Self-Routing in Pervasive Computing Environments using Smart Messages [*]

Cristian Borcea[1], Chalermek Intanagonwiwat[1], Akhilesh Saxena[1], and Liviu Iftode[2]

[1] *Department of Computer Science*
Rutgers University
Piscataway, NJ, 08854, USA
{*borcea, intanago, saxena*}*@cs.rutgers.edu*

[2] *Department of Computer Science*
University of Maryland
College Park, MD, 20742, USA
iftode@cs.umd.edu

Abstract

Smart Messages (SMs) are dynamic collections of code and data that migrate to nodes of interest in the network and execute on these nodes. A key challenge in programming pervasive computing environments using SMs is the ability to route SMs to nodes named by content. This paper describes the SM self-routing mechanism, which allows SMs to route themselves at each hop in the path toward a node of interest. SM applications can choose among multiple content-based routing algorithms, switch dynamically the routing algorithm in use, or even implement the best suited routing algorithm for their needs. The main benefits provided by self-routing are high flexibility and resilience to adverse network conditions. To demonstrate these benefits, we present proof-of-concept implementation, simulation results, and analysis for the SM self-routing mechanism using several content-based routing algorithms. We also show preliminary results for SM routing algorithms executed over our SM prototype implementation.

1. Introduction

Incorporating intelligence in devices encountered in our daily routine, as well as providing them with networking connectivity (mostly wireless), creates the possibility of building large scale networks of embedded systems (NES). NES are inherently ad hoc networks because the sheer number of nodes and their volatility (i.e., nodes join and leave the network often due to mobility, failures, or disposal) preclude any fixed infrastructure. In many environments, these networks will represent the infrastructure for pervasive computing [22, 15]. For instance, we envision home appliances communicating to handle certain domestic activities, intelligent cameras collaborating to track a given object, or cars on a highway cooperating to adapt to the traffic conditions. Sensor networks [11, 10] have represented the first step toward this vision. The target systems that we consider are more powerful than the nodes in sensor networks 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 distributed computing did two decades ago. To program such 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. Recently, we have proposed cooperative computing [4] as a solution for programming user-defined distributed applications over NES, while this paper addresses the routing in NES.

The applications running in NES will target specific data or services within the network, not individual nodes. Fixed naming schemes, such as IP addressing, are almost irrelevant for these applications in most cases. We believe that a naming scheme based on content or properties is more suitable for NES than a fixed naming scheme (the same idea has been proposed for both the Internet and sensor networks [1, 8, 11]).

Different applications in NES can have different routing requirements. For example, an application may use geographic information for routing, and another one may use 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.

The conclusion of the above discussion is that a flexible, application-controlled routing mechanism is needed for NES. The main requirements for it are: generality, capa-

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

bility to perform application-specific content-based routing, ability to adapt to adverse network conditions, and simplicity of implementation.

In this paper, we describe the Smart Messages' self-routing mechanism, which provides solutions for these requirements. Smart Messages (SM) [4] are collections of code and data that migrate to nodes of interest in NES and execute on these nodes. Each node supports SMs by providing a virtual machine and a name-based memory region, called Tag Space. Instead of routing data end-to-end, the SM self-routing mechanism migrates the computation to nodes of interest named by content using application-level content-based routing which is executed at every intermediate node.

Commonly, the SM routing algorithms are provided as pre-defined library functions. Routing flexibility is achieved by allowing applications to choose the best suited routing algorithm for their needs, to implement new routing algorithms, or to switch dynamically their routing. SMs are resilient to network dynamics, being able to control the routing, find alternative routes to nodes of interest, discover similar nodes of interest, or adapt to adverse network conditions as long as a certain quality of result is met.

SMs' design has been influenced by a variety of other research efforts, particularly mobile agents [13, 6] and active networks [5, 17]. We leverage the general idea of code migration, but we focus more on flexibility, scalability, re-programmability, and the ability to perform distributed computing for unattended NES. Specifically, an SM is a lightweight and platform-independent mobile agent with a unique data model (provided by the Tag Space) for cooperative computing in massive networks of heterogeneous, resource constrained embedded systems. Section 5 describes the similarities and differences between SMs and the above work in more details.

The remainder of this paper is organized as follows. Section 2 describes the SM design and system architecture. In Section 3, we present the SM self-routing mechanism, typical applications that benefit from it, and the implementation of four SM routing algorithms (on-demand content-based routing, geographical routing, proactive routing using Bloom filters, and rendez-vous routing). Section 4 evaluates the self-routing mechanism using both an SM prototype to demonstrate the practicality of the proposed solution and an SM simulator to show the benefits of self-routing over large scale networks. Section 5 discusses the related work, and we conclude in Section 6.

2. Smart Messages

Smart Messages (SM) are migratory execution units consisting of code and data sections, termed *bricks*, and a lightweight execution state. The SM execution is embodied in tasks described in terms of computation and migration phases. As opposed to request/reply paradigm, SM applications need to migrate to nodes of interest and execute there. In doing so, SMs execute a routing algorithm, carried as a code brick, on all nodes in the path toward a node of interest. The code bricks are cached by nodes to reduce the cost of transferring the code. Over time, this cost is amortized because many SM applications have temporal and spatial locality.

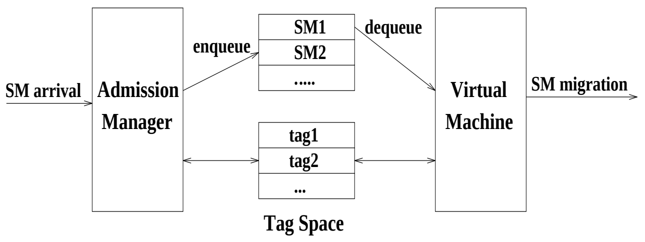

Figure 1. Node Architecture

Since nodes in NES are resource constrained and have limited functionality, the goal of the SM system architecture is to reduce the support required from nodes by placing parts of intelligence in SMs (e.g., routing or various services). Additionally, placing intelligence in SMs provides flexibility and obviates the difficulty of re-programming the network for a new application or protocol [9]. Figure 1 shows the system support provided by nodes: a name-based memory region, called Tag Space, a Virtual Machine (VM), and an Admission Manager. [1]

2.1. Tag Space

The Tag Space is a name-based memory consisting of tags persistent across SM executions. Essentially, each tag consists of an identifier and data. The identifier field is the name of the tag and is similar to a file name in a file system. The data field is application-specific. The identifier is used for content-based naming of nodes. The tags can also be used for routing, synchronization, or data exchange among multiple SMs. Associated with each tag is also a lifetime that specifies the duration after which the tag will expire and its memory will be reclaimed by the node. The Tag Space contains two types of tags: (1) *application* tags which are temporary tags created by SMs, and (2) *I/O* tags which are permanent tags that provide SMs with a unique 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.

[1] Security is an important component of the SM system architecture, but it is outside the scope of this paper. Challenges, solutions, and open issues are discussed in [23].

2.2. SM Lifecycle

Each SM has a well defined lifecycle: (1) *Admission* - the SM has to be admitted at each node in the path toward its target node, (2) *Task generation and execution* - upon acceptance, a task is generated out of SM's code and data bricks and scheduled for execution, and (3) *Migration* - if the execution does not complete at the current node, the SM may decide to migrate to another node.

2.2.1. Admission

To prevent excessive use of resources (e.g., energy, memory, bandwidth), a node needs to perform admission control. The Admission Manager is responsible for receiving incoming messages and storing them into a ready queue, subject to admission restrictions such as resource constraints or the presence of specific tags. An accepted SM has to transfer only the missing code bricks (i.e., the code bricks that are not cached locally). The VM ensures that an SM generated task conforms to its declared resource estimates. Otherwise, the task can be forcefully removed from the system.

2.2.2. Execution

Upon admission, an SM generates a task which is scheduled for non-preemptive execution (other SMs can be accepted, but they will be scheduled only after the current execution completes). The execution time is bounded by the estimated running time presented during admission control. The VM acts as a hardware abstraction layer for executing tasks generated by incoming SMs. The API available to these tasks is given in Table 1. An SM may use *createSM* to assemble a new, possibly smaller SM using some of its code and data bricks. An application that needs to clone itself calls the *spawnSM* function (similar to the *fork* system call in Unix). A new SM created by *spawnSM* or *createSM* is scheduled for execution at the local node. An update-based synchronization mechanism is implemented by the *blockSM* primitive. A task can block on a certain tag until another task performs a write on that tag. A blocked SM yields the processor, and the VM may schedule other tasks. There are two functions for migration: *migrateSM* which is used for high level migration, and *sys_migrate* which is used for low level migration. The *migrateSM* primitive is used by applications to migrate (over multiple hops) to nodes of interest named in terms of arbitrary conditions on tag names and tag values. The *sys_migrate* primitive implements the entire protocol of migrating an SM between neighbor nodes. The *migrateSM* is implemented at user-level and uses *sys_migrate*, which is provided by the system, to migrate an SM to the next hop in the path toward a node of interest.

Category	**Primitives**
Tag Space Operations	createTag, deleteTag, readTag, writeTag
Creation	createSM, spawnSM
Synchronization	blockSM
Migration	migrateSM, sys_migrate

Table 1. API Primitives

2.2.3. Migration

To migrate an SM, the VM has to capture the execution state necessary for resumption at the destination node and send it there along with the code and data bricks. Since an SM generated task can access only its data bricks and the Tag Space, only a small part of the entire execution context has to be saved and transferred through the network. Therefore, we have been able to implement a lightweight migration for SMs.

3. SM Self-Routing Mechanism

Similar to most mobile ad hoc networks, the separation between hosts and routers disappears in NES. In our approach, there is no support for routing at nodes (the entire routing process takes place at application-level). Each application has to include at least one *routing brick* among its code bricks. Applications control routing in two ways: (1) they can select their routing algorithms, (2) they can intervene in routing, being able to change the current *routing brick* during execution.

3.1. Content-Based Migration

The key SM operation is content-based migration. A routing brick defines a high level *migrateSM* function. Applications name the nodes of interest by content and then call *migrateSM* to route them to a node that has the desired content. Additionally, *migrateSM* can be instructed to check if the nodes with this content meet some arbitrary conditions (i.e., it implements a *conditional content-based* migration). The *migrateSM* is a user-level primitive, which can be provided as a library routing brick or implemented directly by programmers. For instance, a simple implementation of *migrateSM* takes a list of tag names as a parameter and migrates the SM to a node that contains all those tags. However, nothing precludes a programmer to express more complex conditions within this primitive. Commonly, *migrateSM* takes a *timeout* as a parameter in order to deal with network volatility. 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. Consequently, it may decide to change the routing, to change the nodes of interest, or to abandon the migration.

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

Figure 2. SM Application Example

Figure 2 illustrates the use of *migrateSM* call. To compute the average temperature over a certain geographic region, the application needs to run on ten nodes providing temperature sensors. To simplify the example, we use a single tag name (TEMP) as a parameter of *migrateSM*. The application starts by creating a tag for average temperature at the source node (line 2). Then, it calls *migrateSM* (line 4) until ten nodes are visited and the sum of temperatures is computed. Finally, it calls *migrateSM* again to return to the source and to write the average value in the AVGTEMP tag (lines 11-12). We make two observations: (1) the second call to *migrateSM* may use another routing brick and implicitly another implementation of *migrateSM*, and (2) if a route to a node of interest is not found, the application will not stay in the network forever (i.e., an SM can use limited resources and if it stays for too long in the network, it will eventually be dropped). If the timeout expires before the SM is able to visit ten nodes (line 8), the application accepts 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 instance, the application might never complete if ten nodes providing temperature readings do not exist in that region.

Figure 3 shows a library implementation of *migrateSM* using *sys_migrate* and additional tags. To be capable of routing, SMs need to maintain routing information within the Tag Space. SMs may create tags at visited nodes, caching discovered routing information in the data portion of these tags. Since tags are persistent across SM executions (as long as their lifetimes have not expired), this routing information can be used by subsequent SMs with similar interests, thus amortizing the route discovery effort. In our example, if a next hop toward a node of interest is available, the entire SM eagerly migrates there (line 4). Otherwise, a route discovery SM is created and the current task blocks waiting for a route (lines 6-7). The task is woken up when the discovery SM returns with a route and writes the routing tag. An interesting problem generated by content-based routing (not shown in this example) is how *migrateSM* ensures that the application does not end up on a node already visited. A simple solution is to let the application record the nodes of interest visited and pass this list as a parameter to *migrateSM*.

```
1 int migrateSM(tagID, timeout){
2  while(!readTag(tagID))
3    if (readTag(Route_to_tagID))
4      sys_migrate(readTag(Route_to_tagID));
5    else{
6      createSM(RouteDiscovery_SM(tagID));
7      blockSM(Route_to_tagID, timeout);
8    }
9 }
```

Figure 3. Migration Implementation

3.2. Application Examples

To illustrate the flexibility provided by self-routing, we present several scenarios for applications that benefit from this mechanism. These scenarios correspond to the two possible ways for an application to control the routing: choosing its routing algorithms, and dynamically changing its current routing algorithm. Section 3.3 describes the SM implementations of the routing algorithms supporting these applications.

3.2.1. Selecting the Routing Algorithm

A first scenario involves an application that needs to perform image recognition on a number of camera nodes that have acquired an image with a certain resolution within a given time interval. In the absence of routing information, a naive solution would be to use an on-demand content-based routing algorithm to discover camera nodes. Once migrated to a camera node, the SM has to check if the resolution of the image and its acquiring time conform to the application's requirements, and then proceed with the computation. The disadvantage of such a method is that the SM has to pay the cost of migrating to nodes that do not satisfy the requirements of the application (e.g., they have low resolutions or old images). The self-routing mechanism allows the application to define its own routing that discovers only the nodes having the desired combination of tag names and values (i.e., they meet the required content-based condition). Thus, the network bandwidth, the energy consumed, and the response time are all reduced for this application. It is important to mention that self-routing offers the power to use any arbitrary condition expressed by a program to select the nodes of interest.

A second example presents an SM routing algorithm that builds an ad hoc content-based topology over a network of hand-held devices belonging to the attendees at a conference. For instance, CEOs attending a conference may de-

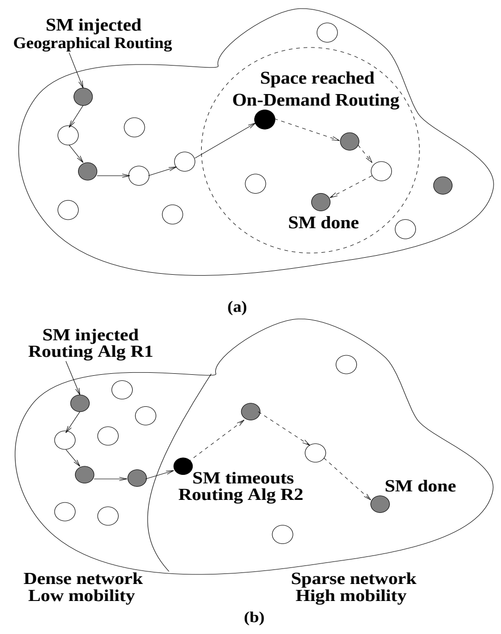

Figure 4. Dynamic Change of Routing

cide to have an important discussion and, for security reasons, they would like to have their messages sent directly to destinations or forwarded toward destinations only by other CEO devices. Under the assumption that it is possible to obtain a connected graph using only CEO nodes, a simple SM routing algorithm can be developed such that the routing entries stored in the Tag Space have the *next hop* value set always to a CEO node.

3.2.2. Dynamically Changing the Routing Algorithm

Using multiple routing bricks during the lifetime of an SM may improve the completion time or even help the application complete in the presence of adverse network conditions.

Figure 4(a) presents an SM that incorporates two routing bricks comprising of a geographical routing and an on-demand content-based routing. The nodes containing the tag of interest are colored grey, but the application is interested only in the grey nodes located in the circular region. Therefore, a simple on-demand content-based routing would perform poorly since it would have to flood the entire network to discover the nodes of interest. The performance can be radically improved if the application has knowledge about the geographical region where the nodes of interest should reside. In such a case, a geographical routing is used to reach the desired region. Once there (the black node in the figure), the SM changes its routing to the on-demand content-based algorithm which will flood only a limited area.

Figure 4(b) shows another example of an SM that changes its routing dynamically. The grey nodes are nodes of interest for the application. In the dense and relatively stable part of the network, the SM may use routes established by a proactive routing algorithm. 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 *migrateSM* call. Let us assume that the SM is executing on the black node when the timeout expires. At this time, the application decides to change its routing. It does so by calling a *migrateSM* which corresponds to an on-demand content-based routing. Using the new routing, the SM is able to visit all nodes of interest and complete its execution.

3.3. SM Routing Algorithms

In the following, we describe briefly the proof-of-concept implementations for several routing algorithms using SMs. [2] It is not our intention to show finely tuned routing implementations. Our goal throughout this paper is to show the potential of the SM self-routing mechanism in implementing flexible content-based routing in NES.

3.3.1. On-Demand Content-Based Routing

Previous research, such as DSR [12] and AODV [19], has shown that on-demand routing is suitable for highly mobile environments. We extend this work to implement an on-demand content-based routing algorithm using SMs. Each time routing information is not available at the current node, an SM floods *Discover* SMs in the network. A *Discover* SM that arrives at a node already visited stops its execution. After finding a node of interest or a route to a node of interest, a *Discover* SM returns to its source. The *Discover* SMs create or update routing tags at each node in the path back to the source. The first *Discover* SM updating the routing tag at the source unblocks the initial SM, which subsequently migrates to the next hop (using *sys_migrate*). Each time the next hop becomes unavailable, the route discovery process is restarted. Thus, routing around broken paths is possible. Such situations emphasize one of the advantages of using self-routing SMs over the traditional request/reply paradigm: an application is able to make progress even in poor network conditions, moving toward nodes of interest and eventually arriving there. In the request/reply paradigm, the round-trip communication may never complete and the application may fail to achieve any result.

3.3.2. Geographical Routing

Unlike traditional distributed systems where the physical location of nodes does not matter, the spatial distribution of nodes across the physical space is a key feature of massive NES. Many times, the applications running in NES will

[2] A more detailed description of these algorithms can be found in the companion Rutgers University Technical Report DCS-TR-477.

prefer to express their interest for content located within well-defined geographical regions. Therefore, a geographical routing algorithm becomes a necessity. We have implemented a simple greedy geographical routing that takes a circular region as a parameter and migrates the SM to the neighbor node closest to the center of the region until it reaches a node located within that area.

3.3.3. Proactive Routing using Bloom Filters

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. We have implemented an algorithm that maintains approximate information (summaries) about content location in the network as Bloom filters [3]. The summaries are disseminated among neighbors and are diluted as they move away from the source. Nodes closer to some content have more accurate knowledge about its existence 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. Initializing the network for this proactive 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. These SMs block on a tag and wake up periodically (or each time new summaries are received) to disseminate information.

3.3.4. Rendez-Vous Routing

We introduce the term “rendez-vous” routing to define a category of routing algorithms that use a combination of on-demand and proactive routing. We have implemented a rendez-vous algorithm that combines geographic dissemination with limited flooding. An SM that creates an important tag disseminates routing information by creating four SMs which migrate using geographical routing in the four cardinal directions (i.e., east, west, north, south). An SM that needs routing information starts by broadcasting *Explore* SMs 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 tag is found, *Explore* SMs return to source and update the routing tag, thereby unblocking the initial SM. If routing information is not available at the neighbors, *Explore* SMs create a tag for the desired routing data and block. If no result is received until timeout, each *Explore* SM broadcasts itself one more hop and doubles the timeout. The routing works recursively until it reaches the established limit of number of hops to be visited. The exploring process is stopped by the application after it receives the required route.

The intuitive idea behind our approach is that the rendez-vous can happen in two situations: (1) one of the dissemination SMs intersects the flooded area, or (2) an *Explore* SM reaches a node storing the disseminated information. There are two advantages to rendez-vous routing: (1) we avoid a global dissemination, which would be too expensive in terms of network resources, but at 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 are discovered faster and the response time for applications decreases.

4. Evaluation

This section presents an experimental evaluation of the SM self-routing mechanism using an SM prototype to demonstrate the practicality of the proposed solution and an SM simulator to show the benefits of our mechanism over large scale networks. In the following, we summarize our SM prototype and SM simulator, describe the evaluation methodology, study the flexibility of application-level self-routing, provide an insight on the re-programmability of our system, and examine the impact of routing dynamics on the application performance. [3]

4.1. Experimental Evaluation over an SM Prototype

We have implemented an SM platform by modifying Sun’s K Virtual Machine (KVM), which is suitable for mobile devices with resource constraints and with as little as 160KB of memory. SM applications are written in Java. The SM API is implemented using native methods and is available as Java libraries.

We present the preliminary evaluation of two simple SM routing algorithms (geographical and on-demand content-based) executed over our SM prototype. Since one of these routing algorithms might be more suitable than the other for some applications, we do not intend to compare them. In fact, a judicious use of both algorithms might yield significantly better results than each of them separately.

Our goals in conducting this experimental evaluation study were three-fold: (1) to demonstrate the flexibility of the SM architecture for application-level self-routing, (2) to understand the re-programmability issues in NES, and (3) to explore the influence of code caching on our unattended re-programmable system. Our testbed consists of eight

[3]Considering the space limitations and the fact that our goal is to show the potential of the SM self-routing mechanism, not that of individual SM routing algorithms, we do not present results for all algorithms described in the previous section. However, the companion technical report contains simulation results for all of them.

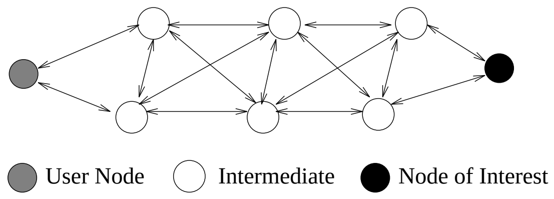

Figure 5. Network Topology

Routing Algorithm	Code not cached (ms)	Code cached (ms)
Geographical	415.6	126.6
On-demand	506.6	314.7

Table 2. Completion Time

HP's iPAQs running Linux and using Orinoco's 802.11b PC cards for wireless communication. The network topology is typically four hops across (see Figure 5). The SM starts at the grey node and discovers the tag of interest at the black node using geographical routing or on-demand content-based routing.

In the first experiment, we measure the completion time of an SM using geographical routing. The SM routes itself from the grey node to the black node and returns on a different path. The round-trip time for this task is 415.6 ms (Table 2). At the beginning of our experiments, there was no SM program (or routing) installed at any node. Therefore, the result also includes the latency imposed by programming the network. The program size of our SM with geographical routing is approximately 4.4KB. To factor out the installation latency, we study the impact of code caching on this experiment by re-running the same SM at the grey node. The second execution of the same SM (the code is cached by all nodes) takes only 126.6 ms (or 3.2 times faster).

We also conduct a similar experiment for an SM with on-demand content-based routing. When the code is not cached, the route discovery time for this SM is 506.6 ms. This result is a bit surprising, given that the program size of this route discovery SM is only 2.8KB. However, the result is reasonable given the significant delay imposed by the wireless contention (due to route discovery flooding). When the code is cached, the route discovery time for this SM decreases to 314.7 ms (or only 1.6 times faster). Understandably, one might also expect a 3-times speedup for this SM after code caching. However, the impact of code caching is less evident when the program size is smaller, given an unavoidable overhead coupled with such wireless contention.

4.2. Simulation for Larger Scale Evaluation

For simulation based experiments, we have developed an event-based simulator, similar to ns-2 [16], extended with support for SM execution. The simulator is written in Java to allow rapid prototyping of applications. To get accurate results, both the communication and the execution time have to be accounted for. The simulator provides accurate measurements of the execution time by counting the number of cycles per VM instruction at the VM level. To account for the execution time, we have simulated each node with a Java thread and we have implemented a new mechanism for scheduling these threads inside VM. The communication model used in our simulator can be considered "generic wireless" with contention solved at the message level. Before any transmission, a node "senses" the medium and backs-off in case of contention. We uniformly distribute 256 nodes in an 1000m by 1000m square. The transmission range for each node is 100m. A node can communicate with an average of 6 neighbors (ranging from 2 to 11 neighbors) at the network bandwidth of 2Mb/s.

Our main goal in conducting the simulation experiments was to quantify the effects of the self-routing mechanism for applications running in large scale NES. We choose two metrics to analyze the performance of our solution: (1) *the completion time* which measures the user-observed response time for an application, and (2) *the total number of bytes sent* which measures the total amount of traffic (generated by an application) throughout the network. This metric implies the energy and bandwidth consumed by an application and consequently, it also indicates the overall lifetime of the network.

Our first set of simulation experiments studies the SM feature that allows programmers to select the most appropriate routing for their applications or even to implement their own routing. The application starts on a node located in the bottom-left corner of the square region that contains the network. The goal of this application is to visit a number of nodes of interest (defined by a given tag name) which satisfy a certain condition. Without loss of generality, we simply check if the value associated with the given tag is over a certain threshold. We use two on-demand routing algorithms for this experiment (similar to those described in 3.2.1): a simple on-demand content-based routing, and a conditional on-demand content-based routing (which enhances the simple on-demand algorithm with a few lines of code that checks the desired condition).

We distribute uniformly over the network area a total of five nodes containing the tag of interest and vary the number of nodes of interest (in this experiment, nodes whose tag values satisfy the desired condition) from one to four (by setting the values of the tags of interest). Since the results of using the simple on-demand content-based routing depend on the order of visiting the nodes, we take all the possible combinations and compute the average for both routing algorithms. Our results indicate that the conditional routing improves the response time with as much as 40% (see Figure 6) because it does not visit any unnecessary nodes (i.e., nodes that have the desired tag, but the tag value does not

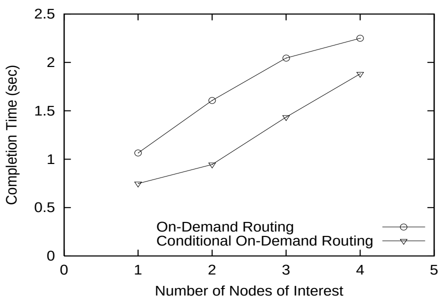

Figure 6. Completion Time for Experiment 1

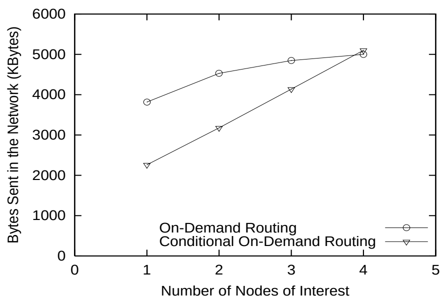

Figure 7. Bytes Sent in the Network for Experiment 1

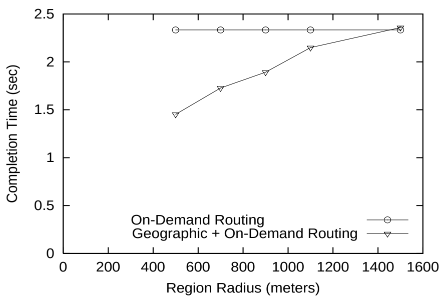

Figure 8. Completion Time for Experiment 2

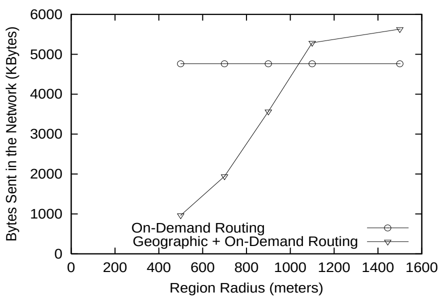

Figure 9. Bytes Sent in the Network for Experiment 2

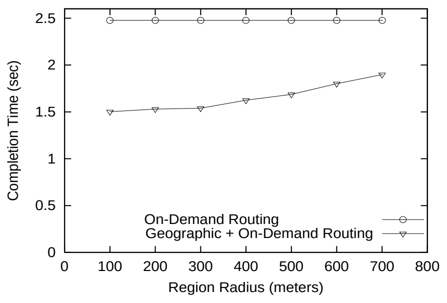

Figure 10. Completion Time for Experiment 3

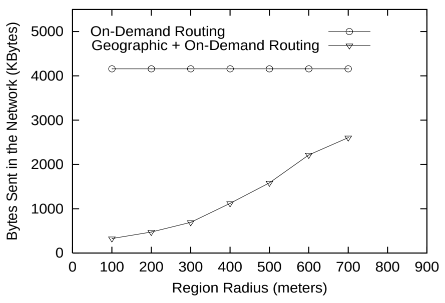

Figure 11. Bytes Sent in the Network for Experiment 3

meet the condition) whereas the simple routing does.

Additionally, our bytes-sent results (see Figure 7) indicate that the conditional routing consumes significantly less energy and bandwidth (40% fewer bytes sent for one node of interest) than the simple routing. As expected, when the number of nodes of interest increases, the savings of our conditional routing are less evident because the simple on-demand routing visits fewer unnecessary nodes. When the number of nodes of interest is close to the number of nodes hosting the tag of interest, the simple routing even performs slightly better than the conditional routing. The primary reason is that the code size of the conditional routing is approximately 150 bytes larger than that of the simple routing. Even though this additional size is small, the impact of the additional overhead for programming the network becomes noticeable, given that the network size is sufficiently large.

In the second set of experiments, we study the SM ability to change its routing during execution. Specifically, we compare applications using only on-demand routing with applications using a combination of geographical and on-demand routing (as described in 3.2.2). The application starts on a node located at the bottom-left corner of the region. The goal of this application is to visit five nodes of interest identified by a given tag name. The network contains exactly five nodes of interest uniformly distributed over a region delimited by a circular area with the center at the opposite corner and a 500m radius. If the application has approximate information of the geographical region containing these nodes, it can migrate to this area using geographical routing. Upon reaching the specified area, the application changes dynamically its routing to geographically-bound on-demand routing (i.e., on-demand routing that floods a limited region) in order to discover the target nodes. In our simulations, we vary the approximate geographical information (of the target nodes) by changing the radius of the circular region defined above (the nodes of interest remain the same).

The performance of the on-demand routing remains constant (regardless of the radius) because this simple on-demand routing always floods the entire network (see Figure 8). Conversely, the more accurate the target area is, the faster the combination scheme completes (as much as 38% reduction in completion time). For the 1500m radius, the combination scheme performs roughly the same as the on-demand algorithm because the target region already covers the entire network.

It is well documented that the use of flooding in large scale networks adversely impacts the system scalability [18]. Figure 9 shows that the combination approach can significantly improve the scalability by reducing the total number of bytes sent in the network (consuming less energy and bandwidth). The combination scheme can achieve up to 80% energy and bandwidth savings. Surprisingly, for a larger radius ($\geq$ 1100m), on-demand routing sends fewer bytes than the combination scheme. There are two reasons for this result. First, given such a large target region, the combination scheme unavoidably floods almost the entire network. Second, the code size of geographically bound on-demand routing is 400 bytes larger than that of simple on-demand routing. This additional code size can significantly decrease the performance, given the sufficiently large network size and the flooding nature of our on-demand routing.

Nevertheless, for some applications, the combination scheme can achieve much better performance. Similar to the previous experiment, we consider an application that starts at the same node at the bottom-left corner. However, unlike the previous experiment, the goal of this application is to visit three nodes, each of which residing in one of the other corners. Additionally, the application has to visit these three nodes (identified by different tag names) in clockwise order. Under our investigated scenarios, the combination scheme (with limited flooding) expectedly completes faster (between 25% and 40%) than the on-demand algorithm which floods the entire network (Figure 10). The difference between full flooding and limited flooding is more evident because the on-demand routing floods the entire network three times. Such faster completion time conforms with the fewer bytes-sent result in Figure 11 (between 62% and 92% bytes savings).

5. Related Work

Recent projects [7, 2] have presented programming models for pervasive computing. In this context, we have proposed cooperative computing [4], a programming model for distributed embedded systems based on Smart Messages (SMs). The self-routing mechanism is an essential component of this model since it allows flexible routing in highly dynamic NES.

SMs are influenced by the design of mobile agents for IP-based networks [13, 6]. A mobile agent may be viewed as a task that explicitly migrates from node to node assuming that the underlying network assures its transport between them. SMs apply the general idea of code migration, but focus more on flexibility, scalability, re-programmability, and ability to perform distributed computing over unattended NES. Unlike mobile agents, SMs are defined to be responsible for their own routing in a network. A mobile agent names nodes by fixed addresses and commonly knows the network configuration a priori, while an SM names nodes by content and discovers the network configuration dynamically. Furthermore, the SM system architecture defines the common system support that each node must provide. The goal of this architecture is to reduce the support required from nodes since nodes in NES possess limited resources.

SM self-routing mechanism shares some of the design goals and leverages work done in the active networks (AN) area [5, 17]. However, SMs differ from AN in several key features. First, AN target IP networks with underlying support for routing whereas SMs does not require any routing support. Additionally, SM communication is based on content rather than node IDs (e.g., IP addresses). Second, AN does not migrate 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 in programming sophisticated tasks which require cooperation and synchronization among several entities. For example, this execution state allows SMs to make routing decisions based on the results of computation done at previously visited nodes. Finally, SMs and AN differ in terms of programmability. Unlike AN, SMs define a computing model whereby several SMs can cooperate, exchange data, and synchronize with one another through the Tag Space.

To demonstrate the SM self-routing mechanism, we have borrowed ideas from the networking literature on routing algorithms, especially DSR [12], AODV [19], GPSR [14], GEAR [24], and Probabilistic Routing [20]. It was not our intention to develop better routing algorithms than the above work, but rather to show the SM flexibility which allows an application to select, implement, or switch dynamically the routing algorithm. With this intention in mind, the idea of using Bloom filters to store information (in Probabilistic Routing) has been leveraged into our SM proactive routing. Similarly, the on-demand feature of DSR and AODV has been further developed into our SM on-demand content-based routing. The SM on-demand routing can be used in combination with our SM geographical routing (a simplified version of GPSR) to improve the performance of some particular applications.

Unlike our approach (which selects dynamically the current routing algorithm from multiple existing algorithms), GEAR combines geographical and on-demand routing into one and provides a more sophisticated technique in forwarding a request toward the specified region. However, hybrid routing algorithms can also be implemented using SMs. An example of an SM hybrid routing is our rendez-vous routing that uses a combination of on-demand and proactive routing algorithms. Rendez-vous routing shares the same idea with a recently introduced paradigm for Internet communication, called rendez-vous communication [21].

6. Conclusions

In this paper, we have presented the Smart Messages (SM) self-routing mechanism. The main feature of SM self-routing is its flexibility in the presence of highly dynamic network configurations. Content-based migration is the high level primitive used by applications to name the nodes of interest by content and to migrate the execution there. Using this primitive, SM applications can choose the most suitable routing for their needs, implement their own routing, or change the routing dynamically. Our experimental and simulation results indicate that the above flexibility can improve the responsiveness of SM applications and provide significant energy and bandwidth savings.

Acknowledgments

The authors would like to thank Deepa Iyer and Porlin Kang for their work in developing the SM prototype. We also thank Rabita Sarker for her contribution in implementing the proactive routing using Bloom filters.

References

[1] 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)*, pages 186–201, December 1999.

[2] 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)*, pages 266–274, August 2000.

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

[4] 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)*, pages 227–236, July 2002.

[5] 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.

[6] R. S. Gray, G. Cybenko, D. Kotz, and D. Rus. Mobile Agents: Motivations and State of the Art. In J. Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2001.

[7] R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. Systems Directions for Pervasive Computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 2001*.

[8] 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)*, March 2001.

[9] 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)*, pages 146–159, October 2001.

[10] 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)*, pages 93–104, November 2000.

[11] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensors Networks. In *Proceedings of the Sixth annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 56–67, August 2000.

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

[13] 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)*, July 1998.

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

[15] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, August 2001.

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

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

[18] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The Broadcast Storm Problem in a Mobile Ad Hoc Network. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 151–162, Seattle, WA, 1999.

[19] C. E. Perkins, E. Royer, and S. R. Das. Ad hoc On Demand Distance Vector(AODV) Routing. In *2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, pages 90–100, February 1999.

[20] S. C. 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)*, 2002.

[21] I. Stoica, D. Adkins, S. Zhaung, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proceedings of ACM SIGCOMM'02*, pages 73–86, August 2002.

[22] M. Weiser. The computer for the twenty-first century. *Scientific American*, September 1991.

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

[24] Y. Yu, R. Govindan, and D. Estrin. Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. Technical Report UCLA/CSD-TR-01-0023, UCLA, May 2001.