Portable Smart Messages Architecture for Ubiquitous Java-enabled Devices

Nishkam Ravi, Cristian Borcea, Porlin Kang, and Liviu Iftode

Department of Computer Science, Rutgers University
{nravi, borcea, kangp, iftode}@cs.rutgers.edu

Abstract. This paper presents the design and implementation of a lightweight and flexible middleware, based on Smart Messages (SMs), for programming distributed applications over networks of ubiquitous Java-enabled devices, such as cell phones and PDAs. SMs are user-defined distributed applications which execute on nodes of interest, named by their properties, and use explicit migration to move between these nodes. The main benefits provided by SMs are ease of deployment for new applications and adaptability to highly dynamic network conditions. To leverage the computing power of existent wireless Java-enabled devices, we have designed a portable SM architecture in which the SM support at nodes is implemented as a runtime system on top of pre-installed Java virtual machines, and the SM migration is implemented by instrumenting the SM Java bytecode. The experimental results for applications executed over a testbed consisting of HP iPAQs communicating through 802.11 wireless cards demonstrate the feasibility of our architecture.

1 Introduction

Recently, we have been witnessing a significant growth in the number of Java-enabled wireless devices [1]. Benefiting from a strong support from industry, cell phones and PDAs are the most representative Java-enabled devices currently available on the market. These devices are mostly used for local computations or downloading data from web servers and personal computers. The lack of appropriate programming models and system architectures, however, preclude users from being capable of running more complex distributed applications over this huge ad hoc computing infrastructure.

In this context, we have proposed Smart Messages (SM), a software architecture similar to mobile agents, for programming user-defined distributed applications in networks of embedded systems. Instead of passing data end-to-end between nodes, an SM application migrates to nodes of interest named by content and executes there. Each node has a virtual machine for SM execution and a name-based memory, called tag space. The SMs use the tag space for content-based naming and persistent shared memory. An SM carries the routing code and routes itself at each node in the path toward a node of interest. SMs represent an attractive alternative to traditional distributed computing based on message passing because they provide ease of deployment for new applications and adaptability to highly dynamic network conditions.

The original implementation of the SM architecture is based on a modified Java virtual machine (Sun's KVM [3]). Having access to the virtual machine (VM) source code, we have been able to implement an efficient migration mechanism. Additionally, all components of the SM architecture have been implemented inside the VM to improve the overall performance of the system. This implementation, although powerful and efficient, comes at a cost of loss of portability. Since this implementation is platform specific, it is not feasible to port it on all devices that are powerful enough to support it. Also, given that most of the new ubiquitous devices come with a pre-installed Java VM, it would be beneficial to have the SMs running over an unmodified VM.

In this paper, we present the design and implementation of a *Portable Smart Messages architecture*. The main issue to be dealt with in a pure Java architecture is how to perform migration without having access to the VM internals. The execution state is located inside the VM and is not directly accessible to the external world. In order to provide migration without involving changes in the VM, we have designed a simple and efficient method for capturing and restoring the execution state by incorporating all the necessary operations in the SM code. The heart of our approach lies in instrumenting the SM bytecode in such a way that the SM can save its state before and restore it before resumption with minimal overhead.

We have developed a prototype implementation using Sun's J2ME CVM virtual machine [4]. Our testbed consists of HP iPAQs running Linux and equipped with 802.11 wireless cards. The experimental results show that the Portable SM implementation, although costlier than the original implementation in terms of execution time, is a feasible solution for programming distributed applications over Java-enabled cell-phones or PDAs.

The rest of this paper is organized as follows. Section 2 describes the SM distributed computing model. Section 3 presents the design and implementation of the Portable SM architecture. In Section 4, we report experimental results. The related work is discussed in Section 5, and the paper concludes in Section 6.

2 Distributed Computing with Smart Messages

A Smart Message (SM) is a user-defined application whose execution is distributed over a series of nodes using execution migration. The nodes on which SMs execute, called *nodes of interest*, are named by content and discovered dynamically using application controlled routing. To move between two nodes of interest, an SM calls explicitly for execution migration. An SM consists of *code bricks*, *data bricks* (mobile data explicitly identified in the program), and execution control state (e.g., *instruction* pointer, *operand stack* pointer). Code bricks are transferred with SMs only if the code is not cached at the destination. An SM can use its code and data bricks to create new, possibly smaller SMs dur-

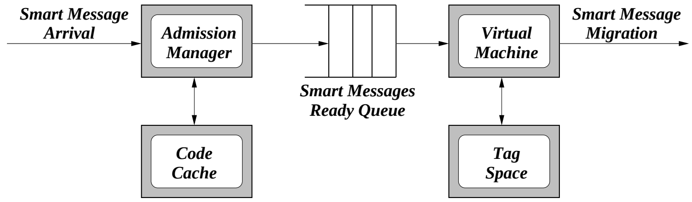

Fig. 1. Node Architecture

ing execution. In this way, an application can eventually generate multiple SMs although it started as a single SM.

2.1 Node Architecture

The SM software architecture is totally decentralized, with nodes in the network acting as peers. SMs do not make any assumptions about the underlying network configuration, except for a minimal system support provided by nodes. The node architecture is presented in Fig. 1.

The admission manager is responsible for receiving incoming SMs, deciding whether or not to accept them, and storing them in the *SM ready queue*. The admission decision is based on a list of resource estimates carried by the SM. The admission manager instructs an accepted SM to transfer only the missing code bricks (i.e., the code bricks that are not stored in the local *code cache*) and stores them in the cache upon reception.

The virtual machine (VM) offers a hardware abstraction layer for SM execution, which shields the SMs from heterogeneous node architectures. The SM execution is non-preemptive; other SMs can be accepted, but they will not be dispatched for execution before the current SM completes. The execution time is bounded by the estimated running time presented during admission. The non-preemptive scheduling simplifies the implementation of inter-SM synchronization and sharing. An SM terminates its execution at a node either if it completes or if it migrates.

The tag space is a name-based shared memory, persistent across SM executions. It consists of a collection of tags, which essentially are *(name, data)* pairs used for data exchange among SMs. Special I/O tags are used as an interface to the host OS and I/O system. Tags are also used to name the destination of SM migrations as well as to store routing information (routing tags).

2.2 Smart Messages Example

To illustrate the SM distributed computing model, let us consider a network of PDAs/cell-phones belonging to students from the same university. At the beginning of each semester the students download on their PDAs/cell-phones an SM that can do two actions: (1) creates a tag for each class the student is

```
i=0;
while(i<N){
  migrate("CS101-Student");
  /* ask student to join */
  if (readTag("Joined"))
    i++;
}
migrate("initiator");
```

Fig. 2. Example of Smart Message Code

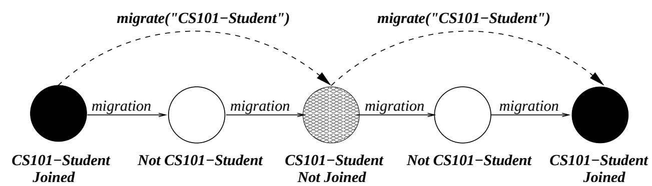

Fig. 3. Execution Path for the Smart Message Presented Above

registered (e.g., CS101), and (2) helps the student set a group study meeting with other students taking the same class. Using this SM, students need not call other people to set a group study meeting, and even more, they need not know the people registered for that class.

Each time a student wants to set a meeting for a group study, she can inject an SM in the network from her PDA/cell-phone. The goal of this SM is to migrate through the network until it finds N students willing to have a group study for a certain class at a given location and time, and then return back to the initiator of the request. This SM is transferred between nodes using short range wireless network interfaces. For instance, Fig. 2 presents the code for an SM that creates a group study for *CS101*. Fig. 3 depicts the execution path of this SM over five nodes.

The key operation in the SM programming model is migration, which implements content-based routing using tags. An SM names the nodes of interest by tags, and then calls *migrate* to route itself to a node that has the desired tags. In our example, *migrate("CS101-Student")* routes the SM to students taking *CS101* using other cell phones (i.e., belonging to students that do not take *CS101*) as intermediate nodes. After migration, the SM resumes from the next instruction following the the migrate call. It is important to notice that migration is explicit (i.e., the programmer invokes a *migrate* primitive when needed), and data transferred during a migration is specified by the programmer as data bricks (i.e., in our case, the location and time for the meeting, as well as the variables i and N are transferred as data bricks).

Table 1. Smart Message API

Category	Primitives

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

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

2.3 Smart Messages API

The SM API is presented in Table 1. To inject a new SM at a node, users invoke *createSMFromFiles* with a list of program file names and one of data bricks. SMs can dynamically create new, possibly smaller SMs by calling *createSM* or *spawnSM*. A *createSM* uses some of the SM's code and data bricks to assemble a new SM and is commonly invoked to build "children" SMs that cooperates with the "parent" SM. An SM may clone itself using *spawnSM* (similar to the *fork* system call in Unix). New SMs generated by *createSM* or *spawnSM* are scheduled for execution at the local node.

There are two functions for migration: *migrate*, and *sys_migrate*. The *migrate* function is used by SMs to migrate (over multiple hops) to nodes of interest named by content. To reach these nodes, *migrate* implements content-based routing algorithms using *sys_migrate* and routing tags [1]. The *sys_migrate* primitive implements the protocol for one-hop migration; it captures the execution state and transfers the SM to the next hop. The VM at destination resumes the SM from the instruction following *sys_migrate*.

An SM can create, delete, or access application tags. The tags are accessed subject to authorization [6]. The same interface is used to access the I/O tags: SMs can issue commands to I/O devices by writing into I/O tags, or can get I/O data by reading from I/O tags (an SM cannot create or delete I/O tags).

An SM can invoke *blockSM* to block on a tag until another SM performs a write on that tag. A blocked SM yields the processor (this is the only exception to the run-to-completion model of execution). When an SM is blocked on a tag, it is inserted into a *wait queue* associated with the tag. When the VM unblocks an SM, it removes that SM from the wait queue and inserts it back in the SM ready queue. To prevent infinite blocking of SMs, *blockSM* has a timeout as parameter; if no write operation takes place within this timeout, the SM is unblocked.

[1] More details about the SM self-routing mechanism are presented in [5]

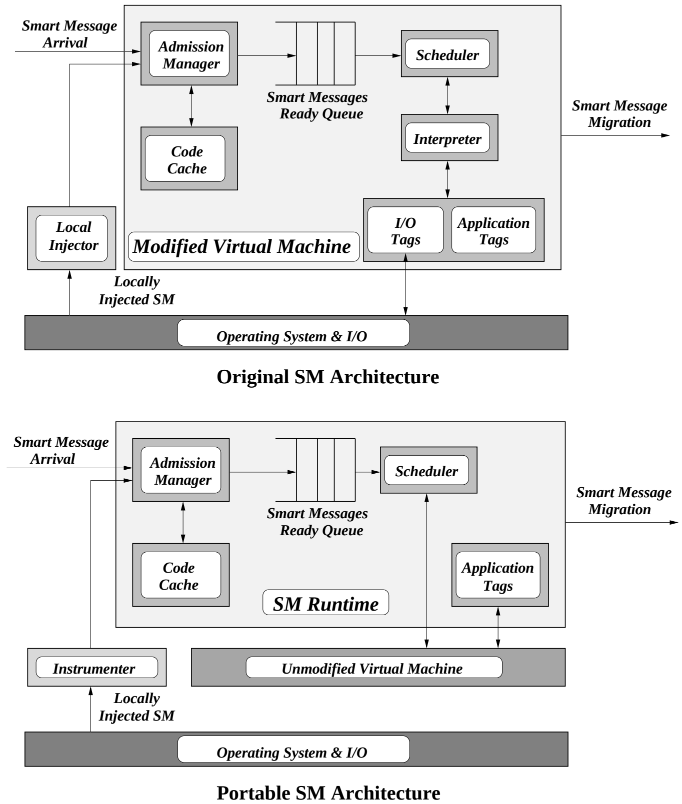

Fig. 4. Comparison of Two Smart Messages Architectures

3 Portable Architecture for Smart Messages

The original SM implementation, was achieved by modifying Sun's Java KVM [3]. The whole architecture, presented in the upper part of Fig. 4, was implemented inside the VM because of the need for VM support in capturing the execution state and restoring it at destination to resume the execution. This implementation, although powerful and efficient, comes at a cost of loss of portability. Since this implementation is platform specific, it is not feasible to port it on all devices that are powerful enough to support it. Also, given that most of the new ubiquitous devices come with a pre-installed Java VM, it would be beneficial to have the SMs running over an unmodified VM. To be able to provide the flexibility that comes with the SM model, but at the same time to avoid the non-portability issue, we have designed and implemented an SM architecture completely in Java.

The main issue to be solved in a pure Java implementation is how to perform migration without having access to the VM internals. Migration is the central part of the SM architecture, and it involves capturing and restoring the execution state. The execution state, however, is located inside the VM and is not directly accessible to the external world.

In order to provide migration without modifying the VM, we have designed a simple and efficient method for capturing and restoring the execution state by incorporating all the necessary operations in the SM itself. The heart of our approach lies in instrumenting the SM bytecode in such a way that the SM can save its state before migration and restore it before resumption with a minimal overhead. Using this method, the state is encoded in the code bricks and data bricks, and no explicit state information is shipped.

The lower part of Fig. 4 presents our pure Java architecture. Unlike the original architecture which was implemented inside the VM (i.e., for efficiency), the architecture for portable SMs is implemented on top of the VM as a runtime system.

3.1 Migration

To migrate an SM, we need to migrate its code bricks, data bricks, and execution control state. The code bricks of SMs are Java class files, and data bricks are Java objects. We use the Java reflection mechanism for loading the classes dynamically at the destination node. The Java serialization mechanism is used to marshal/unmarshal the state of data bricks across migrations. Since SMs do not use local variables across migrations (i.e., the programmers have to include any data that they need across migrations in the data bricks), object deserialization works fine to restore the values of all objects and variables. The main problem that needs to be solved is how to capture and restore the execution control state (i.e., located inside the VM), which consists of the method call stack and the current value of the instruction pointer. Our solution is to instrument the SM bytecode in such a way that SMs can capture and restore their own runtime stack before resuming their normal execution at destination. There are numerous reasons for choosing bytecode transformation [7,8] over source code transformation [9]. First, source code transformation does not provide the fine-grained control as provided by bytecode transformation (e.g., the lack of *goto* statement in Java, the difficulty of instrumenting compound statements statements). Second, instrumenting a loop in source code requires the loop to be unfolded in order to preserve correct execution semantics. Third, instrumenting the source code causes the corresponding bytecode to blow up, and therefore, incurs heavy overheads.

Bytecode Instrumentation for Capturing and Restoring the Execution Control State. We introduce the term *critical method* to refer to any method that can directly or indirectly invoke *sys_migrate* or *blockSM*. As explained in Sect. 2.3, these two methods are the only methods that can lead to capturing and restoring the execution control state. Therefore, only critical methods need to be instrumented. Since a migration (or block) happens at the end of a method call chain, the instrumenter has to detect all the methods from which *sys_migrate* (or *blockSM*) is statically reachable in order to recognize critical methods. To simplify the exposition throughout this section, we will refer only to migration, but the reader should understand both migration and blocking.

Our bytecode instrumenter adds an integer array *ip*[*length*] to every class, where *length* is the number of methods in that class. An element *ip*[*i*] is is used as a pseudo instruction pointer for the *ith* method. The code of a critical method is divided into *code regions* separated by critical method invocations. A critical method invocation marks the end of a code region and the beginning of another new code region. The value of *ip*[*i*] is incremented only before a critical method invocation, and hence, serves as a pointer to the boundaries between code regions. At the time of resumption, the value of *ip*[*i*] also serves as a pointer to the last statement executed inside the *ith* method of the class. The last statement executed inside a critical method before a migration is always a critical method invocation (i.e., either directly a *sys_migrate* call or a chain of method invocations that ends with a *sys_migrate*). This is the reason why incrementing the value of *ip*[*i*] only before *critical* method invocations is sufficient. The value of *ip*[*i*] can be used during resumption to locate the last method invocation made from method *i* before migration. Since every object has a unique *ip* associated with it, *ip* is carried over as a part of data bricks and restored during deserialization.

During resumption, each SM starts its execution from the beginning of the *run()* method of the main class(i.e., SMs execute as Java threads). The instrumenter introduces a *switch* statement at the beginning of every critical method to redirect the instruction pointer, based on the value of *ip*[*i*], to the last statement executed before migration. Hence, the code already executed is skipped. For every method other than the one that directly invoked *sys_migrate*, this will result in an invocation of the method that was adjacent to this method in the runtime stack before migration. As a consequence, the runtime stack is re-created. The *ip*[*i*] of the method that directly invoked *sys_migrate* serves as a pointer to the statement immediately following the *sys_migrate* call.

An SM is said to be in *resumption mode* when it is recreating the runtime stack. To differentiate between *resumption mode* and *normal execution*, the instrumenter adds a global flag: *resumption*. This flag is important for preserving the correct execution semantics. Its purpose is to activate or deactivate the *switch* statement introduced by the instrumenter at the beginning of each critical method depending on whether the SM is undergoing normal execution or is in resumption mode. If the SM is resuming, it is necessary to execute the *switch* statement in order to skip the already executed code. If the SM is undergoing normal execution, it is necessary to ignore the value of *ip*[*i*] to ensure that a method invocation is not influenced by this value (i.e., *ip*[*i*] might be non-zero due to an earlier invocation of the same method). The *resumption* flag of the SM is set by the system before the SM is migrated or blocked and reset by the SM itself once the SM has reconstructed the method call stack, at which point *normal execution* of the SM begins. To achieve this, the *resumption* flag is reset after every statement containing a call to *sys_migrate*.

```
class A{
   void <init>{
     ...
   }

   void Method1(){
     int j = 0;
     int i = 0;
     System.out.println("hello");
     ...
     Method2();
     ...
     Method3();
     ...
     Method4();
     ...
     Migration.sys_migrate();
     ...
     TagSpace.blockSM();
     ...
   }
}
```

Fig. 5. Pseudo-code before Instrumentation

Fig. 5 and 6 illustrate the transformation done by the bytecode instrumenter. Although the transformation is done on the bytecode, for the sake of simplicity, we show a higher level transformation on the corresponding Java pseudo-code. In the example, *classA* has four methods, excluding the constructor. Let us assume that $Method1$, $Method2$, and $Method4$ are critical methods (i.e they can directly or indirectly invoke *sys_migrate* or $blockSM$), while $Method3$ is not a critical method. We present the bytecode instrumentation only for $Method1$, but similar transformations take place on the other critical methods ($Method2$ and $Method4$ in this case) as well. As $Method3$ according to our assumption is not a critical method, it is not instrumented.

Since *classA* has five methods including the constructor, *ip* is declared as an array of length five. We initialize this array in the *<init>* method which is internal to the bytecode, and is invoked every time a new object of the class is created. Given that $Method1$ has four invocations to critical methods (two indirect, and two direct), its code is divided into five code regions labeled from 0 to 4. The value of $ip[1]$ is incremented before every invocation to a critical method. For instance, $ip[1]$ is incremented before an invocation to $Method2$, but not before an invocation to $Method3$ which is not a critical method. This example also shows how the *resumption* flag is used. If the flag is set to *false*, the execution of the methods starts from the beginning. Otherwise, it starts with the code region pointed to by $ip[1]$. As soon as the SM recreates the stack, the *resumption* flag is reset by the SM itself. This ensures that any future invocation to $Method1$ or any other critical method will not be affected by the value of *ip*. Note that *resumption* flag is local to an SM, but global to all the classes that constitute that SM.

```
class A{
   public int[] ip;
   void <init>{
      ip[] = new int[5];
      ...
   }
   void Method1(){
     int j = 0;
     int i = 0;
     if(SM.resumption == true){
        switch(ip[1]){
           case 0: goto label 0;
           case 1: goto label 1;
           case 2: goto label 2;
           case 3: goto label 3;
           case 4: goto label 4;
        }
     }else{
         ip[1] = 0;
     }
     label 0 : System.out.println("Hello");
               ...
               ip[1]++;
     label 1 : Method2();
               ...
               Method3();
               ...
               ip[1]++;
     label 2:  Method4();
               ...
               ip[1]++;
               Migration.sys_migrate();
     label 3:  SM.resumption = false;
               ...
               ip[1]++;
               TagSpace.blockSM();
     label 4:  SM.resumption = false;
               ...
   }
}
```

Fig. 6. Pseudo-code after Instrumentation

Suppose $Method1$ had called $sys\_migrate$ before migration, the value of $ip[1]$ would be 3. When the SM resumes execution at the destination node and enters $Method1$, the instruction pointer would be redirected to *label 3* by virtue of the *switch* statement; from this point on, *normal execution* of the SM begins. If on the other hand $Method4$ had called $sys\_migrate$, then the value of $ip[1]$ would be 2. When the SM enters $Method1$ after resuming at the destination node, the instruction pointer would be redirected to *label 2* which contains a call to $Method4$, thereby skipping the already executed code in $Method1$ and recreating the runtime stack.

Fig. 7 briefly demonstrates the working of our instrumentation scheme. The upper part of the figure gives a pictorial view of ip in the three critical methods at the time of migration. The arrows in the lower part of the figure show the

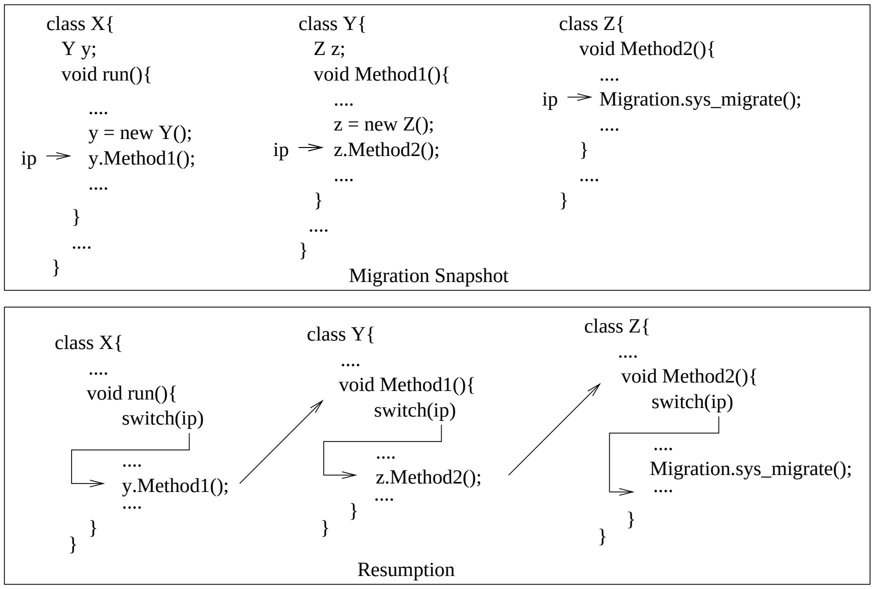

Fig. 7. Example of Resuming the Execution after Migration

control flow of the SM from the time of execution resumption at the destination until it recreates the method stack.

Bytecode Instrumentation for Suspending a Smart Message. In the original SM implementation (which involved modification of the VM), SMs were VM-level threads controlled internally by the VM. In the current implementation, SMs are Java threads, and therefore, the control over SMs is theoretically limited to the amount of control offered by the Java Thread-API. When an SM migrates or blocks on a tag, the corresponding Java thread has to be stopped. The *stop*() method of the Java *Thread* class has been deprecated as it was deadlock-prone. In the absence of a direct way of stopping a Java thread, we have used the Java exception mechanism and bytecode transformations.

For each *sys_migrate* or *blockSM* call, a *StopException* (a class that extends the RuntimeException) is thrown. To ensure that this exception is not caught until it reaches the bottom of the stack, every try-catch block is instrumented to re-throw the *StopException* if it happens to catch it. We ensure that the *run*() method has a try-catch block that catches this exception and consequently finishes the thread's execution. Using a RuntimeException instead of a regular Exception has the advantage that the method signatures do not have to be modified to include a *throws* clause.

3.2 Tag Space

As mentioned in Sect. 2, the tag space contains two types of tags: application tags and I/O tags. Since the implementation of I/O tags is platform dependent, the Portable SM architecture implements only application tags. These tags are Java objects which can be created, deleted, read from, or written into by SMs.

An SM can also block on a tag for a certain period of time. To implement a timeout mechanism, we use Java's built-in scheduler (i.e., provided by the *Scheduler* class), which makes the SM ready for scheduling after the timeout. Using Java's *Scheduler* class avoids a polling timer thread that would otherwise be required to implement the timeout mechanism. Commonly, a blocked SM is woken up by the interpreter when the tag is written by another SM. Each time an SM blocks on a tag, its corresponding Java thread is terminated through the thread stopping mechanism described in 3.1. Each time an SM is unblocked (and consequently dispatched for execution), a new Java thread is created for it.

3.3 Code Cache

We exploit Java's classloader to implement the code cache. The Java reflection mechanism is used to load a class representing a code brick. In the process, a new *Class* instance of the corresponding class is created. The classloader will not unload the class as long as there is a live reference to the *Class* instance. References to the cached classes are stored such that these classes are not unloaded by the classloader. When the caching policy chooses a class for eviction, we just remove the stored reference for that class.

3.4 Scheduler

The SM scheduler [2] is implemented as a Java thread that extracts an SM from the ready queue in FIFO order, dispatches it for execution as a Java thread, and goes to sleep. When the SM completes its execution, it wakes up the scheduler using the Java's thread synchronization mechanism.

3.5 Limitations

As in the traditional SM architecture, the portable SM architecture does not allow the use of local variables across migrations (they can be used *locally* though). Consequently, a local variable cannot occur in two different code regions. All the variables that need to be used across migrations have to be declared as global variables (i.e., they become part of the data bricks). If the programmer wants to use a local variable in two or more code regions, the local variable should be declared and initialized before the beginning of the first code region of that method. This is necessary to satisfy the bytecode verifier.

[2] Note that the SM scheduler is a component of the SM architecture and is different from Java's built-in scheduler mentioned in 3.2.

Table 2. Increase in Bytecode Size Due to Instrumentation

Unmodified Bytecode(KB)	Modified Bytecode(KB)
2330	2395
1084	1122
1230	1266
1527	1564

The lightweight instrumentation scheme comes at a cost of loss of recursion. Our instrumentation scheme relies on the assumption that only one instance of a method is present inside the runtime stack at the time of migration. Recursion, however, can be used *locally* (i.e., not across migrations).

I/O tags imply coupling of the SM platform with the OS. In order to make SMs portable we had to eliminate these tags. The result is a loss of power in the new model, but it can be compensated by various profiles and JSRs provided by J2ME [3] for interacting with the OS or the network. These profiles/JSRs provide Java API for interacting with the OS or the network, thereby hiding the underlying implementation. For instance, the MIDP profile [10] hides the network protocols from the user, provides a generic method of connecting with other devices and is able to store data persistently without referring to the file system. Another example is the Bluetooth API (JSR 82) which allows connectivity through Bluetooth [3].

The security issues related to the SM architecture are presented in [6]. Our current implementation does not address these issues. We are in the process of converging upon the correct approach for dealing with the security issues that come as a byproduct of designing a middleware for mobile agents and mobile ad-hoc networks.

4 Evaluation

Our goals in conducting the experimental evaluation for the Portable SM architecture were three-folds: (1) quantify the impact of bytecode instrumentation on the SM size, (2) compare the costs for the basic SM operations between our portable architecture and the original SM architecture, and (3) execute a simple application over both architectures in order to compare the completion time.

We use Soot1.2.5 [11] to do the off-line bytecode instrumentation. Table 2 shows the increase in the bytecode size as a result of instrumenting four of our SM test cases. On average, we observe an increase of 2.9% in the bytecode size which is negligible compared to existing approaches (see Sect. 5 for details).

We have tested the Portable SM architecture on J2ME CDC platform which uses CVM as the virtual machine. CDC's Personal Profile is the upcoming replacement for Personal Java which is currently used in cell-phones. We have used the Reference Implementation of CDC's Foundation Profile which is upward compatible with both Personal Profile and Personal Java. Foundation Profile is widely used on handhelds like PDAs. Our testbed consists of HP iPAQ 3870 running Linux 2.4.18. Each iPAQ contains an Intel StrongARM 206MHz 32bit RISC processor, 32MB flash memory, and 64MB RAM. For communication we use Lucent Orinoco 802.11b Silver PC cards.

Table 3. Effect of Code Brick Size on Single-Hop Round-Trip Time

Size(Bytes)	Round Trip Time(ms)			
	Portable SM Architecture		Original SM Architecture	
	Uncached	Cached	Uncached	Cached
1430	114	124	50	23
2322	126	124	56	23
3456	150	124	63	23
4454	155	124	69	23
8510	165	124	91	23

Table 4. Effect of Data Brick Size on Single-Hop Round-Trip Time

Size(Bytes)	Round Trip Time(ms)	
	Portable SM Architecture	Original SM Architecture
2088	177	29
4056	196	38
8010	234	57
16010	301	88

Tables 3 and 4 compare the cost of migration between the Portable SM architecture and the original architecture. Table 3 shows the variation of the single-hop round-trip time for an SM as a function of the code brick size (the data brick size is negligible in this experiment). Table 4 shows the variation of the single-hop round-trip time of an SM as the size of data bricks varies from 2KB to 16KB (the code brick size constant at 3.51KB). Table 5 shows the cost of tag space operations.

These results show that the Portable SM architecture is more expensive than the original architecture in terms of execution time. This is, however, the price paid for the ability to inject a new distributed application anytime, anywhere on a Java-enabled device without modifying the system software. Note that these results have been obtained using the Reference Implementation of CDC's Foundation Profile which is much slower than the commercial version which has been optimized for different platforms. We believe that significantly better results could be obtained by using the commercial version.

Table 5. Cost of Tag Space Operations

Operation	Time(μs)	
	Portable SM Architecture	Original SM Architecture
readTag	78	21
createTag	89	43
writeTag	71	32
deleteTag	98	56

Table 6. Completion Time for the *Student Group Study* Application, Varying the Number of Students, *N*, from 1 to 5

N	Completion Time(ms)			
	Portable SM Architecture		Original SM Architecture	
	Uncached	Cached	Uncached	Cached
1	4527	4093	1284	1102
2	5212	5031	1944	1783
3	5604	5308	2036	1968
4	6358	6012	2157	1985
5	7863	6339	2198	2148

We have implemented and evaluated the *Student Study Group* application described in Sect. 2. Table 6 shows the time taken to find *N* students for a group study and return to the initiator. We have executed this application over an ad hoc network consisting of 8 HP iPAQs, while varying *N* from 1 to 5 (the nodes of interest have been distributed uniformly in the network). The results indicate that our architecture yields a completion time greater by as much as 3.7 times. The absolute numbers, however, demonstrate that the SM over the portable architecture can still complete between 4.09s and 6.33s when the code is cached. We consider this time reasonable for mobile ad hoc networks composed of resource constrained devices. The effect of code cache is not very significant for this application because of the unavoidable contention encountered in wireless networks coupled with our on-demand content-based routing which involves many broadcast messages in the network.

To summarize our results, we have found out that our implementation performs approximately 2 to 3.7 times slower than the original implementation. We believe that the main reason is the fact that we use an un-optimized virtual machine (Java CVM based on x86/PC Linux development), while the original implementation uses a virtual machine (Java KVM) designed specifically for resource constrained devices. To quantify the impact of the VM, we plan to run our prototype on the optimized, commercially available CVM. Such an experiment will help us evaluate more accurately the cost of portability (i.e., the cost of implementing the SM architecture as a Java runtime system versus the cost of implementing it within the VM).

5 Related Work

Smart Messages (SMs) share the idea of code migration with mobile agents [12, 13], and active networks [14, 15].

Similar to a mobile agent, an SM may be viewed as a task that explicitly migrates from node to node and executes on nodes of interest. 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 a node architecture suitable for resource constrained devices.

To implement execution migration (i.e., transfer of the execution state), two approaches can be used: VM-based or compiler-based. The first approach implies designing new VMs or modifying existing ones to support the capturing and restoring the execution state. The second approach works for unmodified existing VMs, but it involves either a modified compiler, or other tools that insert new pieces of code in the source code or directly in the executable program in order to capture and restore the execution state. In the following, we discuss several systems that transfer the execution state for Java programs.

Similar to the original SM implementation, a number of systems [16–18] have modified the Java VM (JVM) to provide the required state capturing and restoring. Unlike SMs which were designed specifically for networks of embedded systems, these systems are too heavy for resource constrained devices such as cell phones or PDAs.

Funfrocken [9] implements transparent migration using a source code transformation mechanism (pre-processor). Capturing the method stacks is done within exception handlers. When a program requires to migrate, an exception is thrown, and in every method, an exception handler is instrumented to save the state of the method stack by creating a new *state* object for it. The major difference between our approach and this system is in terms of amount of data sent through the network. Using source code transformation, this system increases the size of the byte code with with as much as 470%. Our bytecode instrumenter increases the size of the bytecode only with as much as 3%. Additionally, this system transfers a significant amount of meta-data for the *state* objects. Other disadvantages of this system compared to ours include changes of the signatures for all methods to accept a *state* object as a parameter, and the need for recompilation of all classes (i.e., the complete source code should be available).

Sakamoto et al [7] implement migration using a bytecode transformation scheme that does bytecode verification. Their approach is similar to Funfrocken's in the sense that they also create a *state* object for every method frame on the stack by using the exception handling mechanism, and therefore, incur comparable growth in the code size. Truyen et al [8] have an implementation that also does bytecode transformation by using the bytecode verification mechanism. Unlike the two previously mentioned approaches, they use *return* and *if* instructions to roll back the stack when they are creating *state* objects for every method on the stack which leads to more degradation in performance. Additionally, in order to target multi-threaded environments, they define their own thread-framework: *Tasks* have to be used instead of threads, and a separate scheduler has been implemented. Although expensive, this approach works well for multi-threaded environments.

The main difference between our approach and the above mentioned bytecode transformation approaches is that we manage to capture and restore execution state without iterating through the runtime stack and creating *state* objects for every method instance on the stack. By assuming no use of recursion and local variables across migrations (i.e., they can be used only locally), we have been able to devise a very lightweight migration approach suitable for embedded systems, without compromising the programming model.

6 Conclusions

The contribution of this paper is two-fold. First, we have presented a portable architecture for ubiquitous Java-enabled devices which opens up many possibilities for user-defined distributed applications over ad hoc networks composed of cell phones or PDAs. Second, we have designed and implemented a lightweight execution migration mechanism without system support. The experimental results for simple applications demonstrate the feasibility of our portable Smart Messages architecture.

References

1. Java-enabled Wireless Devices. http://wireless.java.sun.com/device/.
2. Borcea, C., Iyer, D., Kang, P., Saxena, A., Iftode, L.: Cooperative Computing for Distributed Embedded Systems. In: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002), Vienna, Austria (2002) 227–236
3. K Virtual Machine. http://java.sun.com/products/cldc/.
4. Java 2 Platform, Micro Edition (J2ME). http://java.sun.com/j2me/.
5. Borcea, C., Intanagonwiwat, C., Saxena, A., Iftode, L.: Self-Routing in Pervasive Computing Environments using Smart Messages. In: Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom 2003), Dallas-Fort Worth, TX (2003) 87–96
6. Xu, G., Borcea, C., Iftode, L.: Toward a Security Architecture for Smart Messages: Challenges, Solutions, and Open Issues. In: Proceedings of the 1st International Workshop on Mobile Distributed Computing (MDC'03). (2003)
7. Sakamoto, T., Sekiguchi, T., Yonezawa, A.: Bytecode Transformation for Portable Thread Migration in Java. In: ASA/MA. (2000) 16–28
8. Truyen, E., Robben, B., Vanhaute, B., Coninx, T., Joosen, W., Verbaeten, P.: Portable Support for Transparent Thread Migration in Java. In: ASA/MA. (2000) 29–43
9. Funfrocken, S.: Transparent Migration of Java-Based Mobile Agents. In: Mobile Agents. (1998) 26–37

10. MIDP Profile. http://wireless.java.sun.com/midp/.
11. Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.
12. Karnik, N., Tripathi, A.: Agent Server Architecture for the Ajanta Mobile-Agent System. In: Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), Las Vegas, NV (1998) 66–73
13. Gray, R., Cybenko, G., Kotz, D., Rus, D.: Mobile agents: Motivations and state of the art. In Bradshaw, J., ed.: Handbook of Agent Technology. AAAI/MIT Press (2002)
14. D. Wetherall: Active Network Vision Reality: Lessons from a Capsule-based System. In: Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP 1999), Charleston, SC, ACM Press, New York, NY (1999) 64–79
15. Moore, J., Hicks, M., Nettles, S.: Practical Programmable Packets. In: Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001), Anchorage, AK (2001) 41–50
16. Ranganathan, M., Acharya, A., Sharma, S., Saltz, J.: Network-aware Mobile Programs. In: Proceedings of the USENIX 1997 Annual Technical Conference. (1997)
17. Peine, H., Stolpmann, T.: The Architecture of the Ara Platform for Mobile Agents. In: First International Workshop on Mobile Agents MA'97. (1997) 50–61
18. S. Bouchenak: Pickling threads state in the Java system. Third European Research Seminar on Advances in Distributed Systems (ERSADS'99), 1999.