Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems *

Porlin Kang, Cristian Borcea, Gang Xu, Akhilesh Saxena, Ulrich Kremer, and Liviu Iftode
Department of Computer Science, Rutgers University
{*kangp, borcea, gxu, saxena, uli, iftode*}*@cs.rutgers.edu*

Abstract

In this paper, we present the design and implementation of Smart Messages, a distributed computing platform for networks of embedded systems based on execution migration. A Smart Message (SM) is a user-defined distributed program which executes on nodes of interest, named by their properties, and uses an explicit lightweight migration to reach these nodes. During migrations, an SM carries its code and execution state, and it self-routes at each intermediate node between two nodes of interest. The nodes in the network cooperate to support the SM execution by providing a virtual machine and a shared memory region addressable by names (tag space). To illustrate the flexibility of SMs to program real world applications, we describe EZCab, an application for booking cabs in densely populated urban areas. We also present experimental results to quantify the performance achieved by the SM prototype.

1 Introduction

Following the rapid development of sensor network technologies [1–5], networks of embedded systems (NES) will become ubiquitous during the next decade. We are already witnessing the beginning of this new world in which cars, cameras, cell phones, and even watches have wireless network interfaces and are powerful enough to run Linux [6]. NES offers the opportunity to program a large spectrum of applications, ranging from simple data collection and data dissemination to complex distributed applications such as remote object tracking using robots equipped with video cameras or inter-car collaboration to improve traffic safety and fluidity.

To leverage the raw computing power provided by NES into new distributed applications, we have to overcome the scale, heterogeneity, and volatility that characterize these networks. Current approaches to NES programming are ad hoc and provide limited flexibility; they are designed for specific classes of applications (e.g., querying the network for certain data) and can hardly accommodate new applications or services after

*This work is supported in part by the NSF grant ANI-0121416

the network has been deployed. As NES applications diversify, there will be an increasing demand for a common distributed computing platform to support arbitrary applications over NES.

A distributed platform for NES have to support simple development of new distributed applications. It also has to allow applications to cope with the uncertainty encountered in NES (e.g., the network topology as well as the resources at nodes are unknown a priori and can vary greatly over time). To answer these requirements, we need new programming abstractions since traditional message passing does not work in highly dynamic network configurations. This model includes a number of characteristics that render it unusable in NES: end-to-end data transfer between applications, fixed bindings between names and node addresses, and fixed routing.

A first problem with end-to-end data transfers is that they may complete very slowly, or may not complete at all in volatile networks [7]. Since applications have no control over the network, they are forced to wait indefinitely (or until the connection times out) each time something goes wrong in the network. To be able to adapt quickly to network volatility, applications would like to regain the control as soon as possible. Another problem with traditional end-to-end data transfer is that it does not allow in-network processing in order to reduce the size of data transferred by applications [2]. Reducing the amount of traffic in the network is important in mobile ad hoc networks, such as NES, since it leads to reduced bandwidth and energy consumption. Therefore, NES applications would also like to be capable of performing in-network processing.

The fixed bindings between names and node addresses assumed in the message passing model represent also a serious obstacle for NES applications. After a fixed binding has been established during the name resolution phase, an application is forced to contact the same node each time it needs to access a resource of the same type. Commonly, name resolvers react slowly to network changes, and applications would try to contact a node long time after this node has become unreachable, even though nodes with similar resources exist in the network. To prevent such a situation, more flexible naming is needed in NES. We believe that content-based naming [8, 9] can provide a solution because it allows applications to contact any node has a certain resource.

Since content-based naming makes fixed addresses (e.g., IP) irrelevant, the routing and name resolution should be integrated in NES. Additionally, given the diversity of applications, no single routing will provide good performance for all applications. Therefore, similar to active networks [10–12], it would be desirable to let applications use the best-suited routing for their needs.

In this paper, we present the design and implementation of Smart Messages (SMs) [13], a distributed computing platform for NES based on execution-migration, content-based naming, and self-routing [14]. Instead of passing data end-to-end between nodes, an SM application migrates to nodes of interest named

by content and executes there. Each node has a virtual machine for SM execution and a name-based memory, called *tag space*. The SMs use the tag space for content-based naming and persistent shared memory. An SM carries its own routing code and routes itself at each node in the path toward a node of interest. To perform routing, SMs store routing information in the tag space at nodes.

SMs represent an attractive alternative to traditional distributed computing based on message passing for four reasons. First, SMs allow applications to adapt to highly dynamic network conditions. During migrations between nodes of interest, the routing code can be instructed to return the control to application as soon as a route cannot be found or after an application-set timeout. Since application's code as well as its execution state are already at the same node, the SM can quickly adapt to changes in the network. For instance, an SM can change dynamically its routing or its destination. Second, the content-based routing provides the flexibility to reach a node that offers a certain property in an application-controlled manner. Third, the SM programming model eases the deployment of new applications in the network after the network deployment phase has ended. A user can inject applications at any node in the network, and consequently the applications migrate their code at every node where they need to execute. And fourth, SMs can significantly reduce the amount of traffic generated by certain classes of applications (e.g., process the data at the source for distributed image processing).

Although the SM computing platform leverages work from various research areas, notably from mobile agents [15, 16] and active networks [10–12], its uniqueness comes from both the problem it solves, programming distributed applications in NES, and its design that allows rapid adaptation of applications to volatile network configurations. A detailed discussion and comparison between SMs and other work is presented in Section 10.

To validate the proposed architecture, we have developed an SM prototype in Java by modifying Sun's Java KVM [17]. The prototype runs over a testbed consisting of HP iPAQs equipped with 802.11 cards for wireless communication. This paper presents the implementation and evaluation of our SM prototype. It also describes an application developed over this prototype, called EZCab, for booking cabs in densely populated urban areas.

The rest of this paper is organized as follows. Section 2 introduces the main SM concepts. The system support provided by nodes is described in Section 3. Section 4 discusses the SM API. The self-routing mechanism is presented in Section 5. The basic security architecture is explained in Section 6. Section 7 describes the implementation details of our SM prototype, and Section 8 presents an evaluation of this prototype. Section 9 describes the EZCab application. We discuss the related work in Section 10 and conclude in Section 11.

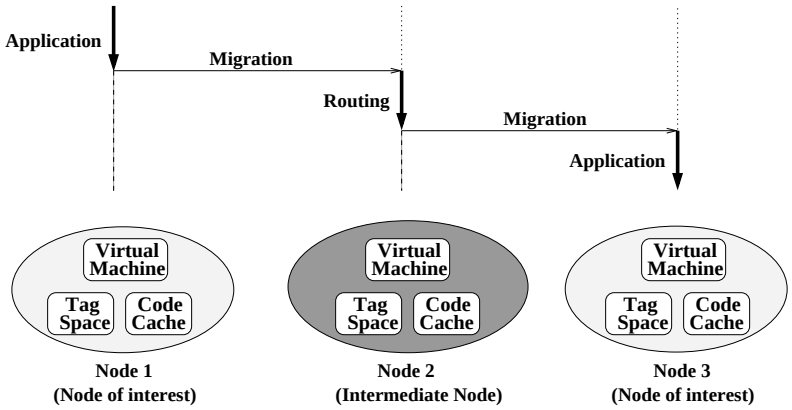


Figure 1: Distributed Computing Using Smart Messages

2 Smart Messages Architecture

An SM is a component of a user-defined application whose execution is distributed over a series of nodes using execution migration. The nodes on which SM applications execute, called "nodes of interest", are named by content (tag names), discovered using application-controlled routing, and switched when the SM application calls for execution migration. The payload of an SM consists of data "bricks", explicitly identified in the application, and execution control state. Code "bricks" may also be transferred if the code is not cached at destination. An SM can carry multiple data and code bricks, and it can use them to create new SMs during its execution. In this way, an application can eventually generate multiple SMs although it has started as a single SM.

The SM computing platform assumes a decentralized architecture, where nodes in the network act as peers. SMs do not make any assumptions about the underlying network configuration, except for a minimal system support provided by nodes: a *virtual machine*, a name-based memory, called *tag space*, and a *code cache*. The virtual machine offers a hardware abstraction layer for SM execution, which shields SMs from heterogeneous node configurations. The tag space offers a name-based memory, persistent across SM executions. It consists of *(name, data)* pairs, called tags, which are used for data exchange among SMs. Special I/O tags are used as an interface to the host OS and I/O system. Tags serve also to name the destination of SM migrations and store routing information (routing tags). The code cache stores frequently accessed code bricks in order to amortize the cost of transferring code over time. Figure 1 depicts the execution of an SM over three nodes. The SM application code starts on *Node1* and finishes on *Node3*. The SM reaches *Node3* by explicitly migrating from node to node. *Node2* is used as an intermediate hop, where only the SM routing code executes.

To illustrate how NES are programmed using SMs, we present a very simple example consisting of an

```
// application data brick
int numCabs, i;
Location loc;
// application code brick
for(i=0; i<numCabs; i++){
    migrate("FreeCab");
    deleteTag("FreeCab");
    writeTag("Location", loc);
}
```

Figure 2: Smart Message Code Example

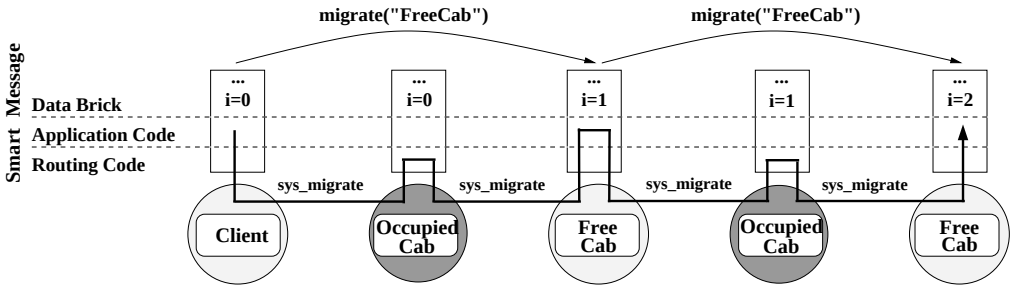


Figure 3: Smart Message Execution Path

SM that books cabs in a densely populated city. Let us consider a group of people attending a conference, who wants to return to the conference venue after an "off-site" lunch. Instead of calling a cab company or waiting on the street for a free cab, one of them uses her handheld device to inject an SM in the network to book a certain number of free cabs. Each cab provides support for SM execution and is identified by a *FreeCab* tag. The code for this application is shown in Figure 2, and the SM execution path through the network is depicted in Figure 3. The SM migrates to free cabs, changes their status from free to occupied (by removing the *FreeCab* tag), and instructs them to come to the client's location (by writing to *Location* tag). The SM completes after booking the desired number cabs.

The key operation in the SM programming model is multi-hop, content-based migration, which implements routing using tags. An SM names the nodes of interest by tags, and then calls a high-level *migrate* function to route itself to a node that has the desired tags. In our example, *migrate("FreeCab")* routes the SM to free cabs using the occupied cabs as intermediate nodes. This high-level function uses the low-level *sys_migrate* primitive, provided by the system, for one-hop migration. After a migration, the SM resumes from the next instruction following the migrate call. It is important to notice that migration is explicit (i.e., the programmer calls *migrate* when needed).

Figure 3 emphasizes two major characteristics of the SM programming model. First, the high-level, content-based migration shields the application programmer from the routing details. Although the routing

code is executed at each node as the SM migrates hop-by-hop through the network, *migrate* returns the control to application only on nodes of interest (i.e., free cabs). Second, the data transferred during a migration is specified by the programmer as data bricks; the variables *numCabs*, *i*, and *loc* are stored in a data brick and carried from node to node during migrations (the figure shows how *i* is modified during execution).

From a user's perspective, this model offers resilience to dynamic network configurations and simple deployment of new distributed applications in the network. An application programmer can write simple sequential programs that migrate to nodes named by content and execute there, while ignoring the routing which is embedded in *migrate* functions. These are user-level functions, typically developed by system programmers. Applications can choose between multiple *migrate* functions and adapt to dynamic network configurations by switching these functions during execution.

To achieve good performance in networks composed of resource constrained nodes, we have decided against involving the VM in determining which data is needed across migrations. In our architecture, the VM captures the minimal execution control state required for SMs to resume at the instruction following a migration. Although this decision puts clearly a burden on programmers, it avoids the overhead of having the VM collect the "live data" of SMs; many times this operation is not only time consuming, but also collects more data than necessary (i.e., conservative approach), thus increasing the amount of traffic in the network.

3 Cooperative Node Architecture

In order to execute SM-based applications, the nodes must cooperate to support SM execution and routing. The entire SM model is built under the assumption that the node architecture must be kept as simple and flexible as possible. Figure 4 shows the system components of a cooperative node.

Virtual Machine. The virtual machine (VM) executes VM-level threads generated by incoming SMs. To migrate an SM, the VM captures the execution state and sends it along with the code and data bricks to the next hop. The VM at destination resumes the SM at the instruction following the *migrate* call.

Local Injector. The local injector allows the users to start new SMs at the local node. A VM-level thread is generated for each new SM. This thread is stored in the *SM ready queue* and dispatched for execution according to the scheduling policies.

Scheduler. The SM execution is non-preemptive; other SMs can be accepted, but they are not dispatched for execution before the current SM completes. The non-preemptive scheduling simplifies the implementation of inter-SM synchronization and sharing. Additionally, we envision that the overhead introduced by more

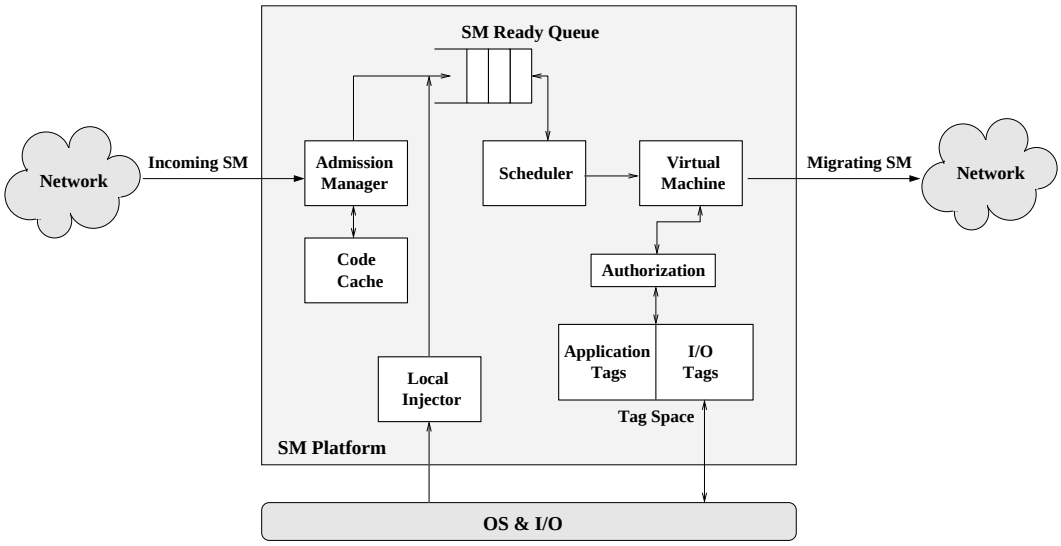


Figure 4: Cooperative Node Architecture

complex scheduling will not be justified for NES applications, which typically have short execution time.

Admission Manager. To prevent excessive use of resources (e.g., processor cycles, tag space memory, runtime memory, bandwidth), the nodes have to perform admission control on incoming SMs. The admission control at nodes ensures the progress of all SMs running in the network. It also prevents SMs from migrating to nodes where they cannot achieve anything due to resource constraints. SM s present their resource requirements in a resource table. The admission manager receives the resource tables, decides whether to accept the SMs or not, and enqueues the accepted SMs into the SM ready queue. It also instructs an accepted SM to transfer only the missing code bricks (i.e., the code bricks that are not stored locally) and stores them in the code cache upon reception.

The admission manager makes the admission decision based on the current state of the node and the SM's resource requirements. This decision is based on the admission policy in effect at that node. An accepted SM is guaranteed non-preemptive execution as long as its resource usage does not exceed certain limits defined by the admission policy. For instance, a node may run out of battery and decide to accept only SMs for which it is a node of interest, but reject all SMs that need to route through it. If an SM is rejected, the migration call fails at the source, and the SM regains the control.

Precise resource usage for SMs cannot be predicted in advance because their computations depend not only on user-provided input data, but also on data gathered from the network during execution. To be able to perform admission, the admission manager needs, however, at least approximate information about SMs' resource requirements. One solution would be to specify upper bounds for the resource requirements. We have dismissed this idea for two reasons: computing relatively precise upper bounds is as hard as predicting

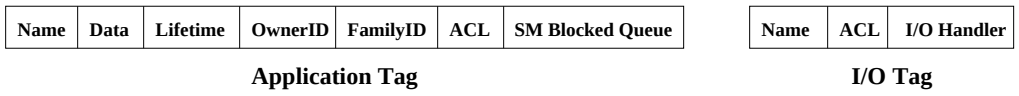


Figure 5: Application and I/O Tag Structures

the actual resource usage (i.e., we do not have knowledge about data acquired at runtime), and large upper bounds may lead to frequent rejections at nodes even though the SM may consume significantly less resources during its execution.

Our solution requires each SM to specify its lower bounds for resource requirements. The programmers can set them before any one-hop migration, and they define the minimum amount of resources that may lead to SM completion or migration. The programmers may use compiler support to derive lower bounds for resource requirements. The declaration of these lower bounds serves two purposes: protect SMs from migrating to a node that cannot offer enough resources for any semantically acceptable result, and protect the resources at the node from being wasted on such SMs. Based on the admission policy, the system may grant more resources to SMs that have exceeded their lower bounds during execution. If no more resources could be granted, the system raises an exception which, by default, terminates the SM. The SM is allowed, however, to catch this exception, to save data of interest in data bricks, and to migrate. A limited amount of resources is reserved during admission for the exception handler. To ensure a successful migration for this case, the SM has to declare, during admission, the maximum amount of data it plans to carry to the next hop.

Tag Space. The tag space provides a name-based memory and a unique interface to the local OS and I/O system. It consists of a collection of tags that can be divided into two categories: (1) *application* tags which are created by SMs and used for inter-SM communication and synchronization, and (2) *I/O* tags which belong to nodes and allow SMs to access system resources. The structures of these tags are shown in Figure 5. Each tag has a name (unique at a node, but not globally unique) which is similar to a file name in a file system. SMs use this name for content-based naming.

Application tags are commonly used for data exchange among SMs because their data portion can store application-specific data. For instance, an SM can build a routing table in a tag, and other SMs can subsequently read the routes from this tag. Each application tag has a lifetime that specifies the duration after which the tag expires and its memory is reclaimed by the node.

I/O tags act as a gateway between SMs and the underlying OS and I/O system. Usually, each I/O tag is associated with an external process, which communicates with the VM through a standard interface. Each time an I/O tag is accessed by an SM, its associated external process interacts with the local resources and returns a response to the SM.

The access to tag space is protected using an access control list (ACL). The application tags have also ownership information (i.e., OwnerID and FamilyID). We defer the description of the protection mechanism to Section 6.

Similar to existent solutions [18,19], we use namespaces to avoid naming conflicts; a tag name is preceded by a namespace (i.e., *namespace:tagname*). The I/O tags have a pre-defined namespace, *ions*, which is known by any SM. The namespaces for application tags, on the other hand, are defined by the SMs that create them. Each SM has a unique default namespace which is used when a reference to a tag name is not preceded by a namespace. The system where the SM is injected generates this unique namespace, and every SM created dynamically inherits it from its parent SM.

An SM may use other namespaces to cooperate with SMs that do not belong to its family. Accessing tags in other namespaces does not create problems because the access is subject to access control. Creating new tags, however, may lead to naming conflicts. For instance, two different SMs may create two tags with the same name, but with different semantics. A solution to this issue is to ensure that conflicting namespaces are extremely rare in practice (e.g., a namespace is a long random string of bits). The developers that need to cooperate can exchange these namespaces off-line.

Although simple, this solution is not bullet-proof. If an SM needs to ensure that conflicts are avoided, it has to use *secure* namespaces (i.e., by definition, a secure namespace is preceded by the keyword *secure*). At the compilation time, the compiler builds the list of secure namespaces used in tag creation invocations throughout each code brick. The compiler has to be able to generate the list of namespaces (i.e., the namespaces are either directly specified, or the compiler is able to determine them using static analysis); if the compiler is not able to find at least one possible namespace for a tag, the compilation fails.

At injection time, the SM must present a capability for each namespace in the compiler-generated list. Therefore, the developer of a code brick (or the developer of an SM) has to acquire these capabilities such that each code brick of an SM has an associated list of capabilities. During SM injection, the system verifies the capabilities and creates a list of namespaces for each code brick. This list together with the default namespace is maintained in the SM structure and cannot be modified over time. A child SM inherits the list of namespaces for the code bricks that compose it. If an SM does not present a capability for every namespace in the list generated by the compiler, it will be rejected during the injection phase.

A central authority (CA) keeps track of all secure namespaces and their owners. Each time a namespace owner decides to allow a code brick to create tags within that namespace, she associates a capability, digitally signed by the CA, with this code brick; the capability contains the hash value of the code brick. Similar to ANTS [10], this value is obtained by applying a hash function on the code itself. Each node has the public key of the CA and the common hash function. During SM injection, the VM uses the CA's public key and

Category	Primitives

Smart Messages	createSMFromFiles(code_files);
	createSM(code_bricks, data_bricks);
	spawnSM();
	sys_migrate();
	blockSM(tag_name, timeout);
	setResources(resources);

Tag Space	createTag(tag_name, lifetime, data);
	deleteTag(tag_name);
	readTag(tag_name);
	writeTag(tag_name, data);

Table 1: SM API

the capability to verify that the code bricks are authorized to use the secure namespaces.

Synchronization Mechanism. Given the non-preemptive SM execution, we have devised a simple update-based synchronization mechanism for inter-SM communication. An SM can block on an application tag until another SM performs a write on that tag. A blocked SM SM is appended to the *SM blocked queue* and yields the processor (this is the only exception to our run-to-completion model of execution). After an SM SM blocks, the scheduler may dispatch other SMs for execution. When an SM writes to an application tag with a non-empty SM blocked queue, all SMs in the queue are woken up and made ready for scheduling. To prevent infinite blocking, if no write operation takes place within a given timeout, SMs are unblocked and made ready for scheduling.

4 Smart Messages API

The SM API is presented in Table 1. SMs are allowed to create new SMs dynamically, migrate one-hop to neighbor nodes, access the tag space, set lower bounds for resource requirements and synchronize on tags. Also, the SMs can use the uniform interface provided by the tag space to execute system calls on the local host (i.e. through I/O tags).

SM Creation. Initially, an SM is injected at a node as a program file, and it calls *createSMFromFiles* with a list of program file names to create a new SM structure. An SM may use *createSM* to assemble a new, possibly smaller SM using some of its code and data bricks. A *createSM* call is commonly used to build an SM that cooperates with the current one (e.g., a route discovery SM). An application that needs to clone itself calls *spawnSM* (similar to the *fork* system call in Unix). Typically, *spawnSM* is invoked when the current SM needs to migrate a copy of itself to nodes of interest while continuing the execution at the local node. A new SM generated by *createSM* or *spawnSM* is scheduled for execution at the local node.

SM Migration. The *sys_migrate* primitive implements one-hop migration. It captures the execution state, sends the resource table for admission, transfers the accepted SMs, and resumes these SMs at destination. The *sys_migrate* is used by high-level *migrate* functions to route SMs to nodes of interest. More details about the SM self-routing mechanism are presented in Section 5.

Synchronization. The *blockSM* primitive allows SMs to block on a tag pending a write by another SM. Typically, an SM uses this primitive to wait for a route. For instance, an SM can create a route discovery SM and block on a routing tag until the route discovery SM returns (i.e., the route discovery SM writes to the routing tag, and thus wakes up the blocked SM).

SM Resource Requirements Programmers invoke *setResources* each time they need to set new lower bounds for resource requirements. Typically, this primitive is called once per high-level migration invocation and specifies two categories of lower bounds: resources needed for routing, and resources needed for computation at the node of interest (i.e., the target of migration). The resources are implementation specific, but they include at least: number of VM cycles, amount of runtime memory, amount of tag space memory and the duration for which this memory is needed, I/O tags to be accessed, and maximum number of bytes that would be generated when migrating this SM to another node. An SM is not required, however, to set the resource requirements. In such a case, the admission is based only on the size of the SM, but the node does not provide any type of guarantees (i.e., the SM can be terminated or asked to migrate at any moment). Our current prototype, described in Section 7, uses this very simple solution.

Tag Space Access. An SM can create, delete, or access application tags. As mentioned in Section 3, the tags are accessed subject to authorization. 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 from I/O tags (an SM cannot create or delete I/O tags).

5 Self-Routing Mechanism

Similar to most mobile ad hoc networks, the distinction between hosts and routers disappears in NES. In our approach, there is no support for routing at nodes. Each application has to include at least one *routing brick* among its code bricks. An application can control routing in two ways: select its routing algorithms, or change the routing during execution. Each routing brick defines a user-level *migrate* function, which contains the routing code and is commonly provided as a library implementation. The programmers, however, are free to implement their own *migrate* functions. Any arbitrary condition on tag names and tag values can be used in *migrate* to define a node of interest. For instance, a simple implementation of *migrate* takes a list of tag names as parameter and migrates the SM to a node that contains all those tags. To cope with

```
1   // routing data brick
2   String tag, routeToTag;
3   int timeout;
4   // routing code brick
5   boolean migrate(String tagName, int timeOut){
7      // use parameters to set tag, routeToTag, and timeout in data brick
6      while(readTag(tag) == null){
7         Address nextHop = readTag(routeToTag);
8         if (nextHop != null)
9            sys_migrate(nextHop);
10        else{
11           createSM(RouteDiscoverySM, tag);
12           if (blockSM(routeToTag, timeout) == TIMEOUT)
13              return FALSE;
14        }
15     }
16     return TRUE;
17  }
```

Figure 6: Migration Example Using Simple On-demand Routing

network volatility, *migrate* can take a *timeout* as parameter. If a timeout occurs (i.e., the routing algorithm has not been able to find a node of interest during the given period), the application regains the control at an arbitrary node on the path between two consecutive nodes of interest. Consequently, it may decide to change the routing or the destination of migration.

Our current prototype provides two pre-defined routing algorithms, a content-based on-demand routing (similar to AODV [20]) and a geographical routing (similar to GPSR [21]). SMs implement routing using tags and the *sys_migrate* (4) primitive. Since tags are persistent across SM executions, the routing information can be shared by SMs with similar interests, thus amortizing the route discovery effort. Given the volatility of NES, most of the routing information will have a short lifetime. Thus, the amount of tag space consumed for storing routing information will be limited. To improve the scalability for networks with many different tags, we have designed an algorithm which maintains approximate information about tags in the network using Bloom filters [14,22].

Figure 6 shows a simplified version of our *migrate* function implemented using on-demand routing. If a next hop toward a node of interest is available, the entire SM migrates there (lines 8-9). Otherwise, a route discovery SM is created, and the current SM blocks waiting for a route (lines 11-12). The SM is woken up when the discovery SM returns with a route and writes the routing tag. If no route is acquired during an application-set timeout, *migrate* returns the control and informs the application that it could not find a node of interest. A problem generated by content-based routing is how to ensure that an application does not end up on a node already visited. We have used two solutions; either instruct *migrate* to discover an un-visited node each time it is called, or let the application record the visited nodes of interest and pass this list to *migrate*.

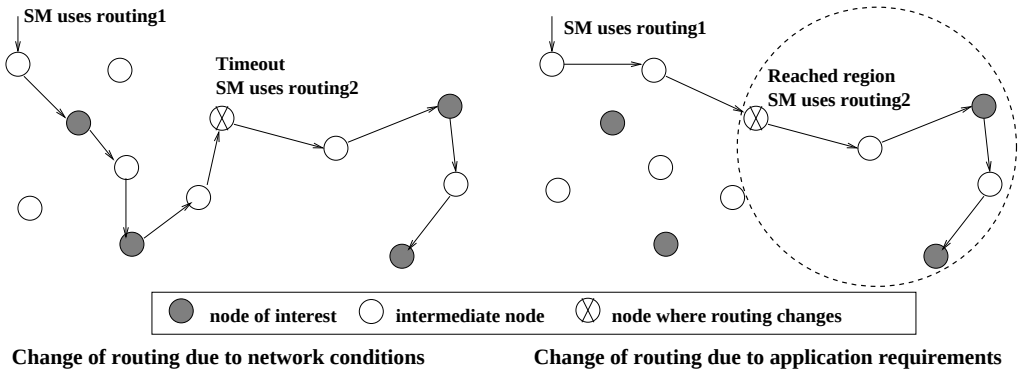


Figure 7: Dynamic Change of Routing

A single routing algorithm might not always reach a node of interest in the presence of highly dynamic network configurations. Therefore, the use of multiple routing bricks during the lifetime of an SM can help an application complete in adverse network conditions. The completion time of an SM can also be improved by using multiple routing algorithms. Two factors can determine a dynamic change of routing: a significant modification in the network configuration, or specific application requirements.

Figure 7 illustrates both factors. In the first example, the SM needs to visit a number of nodes of interest, but it cannot complete because its current routing (suitable for dense and relatively static networks) cannot find a route when the network becomes sparse and mobile. The SM, however, has instructed *migrate* to return the control after a certain timeout. At this time, the SM regains the control at an arbitrary node and calls another *migrate* function, which implements a routing suitable for the new network conditions (e.g., content-based on-demand routing). Using the new routing, the SM is able to make further progress.

The second example depicts a change of routing determined by the application requirements. An SM needs to visit several nodes of interest located in a given geographical region (the circular region). Therefore, an SM may have two routing bricks, a geographical routing used to reach the region, and an on-demand content-based routing used to discover the nodes within that region. In this situation, the SM changes the routing when it reaches the region of interest.

6 Security Architecture

One of the traditional pitfalls of existing systems based on mobile code is security. Similar to mobile agents, there are three main issues that have to be solved: (1) protecting recipient hosts from SMs, (2) protecting SMs from each other, and (3) protecting SMs from malicious hosts. These problems become more severe for SMs due to the volatile nature of NES. Unlike traditional mobile agents for relatively stable IP-based networks, the SMs have to overcome the lack of an infrastructure or a central authority, specific to mobile

ad hoc networks, which increases significantly the difficulty of key authentication and group management.

In this section, we present a basic security architecture for SMs, which focuses on providing protected access to the tag space. This security architecture offers protection against malicious SMs under the assumption that the SM system software at nodes is trusted (i.e., we do not protect SMs against compromised hosts). To protect against compromised systems, we plan to develop a distributed trust mechanism [23], which helps a node assign trust values to its one-hop neighbors; a node deemed untrusted is simply removed from the list of neighbors. Optionally, an SM may ask to be migrated in an encrypted form between neighbor nodes. To support this, each node carries a pair of public/private keys.

6.1 Access Control

A unique characteristic of SMs is that no direct access is allowed to system resources (i.e., the SMs access both their data and system resources through the tag space). The advantage of this design is that the tag space is a single point of access control, which can be implemented and enforced uniformly. Compared to mobile agent systems [24], the tag space simplifies greatly the control mechanisms. The SM creating a tag, called tag's owner, determines the access control policy and delegates the host to enforce this policy on its behalf. Protecting the application tags ensures that SM executions do not interfere with each other, and therefore, provides a secure channel for SM cooperation.

A tag incorporates the ID of its owner, the ID of its owner's family, the address of the node where its owner's family originated, and its ACL (access control list). SMs are uniquely identified by the node address where they originated and the time of their creation. We define a family of SMs as all SMs originated from an SM injected in the network by a user. The family ID is the ID of the original SM. Since an SM can migrate or spawn new SMs at intermediate nodes, its family information can be used to enforce access control for an entire family of SMs. The ACL is a matrix of subjects and their access permissions to tags, read(r) or write(w). The ACL contains five protection domains: *Owner*, *Family*, *Origin*, *Code*, and *Others*.

Each time an SM tries to execute an operation on a tag, the VM performs the authorization process. Based on the credentials presented during admission and the currently executing code brick, the SM is associated with at least one protection domain. A user or the SM itself cannot forge an SM's identification information because this information is set automatically by the system. The request is granted if the SM has the necessary permissions to access the tag in any of the protection domains it has been associated with.

6.2 Protection Domains

The *Owner* and *Others* protection domains define the access permissions for the SM that owns the tag and for any SM, respectively. The group concept, defined as an arbitrary relation over SMs, supports more flexible

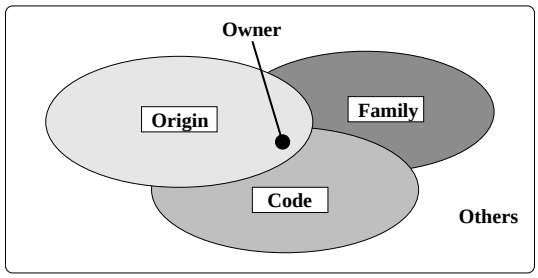
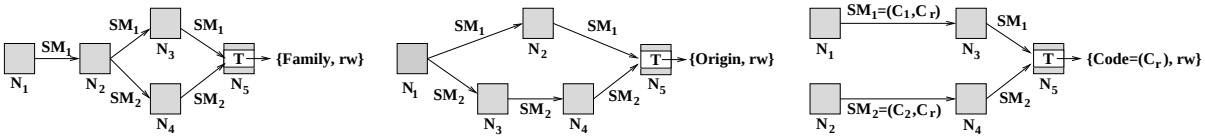


Figure 8: SM Protection Domains for Tag Space Access



(a) Family Cooperation (b) Single Originator Cooperation (c) Code-based Cooperation

Figure 9: Three Access Control Scenarios for SM Group Cooperation (N_i are Nodes, SM_i are Smart Messages, and T is a Tag)

cooperation, but also requires high overhead of managing the group membership on-the-fly. Currently, our architecture does not support dynamic cooperation among totally independent SMs. Instead, we define three protections domains that allow cooperation among well-defined groups of SMs (i.e., *Family*, *Origin*, *Code*). Figure 8 shows that an SM can be associated with multiple protection domains for a tag. In the following, we present three scenarios that illustrate the protection domains for group cooperation.

Family cooperation. In Figure 9(a), all cooperative SMs originate from a common SM ancestor. For instance, SM_1 is created on N_1 and migrates to N_2. At this node it creates a child, SM_2, which migrates and creates a tag T on node N_5. To allow SM_1 to access this tag, SM_2 sets the ACL to {*Family, rw*} (i.e., the familyID of T is the same as the family ID of SM_1).

Single originator cooperation. Figure 9(b) shows the scenario when the group of cooperative SMs originate from a common node. SM_1 and SM_2 are created on node N_1 and migrate to a target node N_5 via different paths. SM_1 arrives at N_5 before SM_2 and creates a tag T. It also sets the ACL as {*Origin, rw*} such that SM_2 will be able to access T (i.e., the unique IDs of SM_1 and SM_2 contain the same origin ID). This scenario is very likely to be encountered since many nodes are small devices, such as PDAs or cell phones, owned by a single user.

Code-based cooperation. In addition to the simple groups described before, the SM group cooperation can be coordinated more flexibly based on code bricks. To ensure cooperation among SMs that are aware of the code used for data sharing or data exchange, each tag has a list of associated hash values for certain code bricks. These hash values define the members of the *Code* group (they may or may not belong to the

owner of the tag). By definition, an SM is a member of the *Code* group if the hash value of its currently executing code brick belongs to this list. For instance, SMs using the same routing brick can add the hash value corresponding to this brick to the tag's list of hash values in order to facilitate route sharing among them. Figure 9(c) presents such an example. SM_1 creates a tag T and sets the ACL to {*Code=*(C_r), *rw*} to grant access to all the other SMs using the C_r routing brick. Hence, SM_2 has the permissions to use T.

7 Implementation

To leverage on the existing user base, we have implemented our SM prototype in the Java programming environment over Linux. Specifically, we have modified Sun Microsystem's KVM (Kilobyte Virtual Machine) [17] because its source code is available and has a small memory footprint (i.e., it is suitable for resource constrained devices such as those encountered in NES). The SM API is encapsulated in two Java classes: *SmartMessage* and *TagSpace*. For efficiency, we have implemented the API as Java native methods. 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 [25], Scylla [26]), programming language, or underlying operating system.

In the rest of this section, we describe the most important components of our prototype implementation: the primitives for SM creation, the memory management mechanism which ensures thread-safety in KVM, the lightweight migration mechanism, the code caching, and the I/O tags. Currently, the admission manager is very simple; it accepts any SM as long as the destination node has enough memory to accommodate this SM.

7.1 Creating New Smart Messages

New SMs can be created at a node by the local injector or the VM interpreter. Each SM in the system is associated with a VM-level thread. The admission manager can also create VM-level threads for SMs arriving from the network.

A user can inject a new SM by passing a Java class name and a list of arguments to the local injector. The injector attempts to load, link, verify, and initialize the class file. Upon successful initialization, the injector creates a new VM-level thread with an initial stack frame for the *main* method of the class and inserts the thread into the ready queue. The arguments passed by the user are pushed onto the stack as arguments of the *main* method. At this point, the VM-level thread has no associated SM structure. When the VM-level thread starts its execution, it has to call *createSMFromFiles* to associate itself with a new SM

structure.

The interpreter thread also creates new VM-level threads in response to *createSM* and *spawnSM* invocations. When an SM calls *createSM*, the data bricks of the new SM are cloned from the current SM, and the code bricks of the new SM refer to the verified code bricks in the code cache. The *spawnSM* call is similar to *createSM*, except that the new SM starts its execution from the next bytecode after *spawnSM*. To implement this primitive, the execution stack frame associated with the VM-level thread of the original SM is duplicated onto the VM-level thread of the new SM.

7.2 Memory Management

The garbage collector in KVM is designed for a single-threaded environment. Since any of the three threads in SM prototype (i.e., interpreter, local injector, admission manager) could allocate memory from the dynamic heap, we protect the garbage collector data structures using a heap lock and restrict the garbage collection to a limited number of locations (i.e. *GC Points* [27]). We have modified the mark-sweep garbage collector in KVM such that garbage collection is performed by the interpreter only during context switches (i.e., the interpreter has a single *GC Point*). The interpreter triggers a garbage collection during a context switch if the available memory falls below a threshold. Before performing garbage collection, the VM ensures that the admission manager and the local injector threads have reached their *GC Points* (defined as the regions where all valid memory references are reachable from the garbage collector's root set). The *GC Points* of the three VM threads are demarcated using a single read-write lock. During garbage collection, the interpreter thread holds the write lock. The admission manager and the injector hold the read lock to protect the critical regions from garbage collection.

7.3 Lightweight Migration

One of the main obstacles in implementing an efficient execution migration arises from the strong coupling between the execution entity and the host. For example, traditional process migration needs to deal with sockets and file descriptors during migration. Two key features in the design of our system helped us circumvent the problem of strong coupling.

First, the tag space shields the SMs from direct coupling with the underlying OS. The read and write operations on tags are complete and atomic transactions; no state of the underlying OS resources is kept in the SM structure. Hence, an SM can be completely extracted from its execution environment, migrated, and resumed at destination.

Second, an SM program never creates a communication endpoint directly since it is based on execution migration, not message passing. Communication channels are managed implicit by the underlying system.

In contrast, traditional message passing programs create communication channels explicitly to transfer data. Hence, SM programs do not have any reference to OS network descriptors.

Our migration is *lightweight* in the sense that we do not migrate the complete memory referred to by SMs. Instead, we migrate data bricks which are explicitly identified in the SM. To simplify the task of programmers, we migrate, however, the *this* self-reference for non-static methods. Therefore, these methods can use object member variables safely after migration.

For clarity of exposition, we will describe the SM migration mechanism as three logical phases: *SM capture*, *SM transfer* and *SM resumption*.

7.3.1 SM Capture Phase

An SM enters into this phase when it invokes *sys_migrate* directly or as part of a routing library. In this phase, we convert the SM into a machine-independent representation. The code bricks are already in the machine-independent Java class format, and therefore, only the data bricks and execution stack frames need to be converted.

Data Brick Capture. To implement this conversion, we have developed a simple object serialization mechanism (i.e., KVM does not provide one). Each data brick is serialized into values and types representing its internal structure recursively. During serialization, we also generate a temporary structure which provides a unique identifier for each data brick reference. The unique identifiers of a data brick object and its sub-objects are determined solely by the structure of the data bricks.

Execution Control State Capture. The execution control state of an SM is represented by the execution stack frames of its associated VM-level thread. Each stack frame is serialized into a tuple of six values: current offset of *instruction* and *operand stack* pointers, method name, signature name, class name, and a flag indicating whether the method is non-static. For non-static methods, we also encode the machine-independent identifier for the *this* self-reference.

7.3.2 SM Transfer Phase

Using the data brick and stack information sizes obtained during the capture phase, the interpreter initiates a three-way handshake protocol with the destination node. The operation of this protocol is shown in Figure 10. If the SM is accepted, the admission manager sends back a list of missing code bricks as part of the *acknowledgment*. Otherwise, the admission just drops the request. Upon the receipt of the acknowledgment, the source node sends the complete SM, which consists of missing code bricks, serialized data bricks, and execution control state. To simplify the implementation, we have used TCP for reliable single-hop communication between neighbors. For better performance, we plan to change it into a reliable single-hop protocol

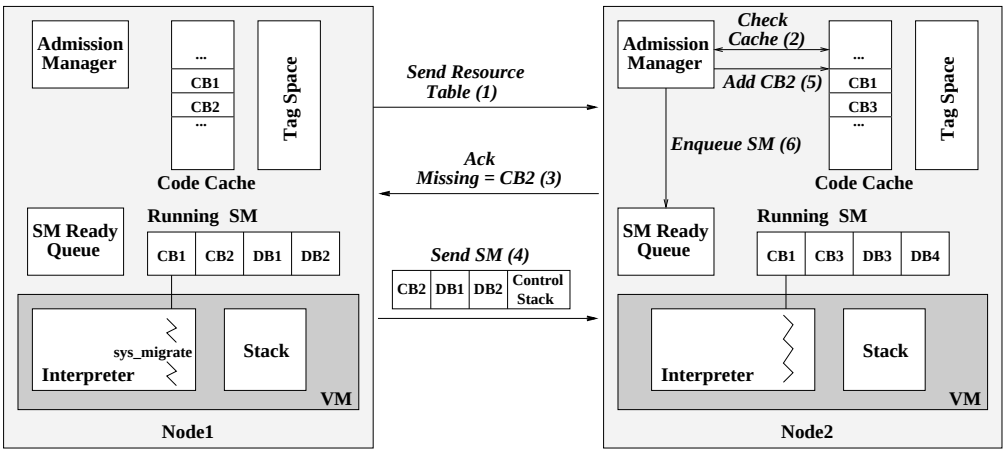


Figure 10: Smart Message Transfer (Main Operations)

over 802.11.

7.3.3 SM Resumption Phase

After the admission manager successfully received the code bricks, data bricks, and execution control information from a source node, a new VM-level thread and its associated SM structure are constructed. The missing code bricks sent from the source node are verified by the KVM verifier and stored in the code cache by the admission manager. We have modified the existing KVM class loader to search the code cache each time the VM needs a class. During data brick de-serialization, the admission manager constructs a temporary structure (similar to the structure constructed during the data brick capture at the source node) which maps a unique identifier to each data brick reference. The execution stack frames are reconstructed using the tuples sent from the source. Finally, the interpreter thread is notified if it is currently idle.

7.4 Code Caching

Each code cache entry consists of the Java class file of a code brick, a reference count, and a reference to the internal VM class representation. The original class format is stored for future migrations to nodes that do not have it cached. The reference count keeps track of the number of SMs currently referring to this code brick. Each time an SM referring to this code brick migrates or terminates, the reference count is decremented. When the reference count becomes zero, the code cache entry is moved to a free list. Should the same code brick be referenced by a new SM, the cache entry is resurrected from the free list. The memory associated with free list entries is reclaimed according to an LRU policy. When a cache entry is evicted, the code brick memory is freed, and the corresponding internal VM class representation is unloaded (since KVM does not have a class unloading capability, we have implemented our own class unloading mechanism).

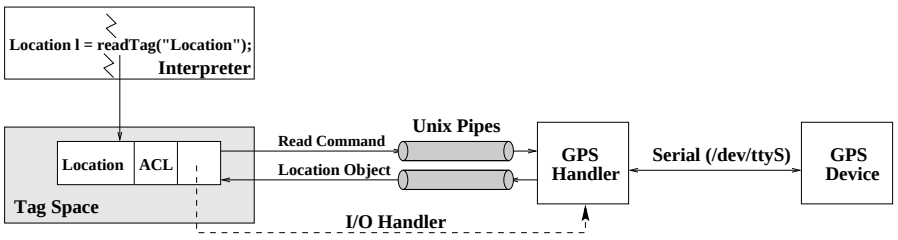


Figure 11: I/O Tag Example (Using GPS to Get the Current Location)

7.5 I/O Tags for Interaction with the OS and I/O System

An application uses the *readTag* and *writeTag* primitives to access an I/O tag. It is up to the system to define the source of the data, but *readTag* typically translates to an OS call. A *writeTag* translates to an OS call which sets certain parameters for an I/O device. Example of I/O tags currently available in our prototype can be found in Table 5.

Since each I/O tag requires specific native code, adding new I/O tags involves adding new native code to the node. We have identified three possible solutions for this issue. The first option is to statically link the native code into the VM. This is not viable because adding new I/O tags would involve shutting down the VM. The second option is to implement new I/O tags as dynamic shared libraries. This is not viable because we cannot assume that every node supports dynamic linking. The third option is to implement new I/O tags as external processes which communicate with the VM using a standard interface. We have chosen the third alternative since it enables users to dynamically extend the I/O tags without requiring the VM to be shut down or the host to support dynamic shared libraries. For efficiency, a few basic I/O tags (e.g., *free_memory* and *system_time*) are implemented and linked permanently into the VM executable.

Commonly, an I/O tag is associated with an external program, termed *handler*, which incorporates the code for reading and writing this I/O tag. When the VM receives a request to add a new I/O tag, it creates a new Unix process for this handler. We use Unix *pipes* for communication between VM and the handler process. Figure 11 shows the interaction between an SM and a handler process for a GPS device. When the SM issues a read request for the *Location* tag, the interpreter sends a *read* command to the handler and blocks waiting for an answer. Once the handler has obtained the data from the GPS device (connected on the serial port in our example), the handler encodes the data and sends it back to the VM. The VM de-serializes the results into a Java Object and returns it to the SM. A write operation is performed similarly.

Certain SMs may have a user interface (in the form of an external process) which allow users to interact with SMs via special I/O tags, termed UI tags. Unlike regular I/O tags, a UI tag behaves similar to a producer-consumer circular buffer. Each UI process can communicate with multiple SMs. This communication is done through a pair of UI tags: a *write* tag for passing data to SMs, and a *read* tag for receiving data

Size(KB)	Time(ms)	
	Uncached	Cached
1	2.622	0.032
2	5.112	0.034
4	9.953	0.042
8	20.151	0.063

Table 2: Effect of Code Brick Size on *createSMFromFiles*

Size(KB)	Time(ms)	
	spawnSM	*createSM*
2	0.270	0.243
4	0.367	0.326
8	0.508	0.469
16	0.913	0.822

Table 3: Effect of Data Brick Size on *spawnSM* and *createSM*

from SMs. These tags persist for the entire duration of the UI process.

8 SM Prototype Evaluation

To evaluate the performance of our prototype, we have measured the cost of the SM primitives: creating new SMs, migrating SMs between two nodes, and accessing the tag space. Our testbed consists of HP iPAQ 3870 running Linux 2.4.18. Each iPAQ contains an Intel StrongARM 206Mhz processor, 32MB flash memory, and 64MB RAM memory. For communication, we use Orinoco 802.11b Silver PC Cards.

8.1 Cost of SM Creation

createSMFromFiles. This primitive allows a user to inject a new SM at a node. After an invocation, the VM loads the class files from the local file system, unless the classes are already in the VM code cache, and creates a new SM structure. To evaluate its cost, we have performed two series of experiments. In the first, we invoke *createSMFromFiles* for an un-cached class of different sizes while keeping the data brick size constant (53 bytes). Then, we repeat the same experiment with the class cached. In both experiments, we have used 1KB class files and we varied the number of class files used to create an SM. Table 2 shows that the cost of *createSMFromFiles* almost doubles (when the code is not cached) as we double the size of the code brick. These results show that the cost of class loading dominates the cost of creating a new SM structure. The cost of creating a new SM structure is essentially the cost measured when the code is cached.

createSM and spawnSM. Table 3 shows the costs of *spawnSM* and *createSM* for different data brick sizes. The code brick and stack size are fixed at 1527 and 131 bytes, respectively. Typically, an SM has a mixture of static and non-static call frames. Therefore, we consider a stack consisting of two stack frames, one for a static method and the other for a virtual method call. Although these two primitives are similar, the results show that the cost of *spawnSM* is slightly higher than the cost of *createSM*. The difference is the time spent to duplicate the execution stack frames for *spawnSM*.

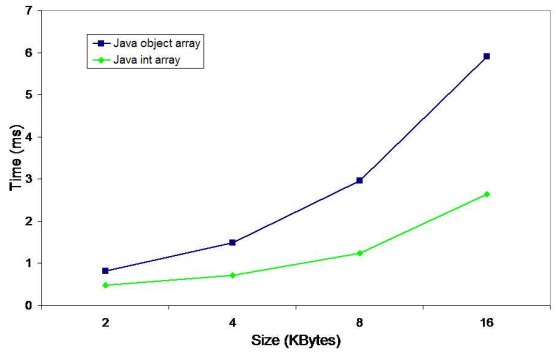


Figure 12: Cost of Data Brick Serialization

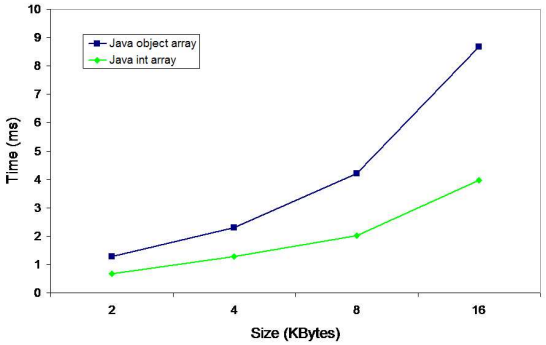


Figure 13: Cost of Data Brick De-Serialization

8.2 Cost of SM Migration

The most significant factors that determine the cost of our migration are the data brick serialization, the SM transfer, and data brick de-serialization.

Data Brick Serialization and De-Serialization. Since the code bricks need not be serialized, we perform this operation only on data bricks and execution stack frames. Our measurements indicate that the serialization cost for the execution stack frames is small compared to the cost of data brick serialization; it varies from 0.204ms to 0.567ms as we vary the execution stack from 2 to 15 frames. To study the effect of data brick serialization, we vary the data brick size from 2KB to 16KB, while using a fixed size code brick (1197 bytes) and two fixed size stack frames (131 bytes).

Commonly, the data bricks in an SM consist of a mixture of objects and primitive types. We use two types of data bricks in this evaluation: an array of integers, and an array of objects. The serialization costs for these two data bricks provide practical lower and upper bounds for the cost of data brick serialization. The object array represents an upper bound since each of its elements causes a call to the top level VM serialization method. The integer array represents a lower bound since it involves only one call to the top level VM serialization method.

Figure 12 shows that the serialization cost is below 6ms for data bricks as large as 16KB. Commonly, the SMs process data at its source, and therefore, they carry small size data. The applications that we have developed carry less than 2KB, which costs less than 1ms to serialize. Figure 13 presents the de-serialization cost for the same data bricks. We observe that de-serialization cost is as much as 30% higher than the cost of serialization due to memory allocation during object de-serialization.

SM Transfer. The variation of execution control state size is small compared to that of code bricks and data bricks. Thus, we only consider the effect of code bricks and data bricks in the subsequent experiments. We have performed two sets of experiments to evaluate the cost of migration (serialization, transfer, de-

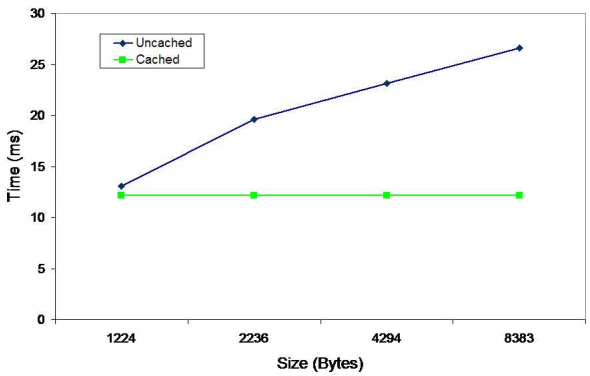


Figure 14: Effect of Code Brick Size on Single Hop Migration

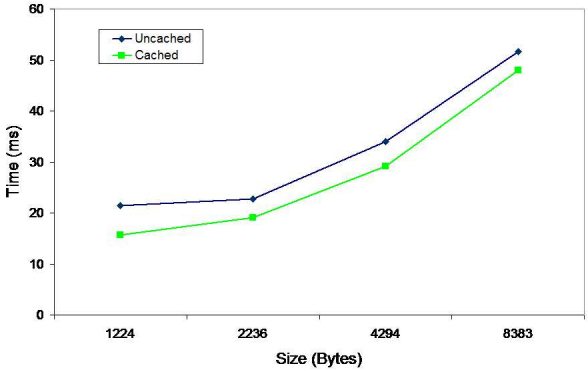


Figure 15: Effect of Data Brick Size on Single Hop Migration

serialization) for different code brick and data brick sizes. In the first set, we vary 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 experiment, we vary the data brick size while keeping the code brick size and stack frame size fixed at 1197 bytes and 131 bytes. Figures 14 and 15 show the results of these two experiments.

The values in Figure 14 represent the total time for single hop migration in two situations: the code is not cached, and the code cached. The time to transfer the SM when the code is cached is constant and represents the overhead of the three-way handshake protocol. Figure 15 shows that the data brick size contributes significantly to the total cost of migration. Thus, it is important to have a serialization scheme with minimal space overhead.

8.3 Tag Space Operations

Table 4 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 operations; when an SM reads a tag, the interpreter acquires a lock, performs a lookup in the tag space, verifies the access rights, and returns the data to the SM. The *writeTag* operation costs slightly more since the interpreter has to check for and unblock any SMs blocked on the tag. The *blockSM* operation costs more than both *readTag* and *writeTag* since it also needs to append the SM to the SM blocked queue and suspend the VM-level thread. The *deleteTag* primitive has the second 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, while the *createTag* primitive has the highest cost since it involves additional steps to register a timer for the tag lifetime and create access control data structures.

Table 5 presents the access time to several I/O tags that are currently implemented in our prototype: GPS

Operation	Time(μs)
createTag	101.781
deleteTag	75.071
readTag	34.548
writeTag	50.289
blockSM	59.844

Table 4: Cost of Tag Space Primitives for Application 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

Table 5: Cost of Reading I/O Tags

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 which reads from the GPS serial interface. The location of the neighbors along with their identifiers are 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 I/O handler converts the image received from camera in YUYV format to RGB format before returning it to the SM. All the other tag values are obtained directly from Linux using system calls.

9 Case Study: EZCab

We envision that the use of embedded devices in cars will soon become a reality [28, 29]. To demonstrate the feasibility of the SM computing platform for real-world applications, we have developed EZCab, an application for locating and booking free cabs in densely crowded traffic environments (like Manhattan, where looking for a free cab can be an annoying experience). Instead of calling a cab company or merely "gesturing" to negotiate a cab for her destination, a client can simply inject an SM through her handheld device to perform seamlessly the same action. Unlike the existing solutions for inter-car communication that are based on certain infrastructures (which are expensive, cannot be deployed on every road, and provide only limited information), EZCab uses a peer-to-peer approach whose key benefits are scalability and practicality. The minimal infrastructure needed by EZCab is the availability of the SM support in the cabs, a location service (e.g., GPS), and wireless connectivity.

The main component of EZCab is an SM that migrates to a cab identified by a *FreeCab* tag, negotiates the price according to a client-established limit, let the cab know the identity of the client, and instructs the cab to go to the client's location. The booking is complete after the cab sends a message with its identification to the client, and the client acknowledges this message. When the cab arrives at the client's

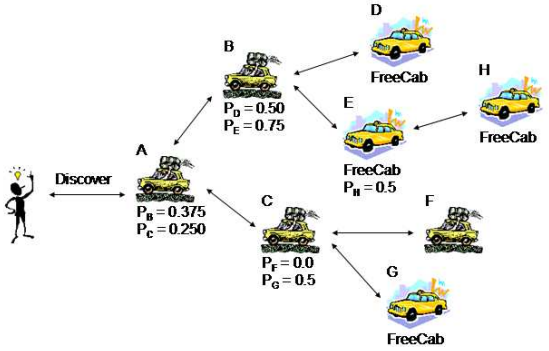


Figure 16: Route Discovery

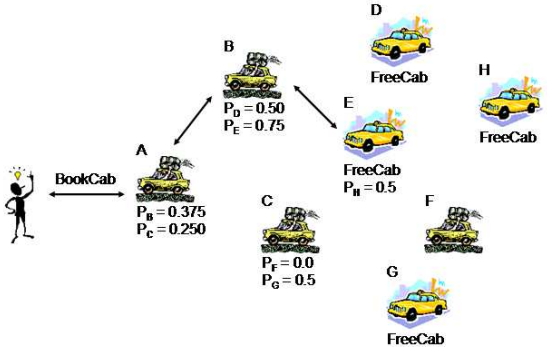


Figure 17: Cab Booking

location, a validation process takes place to ensure that the client gets her booked cab (and the cab takes the client that booked it). In the following, we present a brief description of the basic operations in EZCab: (1) discovering the routes to free cabs, (2) booking a free cab, and (3) performing the validation between the cab and the client. We conclude the section with an analysis of our application based on experimental results.

9.1 Route Discovery

The EZCab application starts at the client node and takes as parameter the radius of the circular geographical region to be covered (the maximum number of hops, *maxHops*, for which any EZCab SM is allowed to migrate is computed based on this radius and the transmission range of nodes). To reach a free cab, the SM uses routing tables that specify the next hop as the probability to reach a free cab from the current node (similar to probabilistic routing [30]). If the probability to find a free cab using the existing routes is too low, or there are no routes at all, the SM creates a *route discovery SM* and blocks waiting for routes (Figure 16 illustrates this process). The code for route discovery is presented in Figure 18. At each node, the discovery SM spawns and blocks (lines 9-11), while the child SM migrates to all the neighbor nodes (line 15). Each SM migrates through the network until it arrives at a node already visited by another discovery SM (i.e., it ends its execution) or it reaches the maximum number of hops that it is allowed to migrate. Once this threshold is reached, the SM migrates back one hop and reports its current information (lines 19-28). This is a recursive process that builds the routing tables at nodes. We have chosen to wait for replies for a given period of time because it is difficult to wait for a fixed number of replies in a volatile network (i.e., those replies may never arrive).

```
1   int probability, status = 0, nHops = 0; // stored in data bricks
2   Address parent, nextHop, source;        // stored in data bricks
3   source = SmartMessage.getLocalAddress();
4   while (nHops < maxHops){
5      if (tagExists("Visited"))
6         System.exit();
7      nHops++;
8      parent = SmartMessage.getLocalAddress();
9      if (SmartMessage.spawnSM()){
10        TagSpace.createTag("Visited", Lifetime, null)
11        TagSpace.blockSM("Discovery", DiscoveryTimeout);
12        break;
13     }
14     else
15        SmartMessage.sys_migrate(AllNeighbors);
16  }
17  if (tagExists("FreeCab"))
18       status = 1;
19  if (nHops == maxHops)
20     probability = status/2;
21  else{
22     if (parent.equals(source))
23        return;
24     probability = (status + maxProbabilityReported("routingTable"))/2;
25  }
26  nextHop = SmartMessage.getLocalAddress();
27  SmartMessage.sys_migrate(parent);
28  addEntry("routingTable", nextHop, probability);
```

Figure 18: Route Discovery Code

9.2 Cab Booking

Booking a cab is a three-way handshake protocol. If a node has routes to free cabs, the application creates a *booking SM* to find a free cab and blocks for a certain amount of time. If the cab is not free, the booking SM chooses the next neighbor greedily (i.e., using the greatest probability in the routing table), as shown in Figure 17. Once a free cab is found, the SM removes the *FreeCab* tag, writes the client's location in the *Location* tag, and creates a *reporting SM* to confirm the booking with the client. Then, it blocks at the cab waiting for an acknowledgment from the client.

The reporting SM migrates to the client's location using geographical routing to improve the efficiency. Once it has informed the client that a cab is on its way, it returns to the cab with an acknowledgment to let the cab know that the handshake has succeeded. If no reply is received from a cab after a timeout, EZCab will re-start with a new best route. Consequently, the booking SM waiting at the cab times out and re-creates a *FreeCab* tag to reflect the change in the cab's status.

9.3 Validation

Upon reaching the client's location, the validation mechanism is initiated. To make the validation possible, the booking SM carries the public key of the client to the cab, and the reporting SM carries the public key of the cab to the client. To validate the client, the cab broadcasts a challenge in the zone by encrypting a text using the client's public key. The client, upon receiving the encrypted text, decrypts it using its private key. In turn, it uses the cab's public key to encrypt the text again and send it to the cab. If the reply text is identical, the client is validated.

9.4 Analysis

For EZCab, it is of particular importance to evaluate its completion time given realistic configurations. In the following, we present an analysis which demonstrates that EZCab can cover a circular area up to 1km radius around the client's location, and the user-perceived response time is less than 2 seconds. Figure 19 shows our EZCab prototype.

The first part of our evaluation tries to determine the maximum distance at which two moving cars can communicate and the time for which the topology is relatively stable. Using two HP iPAQs with 802.11 cards for communication, and various mobility scenarios (as much as 170km/h relative speed between two cars moving in opposite directions), we have experienced a substantial increase in the packet loss rate for distances bigger than 60m. We consider this distance feasible for our target networks (we have also experimented with external antennas and amplifiers to increase to range as much as 400m). Given this distance, two cars are in the communication range of each other for approximately 2 seconds at a relative speed of 120km/h (i.e., typical speed for two cars moving in opposite directions in a crowded city). Therefore, our application should complete faster than that in order to reduce the effects of mobility on the established routes.

The second part of our evaluation tries to see if EZCab can finish using this time bound, and how big the geographical region covered by our application is (i.e., a bigger region increases the probability to locate a free cab). The response time for EZCab is defined as the time spent until the client receives a confirmation from the cab. The design of EZCab makes it easy to bound this response time. All the main operations (route discovery, booking a cab, and reporting a booked cab) are bounded by a timeout. Therefore, the maximum response time for a successful booking is the sum of the timeouts for route discovery and booking a cab.

We compute the timeouts for each SM generated by EZCab as the products of the round trip time of each SM between two nodes (RTT) and the maximum number of hops traveled by an SM ($maxHops$). We further assume that all cabs have the code cached. SMs transfer only small size data bricks and execution

Figure 19: EZCab Prototype

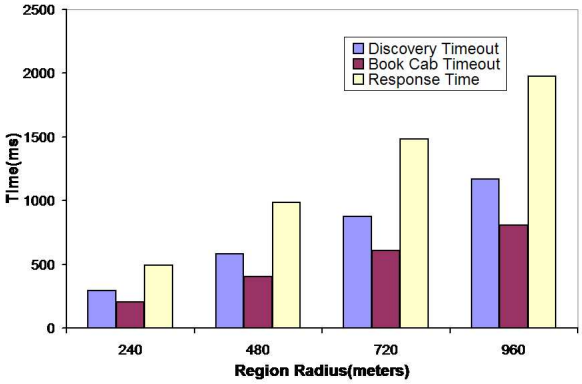


Figure 20: Estimated Completion Time for EZCab

control state when the code is cached. Hence, the measured values of the RTTs for the three SMs are almost identical (24.3ms, 25.4ms, 25.1ms). To include the costs of SM execution and wireless contention, we consider, conservatively, a value three times greater. Since booking a cab and reporting back to the client do not involve any broadcast, we just double the minimum timeout value for booking a cab and reporting back to the client.

Figure 20 shows the response time as a function of the size of the covered region. The results indicate that EZCab can finish in less than 2 seconds for a region radius of approximately 1km even if it needs to perform a route discovery. As determined by the first part of our evaluation, we expect the network topology to be relatively stable during this time period. Therefore, we conclude that SMs offer the flexibility to program the EZCab application without any infrastructure, and the analysis of the actual SM implementation demonstrates the feasibility of EZCab for densely populated cities.

10 Related Work

The SM platform shares the idea of execution migration with process migration [31–33], mobile agents [15,16], and active networks [10–12].

Unlike process migration which has been used to increase performance or availability in stable networks, the main goal of the SM platform is to provide flexible support for programming distributed applications over highly dynamic NES. Additionally, process migration and SM migration differ in two aspects. First, the SM migration is explicit (i.e., the programmer decides when and where to migrate), while process migration is implicit (i.e., the system decides when and where to migrate a process). Second, the SM architecture avoids one of the most difficult problems in process migration: transferring the kernel state (e.g., sockets,

file descriptors). The SM platform does not transfer any kernel state because SMs interact with local hosts through atomic operations performed on the tag space, and they do not open explicitly communication channels.

SMs are influenced by the design of mobile agents. Similar to a mobile agent, an SM may be viewed as an application that explicitly migrates between nodes of interest. Mobile agents, however, name nodes by fixed addresses and commonly know the network configuration a priori, while SMs name nodes by content and discovers the network configuration dynamically. In contrast to mobile agents, SMs are responsible for their own routing at each node in the path between two nodes of interest. This feature allows SMs to adapt quickly to changes that may occur both in the network topology and the availability of resources at nodes. Furthermore, the SM system architecture is suitable for resource constrained devices since it defines a lightweight system support at nodes, with most of the "intelligence" incorporated into SMs.

Although the SM computing platform (especially the self-routing mechanism), shares some of the design goals and leverages work done in active networks (AN), it differs from AN in several key features. A first difference comes from the problems they try to solve: AN target improved performance for end-to-end data transfer in relatively stable networks, while the SM platform helps the development of distributed applications on top of a new computing infrastructure which is significantly under-used due to the lack of programmability support. Unlike AN, we define a computing model whereby several SMs can cooperate, exchange data, and synchronize with each other through the tag space. In terms of migration, AN do not transfer the execution state from node to node whereas the SM model does. The migration of the execution state for SMs trades off overhead for flexibility to react "on-the-spot" to adverse network conditions.

Sensor networks represent the first attempt toward deploying large scale NES. Most of the research in this area has focused on hardware [4, 34], operating systems [1], or network protocols [3, 5, 8]. Even though sensor networks act primarily as huge distributed databases [35, 36], more sophisticated applications might be needed in the future. Toward this end, SensorWare [37] and Mate [25] have proposed solutions for network re-programmability. The SM architecture takes one step further and proposes a distributed computing model that is flexible enough to be implemented for nodes with very limited resources such as those encountered in sensor networks.

Among many projects that target the programmability of ubiquitous computing environments [38–42], *one.world* [38] is similar to our work in the sense that both consider migration as an essential mechanism to adapt to highly dynamic computing environments. Each application in *one.world* has at least one environment that contains tuples (similar to application tags in SM platform), application's components, and other nested environments. When needed, a migration moves a checkpointed copy of an environment to another node. A significant difference between SMs and *one.world* is that our work proposes a computing

model based on execution migration, while *one.world* uses migration just as a mechanism to adapt to changes (i.e., in their programming model, the applications reside on nodes and communicate through remote event passing). Another difference is that the SM architecture is more suitable for resource constrained devices whereas *one.world* is designed for more powerful nodes.

Smart Message is the underlying platform for Spatial Views, a high-level programming model for networks of embedded systems, targeting its dynamic, space-sensitive and resource-restrained characteristics. The core of the model is iterative programming over a dynamic collection of nodes identified by the physical spaces they are in and the services they provide. Hidden in the iteration is execution migration, as the main collaboration paradigm, constrained by user specified limits on resource usage such as response time and energy consumption. A Spatial Views prototype has been implemented and first results are reported in [43]. A Spatial Views compiler with Smart Messages as its target is currently being implemented.

The tag space bears some similarity with tuple spaces [44,45]. While both offer persistent shared memory for applications, the essential difference is that the tag space is local to each node. Also, unlike tuple spaces, the tag space provides SMs with I/O tags for interaction with the local OS and I/O subsystem. The concept of I/O tags share the same goal with *Linux Procfs* [46] which allows user-level programs to access certain kernel information.

Content-based naming has been recently presented for both the Internet [9,47,48] and sensor networks [2]. SMs use content-based migration to reach the nodes of interest. This high-level migration function implements routing algorithms which leverage work done for mobile ad hoc networks [20, 21, 49].

Although the security for both mobile agents [24, 50] and ad hoc networks [51, 52] have been extensively studied, we have faced a new and more difficult problem: how to define a security architecture for a system based on execution migration over mobile ad hoc networks? Given the complexity of this problem, our current architecture provides solutions for protecting the hosts against SMs and SMs against each other. It is much harder, however, to prevent an SM from being tampered by a malicious host. Since SMs have to execute at any host, end-to-end authentication based on digital signatures or encrypting the entire message are not possible. Hardware solutions [53, 54] represent an option, but they involve extra-costs. Complete software solutions are not known yet, but code confusion and encryption techniques have been investigated [55,56] in the context of mobile agents.

Coupled with security comes the issue of admission control at nodes. A significant amount of research has been done to solve this problem for real time systems [57,58] and active networks [11,59]. Given that we did not want to limit the expressibility of the programming language (e.g., SNAP [11]), our solution is based on user-provided lower bounds for resources and non-preemptive execution. Each node has the flexibility to implement its own scheduling and resource allocation policies which are typically integrated. These policies

guarantee enough resources to satisfy the lower bounds and let the SM migrate in case no more resources are allocated. A problem that remains to be solved is how to protect the network, as a whole, against malicious SMs that waste network resources, but respect the admission contract at each node. TTL-based [10] or market-based [24] schemes offer possible solutions.

11 Conclusions

In this paper, we have presented the Smart Messages (SMs) platform for distributed computing in networks of embedded systems (NES). SMs are distributed applications which overcome the scale, heterogeneity, and volatility encountered in NES by migrating the execution to nodes of interest, using application-controlled routing, instead of using end-to-end communication among nodes. The main feature of the SM programming model is its high flexibility in the presence of dynamic network configurations. The experimental results as well as the analysis of our real world application (EZCab) indicate that the SM computing platform can be a feasible solution for programming NES.

Acknowledgments

The authors would like to thank Deepa Iyer for her contribution in developing the SM prototype, and Chalermek Intanagonwiwat for our useful discussions regarding the SM design. We would also like to thank the anonymous reviewers which helped us improve this paper.

References

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

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

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

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

[5] B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, S. Son, and J. Stankovic, "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.

[6] "Linux Devices," http://www.linuxdevices.com.

[7] C. Wan, A. Campbell, and L. Krishnamurthy, "PSFQ: A Reliable Transport Protocol For Wireless Sensor Networks," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications (WSNA 2002)*, Atlanta, GA, September 2002, pp. 1–11, ACM Press, New York, NY.

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

[9] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The Design and Implementation of an Intentional Naming System," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP 1999)*, Charleston, SC, 1999, pp. 186–201, ACM Press, New York, NY.

[10] 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, pp. 64–79, ACM Press, New York, NY.

[11] J. Moore, M. Hicks, and S. Nettles, "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.

[12] B. Schwartz, A. Jackson, W. Strayer, W. Zhou, R. Rockwell, and C. Partridge, "Smart packets: Applying active networks to network management," *ACM Transactions on Computer Systems*, vol. 18, no. 1, pp. 67–88, 2000.

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

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

[15] N. Karnik and A. Tripathi, "Agent Server Architecture for the Ajanta Mobile-Agent System," in *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, Las Vegas, NV, July 1998, pp. 66–73.

[16] R. Gray, G. Cybenko, D. Kotz, and D. Rus, "Mobile Agents: Motivations and State of The Art," in *Handbook of Agent Technology*, Jeffrey Bradshaw, Ed. AAAI/MIT Press, 2002.

[17] "K Virtual Machine," http://java.sun.com/products/cldc/.

[18] "XML," http://www.w3.org/XML/.

[19] "JavaSpaces," http://wwws.sun.com/software/jini/specs/jini1.1html/js-title.html.

[20] C. Perkins and E. Royer, "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.

[21] B. Karp and H.T. Kung, "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, pp. 243–254, ACM Press, New York, NY.

[22] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communication of the ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[23] V. Cahill and et al, "Using Trust for Secure Collaboration in Uncertain Environments," in *Pervasive Computing, IEEE*, 2003, vol. 2(3), pp. 52–61.

[24] R. Gray, D. Kotz, G. Cybenko, and D. Rus, "D'Agents: Security in a Multiple-Language, Mobile-Agent System," in *Mobile Agents and Security*, Giovanni Vigna, Ed., vol. 1419 of *Lecture Notes in Computer Science*, pp. 154–187. Springer-Verlag, London, UK, 1998.

[25] P. Levis and D. Culler, "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, pp. 85–95, ACM Press, New York, NY.

[26] P. Stanley-Marbell and L. Iftode, "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.

[27] O. Agesen, "GC Points in a Threaded Environment," Tech. Rep. SMLI TR-98-70, Sun Microsystems Laboratories, Palo Alto, CA, December 1998.

[28] R. Morris, J. Jannotti, F. Kaashoek, J. Li, and D. Decouto, "CarNet: A Scalable Ad Hoc Wireless Network System," in *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000, pp. 61–65, ACM Press, New York, NY.

[29] P. Koopman, "Critical Embedded Automotive Networks," *IEEE Micro*, vol. 22, no. 4, pp. 14–18, July-August 2002.

[30] S. Rhea and J. Kubiatowicz, "Probabilistic Location and Routing," in *Proceedings of the 21th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'02)*, New York, NY, June 2002, pp. 1248–1257.

[31] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, September 2000.

[32] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite Network Operating System," *IEEE Computer*, vol. 21, no. 2, pp. 23–36, February 1988.

[33] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," in *Proceedings of the USENIX 1986 Summer Conference*, Atlanta, GA, July 1986, pp. 93–113.

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

[35] P. Bonnet, J. E. Gehrke, and P. Seshadri, "Querying the Physical World," *IEEE Personal Communications*, vol. 7, no. 5, pp. 10–15, October 2000.

[36] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "The Design of an Acquisitional Query Processor for Sensor Networks," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, San Diego, CA, June 2003, pp. 491–502, ACM Press, New York, NY.

[37] A. Boulis, C.C. Han, and M.B. Srivastava, "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.

[38] R. Grimm and et al, "Systems Directions for Pervasive Computing," in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau/Oberbayern, Germany, May 2001, pp. 147–151, IEEE Computer Society, Washington, DC.

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

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

[41] S. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd, "ICrafter: A Service Framework for Ubiquitous Computing Environments," in *Proceedings of the Third International Conference on Ubiquitous Computing (Ubicomp)*, Atlanta, GA, September 2001, pp. 56–75, Springer-Verlag, London, UK.

[42] M. Roman and R. Campbell, "GAIA: Enabling Active Spaces," in *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000, pp. 229–234, ACM Press, New York, NY.

[43] Y. Ni, U. Kremer, and L. Iftode, "Spatial Views: Space-Aware Programming for Networks of Embedded Systems," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, College Station, TX, October 2003.

[44] N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM*, vol. 32, no. 4, pp. 444–458, April 1989.

[45] T. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman, "Hitting the Distributed Computing Sweet Spot with TSpaces," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 35, no. 4, pp. 457–472, March 2001.

[46] "Linux Kernel Procfs," http://www.kernelnewbies.org/documents/kdoc/procfs-guide/intro.html.

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

[48] M. Gritter and D. Cheriton, "An Architecture for Content Routing Support in the Internet," in *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS 2001)*, San Francisco, CA, March 2001, pp. 37–48.

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

[50] N. Karnik and A. Tripathi, "Security in the Ajanta Mobile Agent System," *Software Practice and Experience*, vol. 31, no. 4, pp. 301–329, January 2001.

[51] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. Tygar, "SPINS: Security Protocols for Sensor Netowrks," in *Proceedings of the 7th annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2001)*, Rome, Italy, July 2001, pp. 189–199, ACM Press, New York, NY.

[52] Y. Hu, A. Perrig, and D. Johnson, "Ariadne: A Secure On-Demand Routing Protocol for Ad-Hoc Networks," in *Proceedings of the 8th annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2002)*, Atlanta, GA, September 2002, pp. 12–23, ACM Press, New York, NY.

[53] "Trusted Computing," http://www.cl.cam.ac.uk/ rja14/tcpa-faq.html.

[54] E. Palmer, "An Introduction to Citadel - A Secure Cypto Coprocessor for Workstations," in *Proceedings of IFIP SEC'94 Conference*, Curacao, Dutch Antilles, May 1994.

[55] F.Hohl, "Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts," in *Mobile Agents and Security*, Giovanni Vigna, Ed., vol. 1419 of *Lecture Notes in Computer Science*, pp. 92–113. Springer-Verlag, London, UK, 1998.

[56] T. Sander and C.F. Tschudin, "Protecting Mobile Agents Against Malicious Hosts," in *Mobile Agents and Security*, Giovanni Vigna, Ed., vol. 1419 of *Lecture Notes in Computer Science*, pp. 44–60. Springer-Verlag, 1998.

[57] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, vol. 8, pp. 62–72, May 1991.

[58] D. Rosu, K. Schwan, and S. Yalamanchili, "FARA - A Framework for Adaptive Resource Allocation in Complex Real-Time Systems," in *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, Denver, CO, May 1998, pp. 79–84.

[59] V. Galtier, K. Mills, Y. Carlinet, S. Bush, and A. Kulkarni, "Predicting Resource Demand in Heterogeneous Active Networks," in *Military Communications Conference, 2001 (MILCOM 2001). Communications for Network-Centric Operations: Creating the Information Force*, Washington, D.C., October 2001, pp. 905–909.