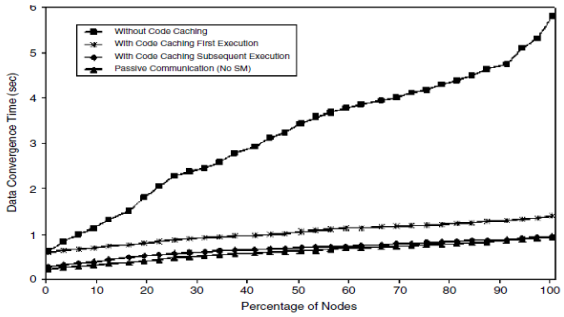

Fig. 11. Directed diffusion using smart messages.

Fig. 12 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, code caching leads to a threefold improvement in performance (i.e., reducing the size of SMs is essential for a protocol based on flooding and three stage communication). The third curve shows a 30% decrease in the completion time (similar to directed diffusion) when the code is

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

The second set of experiments quantified the performance of these applications when multiple SMs run simultaneously in the network. Fig. 13 and Fig. 14 show the data convergence time for directed diffusion and SPIN with the code already cached at nodes. For these experiments, data convergence time is the time when a certain percentage of nodes have received the data (or data rate) for all the SMs running in parallel. The nodes at which the SMs start are distributed uniformly in the network.

The results show that data convergence time increases with the number of SMs, but only during the initial flooding phase because of increased contention in the network. After that, the shapes of the curves are the same, independent of the number of SMs. The results also indicate that SPIN completes faster than directed diffusion in all cases. The cause is that SPIN floods only the neighbours and then brings the data to them, while directed diffusion needs to flood the entire network until it finds the source and then brings the data rate back to all nodes. In the initial phase, directed diffusion generates more messages in the network leading to higher contention, but its performance will increase as soon as the reinforcement phase begins.

IX. RELATED WORK

SMs have been influenced by the design of mobile agents for IP-based networks[5,13,14]. 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. SMs apply the general idea of code migration, but focus more on flexibility, scalability, reprogrammability, and the ability to perform distributed computing over unattended NES.

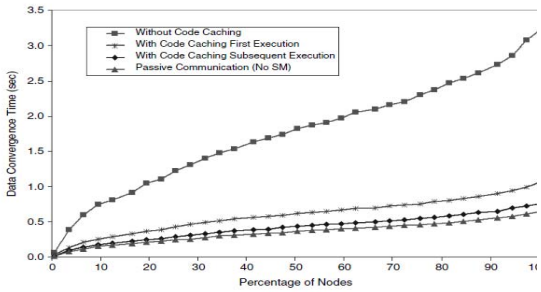

Fig. 12 SPIN using smart messages.

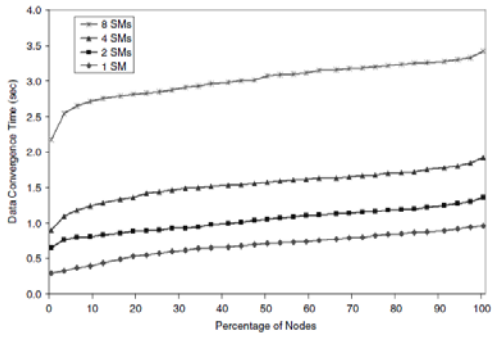

Fig. 13. Directed diffusion: multiple smart messages

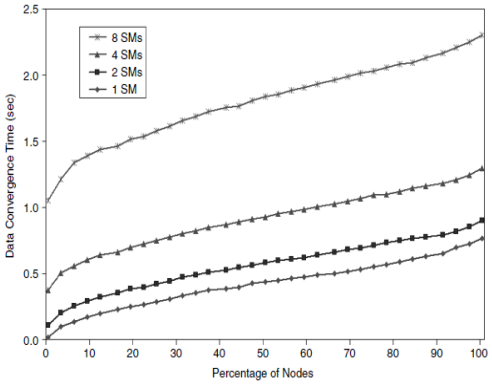

Fig.14. SPIN: multiple smart messages.

Research in mobile ad hoc networking[10,7,11,12] has resulted in numerous routing protocols. These protocols have generally been designed for IP-based networks and have primarily targeted traditional mobile computing applications over networks of mobile personal computers. Some of these protocols have been leveraged into routing algorithms used by the SM self-routing mechanism.

Sensor networks represent the first step toward large networks of embedded systems. Most of the research in this area has focused on hardware operating systems[9] or network protocols[2,4,6].Cooperative computing provides a solution for developing user-defined distributed applications in sensor networks, a crucial issue that has been tackled only marginally so far. As demonstrated, cooperative computing provides enough flexibility to enable implementation of previously proposed protocols over this computing platform.

SensorWare[3] is similar to cooperative computing in that both are frameworks for programmable NES based on code migration. Therefore, both are suitable to reprogram the network. However, Sensor- Ware supports mobile control scripts and accesses the resources through virtual devices, whereas cooperative computing supports mobile Java code (i.e., Java is supported on many embedded systems today execution state migration, and uniform access to resources through tags.

Mate[8] is an efficient VM for sensor networks that can significantly simplify code development and dissemination efforts. The main difference between cooperative computing and this research is that Mate targets only the reprogrammability of the network, but the programming model is still the traditional

X. CONCLUSIONS

In this paper, we have presented a computing model, Cooperative Computing, and a system architecture for distributed embedded systems. The nodes in the network cooperate by providing their computing and communication resources to distributed tasks. The system architecture is based on Smart Messages which are intelligent carriers of data in a network. I have proved that our model and system architecture represent a flexible, yet simple solution for programming large networks of embedded systems by implementing two previously defined applications for data collection and data dissemination (SPIN) in sensor networks.

REFERENCES

1. K Virtual Machine. http://java.sun.com/products/cldc/.
2. Blum, B., Nagaraddi, P., Wood, A., Abdelzaher, T., Son, S., and Stankovic, J. An entity maintenance and connection service for sensor networks. In *1st In. Conf. Mobile Syst., Applications, Services (MobiSys)* (May 2003), 201–214.
3.. Boulis, A., Han, C., and Srivasta, M. Design and Implementation of a framework for programmable and efficient sensor networks. In *Proc. 1st Int. Conf. Mobile Syst., Applications, Services (MobiSys).* (May 2003), 187–200.
4. Heinzelman, W.R., Kulik, J., and Balakrishnan, H. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. 5th Annu. ACM/IEEE Int. Conf. Mobile Computing Networking (MobiCom)* (August 1999), 174–185.
5 . Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. System architecture directions for networked sensors. In *Proc. 9th Int. Conf. Architectural Support Programming Languages Operating Syst. (ASPLOS)* (November 2000), 93–104.
6. Intanagonwiwat, C., Govindan, R., and Estrin, D. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proc. 6th Annu. ACM/IEEE Int. Conf. Mobile Computing Networking (MobiCom)* (August 2000), 56–67.
7. Johnson, D.B. and Maltz, D.A. *Dynamic Source Routing in Ad Hoc Wireless Networks*. In *Mobile Computing*, T. Imielinski and H. Korth (Eds.). Kluwer Academic Publishers, 1996, 153–181.
8. Levis, P. and Culler, D. Mate: a virtual machine for tiny networked sensors. In *Proc. 10th Int. Conf. Architectural Support Programming Languages Operating Syst. (ASPLOS)* (October 2002). 85–95.
9. Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L.-S., and Rubenstein, D. Energy-efficient computing for wildlife tracking: design trade-offs and early experiences with ZebraNet. In *Proc. 10th Int. Conf. Architectural Support Programming Languages Operating Syst. (ASPLOS)* (October 2002), 96–107.
10. Priyantha, N.B., Miu, A.K.L., Balakrishnan, H., and Teller, S. The Cricket compass for contextaware mobile applications. In *Proc. 7th Annu. ACM/IEEE Int. Conf. Mobile Computing Networking(MobiCom)* (2001), 1–14.
11. Karp, B. and Kung, H. Greedy perimeter stateless routing for wireless networks. In *Proc. 6th Annu. ACM/IEEE Int. Conf. Mobile Computing Networking (MobiCom)* (August 2000), 243–254.
12. Perkins, C.E., Royer, E., and Das, S.R. Ad hoc on demand distance vector (AODV) routing. In *2nd IEEE Workshop Mobile Computing Syst. Applications (WMCSA'99)* (February 1999), 90–100.
13. Karnik, N. and Tripathi, A. Agent server architecture for the Ajanta mobile-agent system. In *Proc. 1998 Int. Conf. Parallel Distributed Process. Tech. Applications (PDPTA'98)* (July 1998), 66–73.
14. Milojicic, D., LaForge, W., and Chauhan, D. Mobile objects and agents. In *Proceedings of the 4th USENIX Conf. Object-Oriented Technol. Syst.* (1998), 1–14.

Author's Profile

Dr.M.Usha Rani is an Associate Professor in the Department of Computer Science and HOD for MCA, Sri Padmavati Mahila Visvavidyalayam (SPMVV Womens' University), Tirupati. She did her Ph.D. in Computer Science in the area of Artificial Intelligence and Expert Systems. She is in teaching since 1992. She presented many papers at National and Internal Conferences and published articles in national & international journals. She also written 4 books like Data Mining - Applications: Opportunities and Challenges, Superficial Overview of Data Mining Tools, Data Warehousing & Data Mining and Intelligent Systems & Communications. She is guiding M.Phil. and Ph.D. in the areas like Artificial Intelligence, DataWarehousing and Data Mining, Computer Networks and Network Security etc.

K.Sailaja is a Research scholar in the Department of Computer Science (SPMVV) and working as assistant Professor in the Department of MCA, Chadalawada Ramanamma Engineering College, Tirupati. She is in teaching since 1999. She did her M.Phil in computer science in the area of WDM networks. She presented papers at National and Internal Conferences and published articles in national & international journals.