Self-Routing in Networks of Embedded Systems using Smart Messages *

Cristian Borcea Akhilesh Saxena Deepa Iyer
Porlin Kang Rabita Sarker Liviu Iftode†

Division of Computer and Information Sciences
Rutgers University
Piscataway, NJ, 08854
{borcea, saxena, iyer, kangp, sarker, iftode}@cs.rutgers.edu

Abstract

Smart Messages provide a simple, yet flexible model for programming distributed applications in massive networks of embedded systems. A Smart Message (SM) is a dynamic collection of code and data that migrates through the network, routes itself at each node in the path, and executes on nodes of interest. The nodes of interest are named by properties or content. A key challenge in this model is the ability to discover target nodes, and to route SMs to them.

In this paper we describe the SM self-routing mechanism, which provides high flexibility, scalability, and resilience to adverse network conditions. Using this mechanism, applications can choose among multiple content-based routing schemes, or even define the best suited routing algorithm for their needs. We present proof-of-concept implementation, simulation results, and analysis for three classes of content-based routing algorithms. We also show preliminary results for an SM prototype implementation on Compaq iPAQs using Wavelan 802.11 and Bluetooth for communication.

1. Introduction

Decreasing costs of computing and networking technology open the path towards ubiquitous computing [31]. Incorporating intelligence in any device encountered in our daily routine, as well as providing them with wireless communication, creates the possibility of building large scale networks of embedded systems (NES). These nodes will inherently be heterogeneous. In the near future, we envision home appliances communicating to handle certain domestic activities [2], intelligent cameras collaborating to track a given object, cars on a highway cooperating to adapt to the traffic conditions [3]. Sensor networks [10, 15, 16] have represented the first step towards this vision. The target systems we assume, are more powerful than the nodes in sensornets, in terms of processing power, memory, and network bandwidth. Energy remains an important issue for certain classes of networks. However, in some situations the energy can be provided by a permanent source of power (e.g. home appliances), or the battery can be recharged by users (e.g. handheld devices).

Harnessing such a huge computing infrastructure to execute distributed applications is a research issue that will face us during the next decade, as parallel computing did two decades ago. Merely incorporating processing power and communication capabilities in any device does not solve the programmability problem. To program NES, two main issues have to be solved: (1) how to describe a distributed computation when the network configuration is unknown and volatile, and (2) how to perform flexible naming and routing in these networks.

Data collection and dissemination in sensor networks [15, 19] are examples of simple applications for programming such large scale networks. These applications are developed during the network deployment phase, and it becomes almost impossible to modify the configurations or the protocols on the individual nodes thereafter. Scale and heterogeneity preclude updating all nodes for a new protocol or application. To be able to implement more complex distributed applications, a new computing model must be employed. Traditional distributed computing models are not ap-

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

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

propriate for NES volatility, since they are based on a distribution of tasks across a stable network of similar processing units. Recently, we have proposed Cooperative Computing [6] as a solution for programming user-defined distributed applications in unstable networks with unknown configurations, using application-controlled routing.

The applications running in NES will target specific data or services within the network, not individual nodes. From an application point of view, nodes with the same properties are interchangeable. Fixed naming schemes, such as IP addressing, will be almost irrelevant in most situations. For example, it might be desirable to reach a node that has a temperature sensor, but a fixed binding between the desired property and a unique identifier for a node is inappropriate. If the destination node becomes unavailable, the routing fails, even though multiple nodes providing the same property are available. The need to target specific data or properties within the network raises the issue of a different naming scheme with dynamic bindings between a name and a node address. Recent research has addressed this issue for both Internet and sensor networks [4, 13, 17]. We believe that a naming scheme based on content or properties is more suitable for NES than a fixed naming scheme.

The sheer number of nodes preclude any fixed infrastructure, favoring ad hoc networks formed on-the-fly. Even if the cost of building a fixed infrastructure would not be prohibitive, this solution could hardly work since many nodes are mobile, and others can simply cease to exist due to failures, energy exhaustion, or disposal. In these large scale, volatile networks, it is practically impossible to acquire global knowledge. Since most of the nodes have wireless communication capabilities, they will be able to interact directly only with nodes in their transmission range. Therefore, routing algorithms that achieve global results using only local knowledge are needed. Different applications in NES can have different routing requirements. For example, it might be necessary to route one type of data based on geographic information, and another one based on a certain content name. An application may also need to change the routing dynamically, as different network conditions are encountered. Therefore, the flexibility to use different routing algorithms in the same network is desirable.

End-to-end semantics is hard to achieve in NES. We expect that round-trip communication will fail in many situations, leading to timeouts or even failures of applications since the current programming models treat the unavailability of certain destinations as exceptions. It is important to allow applications to adapt their requirements to the conditions encountered in the network.

The conclusion of the above discussion is that a flexible, application-controlled routing mechanism is needed for networks of embedded system. The main requirements for it are: generality, capability to perform application-specific content-based routing, ability to adapt to adverse network conditions, and easiness of implementation.

In this paper, we propose a mechanism for self-routing in NES using our Smart Messages architecture [29]. Smart Messages (SM) are collections of code and data that migrate through the network, one hop at a time, executing at each node. [1] Nodes in the network support SMs by providing a simple, architecturally independent environment for the receipt and execution of SMs, and a name-based data storage, called Tag Space, consisting of tags persistent across SM executions. The tag names are also used for content-based naming in NES. The nodes cache code upon SM acceptance, and subsequent SMs will avoid transferring code in the common case. Since no routing infrastructure is provided, SMs are self-routing, namely they are responsible for determining their own paths through the network, utilizing the minimal set of facilities provided by nodes. The routing code is just another SM code "brick", usually cached at nodes. Instead of end-to-end data transfer, the SM approach proposes an easy to use, flexible, and resilient "store and forward" content-based routing mechanism. The applications need to execute on target nodes defined by tag names, and in doing so they migrate to destinations using application controlled routing executed at intermediate nodes.

During SM migrations, routing can be completely transparent to the application programmer. The nodes of interest are specified in terms of tag names, and applications can link to pre-defined routing algorithms that promise to take them to a node that contains the desired tags. The routing algorithm takes control, migrates the entire SM to a destination using its policies, and upon arrival returns the control to application. Flexibility is achieved by allowing applications to choose the best suited routing algorithm for their needs, or even to implement a new routing algorithm.

SMs are resilient to network faults, being able to control the routing, to find alternative routes to destination, to discover similar nodes of interest, or to adapt their requirements to adverse network conditions as long as a certain quality of result is met.

The remainder of this paper is organized as follows.

[1] The similarities and differences between Smart Messages and active networks [9, 27, 25], Active Messages [30], and mobile agents [11, 32, 18, 24] are discussed in the Related Work section.

Section 2 describes the Smart Messages model and architecture. In section 3, we present the SM self-routing mechanism. The implementation of the content-based routing algorithms is discussed in section 4. We show the simulation and micro-benchmark results in section 5. Section 6 discusses the related work, and we conclude in section 7.

2. Smart Messages

Smart Messages (SM) are migratory units consisting of dynamically assembled code and data sections, termed "bricks", and a lightweight execution state. Each code brick is an independent program that may be used together with the other code and data bricks to generate a new, possibly smaller SM. The data bricks contain the mobile data carried by SMs during migrations. SMs migrate through the network, searching for nodes of interest, and execute at each node in the path. The SM execution is embodied in tasks described in terms of migration and computation phases.

As opposed to request/reply paradigm, applications in the SM model need to migrate to the nodes of interest, and execute on these nodes in order to achieve their prescribed objectives. SMs execute a routing algorithm on all nodes in the path towards a destination. For example, on nodes of interest, an application may need to read and process sensor data, or it may perform management functions, or even dynamically create/remove services. Placing intelligence in SMs provides flexibility and obviates the need for the potentially impossible task of implementing a new application or protocol [19].

A legitimate question that can be raised is: what is the overhead of transferring code through the network ? In the common case, the code is not transferred, since it is cached at the nodes by previous SMs. In time, the cost of transferring code is amortized, since many NES applications have temporal and spatial locality. Moreover, moving code to nodes of interest improves the performance for certain class of applications that need to process big amounts of data. For example, bandwidth, energy consumption, and response time are all minimized for an object tracking SM that performs image analysis at the node that acquired an image, provided that the node has enough computing power.

Another issue in the SM model is security. To solve it, individual nodes should be protected against SMs, groups of nodes should be protected against SMs that might consume excessive resources in the network, and SMs should be protected against the nodes. Solutions and alternatives for similar problems are presented in [12]. Although defining a security architecture is important, it is outside the scope of this paper.

Figure 1(a) shows an SM, injected in the network at the circular node in the lower left corner, that needs to execute on the square nodes. The network does not maintain any routing infrastructure, and the SM is responsible for determining a path to its destinations.

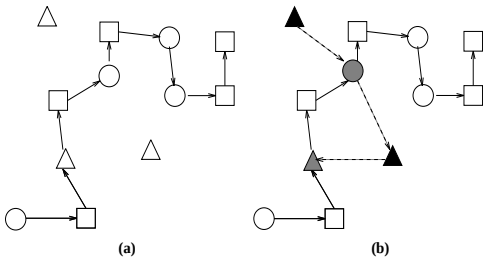

Figure 1. Smart Messages Example

A network may simultaneously contain several executing applications, as illustrated in Figure 1(b). Nodes of interest to the second application are depicted with triangles, and nodes exclusive to its path are colored black. Nodes in the network which may at some point be either part of the first or second application are colored grey. For example, the grey triangle is just a *stepping stone* on the path of the the first application, but is a node of interest to the second application.

To overcome the heterogeneity issue in NES, the SM system architecture defines a minimal, common support that each node must provide. Figure 2 shows the system support provided by nodes: a name-based memory region, called the Tag Space, a Virtual Machine (VM), and an Admission Manager.

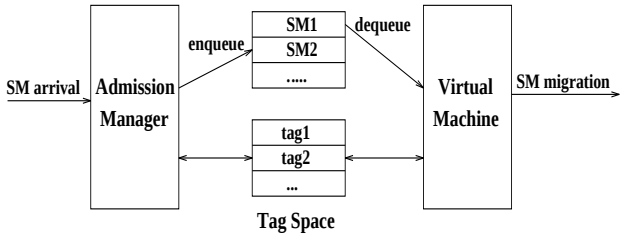

Figure 2. System Architecture

Each SM has a well defined execution cycle. First, it has to be admitted at the destination node. Second, upon acceptance, a task will be generated out of SM's code and data bricks, and scheduled for execution. During its execution the SM may yield the processor and wait for data. Third, when the task completes its execution on the current node, it may decide to migrate to another node.

Category	Primitives
Tag Space operations	createTag, deleteTag, readTag, writeTag
Creating a new SM	create
Spawning the current SM	spawn
Synchronization	block
High level migration	migrate
Low level migration	sys_migrate

Table 1. API primitives

2.1. Tag Space

The Tag Space is a name-based memory consisting of tags persistent across the execution of SMs. Each tag consists of an identifier, a digital signature, lifetime information, and data. The identifier field is the name of the tag, and it is similar to a file name in a file system. This identifier can be used for content-based naming of nodes. The tags can also be used for routing, data exchange, or data sharing among multiple SMs. SMs can create, delete, read, or write tags. The access of SMs to tags is restricted based on the digital signature (i.e. to get access to tags, an SM has to present the same digital signature with the SM that created the tag). The tag lifetime specifies the time at which the tag will be reclaimed by the node from the Tag Space. The Tag Space contains two types of tags: I/O, and application tags. The I/O tags provide SMs with a unique virtual interface to the local OS and I/O system. For example, to read the value provided by a temperature sensor on a node, an application has to read a tag. A task cannot create or delete an I/O tag, but it can create application tags.

2.2. Admission

To prevent excessive use of resources (energy, memory, bandwidth), a node needs to perform admission control. The Admission Manager is responsible for receiving incoming messages, and storing them into a message queue, subject to admission restrictions, such as resource constraints or the presence of specific tags. To avoid unnecessary resource consumption, a three-way handshake protocol is defined for transferring SMs between neighbor nodes. First, only the small size header of the SM is sent to destination for admission control. If the SM admission fails, the task will be informed and the decision about next actions will be taken at the application level. If the SM is accepted, the Admission Manager checks whether the code bricks belonging to this SM are already cached locally, and then informs the SM to transfer only those code bricks not available at destination. This operation is performed fast, because unique identifiers are assigned to each code brick. A hash function applied on the code provides statistically unique identifiers. The VM makes sure that a task conforms to its declared resource estimates. Otherwise, the task can be forcefully removed from the system.

2.3. Execution

Upon admission, an SM generates a task, which is scheduled for non-preemptive execution (other SMs can be accepted, but not scheduled). The execution time is bounded by the estimated running time presented during admission control. The VM acts as a hardware abstraction layer for loading, scheduling, and executing tasks generated by incoming SMs. During its execution, a task may modify the data sections of the SM, as well as the local tags to which it has access, may create new SMs, spawn itself, migrate, or block on tags of interest. A collection of SMs cooperating towards a common goal forms an application.

New SMs can be always received at a node provided there are sufficient resources, but the corresponding tasks will be scheduled only after the current task completes. The VM generates a new task out of SM's code and data bricks, and loads it into the runtime data area according to the scheduling policy. After the loading operation is complete the VM will start the execution. The execution will start from where it was left before a migration, or it will start from the beginning for new SMs .

Although an executing task is never preempted by another task, it may yield the VM by blocking on a tag. When an executing task terminates or blocks, the VM may select the next task that is ready for execution. However, if the execution time estimate based on which the SM was accepted expires, the task can be forcefully terminated.

An application may require synchronization during its execution. A task can block on a specific tag pending a write of the tag by another SM. While the task is blocked, the VM may execute other tasks. When a

task writes a tag, it wakes up all tasks blocked on the tag and makes them ready for execution.

If the data necessary to complete the application is not available within the local Tag Space, or a new node has to be visited, the task may decide to migrate. The VM has to save the execution state necessary for resumption at the destination node. Since a task accesses only mobile data and tags, an efficient, lightweight migration has been implemented, where only a small part of the entire execution context is saved and transferred through the network. When the above operations are done, the SM is sent to destination and resumed there, and the local VM will schedule a new task.

2.4. API

The API for the SM model, given in table 1, provides simple, but powerful primitives in terms of expressibility. The operations on Tag Space allow an SM to create, delete, or access existing tags. An SM may use *create* at any point during the execution to assemble a new SM using its code and data bricks. An application that needs to clone itself calls the *spawn* function. Similar to *fork* system call, *spawn* returns null in the clone, and non-null in the parent. A new SM created by *spawn* or *create* is scheduled for execution at the local node. The update based synchronization mechanism is implemented by the *block* primitive. There are two functions for migration: (1) the high level migration (*migrate*) , usually provided as a library function, is used by applications to name destinations by tag names, and (2) the low level migration (*sys_migrate*) is used by routing algorithms to migrate the SM to the next hop in the path. The entire protocol of physically migrating an SM is implemented in *sys_migrate*.

3. Content-based Migration

The SM model provides a scalable and flexible solution for content-based routing in NES. SMs are self-routing, since they are responsible for determining their own paths through the network by executing their own routing at each hop. The model integrates computation and communication, and similar to most ad hoc routing schemes, the separation between hosts and routers disappears in NES. There is no support for routing in the nodes other than the Tag Space, the entire process taking place at application level. Each application has to include at least one *routing brick* among its code bricks, which is cached at nodes in the common case. The application controls routing in two ways: (1) it can choose the routing algorithm dynamically, or it can implement its own routing, (2) it can intervene in routing, being able to change the current *routing brick* during execution.

```
 1  int n=0, sum=0, avg;
 2  createTag(AVGTEMP, lifetime, null);
 3  while(n < 10){
 4      if ( migrate(TEMP, timeout) ){
 5              sum += readTag(TEMP);
 6              n++;
 7      }
 8      else{
 9              break;
10      }
11  }
12  avg = sum/n;
13  migrate(AVGTEMP, INFINITE);
14  writeTag(AVGTEMP, avg);
```

Figure 3. SM Code Example

The key operation in the SM model is content-based migration. A routing brick defines a high level migrate function. Applications name the nodes of interest by content or properties, represented by tag IDs, and then call *migrate({tags}, timeout)* to take the SM to a node that contains all {*tags*}. *migrate* is a user-level primitive, which can be provided as a library routing brick, or implemented directly by users. The *migrate* call guarantees that the application will regain the control at a destination, or after the *timeout* interval. In the latter case, the routing algorithm has not been able to find a destination during the given period. If a timeout occurs, the application may decide to change the target nodes, or to abandon the migration.

Figure 3 illustrates the use of *migrate* call. To compute the average temperature in a certain geographic region, the application needs to run on ten nodes providing temperature sensors. It is important to mention that we use the "average temperature" example for the sake of simplicity, as our target systems are not represented by sensor networks. To make the example simpler, we use a single tag name (TEMP) as a parameter of *migrate*. In general, migration can take multiple tag names as arguments, and it tries to find destinations that contain all these tags.

The application expects to run on up to 10 different nodes that have the desired information, temperature in our case. An interesting problem generated by content-based routing, not shown in this example, is how *migrate* makes sure that applications avoid ending up in the same place always. A simple solution, that we are using, is to let the application record the target nodes visited, and to pass this list as a parameter to *migrate*.

```
 1  int n=0, sum=0, retry=0, avg;
 2  createTag(AVGTEMP, lifetime, null);
 3  while (retry < 3){
 4    if (spawn()){
 5      if (block(AVGTEMP, block_timeout)){
 6        retry++;
 7      }
 8      else{
 9        break;
10      }
11    }
12    else{ // migrates
13      // Code from Figure 3.
14    }
15  }
```

Figure 4. Improved Result Availability

The application starts by creating a tag for average temperature at the source node (line 2). Then, it calls *migrate* (line 4) until ten nodes are visited, and the sum of temperature is computed. Finally, it calls *migrate* again to return to the source, and to write the average value in the AVGTEMP tag (lines 13-14). Two observations have to be made here. First, the second call to *migrate* may use another routing brick, and implicitly another implementation of *migrate*. Second, if a route to destination is not found, the application will not stay in the network forever. An SM can use a limited number of resources, and if it stays for too long in the network it will end up being dropped. If the timeout expires before being able to visit ten target nodes (line 8), the application is willing to accept a partial result. This is a simple example of application-defined quality of result, which shows the ability of SMs to adapt to adverse network conditions. For example, the application may never complete if ten nodes providing temperature readings do not exist in that region.

In traditional networks, service availability is a metric of primary importance. In NES, we talk about result availability, meaning the probability to get a result in such a volatile environment. To improve the result availability, the application can spawn itself at the source and wait for an answer. The same application, trying three times to get the result before giving up, is presented in figure 4. The additional code is inserted in lines 3-11. After spawn, one instance of the application blocks locally waiting for result, and the other one migrates through the network in search of target nodes.

Figure 5 shows a library implementation for *migrate* using *sys_migrate* and additional tags. To be capable of routing, SMs need to maintain routing information

```
 1  int migrate(tagID, timeout){
 2    do{
 3         if (!Route_to_tagID){
 4            create(RouteDiscovery_SM(tagID));
 5            block(Route_to_tagID, timeout);
 6         }
 7         sys_migrate(Route_to_tagID);
 8    }while(!tagID);
 9  }
```

Figure 5. Migration Implementation

```
 1 RouteDiscovery_SM(tagID){
 2  do{ // forward
 3     sys_migrate(All_Neighbors);
 4     createTag(previous_to_tag, lifetime, previous());
 5  }while(!(readTag(tagID) || readTag(Route_to_tagID)));
 6  do{ // backward
 7     sys_migrate(previous_to_tag);
 8     createTag(Route_to_tagID, lifetime, previous());
 9  }while(previous_to_tag);
10  writeTag(Route_to_tagID, previous());
11 }
```

Figure 6. Route Discovery SM

within the Tag Space. SMs may create tags at visited nodes in the network, caching discovered route information in the data portion of these tags. Since tags are persistent, routing information, once acquired, can be used by subsequent SMs with similar interests, thus amortizing the route discovery effort. In this way, an application may implement traditional routing algorithms using tags to store routing tables.

As we discussed before, *migrate* returns when the SM arrives at a node containing the tag of interest, or after timeout expiration. The verification of the *timeout* is omitted in the figure for the sake of simplicity. If a next hop towards a destination is available, the entire SM eagerly migrates there (line 7). Otherwise, a route discovery SM is created, and the current task blocks waiting for a route (lines 4-5). The task will be waken up by the write performed by the discovery SM when it returns with a route. This implementation is resilient to broken paths (due to mobility or failures), being able to restart a discovery process at any node in the path.

Figure 6 presents the code for the route discovery SM used in the above migration implementation. It floods all neighbors, looking for the desired tagID, or for a route to a node of interest (lines 2-5). In its

way through the network, it records at every node the address of the node it came from (line 4). This information will be used on the way back to the source, but it can also be used as a trace to prevent looping. In general, loop avoidance can be implemented by recording the path inside SM, or storing SM identification information within certain tags when a node is visited. Once a destination, or a route to a destination is found, the discovery SM starts its journey back to the source (lines 6-9), where it updates the routing tag with the newly acquired route, thus unblocking the SM waiting on that tag.

Directed Diffusion [17] has implemented an algorithm similar to the one presented in this section.

4. SM Routing Algorithms

In this section, we present several implementations for content-based routing algorithms using SMs. It is not our intent to show finely tuned routing implementations. Our goal is to understand the potential of SMs in implementing flexible content-based routing in NES, and to illustrate each class of algorithms with a routing scheme using SMs. The simulation results, and the analysis for these algorithms will be presented in the next section. Having the possibility to choose among multiple routing schemes, an application programmer can take informed decisions about which one to select. For applications that expect radical changes in the network conditions during their execution, more than one routing brick should be incorporated, thus allowing an SM to dynamically adapt.

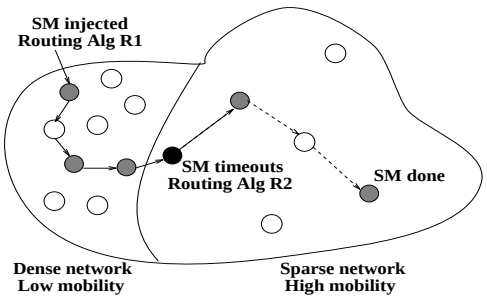

Figure 7. Dynamic Change of Routing

Figure 7 shows an example of an SM that dynamically changes its routing algorithm during execution. The grey nodes are nodes of interest for the application. In the dense and relatively stable part of the network, the SM routing brick may use routes established by an a priori routing algorithm. A possible implementation would employ exchange of information among SMs resident on nodes. Once the SM enters the unstable part of the network, the adverse conditions (low density of nodes, high mobility) lead to a timeout in the *migrate* call. Let us assume that the SM is executing on the node colored black in the figure when the timeout expires. At this point, the application decides to change the current routing algorithm. It does so by calling a new *migrate*, corresponding to another routing brick, which implements an on-demand routing. Using the new routing scheme, the SM is able to visit all nodes of interest, and to complete its execution. This example shows the potential of the SM self-routing mechanism in providing flexible routing for applications.

In the rest of the section, we present three routing algorithms implemented using SM: the first shows an on-demand routing, the second implements an a priori algorithm, and the third is a combination of the first two.

4.1. On-demand Routing

Previous research, such as DSR [20] and AODV [8], has shown that on-demand routing schemes are suitable for highly mobile environments, where nodes come and go at any moment.

For on-demand routing, we have implemented a *migrate* function similar to the one presented in section 3. If no routing information is available locally, *Discover* SMs will be flooded in the network. A *Discover* that arrives at a node already visited will stop its execution. After finding a node of interest or a route to a node of interest, *Discover* returns to source. The first *Discover* updating the routing tag at the source unblocks the main SM, which subsequently migrates to the next hop (using *sys_migrate*). If at any point in time the next hop becomes unavailable, the route discovery process is restarted. In this way, as long as the timeout that bounds the migration does not expire, routing around broken paths is possible. An advantage of the SM model compared to the traditional request/reply model is that an application is able to make progress even in poor network conditions, moving towards destination, and eventually arriving there. In the request/reply model, the round-trip communication may never complete, and the application can fail to achieve any result.

For each tag of interest found, *Discover* creates routing tags on the path back to the source. A routing tag is associated with the ID of a discovered tag, and it maintains routes to all destinations learned so far that contain this tag ID. To avoid the problem of ending up always at the same node, the destination IDs are also maintained in the routing tag. By default, the routing chooses the shortest path. The exception occurs when

the application does not want to end up in the same place more than once. In such a situation, the application passes a list of visited nodes to *migrate*. The routing makes sure that the SM will not execute twice on a given node of interest.

The problem with on-demand schemes is the use of flooding. In networks of such size, flooding can consume most of the resources. To be more realistic, we need to limit the number of hops traveled by a *Discover* SM.

4.2. A Priori Routing

Exchanging routing information among all nodes in NES is practically impossible, but a limited exchange of information among neighbors can be useful, even in the absence of global convergence. This idea has been used for disseminating limited data among nodes in sensor networks by SPIN [15]. We present a probabilistic routing scheme for content-based routing using Bloom filters [5]. A Bloom filter is a bit vector of length n that uses several independent hash functions to map the elements from a set to integers in [0,n). To form a Bloom filter summary, each element in the set is hashed and the bits in the bit vector associated with the hash functions are set. To do an element lookup, the element is hashed and the corresponding bits are checked to see if they are set or not. If all the bits are set, there is a certain probability that the element might be contained in the set. Thus, false positives can occur. Whereas, if any one of the bits is not set, we can guarantee that the element is not in the set. Among several recent works that use Bloom filters to store information summaries, the idea presented in Probabilistic Routing is similar to ours [28].

The basic idea in this algorithm is to maintain approximate information about content location in the network as Bloom filters. The summaries are disseminated among neighbors, and they are diluted as they move away from the source. A nodes closer to a content have more accurate knowledge about the existence of the content than nodes farther away from it. This information continues to degrade as we move farther from the content. However, it is still possible for an SM to discover a route to a content located far away from its current node using the approximate information maintained locally. This knowledge may not be accurate, but it is expected that the next hop will be able to provide more precise information. Thus, choosing as intermediate hops nodes which have an a priori better knowledge about the location of the content may finally lead to the desired destination.

Setting the network for an a priori algorithm can be done on demand by injecting a *Routing* SM that will replicate itself at the participating nodes. The *Routing* SM maintains summaries about the information learned so far, and stores them in the Tag Space. It maintains exact summaries for the local node and its immediate neighbors, and approximate information about its larger neighborhood. The approximate information for a node located N hops away from a content is an OR of the summaries for the nodes located up to N-1 hops away (N is an implementation parameter).

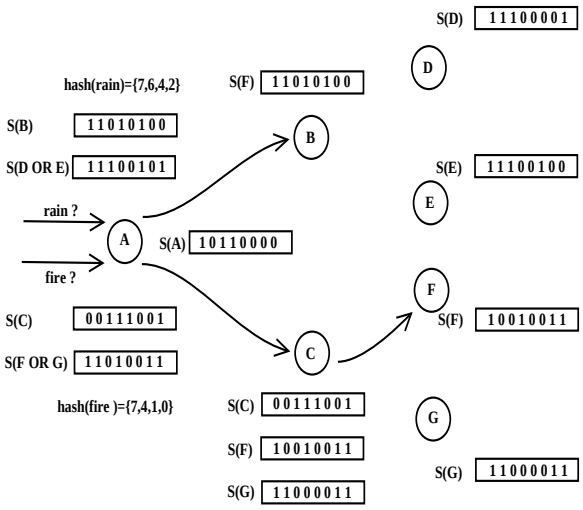

Figure 8. Lookup in A Priori Routing

Routing SMs block on a tag, wake up periodically to disseminate information, or they are waken up by SMs bringing summaries. In the initialization phase, the local summaries are disseminated to each of the neighbors one hop away. After the exact summaries of the neighbors have been received, the local summaries are updated, and new SMs will propagate them. Each *Routing* SM creates periodically a *heartbeat* SM that migrate to each of its immediate neighbors, informing them that the local node is still alive. If no *heartbeat* is received from a node within a timeout period, the node is assumed to be dead and its summary is discarded. When new summaries are modified, the *Routing* SM incorporates the differences (if any) in the *heartbeat*, and informs all its immediate neighbors about the change. This change is further propagated by *heartbeats* created by *Routing* SMs residing on the neighbor nodes.

Figure 8 shows an example of a lookup operation performed in a network with intelligent cameras. The cameras are programmed such that a tag is set when fire is detected. For instance, this tag can be used by an application to monitor a forest. When the application arrives at node A, it looks up the summaries to find the

next hop which has the "fire" tag, or more precise information about the location of this tag of interest. The routing algorithm applies the hash functions on "fire", and checks if the hash value can be matched against the local summaries. It concludes that the neighbors of C might know better about "fire", and migrates to C. The same algorithm is applied at C, and the SM discovers the "fire" on node F.

An SM arriving at a node, which has been previously visited, will look up the summary and choose a different neighbor towards the desired content. If no such neighbor exists, the SM will migrate randomly. If it will arrive again at the same node, it will decide to stop its execution.

4.3. Rendez-Vous Routing

In the above discussion we have seen two completely separate classes of routing algorithms: one is based on pulling the routing information after discovery, and the other one is based on pushing it. An interesting approach for content-based routing is to combine "push" and "pull". For example, we might like to help the applications by disseminating routes for important content in their physical proximity, and then let them discover the route when needed.

We describe a rendez-vous routing which is a routing scheme that uses a combination of "push" and "pull" mechanisms. The idea of rendez-vous communication has been recently introduced in Associative Overlay Networks [1]. Rendez-vous routing is similar to a publish/subscribe system. Unlike the usual publish/subscribe systems that run over stable IP networks, the big issue in NES is to determine where to publish the information, or where to subscribe. A solution based on hashing the name into a key, and mapping the key to a geographic region has been proposed recently for networks of sensors [26].

Figure 9 illustrates a simple idea of localized rendez-vous routing. We have implemented a rendez-vous algorithm that combines geographic dissemination with limited flooding. A task, running at the grey circle in the figure, needs routing information for a certain tag name. The algorithm starts by broadcasting an *Explore* SM to one-hop away neighbors, and then blocks waiting for routing tag updates. The *Explore* SMs look for the given routing tag at the neighbor nodes. If the information is found, *Explore* SM returns at source, and updates the routing tag; this operation unblocks the application. If routing information is not available at the neighbors, *Explore* SMs create a tag for the desired routing data, and block on it. If no result is received until timeout, each *Explore* will broadcast itself one more hop, and will double the timeout. The algorithm works recursively until it reaches the established limit of the number of hops to be visited. The exploring process is stopped by the application after it receives the required route. The application floods *Cancel* SMs, which will let the *Explore* SMs know that they have to finish. A *Cancel* SM stops at the first node that does not contain the desired routing tag (i.e. no *Explore* has passed through that node).

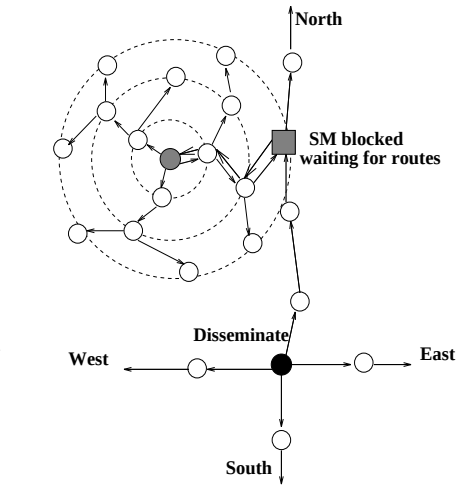

Figure 9. Rendez-Vous Routing

Applications that create important tags generate a new SM to disseminate routing information. According to our algorithm, an application, running at the black node in the figure, creates 4 SMs that will travel in 4 directions, based on geographic coordinates: East, West, North, South. These SMs are inherently loop free. We assume that each node knows its own location and its neighbors locations. The intuitive idea behind our approach is that the rendez-vous can happen in two situations: one of the dissemination SM will intersect the flooded area, or an *Explore* SM will reach a node storing the disseminated information. There are two advantages to this scheme: (1) we avoid a global dissemination, which would be too expensive in terms of network resources, but in the same time we propagate routing information eagerly, (2) we limit the flooding process that takes place in on-demand algorithms. Consequently, routes to important information will be discovered faster, and the response time for applications will decrease. In our example, the SM moving North will update the routing tag at the grey square node, and the *Explore* SM waiting there will bring the routing information to the source (grey circle).

5. Evaluation

In this section we present preliminary results for an SM prototype implementation as well as simulation results for the routing algorithms described in the previous section.

5.1. SM Implementation

We are in the process of implementing an SM architecture by modifying Sun's KVM on Compaq iPaqs. The KVM is a virtual machine designed for mobile devices with resource constraints, suitable for devices with 16/32-bit RISC/CISC microprocessors/controllers, and with as little as 160 KB of total memory available. The iPaqs use Wavelan 802.11b and Bluetooth for communication.

SM applications are written in Java. Code bricks are Java class files, and data bricks are objects. Initially, an SM is created from class files and data objects specified by the application. Once an SM is injected into the network, other SMs can be created dynamically at nodes. The VM implements this process by cloning the data objects, setting pointers to code bricks, and adding the SM to the message queue.

Tag Space Operation	Time (microseconds)
createTag	55.8
deleteTag	30.8
readTag	25
writeTag	28
block	24.6

Table 2. Time for Tag Space operations

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

We have evaluated the time for *sys_migrate* and one SM round-trip using two wireless technologies, 802.11b (ad hoc mode) and Bluetooth. We have used TCP for communication in the initial phase, but we plan to implement a simple link layer protocol for better performance.

For Bluetooth, we use Ethernet encapsulation over L2CAP. Slave-to-slave communication is implemented using Linux Ethernet bridging. Once the scatternet capability will become robust for commercially available Bluetooth devices, we plan to implement a scatternet formation protocol.

The time taken by our three-way handshake protocol for a *sys_migrate* of an SM with one code brick of size 1480 bytes, stack size of 129 bytes, and multiple data brick sizes is presented in table 3.

Data Brick size (bytes)	Wavelan (ms)	Bluetooth (ms)
1	23	132
100	24.5	133.2
1000	26.4	169.1
4000	41	257.8

Table 3. sys_migrate time

To measure the average time required for a round-trip communication and execution, we load a small SM at a node, which creates a new tag and a second SM (*RTT SM*), and then blocks on the tag. The *RTT SM* migrates one hop, then migrates back to the source node and writes the tag. As a result, the original SM is unblocked. The *RTT SM* has a single code brick of size 731 bytes, no data bricks, and a stack of 78 bytes. The time elapsed between the original SM's *block* call and the resumption of execution is 55.2 ms using Wavelan, and 452.8 ms using Bluetooth.

We have implemented a neighbor discovery protocol in 802.11b ad hoc mode. Periodically, a kernel thread broadcasts a NEIGH_REQ message. Every neighbor responds with a NEIGH_RESP message, containing the node address. A cache of known neighbors is kept in kernel memory. SMs can query a read-only I/O tag *neighbor_list* for the list of neighbors. A typical read operation takes 180 microseconds.

Currently, we use a native method to migrate one hop to neighbors. We will provide a *send* tag for this purpose in the future. To migrate to another node, an SM writes the destination address to the *send* tag. Since we can write an arbitrary object onto the tag, multicast can be supported by writing a list of neighbor addresses to the tag.

5.2. Simulation Results

For experiments we use an event-based simulator, similar to ns-2 [23], extended with support for SM execution. To get accurate results, both the communication and the execution time have to be counted. The simulator is written in Java to allow rapid prototyping

of applications. The simulator provides accurate measurements of the execution time by counting, at the VM level, the number of cycles per VM instruction. The problems introduced by counting the execution time have been solved by simulating each node with a Java thread, and implementing a new mechanism for scheduling these threads inside JVM.

The communication model used in our simulator can be considered "generic wireless" with contention solved at the message level. The nodes can communicate within an area limited by their transmission range. Before any transmission, a node "senses" the medium and backs-off if somebody else is sending. The percentage of mobile nodes, as well as the percentage of nodes to fail are given as simulation parameters. Mobility is simulated using the random waypoint model [23]; nodes alternate between moving to randomly chosen points, and pausing.

An interesting problem that has to be solved is how to characterize these networks. By taking a random network and placing the nodes of interest randomly, it is difficult to evaluate and compare different applications, or routing algorithms. We have defined two parameters that characterize the network for given applications. The first one is the network density, expressed as the average number of neighbors per node. The second is the average number of hops between any two nodes of interest for the application.

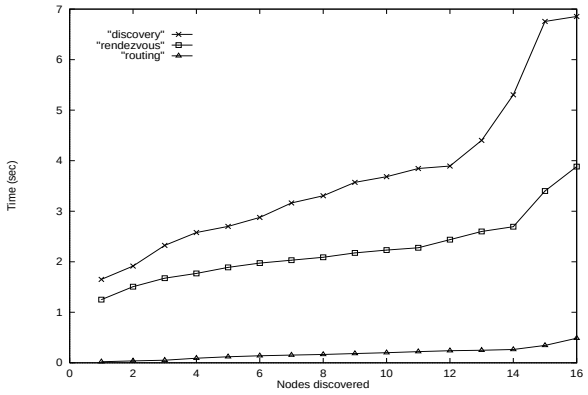

Figure 10. On-demand vs. Rendez-Vous

In all experiments, the nodes are uniformly located in a square area, and each node has the same transmission range. In the first experiment we analyze the simulation results for on-demand and rendez-vous routing algorithms in a static network with 256 nodes. We inject 8 SMs that need to visit all 16 nodes of interest that exist in the network. They run in parallel, and the contention effect is accentuated since nobody has routing information at the beginning. The average time to discover the nodes for the first time is plotted in figure 10. Other set of 8 SMs, looking for the same content, are injected in the network after the previous set finished its execution. The effects of maintaining routing information in the Tag Space and caching code in the network are immediately observed, as the average time to visit all target nodes drops more than an order of magnitude.

The rendez-vous algorithm can be used to reduce the cost of discovering the nodes when no routing information is available in the Tag Space. As we can see, the time to visit all nodes decreases with more than 30%.

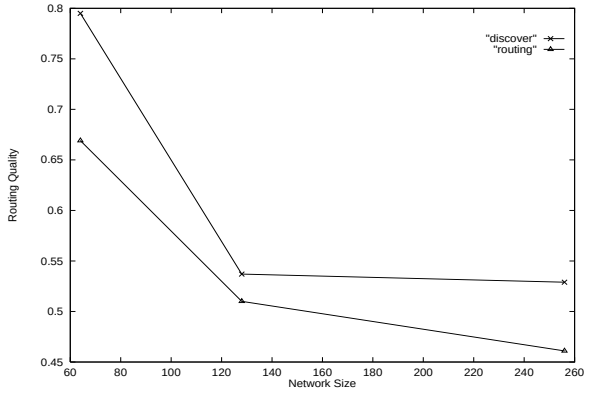

Figure 11. On-demand dense networks

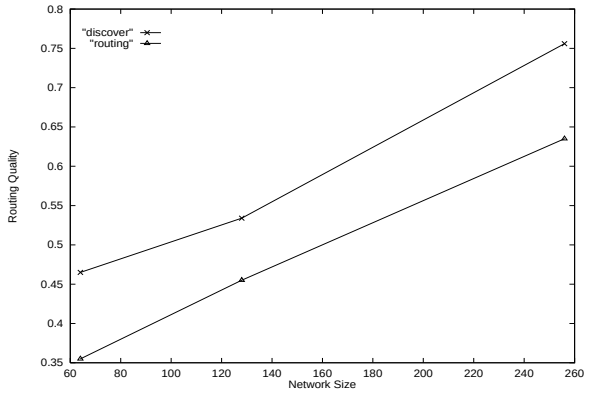

Figure 12. On-demand sparse networks

To be able to evaluate routing performance for different networks, and different placement of the nodes of interest, we define *Routing Quality* as follows. Let M be the average number of hops between two nodes of interest for the current path, and N the average number of hops between any two nodes of interest in the network. Routing Quality is defined as the report M/N.

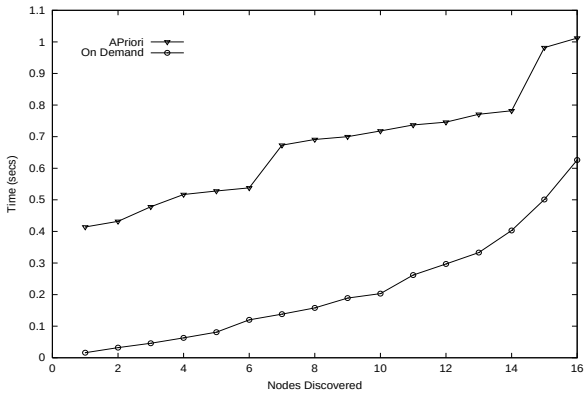

Figure 13. A priori vs. On-demand

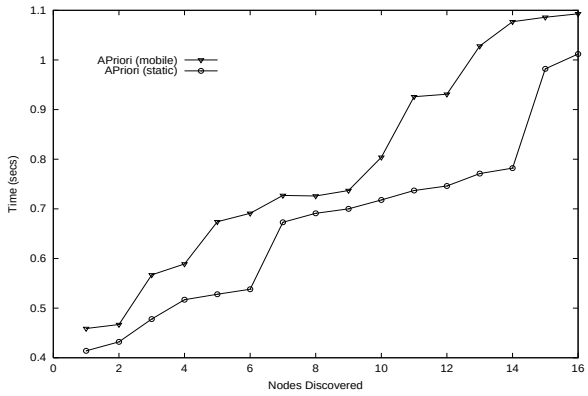

Figure 15. Mobility effect

Figures 11 and 12 present the evaluation of Routing Quality for our on-demand implementation. We use 3 network sizes for each experiment (64, 128, 256 nodes). The networks are differentiated by node density. The dense networks have an average of 10 neighbors per node, compared to 4 neighbors per node in the sparse networks. The routing performs much better in the dense networks, since multiple paths are created towards the same destinations.

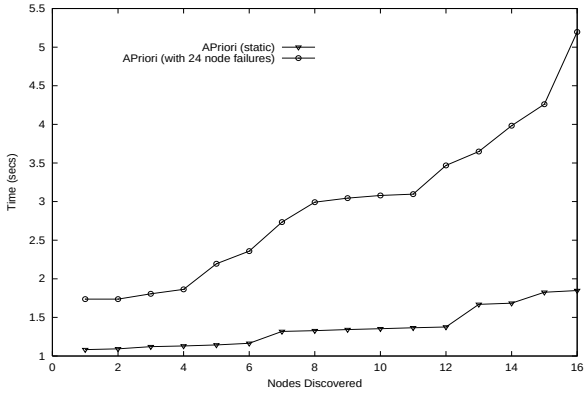

Figure 14. Node failure effect

In figure 13 a comparison between on-demand and a priori routing algorithms is presented. The network is dense, static, and the size is 128 nodes. The routing for the on-demand algorithm performs better, since it uses routing information stored previously at nodes, while the a priori algorithm uses approximate information, which may lead to false positives, thus longer execution time.

Figure 14 depicts a case where node failures affect the performance of the a priori algorithm. In a static and dense network with 128 nodes, 24 nodes will fail during the execution; nodes of interest do not fail. Node failures may lead to wrong choices in the summary lookups, which result in greater delays and hence a degradation in performance.

In figure 15 we present the effect of mobility on routing performance. The network is dense, with 128 nodes, out of which 16 are mobile. The nodes move at an average speed of 20 m/s. As expected, the algorithm performs better on the static configuration, and although mobility leads to loss of prior information due to dropping of summaries of the neighbors who have moved, the application is still able to visit all nodes of interest.

6. Related Work

Smart Messages bear some similarity to Active Messages [30], active networks [9, 27, 25], and mobile agents [11, 32, 18, 24].

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

The ANTS [9] capsule model of programmability allows forwarding code to be carried and safely executed inside the network by a Java VM. A first difference is that this model does not migrate the execution state from node to node. It caches and transfers code that always starts and finishes on the same node. A second

difference is that ANTS targets IP networks, while SMs does not require any routing support. The main difference in terms of programmability is that, unlike capsule model, SMs define a distributed computing model where the execution state is transferred with the application.

The Smart Packets [27] architecture provides a flexible mean of network management through the use of mobile code. Smart Packets are implemented over IP, using the IP options header. They are routed just like other data traffic in the network, and only execute on arrival at a specific location. Unlike Smart Packets, SMs encapsulate and carry the state of the application, and they run at each hop in the path through the network.

Programmable Packets [25] are centered around a low-level packet language that adds flexibility over IP. They share some of the design goals with SMs, like safety and flexibility, that allow for in-network processing of application specific code, but SMs provide more expressibility for user-defined applications. High level declarative languages can be constructed over SMs to offer a simpler interface to users.

Smart Messages are similar to mobile agents, which also use migration of code in the network. A mobile agent may be viewed as a task that explicitly migrates from node to node assuming the underlying network assures its transport between them. Unlike mobile agents, SMs are defined to be responsible for their own routing in a network. The SM architecture further defines the infrastructure that nodes in a network supporting SMs must implement, which makes self-routing of SMs possible.

Research in mobile ad hoc networking [20, 7, 22, 21] has resulted in numerous routing protocols for peer-to-peer multi-hop networking in infrastructures without base stations. These protocols have generally been designed for networks based on IP, and have primarily targeted traditional mobile computing applications such as mobile personal computers and PDAs. These protocols can be leveraged and implemented using the SM self-routing mechanism.

Recent work on large networks of embedded systems has focused on network protocols for wired and wireless sensor networks [10, 17, 15, 14] and system architectures for fixed-function sensor networks [16]. This research is complementary to the SM model, which provides enough flexibility to enable the implementation of these models over the SM architecture.

7. Conclusions

In this paper we have presented the Smart Messages self-routing mechanism. The two main features that distinguish SM self-routing from other routing mechanisms are flexibility and resilience to adverse network conditions. Content-based migration is the high level primitive used by applications to name their target nodes by content. Multiple library implementations of migrate are available in the form of routing bricks. Applications can choose dynamically the routing algorithm, or they can implement their own routing. In face of adverse network conditions, an application may change its routing during execution. The implementation of three routing algorithms, corresponding to on-demand, a priori, and rendez-vous routing, has demonstrated the potential of SM self-routing mechanism in supporting flexible content-based routing in NES.

Acknowledgments

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

References

[1] Associative overlay networks. http://www.cs.berkeley.edu/∼istoica.

[2] Electrolux screenfridge. http://www.electrolux.se/screenfridge/.

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

[4] Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., and Lilley, J. The Design and Implementation of an Intentional Naming System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (1999), pp. 186–201.

[5] Bloom, B. Space/time trade-offs in hash coding with allowable errors. *Communication of the ACM 13*, 7 (July 1970), 422–426.

[6] Borcea, C., Iyer, D., Kang, P., Saxena, A., and Iftode, L. Cooperative computing for distributed embedded systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), July 2002. To Appear.*

[7] Broch, J., Maltz, D., Johnson, D., Hu, Y., and Jetcheva, J. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proceedings of the Fourth annual ACM/IEEE International Conference on Mobile Computing and Networking* (1998), pp. 85–97.

[8] Charles E. Perkins, Elizabeth Royer and Samir R. Das. Ad hoc on demand Distance Vector(AODV)routing. In *2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)* (February 1999), pp. 90–100.

[9] David Wetheral. Active network vision reality: lessons from a capsule-based system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* (1999).

[10] Estrin, D., Govindan, R., Heidemann, J., and Kumar, S. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the Fifth annual ACM/IEEE International Conference on Mobile Computing and Networking* (1999), pp. 263–270.

[11] Gray, R. S., Cybenko, G., Kotz, D., and Rus, D. Mobile agents: Motivations and state of the art. In *Handbook of Agent Technology*, J. Bradshaw, Ed. AAAI/MIT Press, 2001.

[12] Gray, R. S., Kotz, D., Cybenko, G., and Rus, D. D'Agents: Security in a multiple-language, mobile-agent system. In *Mobile Agents and Security*, G. Vigna, Ed., vol. 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998, pp. 154–187.

[13] Gritter, M., and Cheriton, D. An architecture for content routing support in the internet. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)* (March 2001).

[14] Heinzelman, W., Chandrakasan, A., and Balakrishnan, H. Energy-efficient communication protocol for wireless microsensor networks. In *Proc. Hawaii Int. Conf. on System Sciences* (January 2000).

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

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

[17] Intanagonwiwat, C., Govindan, R., and Estrin, D. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensors Networks. In *Proceedings of the sixth annual ACM/IEEE international conference on Mobile computing and networking* (2000).

[18] Johansen, D., van Renesse, R., and Schneider, F. Operating system support for mobile agents. In *5th IEEE Workshop on Hot Topics in Operating Systems* (1995).

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

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

[21] Li, J., Jannotti, J., Couto, D. S. J. D., Karger, D. R., and Morris, R. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom '00)* (Boston, Massachusetts, August 2000), pp. 120–130.

[22] Marti, S., Giuli, T., Lai, K., and Baker, M. Mitigating routing misbehavior in mobile ad hoc networks. In *Proceedings of the Sixth annual ACM/IEEE International Conference on Mobile Computing and Networking* (2000), pp. 255–265.

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

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

[25] Moore, J. T., Hicks, M., and Nettles, S. Practical programmable packets. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'01)* (April 2001).

[26] Ratnasamy, S., Estrin, D., Govindan, R., Karp, B., Shenker, S., Yin, L., and Yu, F. Data-centric storage in sensornets. Submitted to SIGCOMM 2002.

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

[28] Sean C. Rhea, J. K. Probabilistic location and routing. In *Proceedings of the 21th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'02). To Appear* (2002).

[29] Stanley-Marbell, P., Borcea, C., Nagaraja, K., and Iftode, L. Smart messages: A system architecture for large networkws of embedded systems. In *Proceedings of HotOS-VIII, May 2001. Position Paper*. Longer version: Rutgers University Technical Report DCS-TR-430.

[30] von Eicken, T., Culler, D., Goldstein, S., and Schauser, K. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual international symposium on Computer architecture* (May 1992), pp. 256–266.

[31] Weiser, M. The computer for the twenty-first century. *Scientific American* (1991).

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