Split Smart Messages: Middleware for Pervasive Computing on Smart Phones

Nishkam Ravi[1], Cristian Borcea[2], and Liviu Iftode[1]
[1] *Department of Computer Science, Rutgers University, USA*
[2] *Department of Computer Science, New Jersey Institute of Technology, USA*
nravi@cs.rutgers.edu, borcea@cs.njit.edu, iftode@cs.rutgers.edu

Abstract

Smart Phone is a recently emerged technology that supports Java program execution and provides both short-range wireless connectivity (Bluetooth/IrDA) and Internet connectivity (GPRS/3G). Smart Phones represent the first viable ubiquitous computing devices because they are becoming an integral part of our daily life. Although these phones are closed systems with limited resources, we believe that a multitude of distributed applications in which Smart Phones act as peers in ad hoc networks can be developed. To realize the potential, there is a need for a middleware that supports such applications and a systematic study of the communication/computation trade-offs. The middleware should provide functionality to support service execution, discovery and migration and should be able to score well on three criteria: portability, security, and performance.

To achieve this goal, we have implemented and evaluated Split Smart Messages (SSM), a lightweight middleware architecture similar to mobile agents, that exploits dual connectivity on Smart Phones. Services can be executed, discovered, and migrated on top of the SSM middleware. To facilitate portability, we have designed an execution migration scheme that works on top of unmodified Java virtual machines. To improve upon security while preserving performance, code is uploaded to and downloaded from a trusted web server, while data and state are transferred across the local network. We have implemented an SSM prototype on Sony Ericsson P800/P900 Smart Phones and compared its performance with that achieved on HP iPAQ PDAs.

1. Introduction

Smart Phone technology is the result of the convergence of cell phones and PDAs and is steadily becoming ubiquitous with all the big mobile phone manufacturers such as Nokia, Sony Ericsson, and Motorola vigorously supporting it. Smart Phones support Java program execution and provide both short-range wireless connectivity (Bluetooth/IrDA) and Internet connectivity (GPRS/3G). Currently, Symbian OS is the dominant platform (holding 67% of the market share), but Microsoft and Palm are interested to increase their presence in the Smart Phone market. Linux can also be found on certain phones such as Motorola A760. Symbian OS supports Personal Java, J2ME, MIDP as well as C++. The *dual connectivity* on these phones allows them to connect to other Bluetooth enabled devices in an ad hoc fashion, while being connected to the Internet all the time. With significant amount of memory and computing power, Smart Phones are becoming increasingly interesting to academia and industry as platforms for realizing the pervasive computing vision [36, 23].

Mobile agents is a well-established idea whose utility has been debated upon numerous times. There are intriguing reasons for mobile agents not having taken off. The well known *chicken-and-egg* problem being the major hurdle. For running mobile agents ubiquitous devices are needed, while for such devices to exist and support mobile agent execution, killer applications are required. Lack of good solutions for establishing security and trust is another reason for rejecting mobile agents and a well justified one in that. We believe that Smart Phones can be instrumental in breaking the perpetual chicken-and-egg problem as they provide a ubiquitous and easy to use platform for mobile agent execution.

So far, Smart Phones have been mostly used for local applications (e.g., games), interacting with web services (e.g., finding restaurants in the nearby areas using a location based web service [3], downloading/uploading media content such as pictures), or sending emails. However, to fully exploit the potential of Smart Phones of becoming a universal interface between end-users and surrounding ubiquitous computing environments, we must be able to leverage their computing and networking capabilities into distributed applications. Essentially, Smart Phones can be used for two types of applications: (1) interacting with the devices/services embedded in the physical world (e.g., open-

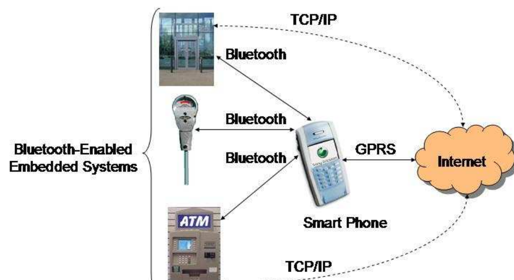

Figure 1. One-hop device/service interaction

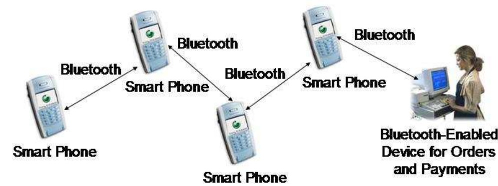

Figure 2. Multi-hop peer-to-peer interaction

ing doors equipped with "smart" locks, paying the parking meters, or controlling home appliances), and (2) interacting with an ad hoc network of stationary or mobile devices in a peer-to-peer fashion (e.g., booking a free cab using a network of wireless-enabled cabs, paying and sharing the bill among a group of friends in a restaurant, peer-to-peer games among ad hoc groups of students in a campus or creating study groups). Figures 1 and 2 illustrate these two types of applications. Currently, the lack of a common execution environment precludes the development of such distributed applications over Smart Phones. Additionally, no systematic study of the communication/computation trade-offs for distributed applications over Smart Phones exists.

In this paper, we propose *Split Smart Message(SSM) Architecture*, which extends the Smart Message(SM) model by addressing the issues of portability and security while preserving performance at the same time. Smart Messages(SM) [12, 10, 20, 30] is a migration based programming model for executing distributed applications, that has been shown to perform excellently over mobile ad-hoc networks of PDAs (HP iPAQs) communicating via IEEE 802.11. The SM middleware is based on execution migration, self-routing, and content-based naming. SMs are migratory units that execute on top of the SM middleware. They execute on nodes of interest named by content and use explicit execution migration to reach these nodes, self-routing themselves on the way. SMs in addition to being lightweight, are capable of providing functionality to support service execution, discovery, and migration in highly volatile mobile ad hoc networks. All these properties make the SM middleware an ideal candidate for Smart Phones. However, like most mobile-agent systems, SMs suffer from lack of portability and security.

Any middleware designed for heterogeneous resource constrained devices should score well on three criteria : portability, security and performance. SSM architecture is portable to any Java virtual machine, is relatively secure and has been observed to perform well on both HP iPAQ PDAs and Sony Ericsson P800/P900 Smart Phones. Services can be executed, discovered and migrated over the SSM middleware.

To facilitate portability, we have designed a lightweight migration mechanism based on Java bytecode instrumentation that works on top of unmodified Java virtual machines. This approach is especially well suited for mobile ad hoc networks which are typically resource and bandwidth constrained.

Dual connectivity on Smart Phones allows them to communicate with other Bluetooth enabled devices, while being connected to the Internet at the same time. Thus, both local services and web services can be accessed. The SSM middleware makes use of this property to implement security. An SSM is a migratory unit similar to a mobile agent and is composed of data, code, and execution state. During migrations, only data and state are transferred across the local network, while code is uploaded to and downloaded from a trusted web server, thereby ensuring that the code is not malicious.

We have evaluated the Split Smart Message Architecture on a network of Sony Ericsson P800 and P900 phones which run Symbian OS. Results indicate that SSMs perform reasonably well on Smart Phones, despite the fact that phones have limited computing power and limited bandwidth over Bluetooth. Single-hop round-trip time of an SSM on phones varies from 1.3 seconds to 2.6 seconds as the size of the SSM is varied from 1KB to 16KB (which is good enough for most SSM applications).

This paper makes the following contributions: (i) it presents a lightweight, portable, and relatively secure middleware for distributed services and applications over mobile ad hoc networks of Smart Phones, (ii) it describes solutions for portability and security that can be applied to any migration-based system designed for resource constrained devices, and (iii) it provides useful results that give an insight into the performance of Smart Phones as well as distributed applications executed over ad hoc networks of Smart Phones.

The rest of the paper is organized as follows. In Sec-

```
i=0;   /* i stored in data brick */
while(i<N){
  migrate("Ubiquitous");
  /* ask attendee to join */
  if (readTag("Joined"))
    i++;
}
migrate("Initiator");
```

Figure 3. Example of Smart Message Code: Ad hoc Creation of a Research Discussion Group at a Conference

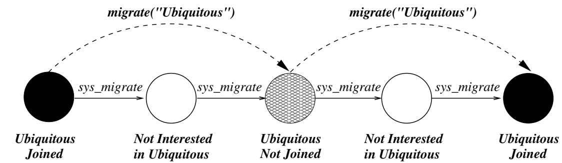

Figure 4. Execution Path for the Smart Message Presented in Figure 3

tion 2, we introduce the Smart Message computing model. In Section 3 we describe the Split Smart Message architecture and discuss security and portability. Section 4 discusses peformance of our architecture. In Section 5 we discuss related work. Conclusions and future research directions are presented 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 properties 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* pointers, *operand stack* pointers).

To support SM execution, the SM runtime system runs inside a Java virtual machine and consists of: (1) *admission manager* that performs admission control on incoming SMs, (2) *code cache* that stores frequently executed code bricks, (3) *scheduler* that dispatches ready SMs (from SM ready queue) for execution on the Java virtual machine, and (4) *tag space* that provides a shared memory addressable by names for inter-SM communication and synchronization. An accepted SM generates a task which gets scheduled for non-preemptive execution. The execution starts with the next instruction following a migration invocation. During execution, an SM can interact with the host or other SMs through the tag space. Tags are used for for “persistent” memory across SM executions. Each tag has a name, similar to a file name in a file system, which is used for content-based naming of nodes.

To illustrate the SM distributed computing model, let us consider a network of handhelds belonging to people attending a conference. At the beginning of the conference, people download on their handhelds a simple SM that creates tags for their research interests. These tags can be used by other SMs to identify people with certain research interests. For instance, a certain person can download an SM that sets up a discussion with *N* people interested in ubiquitous computing (i.e., identified by a tag named *Ubiquitous*) or invites them to have lunch together. This SM works in an ad hoc fashion over short range wireless links and achieves its task even if the attendees do not know each other beforehand.

Each time an attendee wants to start a discussion on a given research topic, or invite people for lunch, she injects this SM in the network from her handheld. The SM migrates through the network until it finds *N* people willing to have such a discussion or meet for lunch. Once the group is set, it returns and informs the initiator. For instance, Figure 3 presents the code for an SM that creates a group discussion for *Ubiquitous* computing. Figure 4 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 [11]. 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(“Ubiquitous”)* routes the SM to people interested in ubiquitous computing using other handhelds (i.e., belonging to people who may or may not be interested in ubiquitous computing), as intermediate nodes. The *migrate* function uses the *sys_migrate* primitive for transferring the SM to the next hop. After migration, the SM resumes from the next instruction following the migrate call.

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. The programmers can choose among multiple library implementations of *migrate*. To reach these nodes, *migrate* implements content-based routing algorithms using *sys_migrate* and routing tags. 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*.

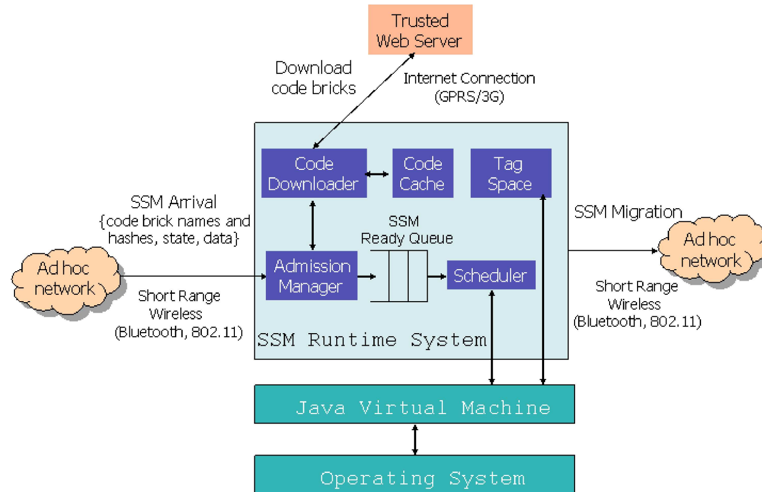

Figure 5. Split Smart Messages Architecture

The SM middleware provides primitives to create, spawn, and block SMs, as well as primitives to access the tag space (subject to authorization). SMs can dynamically create new, possibly smaller SMs by calling *create* or *spawn*. A *create* uses some of the SM's code and data bricks to assemble a new SM and is commonly invoked to build "children" SMs that cooperate with the "parent" SM. An SM may clone itself using *spawn* (similar to the *fork* system call in Unix).

An SM can invoke the *block* primitive to block on a tag until another SM performs a write on that tag. A blocked SM yields the virtual machines; the scheduler inserts it into a *wait queue* associated with the tag. When the tag is written, the scheduler removes that SM from the wait queue and inserts it back in the SM ready queue. To prevent infinite blocking of SMs, *block* has a timeout as parameter; if no write operation takes place within this timeout, the SM is unblocked.

3. Split Smart Messages

We have developed the Split Smart Messages middleware for pervasive computing over Smart Phones based on the Smart Messages [12, 10, 20] architecture. In the design of SSM, we have payed special attention to portability, security and performance, which are essential requirements for any middleware or distributed application for Smart Phones.

An incoming SSM consists of execution state, data, and the hash values of its code bricks (obtained by applying a common hash function on the code bricks). Upon acceptance at a node, according to the admission policy, the admission manager informs the code downloader to download the code bricks, specified by the list of code hashes carried by the SSM, from a trusted web server. An accepted SSM generates a task which gets scheduled for non-preemptive execution.

In addition to executing distributed applications, migratory services can be executed on top of the SSM middleware. This is achieved by using *Service SSMs* which implement migratory services and *Discovery SSMs* which discover services. When a Service SSM starts executing, it creates a tag containing the service profile. Every time the service has to migrate to another node, it creates the same tag on the new node while the old one is deleted. Discovery SSMs discover services by looking for desired tags in the network.

Figure 5 illustrate the runtime system that implements the SSM system support at nodes. In what follows, we will briefly discuss the implementation of the different components of the SSM architecture:

Tag Space: As mentioned in Section 2, tag space is name-based memory. Tag space is composed of tags which are *(name, data)* pairs. These tags are Java objects which can be created, deleted, read from, or written into by SSMs. They provide storage as well as inter-SM communication and synchronization. They are integral to service discovery over SSMs. Commonly, a blocked SSM is woken up by the interpreter when the tag is written by another SSM. Each time an SSM blocks on a tag, its corresponding Java thread is terminated. Each time an SSM is unblocked (and consequently dispatched for execution), a new Java thread is created for it. We are looking into the issue of avoiding naming conflicts for tags.

Admission Manager: The admission manager is responsible for receiving and admitting incoming SSMs over different network interfaces. Our admission manager listens on the TCP/IP socket interface (for receiving SMs over

802.11b) as well as Bluetooth L2CAP interface (for receiving SMs over Bluetooth). While admitting SSMs into the system, the admission manager verifies the data and state against certain verification policies. We are still working on adding verification policies to the admission manager.

Code Downloader: The code downloader for Smart Phones is implemented as a MIDlet that runs over MIDP. MIDP supports OTA (Over-The-Air) Provisioning which is used for implementing dynamic downloading of code. The code downloader is invoked by the Admission Manager everytime code needs to be downloaded.

Code Cache: We exploit Java's classloader to implement the code cache. The Java dynamic class loading 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.

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

3.1. Security

The security issues associated with SSMs are the same as those associated with mobile agents. As mentioned in [19], the security threats for mobile agents can roughly be classified into four categories: *agent to platform*, *platform to agent*, *agent to agent*, *others to agent*. The *others to agent* threat is not specific to mobile agents, but in general applies to any form of data transfer between two untrusted peers. Broadly speaking, the other three categories contain two different security threats: snooping/changing/dropping data, and running malicious code. When we look at a mobile agent as composed of code and data, the security threats specific to them involve malicious code running on a certain platform. Protecting data against threats like replay attacks, middleman attacks, snooping or changing data is a problem common to any form of network communication. Therefore, we assume that state-of-the-art solutions can be applied to protect mobile agents' data, and in the following, we focus on protecting against malicious code.

The agent-to-platform category represents the set of threats in which agents exploit security weaknesses of an agent platform or launch attacks against an agent platform. This set of threats includes denial of service or unauthorized access. Mobile agents can launch denial of service attacks by consuming an excessive amount of the agent platform's computing resources. Mobile agents can gain unauthorized access to confidential data on the platform if they can bypass the platform's security policy. Several techniques have been proposed for protecting the agent platform, namely Signed Code [13], Proof Carrying Code [25], Path Histories [26], Authorization Certificates [33], Safe Code Interpretation [27], State Appraisal [16], and Software-based Fault Isolation [35]. Some of these techniques aim at authenticating the mobile agent or the source of the agent, while others are focused on safe code execution.

The solution that we propose for SSM aims at inducing trust between the agent and the platform by establishing trust between the target platform and the agent source. The dual connectivity on Smart Phones (i.e., short range wireless connectivity over Bluetooth/802.11 and Internet connectivity over GPRS/3G) is the cornerstone of our approach. This dual connectivity provides a simple infrastructure for establishing trust in the local ad hoc network. An SSM is composed of three essential components: data bricks (Java objects), code bricks (Java class files), and execution state. In our portable runtime system implementation (described in detail in the following subsection), execution state is merged with data bricks. To protect Smart Phones against malicious code, the SSM middleware transfers only data bricks over the ad hoc network. The code bricks are downloaded from a trusted web server. We have implemented code downloading from the Internet using OTA (Over-The-Air) Provisioning provided by most cell-phone service providers. Trusted code bricks ensure a certain level of security which can be improved upon by using one of the aforementioned techniques in conjunction. It also promotes cooperation among Smart Phones users.

Downloading code from a trusted web server is safer than relying on authentication certificates presented by an incoming SSM because the SSM could have been tampered with. No safe assumptions can be made about the data/code coming from a machine, unless the machine follows the trusted computing model [8]. Proof carrying code or signed code provide a certain level of security which is not as safe as directly obtaining the code from a trusted entity.

In our current architecture, the code bricks are uploaded to the web server beforehand. The web server would make the code available after suitable authentication which may involve manual work or simulated execution of the code to ensure that it is safe. To support on-the-fly uploading of code bricks, Proof-carrying-code technique or Microsoft's Authenticode [5] could be employed. For this paper, we assume the existence of an authentication web service.

There are many reasons for not migrating the whole SSM over the Internet. First, our present architecture exploits a web service only when available, but does not depend on

Figure 6. Life-cycle of an SSM code brick

it. If the web service is not available, the code bricks can be fetched from the source over short range wireless, by compromising on security. Having an architecture which migrates the whole SSM over the Internet would make it strongly coupled with Internet availability. Second, code bricks can be uploaded to the web server beforehand offline and downloaded on demand, because an SSM always uses the same code bricks. This is an upload-once-download-many strategy. Data bricks keep growing and shrinking in size and number as the SSMs travel across the network. Uploading and downloading data bricks from a web server on the fly, restricts the level of authentication that can be provided by the web server. Third, it is important to minimize Internet usage as there is a cost associated with downloading data from the Internet. Many cell-phone service providers for instance, charge an amount proportional to the amount of data downloaded from the Internet.

3.2. Portability

For implementing migratory applications or services, it is important that they be portable and transferable with minimal overhead. The original Smart Messages architecture was implemented by modifying Sun's Java Kilobyte virtual machine (KVM) [20]. The whole architecture 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, is not portable. Since devices like Smart Phones and Smart Watches come with a pre-installed Java VM (and most of the time users do not want to or cannot modify the system software on their devices), we have designed and implemented a portable SSM runtime system. This system is implemented completely in Java and runs on top of unmodified Java virtual machines.

The main issue to be solved in a pure Java implementation of a portable system is how to perform migration without depending on the VM to capture and restore the execution state. The execution state 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 mechanism for capturing and restoring the execution state by incorporating all the necessary operations in the SSM itself. The heart of our approach lies in instrumenting the SSM bytecode in such a way that the SSM can save its state before migration and restore it before resumption with a minimal overhead. Using this mechanism, the state is encoded in the data bricks, and no explicit state information is shipped. Being resource and bandwidth constrained, mobile ad hoc networks impose constraints on the amount of data that can be transferred for reliable communication. With this in mind, we have focused on making the migration mechanism extremely lightweight and efficient.

We have used this migration mechanism to port SMs on HP iPAQ PDAs in [30], now we have adapted it to port SSMs on Smart Phones. The mechanism is generally applicable to any system based on execution migration for Java programs, however for simplicity we explain it in context of SSMs. To migrate an SSM, we need to migrate its data bricks, and execution control state. The code bricks of SSMs are Java class files and are transferred over the internet and loaded at the destination node using Java dynamic class loading. The Java serialization mechanism is used to marshal/unmarshal the data bricks across migrations. Since SSMs 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 instruction pointer and the method call stack.

Our solution is to instrument the SSM bytecode in such a way that SSMs can capture and restore their own runtime stack before resuming their normal execution at destination. We introduce the term *critical method* to refer to any method that can directly or indirectly invoke *sys_migrate* or *block*. 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 *block*) is statically reachable in order to recognize critical methods. To simplify the exposition throughout this section, we will refer only to migration.

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 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 re-

gions. 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 SSM starts its execution from the beginning of the *run()* method of the main class (i.e., SSMs 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 recreated. 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 SSM 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 SSM is undergoing normal execution or is in resumption mode. If the SSM is resuming, it is necessary to execute the *switch* statement in order to skip the already executed code. If the SSM 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 SSM is set by the system before the SSM is migrated and reset by the SSM itself once the SSM has reconstructed the method call stack, at which point *normal execution* of the SSM begins. To achieve this, the $resumption$ flag is reset after every statement containing a call to $sys_migrate$.

Figure 6 shows the life-cycle of an SSM code brick. Figure 7 briefly demonstrates the working of our instrumentation scheme. The upper part of the figure gives a pictorial view of ip in three critical methods at the time of migration. The arrows in the lower part of the figure show the control flow of the SSM from the time of execution resumption at the destination until the method stack is recreated.

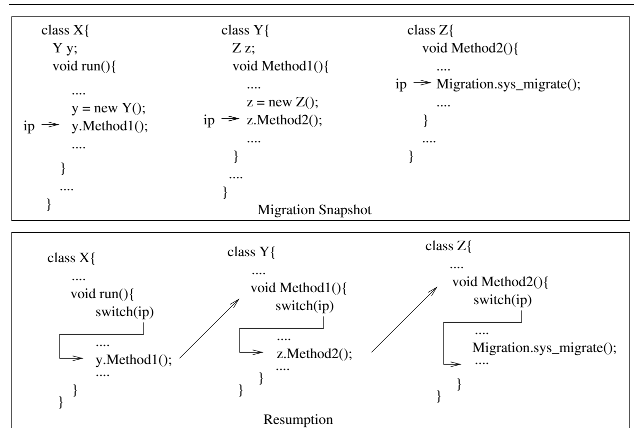

Figure 7. Example of Resuming the Execution after Migration

4. Performance

We evaluated the SSM middleware on Smart Phones as well as HP iPAQs to get an insight into the performance of SSMs as well as Smart Phones. From this comparison, we have learned valuable lessons about the potential and limitations of distributed computing on top of Smart Phones. Our goals in conducting the experimental evaluation were threefold : (1) quantify the impact of bytecode instrumentation on the SSM bytecode size, (2) compare the costs of basic SSM operations on Smart Phones with that on HP iPAQs, (3) compare single-hop round-trip time of an SSM on Smart Phones with that on HP iPAQs to get an estimate of communication costs. Our testbed consists of Sony Ericsson P800 and P900 phones communicating via Bluetooth and HP iPAQs communicating over 802.11b.

Sony Ericsson P800 and P900 phones run Symbian OS, an operating system designed specifically for resource constrained devices such as mobile phones. They also come equipped with two version of Java technology : Personal Java [6], and J2ME CLDC/MIDP [1]. Additionally, they support C++ which provides low level access to the operating system and the Bluetooth driver. The P800 phone, for instance, has 16MB of internal memory and up to 128MB external flash memory.

We have ported the SSM middleware over Personal Java which supports serialization. The middleware can be ported to MIDP as well, but external serialization support would be required which is provided by tools like JSX [4]. We have used Symbian C++ to implement Bluetooth communication and linked it to Personal Java using JNI. OTA (Over-The-Air) Provisioning supported by MIDP and provided by

Table 1. Cost of Individual Operations that Make up the Round-Trip Time on Smart Phones

Size(Bytes)	Time(ms)			
	Bluetooth Connection	Data Transfer	Serialization	Thread Switching
1044	1010	50×2	110×2	20×6
2088	1010	100×2	120×2	20×6
4056	1010	210×2	120×2	20×6
8010	1010	350×2	130×2	20×6
16010	1010	590×2	150×2	20×6

Table 2. Effect of Data Brick Size on Single-Hop Round-Trip Time

Size(Bytes)	Round-Trip Time(ms)	
	HP iPAQ	Sony Ericsson P800/P900
1044	150	1310
2088	177	1500
4056	196	1750
8010	234	2120
16010	301	2590

Table 3. Increase in Bytecode Size Due to Instrumentation

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

most cell-phone providers was used to implement dynamic downloading of code.

We have also ported the architecture on J2ME CDC platform which uses Java CVM as the virtual machine. We have used the Foundation Profile. CDC's Personal Profile is the upcoming replacement for Personal Java and is backward compatible with Foundation Profile which is widely used on PDAs. We have tested it on HP iPAQs running Linux and communicating via 802.11b. Each iPAQ contains a StrongARM 206MHZ processor, 32MB flash memory, and 64MB RAM.

Table 3 shows the increase in bytecode size as a result of instrumenting four of our SSM test cases. We have used Soot1.2.5 [2] to do off-line bytecode instrumentation. On average, we observe an increase of 2.9% in the bytecode size which is negligible as compared to existing approaches (see Section 5 for details).

Table 4 shows the cost of tag space operations. Table 2 compares the cost of SSM execution (including migration) on Smart Phones with that on HP iPAQs. The results indicate that for establishing a Bluetooth connection it takes on an average a constant of 1 second, and the round-trip time varies from 300ms to 1600ms (excluding the cost of establishing a Bluetooth connection) as data brick size is varied from 1KB to 16KB. For all practical purposes, this is good performance. The performance on iPAQs is much better compared to that on Smart Phones, which is expected because iPAQs have more computation power than Smart Phones and 802.11b offers a much higher bandwidth than is offered by Bluetooth.

Table 4. Cost of Tag Space Operations

Operation	Time(μs)	
	HP iPAQ	Sony Ericsson P800/P900
readTag	78	188
createTag	89	578
writeTag	71	203
deleteTag	98	156

Table 1 shows the time taken by individual operations during a single-hop round trip on Smart Phones (rounded to the nearest ten). The cost of thread switching is a constant of 20ms. The cost of serialization varies from 110ms to 150 ms as data brick size is varied from 1KB to 16KB, while the cost of data transfer over Bluetooth varies from 50 ms to 590 ms. The remaining time (not accounted for) is taken by SSM code execution. Note that the time values for data transfer and serialization have been multiplied by 2 to account for the round-trip. During a single-hop round trip, 6 application thread switches are involved. The performance of SSMs on Sony Ericsson P900 phones is identical to that on P800 phones.

The cost of downloading code from the web server has a lower bound of 3 seconds, which is determined by the size of the corresponding *jad* file which is at least 250 bytes.

To summarize our results, we have found that the performance of SSMs on Smart Phones is good for all practical purposes, thereby implying that Smart Phones are suitable devices for migration-based middlewares such as the one proposed in the paper. The performance of SSM on HP iPAQs is better, but this can be attributed to the additional computation power and bandwidth.

5. Related Work

Aalto et al [7] describe a system for Bluetooth and WAP Push based advertising for Smart Phones. Beaufour et al [9] propose the idea of storing digital keys on Smart Phones for opening doors. I-mode [3] makes web service provisioning on Smart Phones easier. Cooltown [22] and Splendor [37] utilize the idea of associating devices and services with the web. In [31] we have described a protocol for provisioning services on Smart Phones that involves using SMs. SSMs is a middleware based on SMs, tailored with Smart Phones in perspective, that provides a common denominator for many kinds of distributed applications and services introduced and published earlier.

Split Smart Messages (SSMs) share the idea of code migration with mobile agents [21, 18], and active networks [15, 24] as well as the security and portability issues.

Unlike mobile agents, SSMs are defined to be responsible for their own routing in a network. This feature combined with content-based routing allows SSMs to adapt quickly to changes that may occur both in the network topology and the availability of resources at nodes. Furthermore, the SSM system architecture is lightweight and defines a node architecture suitable for resource constrained devices. Services can be executed, discovered and migrated on top of the SSM middleware.

SSMs differ from active networks (AN) in several key features. A first difference comes from the problems they try to solve: AN target improved performance for end-to-end data transfers in relatively stable networks, while SSMs help the development of distributed applications on top of a new computing infrastructure which is significantly underused due to the lack of programmability support. Unlike AN, we define a computing model whereby several SSMs can cooperate, exchange data, and synchronize with each other through the tag space. In terms of migration, AN do not transfer the execution state from node to node whereas the SSM model does.

To implement execution migration (i.e., transfer of the execution state), two approaches can be used: VM-based or compiler-based. Similar to the original SM implementation, a number of systems [29, 28, 14] have modified the Java VM (JVM) to provide the required state capturing and restoring, at the cost of loss of portability.

Funfrocken [17] implements transparent migration using a source code transformation mechanism (pre-processor). Sakamoto et al [32] and Truyen et al [34] implement migration using a bytecode transformation scheme that does bytecode verification. The main drawback of these systems is that they are very heavy and inefficient and have been known to increase the size of the bytecode by as much as 470%. Our bytecode instrumenter increases the size of the bytecode only with as much as 3%. Additionally, their system transfers a significant amount of meta-data for the $state$ objects.

By assuming no use of recursion and local variables across migrations, we have been able to devise a lightweight migration approach suitable for embedded systems, without compromising the programming model.

Section 3.1 discusses the various solutions proposed for mobile agent security. While some of the solutions aim at authenticating the mobile agent or the source of the agent, others are focused on safe code execution. None of the solutions has led to the deployment of mobile agents because of the lack of simplicity of the solutions. In the paper, we have shown a simple solution for countering malicious code attacks by making use of internet connectivity on Smart Phones, which can be used in conjunction with state of the art solutions.

6. Conclusions and Future Work

This paper has described Split Smart Messages, a middleware based on execution migration, which to the best of our knowledge is the first middleware for distributed computing over Smart Phones. SSM is portable, relatively secure, lightweight, and has been shown to perform well over Smart Phones. We have devised a lightweight execution migration scheme that works on top of unmodified Java virtual machines. We have shown a simple solution to counter malicious code attacks by exploiting dual connectivity on Smart Phones. The experimental results provide an insight into the performance and capabilities of Smart Phones. This middleware has also demonstrated that Smart Phones represent a suitable target infrastructure for programming models similar to mobile agents.

The issue that we have not addressed in the paper is that of energy efficiency. So far, there is no systematic characterization of the energy consumption on Smart Phones. Energy consumption is going to be a deciding factor in the success of Smart Phones. Since phones, as of now, are closed platforms, our next step is to look into source code transformations for energy efficiency. We are also planning to investigate biometric security solutions (such as fingerprints and voice recognition) for protecting Smart Phones from unauthorized access. We are in the process of implementing applications and services on top of the SSM middleware.

References

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

[2] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[3] i-mode. http://www.nttdocomo.com/corebiz/imode.

[4] JSX. http://jsx.org.

[5] Miscrosoft Authenticode Technology. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnauth/html/msdncodewp.asp.

[6] PersonalJava. http://java.sun.com/j2me/.

[7] L. Aalto, N. Gothlin, J. Korhonen, and T. Ojala. Bluetooth and wap push based location-aware mobile advertising system. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 49–58. ACM Press, 2004.

[8] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture, 1997.

[9] A. Beaufour and P. Bonnet. Personal Servers as Digital Keys. In *Second International Conference on Pervasive Computing and Communications*, 2004.

[10] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial Programming using Smart Messages: Design and Implementation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 690–699, Tokyo, Japan, March 2004.

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

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

[13] N. Borisov and E. Brewer. Active certificates: A framework for delegation.

[14] S. Bouchenak and D. Hagimont. Pickling threads state in the java system. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 22. IEEE Computer Society, 2000.

[15] 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)*, pages 64–79, Charleston, SC, December 1999. ACM Press, New York, NY.

[16] W. M. Farmer, J. D. Guttman, and V. Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the Fourth European Symposium on Research in Computer Security*, 1996.

[17] S. Funfrocken. Transparent Migration of Java-Based Mobile Agents. In *Mobile Agents*, pages 26–37, 1998.

[18] R. 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, 2002.

[19] W. Jansen and T. Karygiannis. Nist special publication 800-19 - mobile agent security, 2000.

[20] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode. Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems. *The Computer Journal, Special Focus-Mobile and Pervasive Computing*, pages 475–494, 2004. The British Computer Society. Oxford University Press.

[21] 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)*, pages 66–73, Las Vegas, NV, July 1998.

[22] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, and M. Spasojevic. People, places, things: Web presence for the real world.

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

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

[25] G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[26] J. J. Ordille. When agents roam, who can you trust? In *First Conference on Emerging Technologies and Applications in Communications (etaCOM)*, 1996.

[27] J. K. Ousterhout, J. Y. Levy, and B. M. Walsh. The Safe-Tcl Security Model. Technical Report SMLI TR-97-60, Sun Microsystems, 1997.

[28] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *First International Workshop on Mobile Agents MA'97*, pages 50–61, April 1997.

[29] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware Mobile Programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 91–104, 1997.

[30] N. Ravi, C. Borcea, P. Kang, and L. Iftode. Portable Smart Messages for Ubiquitous Java-enabled Devices. In *First International Conference on Mobile and Ubiquitous Computing*, pages 412–421, 2004.

[31] N. Ravi, P. Stern, N. Desai, and L. Iftode. Accessing Ubiquitous Services Using Smart Phones. In *Third International Conference on Pervasive Computing and Communications*, 2005.

[32] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *ASA/MA*, pages 16–28, 2000.

[33] M. Thompson, W. Johnston, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari. Certificate-based access control for widely distributed resources. In *Proceedings of the Eighth USENIX Security Symposium*, 1999.

[34] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *ASA/MA*, pages 29–43, 2000.

[35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 1993.

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

[37] F. Zhu, M. Mutka, and L. Ni. Splendor: A secure, private, and location-aware service discovery protocol supporting mobile services. In *First International Conference on Pervasive Computing and Communications*, 2003.