Cooperative Computing in Sensor Networks

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

Abstract

We envision that during the next decade the advances in technology will make sensor networks more powerful. Therefore, they will become part of a larger class of networks of embedded systems that have sufficient computing, communication, and energy resources to support distributed applications. The current software architectures and programming models are not suitable for these new computing environments.

We present a distributed computing model, Cooperative Computing, and the Smart Messages software 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. To illustrate this flexibility, we describe the implementation, using Smart Messages, of two previously proposed applications for sensor networks, SPIN and Directed Diffusion. The simulation results for these applications together with micro-benchmark results for our prototype implementation demonstrate that Cooperative Computing is a viable solution for programming networks of embedded systems.

1. Introduction

As the cost of embedding computing becomes negligible compared to the actual cost of goods, there is a trend toward incorporating computing and wireless communication capabilities in most of the consumer products. Therefore, we believe that 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.

The first illustration of these systems that has received considerable interest in the last couple of years are sensor networks [12, 10, 11]. These networks have severe resource limitations in terms of processing power, amount of available memory, network bandwidth, and energy. We envision, however, that 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. For instance, there are already companies that propose computer systems embedded into cars or video cameras which are able to communicate to each other [3, 2]. For some of these networks, such as a networks of intelligent cameras performing object tracking over a large geographical 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 the appropriate computing model is, and what system support is necessary to execute distributed applications in these networks.

NES pose a unique set of challenges which makes traditional distributed computing models difficult to employ in programming them. The number of devices working together to achieve a common goal are orders of magnitude greater than those seen so far. These systems are heterogeneous in their hardware architectures since each embedded system is tailored to perform a specific task. Unlike the In-

ternet, NES are typically deployed in environments void of human attention, where it is unacceptable to expect a human to hit a "reset" button to recover from a failure. NES are inherently fragile, with node and connection failures being the norm rather than the exception. The availability of nodes may vary greatly over time; the nodes can become unreachable due to mobility, depletion of energy resources, or catastrophic failures.

The nodes in NES communicate through wireless network interfaces. Hence, they can communicate directly only with nodes within their transmission range. Similar to most ad hoc networks, the separation between hosts and routers disappears (i.e., each node has to 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 algorithm in the same network is desirable.

The applications running in NES 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, are 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 names and node addresses. A naming scheme based on content or properties is more appropriate for NES than a fixed naming scheme [9].

We propose a distributed computing model, Cooperative Computing, and a software architecture for NES based on execution migration. Cooperative Computing applications consist of migratory execution units, called Smart Messages (SMs), working together to accomplish a distributed task. SMs are user-defined distributed programs (composed of code, data, and execution control state) that migrate through the network searching for nodes of interest (i.e., nodes on which the program needs to run) and execute their own routing 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 the volatility and dynamic binding of names to nodes specific to 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.

Nodes in the network support SMs by providing: (1) a name-based shared memory (tag space) for inter-SM communication and access to the host system, and (2) an architecturally independent environment 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.

To validate the Cooperative Computing model we have designed and implemented a prototype by modifying Sun Microsystem's Java KVM (Kilobyte Virtual Machine) [1]. We report micro-benchmark results for this prototype running over a testbed consisting of Linux-based HP iPAQs equipped with 802.11 cards for wireless communication. These results indicate that Cooperative Computing is a feasible solution for programming real world applications. For larger scale evaluation, we have developed a simulator that executes SMs and allows us to account for both execution and communication time. In this simulator, we have implemented two previously proposed applications for data collection and data dissemination in sensor networks, Directed Diffusion [12] and SPIN [10]. 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.

The rest of this chapter is organized as follows. The next section describes Cooperative Computing. Section 3 presents the node architecture for our the model. In Section 4, we discuss the details of Smart Messages, and Section 5 presents the API for Cooperative Computing. Section 6 shows micro-benchmark results for our prototype implementation. Section 7 describes the applications implemented using SMs, while their simulation results are presented in Section 8. Section 9 discusses related work. We conclude in Section 10.

2. The Cooperative Computing Model

Cooperative 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 Smart Messages (SM), that cooperate in achieving a common goal. The execution of an SM is described in terms of computation and migration phases. The ex-

ecution 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 the Cooperative Computing infrastructure which allows programming user-defined distributed applications in NES.

In our 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 [9].

SMs are resilient to network volatility. Over time, certain nodes may become unavailable due to mobility or energy depletion, but SMs are able to adapt by controlling the routing. SMs can carry multiple routing procedures and choose the most appropriate one based on the conditions encountered in the network. Using this feature, SMs can discover routes to nodes of interest even in adverse network conditions.

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 on performance of transferring code can be limited by caching code at the nodes.

Figure 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 [16]. In the figure, the target is represented by a person that moves across a given geographical region. A user can inject the tracking SM into any

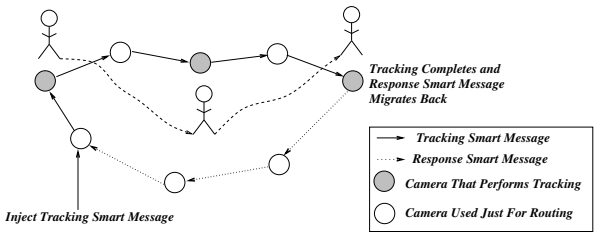

Figure 1: Distributed Object Tracking Using Cooperative Computing

node of the network. The SM migrates to a node that acquired an image of a possible target object, analyzes this image, and then it may decide to follow the object. The network maintains no routing infrastructure, and the SM is responsible for determining its path to cameras that detected the object. The SM can use the direction of motion and geographical information to "chase" the object. Once the SM arrives at a new node that has a picture of the object, it generates a task to further analyze the object and its motion. The SM may migrate to neighbor nodes to obtain pictures of the object from a different angle or lighting conditions. When the tracking completes, the SM generates a response SM that will transport the gathered information back to the user node.

3. 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. Figure 2 shows the common system support provided by nodes for Cooperative Computing. The admission manager receives incoming SMs, decides whether or not to accept them, and stores these messages into the SM ready queue. The code cache stores frequently used code 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 host's OS and I/O system.

3.1. Admission Manager

To prevent excessive use of its resources (energy, memory, bandwidth), a node needs to perform admission control. Each SM presents its resource re-

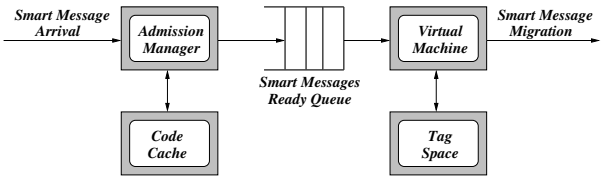

Figure 2: Node Architecture

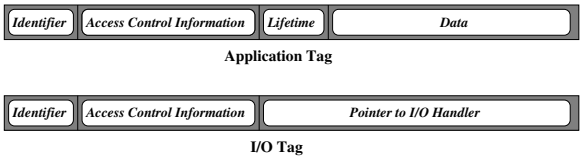

Figure 3: The Structure of Tags

quirements within a resource table. The admission manager is responsible for receiving incoming messages and storing them into the SM ready queue, subject to admission restrictions.

3.2. Code Cache

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

3.3. Virtual Machine

The virtual machine (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.

3.4. 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 which 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.

Figure 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. This identifier is used by SMs for content-based node naming. The access of SMs to tags is restricted based on the access control information associated with each tag. 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 [1]. 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) which is capable of serving I/O requests. Below, we list all the possible utilizations of tags:

- 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 woken 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.

1 More information about access control, protection domains, and SM security in general can be found in [27].

4. Smart Messages

SMs are execution units which 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 are comprised of 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 the size of these SMs, 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 traffic. 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.

The SM computation is embodied in tasks. During its execution, a task may modify the data bricks of the SM as well as the local tags to which it has access. It can also migrate, create new SMs, or block on tags of interest. A collection of SMs cooperating toward a common goal forms a distributed application.

4.1. 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 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.

4.1.1. 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 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 offline by applying a hash function on the code itself), whether the code bricks belonging to this SM are cached locally. Then, it informs the source to transfer only the missing code bricks (i.e., these code bricks will also be cached upon arrival).

4.1.2. Scheduling and Execution. Upon admission, an SM becomes a task which is scheduled (in FIFO order) for execution. The execution is non-preemptive; 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).

We use a non-preemptive scheduling 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, there is no need 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-preemptive scheduling makes inter-SM synchronization and sharing particularly simple to implement.

4.1.3. 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. 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 and transferred through the network. Essentially, we transfer the instruction and stack pointers 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 is specified by the programmer as data bricks.

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);

Table 1: Cooperative Computing API

4.2. 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 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. Since 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 has to 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 the network conditions change significantly during its execution. A complete description of the self-routing mechanism can be found in [5].

5. Programming Interface

The API for the Cooperative Computing model, given in Table 1, 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" SM. Typically, *spawnSM* is invoked when the current computation needs to migrate a copy of itself to nodes of interest while continuing the execution at the local node. A newly created SM is inserted into the SM ready queue.

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.

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 the control at one of the intermediate nodes in the path. Figure 4 presents an example of a typical SM which uses *migrateSM*. For instance, this SM can be used

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

Figure 4: Code Skeleton for Typical Smart Message

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 *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 *migrateSM* 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*.

6. Prototype Implementation and Evaluation

We have implemented our SM prototype in Java over Linux, thus harnessing well developed and supported Java application development tools and knowledge base [2]. Specifically, we have modified Sun Microsystem's KVM (Kilobyte Virtual Machine) [1] since 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, we have implemented the API as Java native methods. We have also implemented our own serialization mechanism since KVM does not support serialization. Besides the KVM interpreter thread, we have introduced two additional threads 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 virtual machine (e.g., Mate [19], Scylla [25]), any language, or underlying operating system.

In the following, we report micro-benchmark results for our SM prototype. Specifically, we have evaluated the cost of one-hop migration and the cost of tag space operations. Our testbed consists of HP iPAQs 3870 running Linux 2.4.18-rmk3-hh24. Each iPAQ contains an Intel StrongARM 1110 206Mhz 32bit RISC processor, 32MB flash memory, and 64MB RAM memory. For communication, we use Lucent Orinoco 802.11b Silver PC Cards in ad hoc mode. To factor out the cost of Java method call overhead (approximately 6μs), we have inserted the code for measuring the costs inside the native methods associated with the SM API.

6.1. Cost of SM Migration

The one-hop migration has three phases: execution capture at source, SM transfer, and execution resumption at destination. To be capable of capturing and resuming an SM, we convert the SM into a machine-independent representation. Since 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 our simple object serialization mechanism. The serialization of the execution state does not have a significant impact because we do not capture and transfer the local variables, but only the execution control state. Therefore, the important factors that determine the cost of one-hop migration are the data brick serialization, the SM transfer, and data brick de-serialization.

6.1.1. Data Brick Serialization and De-Serialization. To study the effect of data brick serialization, we have used a fixed size code brick (1197 bytes) and have varied the data brick from 2 KB 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. We have used two types of data bricks in this evaluation which 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 since each of its elements causes a call to the top level VM serialization method, while the integer array represents a lower bound since there is only one call to the top level VM serialization method.

2 The SM software distribution is freely available at http://discolab.rutgers.edu

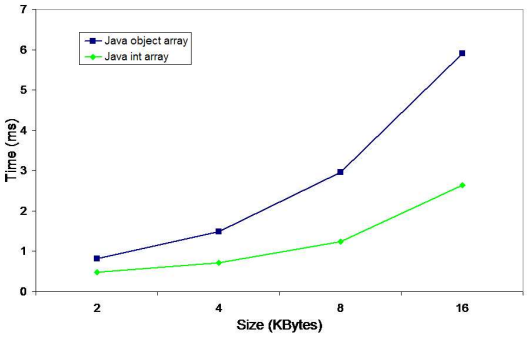

Figure 5: Cost of Data Brick Serialization

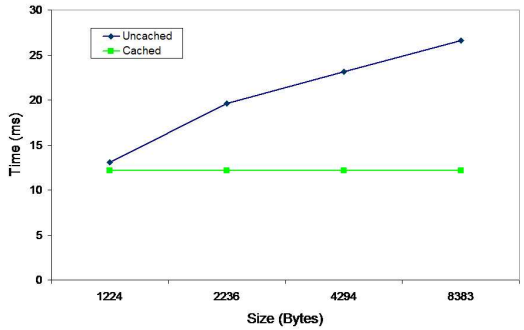

Figure 7: Effect of Code Brick Size on Single Hop Migration

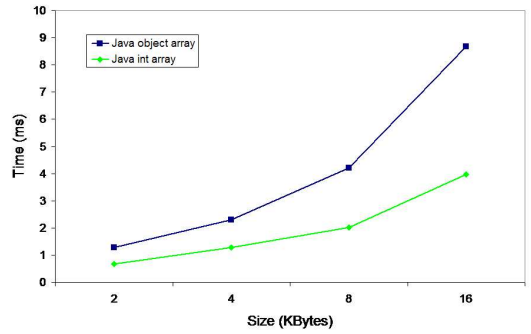

Figure 6: Cost of Data Brick De-Serialization

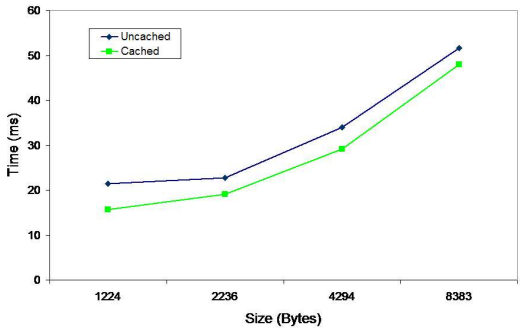

Figure 8: Effect of Data Brick Size on Single Hop Migration

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 that we developed carry less than 2 KB, which costs less than 1 ms to serialize. Figure 6 presents the cost of de-serialization for the same data bricks. We observe that this cost is with as much as 30% larger than the cost of serialization. This increase is caused by the memory allocation costs during object de-serializations.

6.1.2. SM Transfer. To evaluate the total cost of migrating an SM (serialization, transfer, de-serialization), we have performed two sets of experiments. In the first, we have varied the code brick size while keeping the data brick size and stack frame size fixed at 53 bytes and 131 bytes, respectively. In the second, we have varied the data brick size while keeping the code brick size and stack frame size fixed at 1197 bytes and 131 bytes.

Figures 7 and 8 show the results of these two experiments for two cases: the code is not cached, and the code is cached. In Figure 7, the time to transfer the SM when the code is cached represents, essentially, the overhead of the three-way handshake protocol since the size of the data bricks and stack frames is small. Figure 8 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.

6.2. Cost of Tag Space Operations

Tables 2 shows the cost of the tag space operations for application tags. The *readTag* primitive has the lowest cost since it performs the least number of

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

Table 2: Time for Tag Space Operations

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

Table 3: Cost of Reading I/O Tags

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 slightly higher since the interpreter has to 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 since 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). A typical node with a video camera and a GPS receiver attached is shown in Figure 9. The *gps_location* is updated by a user level process which reads from the GPS serial interface. The location of the neighbors along with their identifiers can be returned by reading the *neighbor_list* tag. This tag is typically used by geographical routing algorithms carried and executed by SMs. To get the information about neighbor nodes, we have implemented a neighbor discovery protocol which maintains a cache of known neighbors. For the *image_capture* tag, the system also performs YUYV to

Figure 9: Prototype Node with Video Camera and GPS Receiver Attached

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.

7. Applications

To prove that virtually any protocol or application can be written using SMs, we have implemented two previously proposed applications: SPIN [10] and Directed Diffusion [12]. 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.

7.1. SPIN using Smart Messages

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. Each time 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 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.

As an example of a Cooperative Computing program, Figure 10 presents the code for our implementation of SPIN using SMs. The tag space at each node hosts two tags: the value of the most recent data received (*tagData*), and the timestamp associated with this data (*tagTimestamp*).

```
 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 }
```

Figure 10: SPIN with Smart Messages

The protocol is initiated by injecting a *Disseminate SM* into a node that produces data. This SM blocks on *tagData* (line 8) waiting for new data. Each time new data is produced, the SM reads the *tagTimestamp* and spawns itself (lines 9-10). The "child" SM migrates to the neighbors to advertise the new data (line 13). If a destination node does not have this data or more recent data, the "child" SM updates the *tagTimestamp* and migrates back to the source to bring the data (lines 14-22). Upon data arrival, the "child" SM executes recursively the same algorithm until the data is disseminated in the entire network.

7.2. Directed Diffusion using Smart Messages

In Directed Diffusion, a sink node requests data by sending "interests" for named data. Data matching an interest is 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.

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.

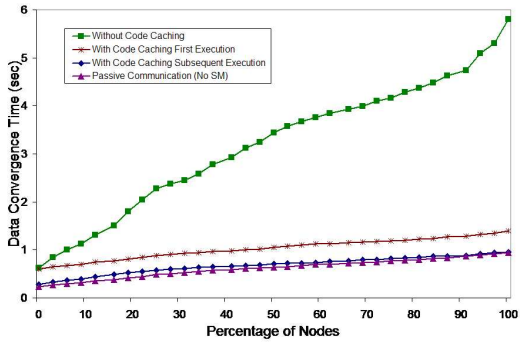

Figure 11: Directed Diffusion using Smart Messages

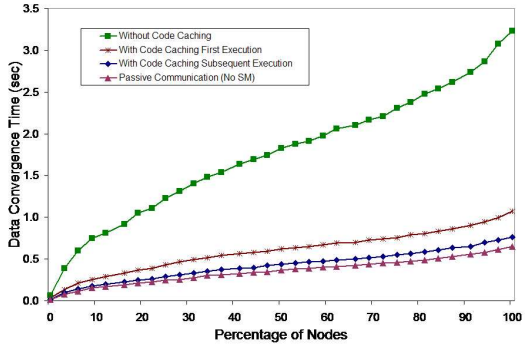

Figure 12: SPIN using Smart Messages

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 awaken by the data arrival, and it will carry the data back until it reaches the sink.

8. Simulation Results

For large scale evaluation, we have developed an event-driven simulator extended with support for SM execution. The simulator is written in Java to allow rapid prototyping of applications. To get accurate results, both the communication and the execution time have to 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, 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. 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 our implementations of SPIN and Directed Diffusion using SMs and to compare these results with the results for traditional message passing implementations. We define the data convergence time as the time when a certain percentage of the total number of nodes have 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, while Directed Diffusion will start the reinforcement phase after all nodes have received the data rate. 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.

The first set of experiments evaluate the data convergence time when only one SM is injected in the network. Figure 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. We plot the data convergence time for three different cases of the same SM and 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 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 both forward (looking for the source) and backward (diffusion of data rate). In the third curve we can observe a 30% decrease in the 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 is transferred through the network, and the execution time is not accounted for. We observe that the degradation in performance

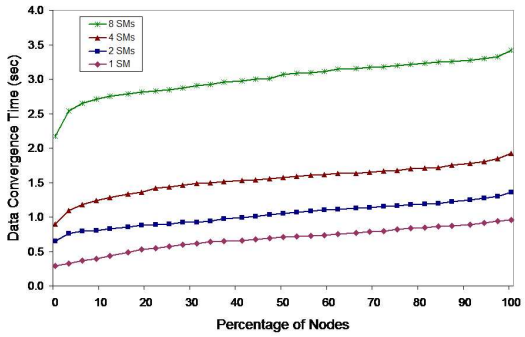

Figure 13: Directed Diffusion - Multiple Smart Messages

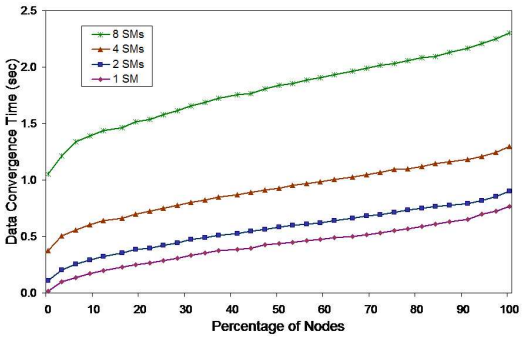

Figure 14: SPIN - Multiple Smart Messages

for our implementation, when the code is cached at all nodes, compared to the traditional implementation is only 5%. We believe that this is a reasonable price for the flexibility to program any user-defined distributed application in NES.

Figure 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 3 times 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 quantify the performance of our applications when multiple SMs run simultaneously in the network. Figures 13 and 14 show the data convergence time for both 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 (i.e., 2.3 s compared to 3.4 s for the top curves in the figures). The cause is that SPIN 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 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.

9. Related Work

SMs have been influenced by the design of mobile agents for IP-based networks [26, 17, 8, 20]. 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 ability to perform distributed computing over unattended NES. Unlike mobile agents, SMs are defined to be responsible for their own routing in a network. A mobile agent names nodes by fixed addresses and commonly knows the network configuration a priori, while an SM names nodes by content and discovers the network configuration dynamically. Furthermore, the SM software architecture defines the common system support that each node must provide. The goal of this architecture is to reduce the support required from nodes since nodes in NES possess limited resources.

The SM self-routing mechanism shares some of the design goals and leverages work done in active networks (AN) [7, 24, 21]. SMs differ, however, from AN in several key features. The main difference between them is in terms of programmability. Unlike AN which target faster communication in IP-based networks, Cooperative Computing define a distributed computing model for NES whereby sev-

eral SMs can cooperate, exchange data, and synchronize with each other through the tag space. Additionally, the AN model does not contain the migration of execution state whereas our model does. The migration of execution state for SMs trades off overhead for flexibility in programming sophisticated tasks which require cooperation and synchronization among several entities. For example, this execution state allows SMs to make routing decisions based on the results of the computation done at previously visited nodes.

Research in mobile ad hoc networking [14, 22, 18, 13] 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. We have leveraged some of these protocols 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 [15, 23], operating systems [11], or network protocols [12, 10, 4]. Cooperative Computing provides a solution for developing user-defined distributed applications in sensor networks, a crucial issue which has been only marginally tackled so far. As we have demonstrated, Cooperative Computing provides enough flexibility to enable the implementation of previously proposed protocols over our computing platform.

SensorWare [6] is similar to Cooperative Computing in the sense that both are frameworks for programmable NES based on code migration. Therefore, both are suitable to re-program the network. However, SensorWare supports mobile control scripts and accesses the resources through virtual devices whereas Cooperative Computing support mobile Java code (i.e., Java is supported on many embedded systems today [1]), execution state migration, and uniform access to resources through tags.

Mate [19] is an efficient virtual machine for sensor networks which can significantly simplify the code development and dissemination effort. The main difference between Cooperative Computing and this research is that Mate targets just the reprogrammability of the network, but the programming model is still the traditional message passing. SMs on the other hand are based on execution migration. An SM transfers not only the code, but also the execution state through the network.

10. Conclusions

We have described a programming model for large scale networks of embedded systems, in which distributed applications are implemented as collections of Smart Messages. The model overcomes the scale, heterogeneity, and connectivity issues encountered in these networks by using execution migration, content-based naming, and self-routing. The experimental results for our prototype implementation demonstrate the feasibility of Cooperative Computing. The implementation and simulation results for two sensor network applications show that our model represents a flexible, yet simple solution for programming large networks of embedded systems.

Acknowledgments

This work was supported in part by the NSF under the grant ANI-0121416. The authors would like to thank Ulrich Kremer and Chalermek Intanagonwiwat for our frequent discussions regarding the design of Cooperative Computing. We also like to thank Philip Stanley-Marbell, Deepa Iyer, and Akhilesh Saxena for their contributions at various stages of this project.

References

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

[2] Axis Communications. http://www.axis.com.

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

[4] Blum, B., Nagaraddi, P., Wood, A., Abdelzaher, T., Son, S., and Stankovic, J. An Entity Maintenance and Connection Service for Sensor Networks. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)* (San Francisco, CA, May 2003), pp. 201–214.

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

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

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

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

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

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

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

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

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

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

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

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

[17] Karnik, N., and Tripathi, A. 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)* (Las Vegas, NV, July 1998), pp. 66–73.

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

[19] Levis, P., and Culler, D. Mate: A Virtual Machine for Tiny Networked Sensors. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)* (San Jose, CA, October 2002), ACM Press, New York, NY, pp. 85–95.

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

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

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

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

[24] Schwartz, B., Jackson, A., Strayer, W., Zhou, W., Rockwell, R., and Partridge, C. Smart packets: Applying active networks to network management. *ACM Transactions on Computer Systems 18*, 1 (2000), 67–88.

[25] Stanley-Marbell, P., and Iftode, L. Scylla: A smart virtual machine for mobile embedded systems. In *3rd IEEE Workshop on Mobile Computing Systems and Applications, WMCSA2000* (Monterey, CA, December 2000), pp. 41–50.

[26] White, J. *Mobile Agents.* J. M. Bradshaw (Ed.), MIT Press, 1997.

[27] Xu, G., Borcea, C., and Iftode, L. Toward a Security Architecture for Smart Messages: Challenges, Solutions, and Open Issues. In *Proceedings of the 1st International Workshop on Mobile Distributed Computing (MDC'03)* (May 2003).