Outdoor Distributed Computing with Split Smart Messages

Nishkam Ravi and Liviu Iftode

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

Abstract. In this paper, we exemplify outdoor distributed computing and point out the key challenges. We present Split Smart Messages, a lightweight, portable, network failure resilient and relatively secure middleware that enables a large subset of outdoor distributed computing applications. We also present a Service Discovery, Interaction and Payment Protocol (SDIPP) tailored for mobile phones. We evaluate our middleware and protocol on Sony Ericsson P900 phones and present experimental results.

1 Introduction

Traditional distributed computing techniques were designed specifically for the client/server paradigm. In this approach, a connection is established with one or more stationary servers and messages are exchanged to complete a task. The connection needs to be maintained during the entire lifetime of a task. The computation is distributed among the different servers for the purpose of optimizing performance. The key properties of typical distributed systems are resource sharing, concurrency, scalability and openness. The machines and the underlying networking medium are assumed to be fairly robust and trustworthy. Failures are treated as anomalies. Consequently, distributed systems are designed with a robust, secure and fairly static infrastructure in mind. While failures and disconnections are taken care of, they are not considered part of normal operation.

In contrast, outdoor distributed computing systems have to be designed to cope with frequently occurring failures and disconnections due to dynamically changing topologies. Failures have to be treated as part of normal operation and therefore, systems have to be designed with a weakly connected challenged network in mind. This network is typically composed of heterogeneous nodes that join and leave the network dynamically and are better identified by properties than by statically assigned names (e.g IP addresses). Nodes typically act as both clients and servers, and can potentially exhibit malicious behavior. Ad-hoc wireless connections with short life-times are dominant. Mobility is common. Prior knowledge of the configuration of the system is limited. While the goal of resource sharing is common with traditional distributed computing, the kind of resources that are shared and the mechanisms used for sharing those resources are quite different.

Properties apart, the goals of outdoor distributed computing are fundamentally different from the goals of traditional distributed computing. While traditional distributed computing is focused primarily on optimization of performance through sharing of resources, outdoor distributed computing enables new functionalities and applications. In this paper, we identify the set of applications that define this new form of computing (Section 2.1) and point out the challenges common to most applications (Section 2.2). We then present the design and implementation of a middleware for outdoor distributed computing (Sections 3 and 4).

2 Outdoor Distributed Computing

What is outdoor distributed computing? We answer this question by identifying the set of applications that motivate this new form of computing and identifying the challenges synonymous with this class of applications.

2.1 Motivating Applications

Vehicular Computing There is growing interest in equipping vehicles with portable computers to enable applications such as real-time congestion estimation, collision avoidance, route planning, content sharing, etc. Vehicles form a mobile ad-hoc network and disseminate information among themselves. Various data propagation models can be conceived, such as broadcast, geographical routing and publish/subscribe. Vehicles typically exchange information about each other (e.g location and speed) and about the environment (e.g accidents and signs). The information can be exchanged proactively (e.g through broadcast) or queried on demand and routed between the source and destination. Furthermore, the information can be collected and dissipated incrementally using store-and-forward mechanisms.

Social Networking Akin to vehicular computing, which involves information exchange in a mobile ad-hoc network of vehicles, social networking involves information exchange in a mobile ad-hoc network of people. For example, researchers at a conference may wish to exchange profiles, or invite people with similar research interests for lunch. A distributed application that executes on the handhelds of researchers and finds people with similar research interests is a typical example of a social networking application. Yet another example is *ad-hoc carpooling*, where a user initiates a request to carpool to a certain destination and his neighbors answer the query if they are interested. We can extend this class of applications to include distributed information exchange between handhelds of soldiers on a battlefield, or firemen on duty.

Location-based Services There is great value in making information services highly personalized. Using location information is one of the best ways of personalizing services. Emergency services led by E911 in North America and E112

in Europe have motivated the wireless carriers to deploy localization technologies. Tracking personnel (e.g patients in a hospital) and assets (e.g objects in a store) using location is predicted to become fairly popular. A typical example is a *friend-tracking* application, where a group of friends keep track of each other's locations using their handhelds while traveling or when in a museum. Provisioning information to the user's handheld (such as list of restaurants) or sending notifications (such as a sale on men's suits in the proximity) based on user's location can enable new business models. Location-based billing is yet another example. A user can establish personal zones, such as a home zone or work zone and arrange preferential billing with his wireless service provider. Similarly, telematics-centered location-based services can aid traffic monitoring and congestion avoidance. It is speculated that location-based services will create a multi-billion dollar market.

Environment Query Deployment of sensors that gather information about the environment has already begun. Cameras on New York streets, temperature sensors in forests of California, pressure sensors in bridges are just a few examples. RFID tags and readers are commonly used today in many applications. Networking these sensors and then linking them to more powerful devices such as car PCs or handhelds can enable plethora of distributed computing applications.

2.2 Challenges and Requirements

Numerous research challenges need to be overcome in order to realize the applications described above. In this section, we point out some of the key challenges that are common to these applications.

Opportunistic Networking Mobility leads to periods of disconnection. Networking infrastructure may only be intermittently available. This makes the design of outdoor distributed computing applications challenging. An ideal design should treat disconnection as part of normal operation, and look for *opportunities* to pass information/data along. IP is clearly insufficient; delay tolerant networking solutions [28], store-and-forward mechanisms, and opportunistic routing algorithms [34] are needed. Since the network is not pre-configured, protocols are needed for resource and service discovery [14, 43, 16, 4]. For this to happen, new naming conventions are needed to uniquely identify resources and services, as IP addresses do not scale well to highly dynamic scenarios.

Portable Middleware Devices over which distributed computing applications execute will range from tiny sensor nodes to mobile phones to car PCs. In other words, we are looking at a network of nodes with different operating systems, varying computation power and heterogeneous networking capabilities. For such heteregeneous devices to successfully cooperate in the execution of a common task, there is a pressing need for designing middleware that can hide the underlying software abstractions from the applications and create a homogeneous

virtual environment for applications to execute in [56]. At the same time, such middleware should be easily portable to different operating systems in order to maximize code reuse and minimize development effort. For performance reasons, the middleware should impose low computation and networking overheads.

Context Awareness User context can be helpful in personalizing services and applications [57]. Examples of context information include location, time of day, user activity, user profile, available networking interfaces, environment, etc. There are numerous challenges in creating context-awareness for outdoor distributed computing applications. Active research is being carried out in sensing user activity and location [61, 50, 18, 54, 20, 52]. Context information from different sensors needs to be fused and reasoned with to obtain directly usable information. Ontologies are being developed to reason with context information [9, 25]. Practical problems, such as storage and retrieval of context, cannot be overlooked.

Security and Privacy Outdoor distributed computing relies on cooperation between alien devices. Some of these devices would interact with each other for a few seconds, a few minutes at the most, and would likely not cross paths again ever after. Personal data in the form of context information would be heavily shared. In such a computing model, both user devices and data are prone to attacks. Reputation-based security schemes are hard to devise, due to the ad-hoc nature of the network. Also, reputation schemes require strong identities which exacerbate the privacy problem. Novel security models that induce trust in the network are needed. Privacy mechanisms that restrict flow of sensitive information need to be devised. Location information, in particular, is very sensitive. Users would not like their location to be known to others all the time. The US government realized the seriousness of the this problem and released the *Location Privacy Protection Act* [5] in 2001. Since then, there has been some research focused on safeguarding location privacy [32, 53].

Energy Optimization Due to mobility, majority of the devices on which outdoor distributed computing applications execute, are battery powered. This includes mobile phones, laptops, PDAs and sensors. Battery has a limited lifetime, and therefore energy optimization mechanisms need to be applied at all levels, including hardware, operating system and compiler. The applications themselves need to be designed with this constraint in mind and should be inherently *lightweight*. One way to accomplish this is to offload computing to a wall-powered server whenever opportunities to connect to the server are available. This approach is called *cyber foraging* [19]. Another approach is to trade the fidelity of applications for energy [44]. Such mechanisms need to be applied during application design phase, as opposed to compiler or OS based energy optimization mechanisms, which are hidden from the applications.

Incentive for Cooperation Cooperation is the key to outdoor distributed computing applications. Why should devices cooperate? Why should a device forward packets for another device? What if a device exhibits selfish, reserved or parasitic behavior? Although the symbiotic nature of these applications promotes cooperation, there is a need for devising incentive models so that devices can benefit from cooperating with other nodes, even if other nodes refuse to cooperate. Such models will drive the equilibrium towards cooperation. A good starting point is to follow the design of reputation-based schemes, which are used to promote trust in peer-to-peer networks, and tailor them to work for short-lived interactions.

Human-Computer Interaction Human-computer interaction issues will play a big role in the success or failure of these applications. Since user attention is a limited resource, the complexity of using and interacting with such applications has to be minimized. Novel user interfaces are needed. Speech recognition, gesture recognition and vision-based solutions can be very helpful. The applications have to be robust and reliable even in the presence of network failures to ensure satisfactory user experience.

Bootstrapping Bootstrapping of outdoor distributed applications is a hard problem, and there are numerous reasons for it. First, while these applications are assumed to be ad-hoc in nature, there is a minimal amount of configuration required on every device to bootstrap these applications. The configuration effort may outweigh the convenience that a user gets in return. There is a need to hide this configuration effort from the users. Second, the users have to carry with them a minimal amount of hardware and software to avail of such services and applications. This contradicts the spontaneous nature of these applications. The solutions should be designed around hardware that users already carry. Also, the software layer required for bootstrapping these applications should be bare minimum. Third, a certain amount of infrastructure is required to support many of these applications/services. For such an infrastructure to exist, right economic models are needed. While these applications have a lot of utility, users must be willing to pay a price for using these applications. There is a need to minimize the infrastructure requirement to improve chances of deployment. Fourth, due to the fact that these applications rely on cooperation between devices, a user may not be convinced about paying a price for a technology that he/she cannot use unless others have it too. The dependencies need to be minimized. Fifth, technology is not matured enough to support robust, secure and distraction-free distributed applications in ad-hoc networks. The research challenges pointed out in this section need to be overcome before such applications can become a reality.

2.3 Technology Enablers

As mentioned before, in order to make bootstrapping easier, it is important to design solutions around devices that have the greatest potential of becoming ubiquitous. Here we identify such devices.

Mobile Phones Mobile phones are carried by almost everyone. Hitherto, they were mostly used for the purpose of making and taking phone calls on the move. With advances in hardware and improving storage trends, mobile phones are evolving into personal computing devices. Sony Ericsson P900, which is a commonly used phone today, runs Symbian OS, an operating system designed specifically for mobile phones, and comes equipped with two different versions of Java: Personal Java [10] and J2ME CLDC/MIDP [1]. It also supports C++ in order to provide low-level access to the system. The phone has 16MB of internal memory and upto 128MB external flash memory, which is small compared to newer phones that have GBs of storage capacity. What is particularly noteworthy about these phones, is the hybrid wireless networking capabilities that they come equipped with, which includes Bluetooth, WLAN, IR and internet connectivity via GPRS. These phones can serve as personal computers for all practical purposes, if only the display could be made bigger and battery lifetime improved. There is already some research initiative in exploring the potential of mobile phones as next generation personal computers [35, 55].

Car PCs Cars have carried on-board computers for decades and modern mobile microprocessors control everything from engine performance to car's instrument cluster. There is ongoing effort in making car PCs more popular and sophisticated than they are today. There are numerous vendors that assemble and sell car PCs (e.g Xenarc, Logisys etc). Several cars already come equipped with full car PCs that run Windows XP (e.g Nissan 350Z and Peugeot 307 XSi). So far these systems are used for stand-alone applications such as navigation systems. DSRC is a block of spectrum in the 5.850 to 5.925 GHz band allocated by US FCC to enable communication between cars. Several car manufacturers, including Toyota, Daimler Chrysler and GM are funding research for the development of distributed applications based on car-to-car communication.

Sensors GPS, thermometer, calorimeter, multi-meter, magnetic compass, barometer, RADAR sensor, infra-red sensor, RFID reader, camera are just a few examples of sensors that we use in our daily life. Sensors are the "eyes and ears" of outdoor distributed computing applications. They gather information about the environment, which serves as context and aids in provisioning services to the user. Location sensors, activity sensors, RFID readers, temperature sensors, cameras, are some of the more directly usable sensors for distributed applications. The size of these sensors varies and so does their cost and utility. Many of these sensors are programmable and can be networked together and linked to more powerful computing devices to enable many interesting outdoor distributed computing applications. While some of them are meant to be wall-powered, others are meant to be scattered in the environment and survive on batteries.

3 Split Smart Messages: A Middleware for Outdoor Distributed Computing

In this section, we describe the design of Split Smart Messages, a middleware for outdoor distributed computing. Split Smart Messages is an extension of the Smart Message [37] model and has been tailored to suit the requirements of resource constrained devices such as Smart Phones which come with a pre-installed JVM. In the design of SSM, we have payed special attention to opportunistic networking, portability, security and lightweightness.

The design of Split Smart Messages has been inspired by mobile agents. A Split Smart Message (SSM) is a user-defined application whose execution is distributed over a series of nodes using execution migration. The nodes on which SSMs execute, called *nodes of interest*, are named by properties and discovered dynamically using application controlled routing. To move between two nodes of interest, an SSM calls explicitly for execution migration, and routes itself without any underlying routing support. An SSM consists of *code bricks*(e.g Java class files) and *data bricks*(e.g Java objects which store data and execution state). SSMs in addition to being lightweight, provide the functionality to support service execution, discovery, and migration in highly volatile mobile ad hoc networks. SSMs are resilient to network failures, as they carry the code for routing themselves, and can therefore store-and-forward themselves opportunistically.

3.1 SSM Middleware Architecture and Implementation

Every participating node has to be equipped with the SSM middleware. The SSM middleware is written completely in Java (using J2ME CDC and CLDC), and can be ported to the common JVMs. It consists of the following components (as shown in Figure 1):

Tag Space: Tag space is name-based virtual memory. It is composed of tags which are *(name, data)* pairs. These tags are Java objects that can be created, deleted, read from, or written into by SSMs. Nodes are identified by properties that are stored in tags. Also, services running on these nodes create tags for advertising themselves. Tags are, therefore, integral to content-based routing and service discovery over SSMs.

In addition to providing storage, tags also provide inter-SSM communication and synchronization. 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.

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 bricks and state against certain verification policies.

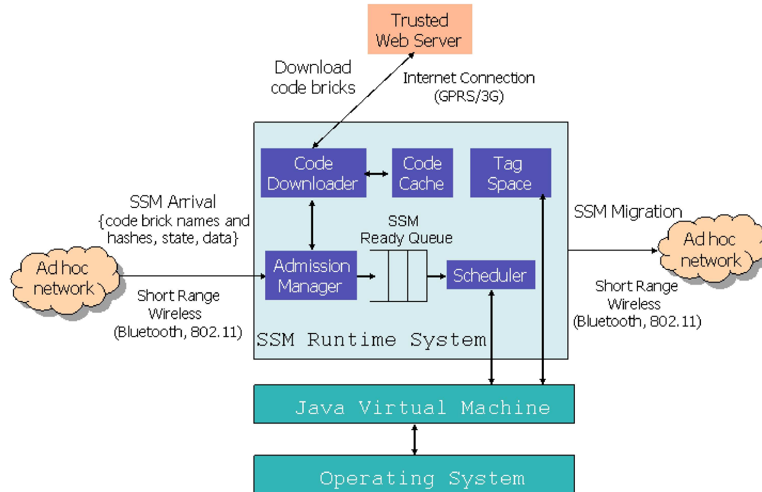

Fig. 1. Split Smart Messages Architecture

Code Downloader: Smart Phones are equipped with multiple network interfaces: WLAN, Bluetooth, GPRS/3G. Future car PCs are believed to have multiple network interfaces too. By virtue of these hybrid networking capabilities, these devices are capable of communicating with each other over short-range wireless (Bluetooth or WLAN), while being connected to the internet at the same time (via GPRS/3G). SSMs have been designed with this feature in mind.

As mentioned before, an SSM is composed of: code bricks (which are Java class files in our implementation) and data bricks (which are Java objects in our implementation). During migration, if internet connectivity is available, only data bricks are transferred across the local network (using WLAN/Bluetooth), while code bricks are uploaded to and downloaded from a trusted web server (using GPRS/3G) [1]. This helps security as described later in this section. If internet connectivity is not available, then code bricks are transported along with data bricks.

The component that downloads code from the trusted web server is implemented as a MIDlet that runs over MIDP [1]. 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: Code cache stores frequently used code bricks. In order to implement Code cache, we exploit Java's classloader. 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 un-

[1] Hence, the name *Split* Smart Messages

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

Fig. 2. Example of Split Smart Message Code: Ad hoc Creation of a Research Discussion Group at a Conference

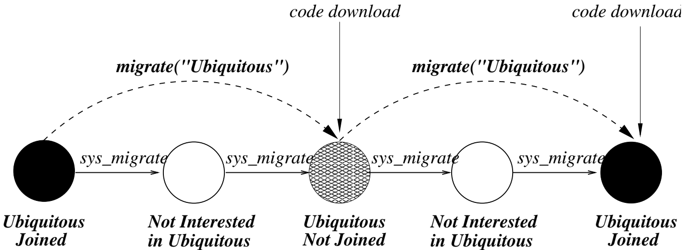

Fig. 3. Execution Path for the Split Smart Message Presented in Figure 2

loaded by the classloader. When the caching policy chooses a class for eviction, we just remove the stored reference for that class.

Scheduler: The scheduler is responsible for dispatching SSMs (from the SSM ready queue) for execution on the JVM. 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.

3.2 Example

To illustrate the SSM 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 SSM that creates tags for their research interests. These tags can be used by other SSMs to identify people with certain research interests. For instance, a certain person can download an SSM 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 SSM 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 SSM in the network from her handheld.

The SSM 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 2 presents the code for an SSM that creates a group discussion for *Ubiquitous* computing. Figure 3 depicts the execution path of this SSM over five nodes.

The key operation in the SSM programming model is migration, which implements content-based routing using tags [21]. An SSM 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 SSM 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.

3.3 Portability

For implementing migratory applications or services, it is important that they be portable and transferable with minimal overhead. The original Smart Message architecture [37] was implemented by modifying Sun's Java Kilobyte virtual machine (KVM). 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 SSM middleware such that it can execute on top of unmodified Java virtual machines.

The main issue to be solved in a pure Java implementation of a migration-based middleware is performing migration without requiring 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 focussed on making the migration mechanism extremely lightweight and efficient. Our Java bytecode instrumentation mechanism increases the Java bytecode size by only 3% as opposed to previously proposed portable Java migration mechanisms, which increase the bytecode size by as much as 400%. The mechanism is generally applicable to

any system based on execution migration of Java programs For details on our instrumentation mechanism, refer to [45].

3.4 Security

The security issues associated with SSMs are the same as those associated with mobile agents. As mentioned in [36], 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 [22], Proof Carrying Code [42], Path Histories [46], Authorization Certificates [59], Safe Code Interpretation [47], State Appraisal [29], and Software-based Fault Isolation [60]. 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. As mentioned before, devices such as Smart Phones and Car PCs support *dual connectivity*. Dual connectivity provides a simple infrastructure for establishing trust in the local ad hoc network. An SSM is composed of two essential components: data bricks (Java objects) and code bricks (Java class files). To protect devices against malicious code, the SSM middleware transfers data bricks over the ad hoc network, while code bricks are downloaded from a trusted web server (when internet connectivity is available). Trusted code bricks ensure a certain level of security, which can be improved upon by using one of the aforementioned techniques in conjunction.

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 [17].

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 analysis or abstract interpretation 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 [7] could be employed. We assume the existence of an authentication web service.

There are many reasons for not migrating the whole SSM over the internet. First, our current design is opportunistic in nature and exploits a web service only when available, but does not depend on it. If the web service is not available, the code bricks can be fetched from the source over short range wireless. Having an architecture that 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 3G/GPRS service providers charge an amount proportional to the amount of data downloaded from the internet.

3.5 Performance

We evaluated the SSM middleware on Smart Phones as well as HP iPAQs to get an insight into the performance of SSMs on resource constrained devices. 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 over Bluetooth, and HP iPAQs communicating over 802.11b.

Table 1 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 2 shows the cost of tag space operations. Table 3 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

Table 1. Increase in SSM Bytecode Size Due to Instrumentation

Unmodified Byte-code(KB)	Modified Byte-code(KB)
1084	1122
1230	1266
1527	1564
2330	2395

Table 2. Cost of SSM Tag Space Operations

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

than Smart Phones and 802.11b offers a much higher bandwidth than Bluetooth. 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.

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

Size(Bytes)	Round-Trip Time(ms)	
	HP iPAQ	Sony Ericsson P800/P900
1044	150	1450
2088	177	1600
4056	196	1790
8010	234	2120
16010	301	2630

4 SDIPP: Service Discovery, Interaction and Payment Protocol

In this section, we describe the design and implementation of a protocol for provisioning services on devices with dual connectivity (e.g smart phones). The services execute on top of the SSM middleware as *Service SSMs*, and can therefore migrate themselves. For advertising themselves, the services create tags on the nodes they execute on, which can be discovered using *Discovery SSMs*. In

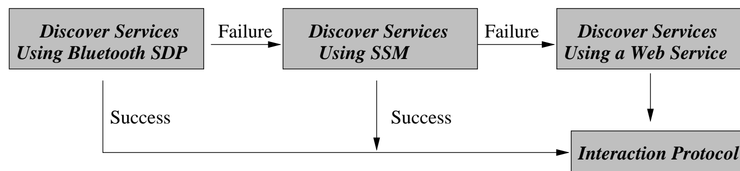

Fig. 4. SDIPP Discovery Model

addition, they register themselves with a web server that is publicly known. When Bluetooth is available on the device, the services can also be discovered using the Bluetooth Service Discovery Protocol(SDP).

4.1 Architecture

Bluetooth engine, GPRS Engine and *Cache* are the building blocks of the protocols. Bluetooth Engine is invoked by the protocols to discover or interact with the services in the proximity. It is a layer above the Bluetooth stack and provides a convenient Java API similar to JSR-82 for accessing the Bluetooth stack. *GPRS Engine* is invoked to carry out the communication with the web services over GPRS.

Cache is persistent storage. The personal information of the user along with her preferences regarding services are stored in the cache. Personal information of the user may include name, age, address, credit card number etc. Storing personal information serves two purposes: first, it provides a way of identifying the user and authenticating her if need be; second, personal information along with preferences and location help in identifying the best suited service for the user during service discovery phase thereby making SDIPP context-aware. Cache also stores the interface/protocol downloaded by the interaction protocol and the data needed across protocols or across sessions.

4.2 Discovery Protocol

Bluetooth SDP provides service browsing without apriori knowledge of the service characteristics. It does not include functionality for accessing services, however, it can be used in conjunction with some other protocol for accessing services. Our discovery protocol is hierarchical in nature, and is a 3-step process as summarized below:

One-hop discovery: Services in the proximity (one-hop) are discovered using Bluetooth SDP. If the list of services discovered by Bluetooth SDP includes the desired service, the discovery phase is over. If it does not include the desired service, but instead lists a Service Discovery Service(SDS), the SDS is invoked to locate the desired service in a multi-hop fashion.

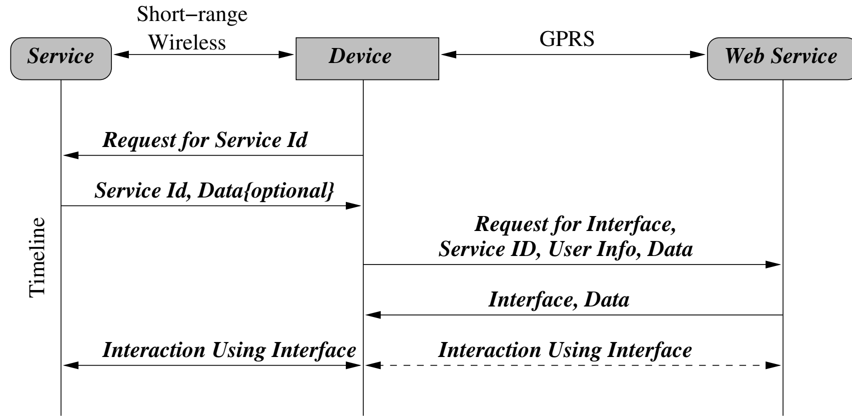

Fig. 5. SDIPP Interaction Protocol

Multi-hop discovery: In our implementation, SDS is implemented using Split Smart Messages. Discovery SSMs broadcast themselves in the ad-hoc network and look for tags with the desired properties. These tags are created by services for the purpose of advertising themselves. When a desired tag (i.e service) is found, the requester is informed.

Web-based discovery: Services also register themselves with a public web service and periodically update their information. If internet connectivity via GPRS is available, the public web service is contacted and requested for information about the desired service (e.g location).

4.3 Interaction Protocol

In outdoor distributed computing scenarios, the interaction of the user with a service is assumed to be spontaneous, and therefore the protocol for interacting with the service would need to be learnt on the fly. Our interaction protocol is inspired by Jini [4]. Every service registers itself with a web server, which assigns it a unique id and stores the interface which can be downloaded for interacting with the service. Figure 5 shows the interaction protocol. The protocol can be summarized as follows:

– The device lists the services discovered during discovery phase. The desired service is requested for its id over short-range wireless.
– The service responds with its id.
– The id along with the personal information of the user stored on the device is sent over to a trusted web server over the GPRS connection. The personal information of the user would be used for authenticating them if the service requires that.

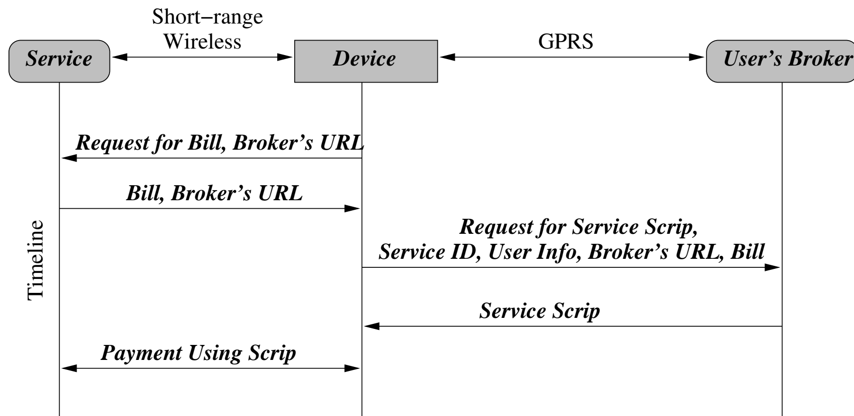

Fig. 6. SDIPP Payment Protocol

- The web server, after authenticating the request, responds with the code and data needed for interacting with the service. The code is a Java program that contains the protocol and interface for interacting with the service.
- Since the code is obtained from a trusted server it is assumed to be safe and is dispatched for execution on the device. All further communication between the device and the service takes place as a result of executing the downloaded code.

Note that the web server(s) for storing the downloadable interface for the services may be different from the web server(s) that act as service directories. We implement downloading of code from the internet using OTA (Over-The-Air) provisioning [8].

4.4 Payment Protocol

Our protocol for paying services is based on the electronic cash representation proposed by the Millicent protocol [30]. Millicent proposes the idea of using accounts based on scrip and brokers to sell scrip. A piece of scrip represents an account the user has established with a vendor. At any given time, a vendor has outstanding scrip (open accounts) with the recently a users. The balance of the account is kept as the value of the scrip. When the customer makes a purchase with scrip, the cost of the purchase is deducted from the scrip's value and new scrip (with the new value/account balance) is returned as When the user has completed a series of transactions, he can "cash in" the remaining value of the scrip (close the account). Brokers serve as accounting intermediaries between users and vendors. Customers enter into long-term relationships with broke the same way as they would enter into an agreement with a bank, credit card company, or internet service provider. Brokers buy vendor scrip as a service

to users and vendors. Broker scrip serves as a common currency for customers to use when buying vendor scrip, and for vendors to give as a refund for unspent scrip.

We try to satisfy the design principals described in [48]. In our model, the broker is a web service that the user already has an account with. The vendor is the service that the user wishes to use and pay for.

Figure 6 illustrates the payment protocol. It as can be summarized as following:

- The device requests the service for its broker's URL and the bill over short-range wireless.
- The service responds with its broker's URL and the bill.
- The service id, broker's URL and bill amount is sent over to the user's broker over GPRS along with the personal information of the user stored on the device.
- User's broker buys service scrip from service's broker on user's behalf. The amount of scrip bought is greater than or equal to the bill amount.
- User's broker responds to the device with the service scrip .
- User pays the service using the service scrip.

Brokers are assumed to be trusted services that have service providers as their clients and other brokers as their peers. Even if the broker tries to cheat, the customer and the service provider can independently check the scrip and detect broker fraud. Service provider fraud consists of not providing service for valid scrip or deducting more amount from the scrip than is valid. If the service provider tries to cheat, the customer can detect the fraud and complain to the broker who will take care of it.

If the customer is cheating, then the service provider's only loss is the cost of detecting the bad scrip and denying service. Every transaction requires that the customer knows the secret associated with the scrip. The protocol never sends the secret in the clear, so there is no risk due to eavesdropping. No piece of scrip can be reused, so a replay attack will fail. Each request is signed with the secret, so there is no way to intercept scrip and use the scrip to make a different request.

This payment protocol provides a security model that is well suited for profit-based services, where the service and the user need to be authenticated to each other and anonymity maintained at the same time.

4.5 Evaluation

The SDIPP protocol was implemented and tested on Sony Ericsson P900 phones which have both Personal Java and MIDP in addition to C++. We used MIDP and JSR-82 (Java Bluetooth API) to implement the architecture. Table 4 shows the time of completion for the different phases of the SDIPP protocol. The time of completion of the Interaction Protocol depends on the size of the code downloaded from the internet. The lower bound is determined by the size of the *jad* file of the corresponding code which is typically 250 Bytes. The time of

Table 4. Performance Evaluation of SDIPP

Operation	Average Time of Completion
Bluetooth Service Discovery	22.5 sec
Ad-hoc Service Discovery	2 sec $\times$ No. of Hops
Web directory lookup	2.5 sec
Interaction Protocol(Lower Bound)	3 sec
Payment Protocol	6 sec

completion of the ad-hoc service discovery over Split Smart Messages depends on the number of nodes (hops) involved.

We have implemented and tested a few applications on top of this protocol. We have also gained some experience in the process. For details refer to [55].

5 Related Work

Split Smart Messages (SSMs) share the idea of code migration with mobile agents [38, 31], and active networks [26, 41], 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. Primary 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 that is unreliable and under-utilized due to the lack of programmability. 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.

Tag Space bears resemblance with tuple spaces [24, 40]. While both offer persistent shared memory for applications, the essential difference is that the tag space is local to every node.

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 of the execution state. The second approach works for unmodified 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.

Similar to the original SM implementation, a number of systems [51, 49, 23] have modified the Java VM (JVM) to provide the required state capturing and restoring. Unlike SMs, which were designed specifically for networks of resource constrained devices, these systems are too heavy for devices such as cell-phones or PDAs.

Numerous service discovery protocols(SDPs) have been proposed. Each has its own infrastructure requirements and target audiences. Bluetooth SDP [14] follows the client-service model and enables nearby devices to discover services on each other. Bluetooth is a low power protocol making it suitable for energy-constrained devices. Bluetooth SDP is query based, which means that clients query for available services rather than services pro-actively announcing their presence. It uses unicast and broadcast as the communication mechanism and does not provide any service invocation mechanism. DEAPspace [43] proposed by IBM research, is aimed for single-hop ad hoc environments. Nodes cache service information and periodically broadcast it to share the information with each other. DEAPspace follows the client-service model. INS [16] is a hierarchical resource discovery and service location protocol that uses a late binding mechanism to provide resilience against name-to-location mapping changes. INS follows the client-service-directory model where the directory is distributed. INS is scalable and targets peer-to-peer networks.

Salutation [13] follows the client-service-directory model and uses RPC for service invocation. Salutation provides a transport-independent interface to applications making it very flexible. Effort has been made to map Salutation APIs to Bluetooth Service Discovery Layer [27]. Service Location Protocol [33] is a lightweight protocol that targets service discovery within a site. It uses URL-based service invocation mechanism. Directories are optional. SLP supports both service announcements and client queries. Universal Plug and Play(UPnP) [6] is a device oriented service discovery protocol that targets home and office environments. UPnP follows the client-service model and uses XML for service invocation. UPnP is being actively advocated by Microsoft. Splendor [62] follows a *client-service-proxy-directory* model, where *proxy* is used to achieve privacy, authentication and load-sharing. Jini [4] follows client-service-directory model, where the directory not only provides service look-up but also downloadable Java code/objects for interacting with the service using RMI. Jini targets enterprise environments. UDDI [12] uses a web-based distributed directory that enables profit-based web services to list themselves on the internet and discover each other, similar to *yellow pages.*

Aalto et al [15] describe a system for Bluetooth and WAP Push based advertising for Smart Phones. They utilize Bluetooth for positioning the end-user's handheld and obtaining the Bluetooth address. The advertisements are then pushed to the phone over WAP. Scott et al [58] propose machine readable *visual tags* for bypassing Bluetooth service discovery on phones. Visual tags can be recognized by phone cameras and can improve device discovery time. However,

this restricts discovery to only devices that are visible and increases human-intervention (for scanning the environment with phone camera).

Cooltown [39] and Splendor [62] utilize the idea of associating devices and services with the web. UDDI [12] is a web-based distributed directory that enables profit-based web services to list themselves on the internet and discover each other, similar to *yellow pages*. NTT DoCoMo's I-mode [3] makes web service provisioning on Smart Phones easier. Recently, they adopted Sony's contactless smart card technology called Felica [11], which can be used for electronic payment. Felica implements RF functions for wireless communication and is therefore extremely short-range.

6 Conclusions

In this paper, we exemplified outdoor distributed computing and identified the key challenges. We presented a middleware, called Split Smart Messages, that enables a large subset of outdoor distributed computing applications. We evaluated it on a testbed of HP iPAQs and Sony Ericsson P900 phones and found it lightweight, portable, resilient to network failures and relatively secure. We also presented a protocol, called SDIPP, which exploits dual connectivity on devices (e.g smart phones) for service provisioning (discovery, interaction and payment); and evaluated its performance on Sony Ericsson P900 phones.

Acknowledgements This material is based upon work supported by NSF under grants ANI-0121416 and CNS-0520123.

References

1. MIDP Profile. http://wireless.java.sun.com/midp/.
2. Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.
3. i-mode. http://www.nttdocomo.com/corebiz/imode.
4. Jini Network Technology. http://wwws.sun.com/software/jini.
5. Location privacy protection act, 2001. http://www.theorator.com/bills107.
6. Microsoft Upnp Specification.
7. Miscrosoft Authenticode Technology. http://msdn.microsoft.com/library/default.asp.
8. OTA Provisioning. http://java.sun.com/products/midp.
9. Owl web ontology language. http://www.w3.org/TR/owl-features/.
10. PersonalJava. http://java.sun.com/j2me/.
11. Sony Felica Technology. www.sony.net/Products/felica.
12. UDDI. http://www.uddi.org.
13. White Paper : Salutation Architecture, 1998. http://www.salutation.org/whitepaper/originalwp.pdf.
14. Bluetooth Specification Part E. Service Discovery Protocol (SDP), 1999. http://www.bluetooth.com.
15. 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, Boston, MA, June 2004.

16. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 186–201, Charleston, SC, December 1999.
17. W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, pages 65–71, Oakland, CA, May 1997.
18. P. Bahl and V. N. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *INFOCOM (2)*, Tel-Aviv, Israel, March 2000.
19. R. Balan, J. Flinn, M. Satyanarayanan, S. Sin, and H. Yang. The case for cyber foraging. In *Proceedings of 10th ACM SIGOPS European Workshop*, Saint Emilion, France, September 2002.
20. L. Bao and S. S. Intille. Activity recognition from user-annotated acceleration data. In *Proceceedings of the 2nd International Conference on Pervasive Computing*, Vienna, Austria, April 2004.
21. 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)*, pages 87–96, Dallas-Fort Worth, Texas, March 2003.
22. N. Borisov and E. Brewer. Active certificates: A framework for delegation, 2000.
23. 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)*, Santa Barbara, CA, August 2000.
24. N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.
25. H. Chen, T. Finin, and A. Joshi. An ontology for context-aware pervasive computing environments. *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review*, 18(3):197–207, 2003.
26. D. Wetherall. Active Network Vision Reality: Lessons from a Capsule-based System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 64–79, Charleston, SC, December 1999.
27. C. Dabrowski, K. Mills, and J. Elder. Understanding consistency maintenance in service discovery architectures during communication failure. In *Proceedings of the third international workshop on Software and performance*, pages 168–178, Rome, Italy, July 2002.
28. K. Fall. A delay tolerant network architecture for challenged internets. In *Proceedings of SIGCOMM*, Karlsruhe, Germany, August 2003.
29. 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*, Rome, Italy, September 1996.
30. S. Glassman, M. Manasse, M. Abadi, P. Gauthier, and P. Sobalvarro. The Millicent protocol for inexpensive electronic commerce. In *Proceedings of the 4th World Wide Web Conference* , pages 603–618, Boston, MA, December 1995.
31. 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.
32. M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Fransisco, CA, May 2003.

33. E. Guttman. Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing*, 3(4):71–80, 1999.
34. P. Hui, A. Chaintreau, J. Scott, R. Gass, J. Crowcroft, and C. Diot. Pocket switched networks and human mobility in conference environments. In *WDTN '05: Proceeding of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, Philadelphia, PA, August 2005.
35. L. Iftode, C. Borcea, N. Ravi, P. Kang, and P. Zhou. Smart phone: An embedded system for universal interactions. In *Smart phone: An embedded system for universal interactions. In Proceedings of the tenth International Workshop on Future Trends in Distributed Computing Systems*, Suzhou, China, May 2004.
36. W. Jansen and T. Karygiannis. Nist special publication 800-19 - mobile agent security, 2000.
37. 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*, 47(4):475–494, 2004.
38. 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)*, pages 66–73, Las Vegas, NV, July 1998.
39. 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.
40. T. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman. Hitting the distributed computing sweet spot with tspaces. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 35(4):457–472, March 2001.
41. 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)*, pages 41–50, Anchorage, AK, April 2001.
42. G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.
43. M. Nidd. Service Discovery in DEAPspace. In *IEEE Personal Communications*, August 2001.
44. B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, St Malo, France, October 1997.
45. N.Ravi, C. Borcea, P. Kang, , and L. Iftode. Portable Smart Messages for Ubiquitous Java-enabled Devices. In *The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous)*, Boston, MA, August 2004.
46. J. J. Ordille. When agents roam, who can you trust? In *First Conference on Emerging Technologies and Applications in Communications (etaCOM)*, Portland, OR, May 1996.
47. J. K. Ousterhout, J. Y. Levy, and B. M. Walsh. The Safe-Tcl Security Model. Technical Report SMLI TR-97-60, Sun Microsystems, 1997.
48. P.Boddupalli, F.Al-Bin-Ali, N.Davies, A.Friday, O.Storz, and M.Wu. Payment support in ubiquitous computing environments. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Monterey, CA, October 2003.

49. H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *First International Workshop on Mobile Agents MA*, pages 50–61, April 1997.
50. N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, Boston, MA, August 2000.
51. 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, Anaheim, CA, January 1997.
52. N. Ravi, N. Dandekar, P. Mysore, and M. Littman. Activity recognition from accelerometer data. In *Proceedings of the Seventeenth Conference on Innovative Applications of Artificial Intelligence(IAAI)*, Pittsburgh, PA, July 2005.
53. N. Ravi, M. Gruteser, and L. Iftode. Non-inference: An information flow control model for location-based services. In *Proceedings of the Third International Conference on Mobile and Ubiquitous Systems (Mobiquitous)*, San Jose, CA, July 2006.
54. N. Ravi, P. Shankar, A. Frankel, A. Elgammal, and L. Iftode. Indoor localization using camera phones. In *Proceedings of the Workshop on Mobile Computing Systems and Applications(WMCSA)*, Washington, USA, April 2006.
55. N. Ravi, P. Stern, N. Desai, and L. Iftode. Accessing ubiquitous services using smart phones. In *Third International Conference on Pervasive Computing and Communications*, Kauai, Hawaii, March 2005.
56. M. Roman and R. Campbell. Gaia: Enabling active spaces. In *Proceedings of 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.
57. D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of CHI*, Pittsburgh, PA, May 1999.
58. D. Scott, R. Sharp, A. Madhavapeddy, and E. Upton. Using visual tags to bypass bluetooth device discovery. *SIGMOBILE Mobile Computing and Communications Review*, 9(1):41–53, January 2005.
59. 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*, Monterey, CA, June 1999.
60. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, Asheville, NC, December 1993.
61. R. Want, A. Hopper, V. Falco, and J. Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, December 1992.
62. 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*, Dallas-Fort Worth, Texas, March 2003.