Context-Aware Migratory Services in Ad Hoc Networks

Oriana Riva, Tamer Nadeem, *Member*, *IEEE*, Cristian Borcea, *Member*, *IEEE*, and Liviu Iftode, *Senior Member*, *IEEE*

Abstract—Ad hoc networks can be used not only as data carriers for mobile devices but also as providers of a new class of services specific to ubiquitous computing environments. Building services in ad hoc networks, however, is challenging due to the rapidly changing operating contexts, which often lead to situations where a node hosting a certain service becomes unsuitable for hosting the service execution any longer. We propose a novel model of service provisioning in ad hoc networks based on the concept of context-aware migratory services. Unlike a regular service that executes always on the same node, a migratory service can migrate to different nodes in the network in order to accomplish its task. The migration is triggered by changes of the operating context, and it occurs transparently to the client application. We designed and implemented a framework for developing migratory services. We built TJam, a proof-of-concept migratory service that predicts traffic jams in a given region of a highway by using only car-to-car short-range wireless communication. The experimental results obtained over an ad hoc network of personal digital assistants (PDAs) show the effectiveness of our approach in the presence of frequent disconnections. We also present simulation results that demonstrate the benefits of migratory services in large-scale networks compared to a statically centralized approach.

Index Terms—Mobile computing, mobile applications, ubiquitous computing, distributed programming, distributed systems.

1 Introduction

Short-range wireless technology is on its way to becoming ubiquitous, and it will soon be possible to program real-world ad hoc networks, which can be formed spontaneously (for example, vehicles on the road) or deployed for specific tasks in specific regions (for example, monitoring a certain region during an emergency situation). Traditionally, ad hoc networks have been viewed as data carriers between a mobile device and an Internet server or between two mobile devices. However, besides transferring static data to/from mobile nodes, these networks can be leveraged to provide a new class of services that acquire, process, and distribute real-time information from nodes located in the immediate proximity of geographical regions, objects, or activities of interest. For instance, a mobile ad hoc network of vehicles can provide traffic information from a region 10 miles ahead of a given car on a highway, whereas an ad hoc network of intelligent video cameras can transmit images from the proximity of a disaster area.

Building such services is difficult because the rapidly changing nodes' operating contexts can often lead to situations where a node currently providing a certain service becomes unsuitable for hosting that service any longer. For example, the target of an object-tracking service can move out of the sensing range of the node (for example, video camera) where the service currently executes, or a node may stop providing a service currently in use due to limited resource availability (for example, its energy is exhausted). Essentially, a node may become incapable of hosting a service when certain context parameters (for example, location, time, node capabilities, and network topology) change.

Typically, service interaction models are connection oriented: clients select services, bind to the service interfaces, and then invoke operations on these interfaces [1]. As the environment and network connectivity change, different rebinding techniques can be employed in an attempt to maintain the illusion of a connection-oriented communication. The simplest way to address such an issue is to require the client to discover a similar service running on a different node every time the old node becomes unsuitable, and then restart the interaction with the new one. There are two potential issues posed by this solution. First, it is possible that no other node providing the service of interest exists in the ad hoc network; rather than offering all possible services, each node will tend to offer just a small set of services determined by its owner or resources. Furthermore, even though another node providing the service of interest may exist, such a node could still be incapable of hosting the service execution due to its current operating context (for example, low battery power). Second, any state associated with the old service execution is lost unless a handoff mechanism is employed.

In addition to these issues, the deployment of services in ad hoc networks is hampered by the possible unavailability of Internet connectivity. Due to costs, limited resources, or deployment issues, Internet connectivity is not always

- *O. Riva is with the Helsinki Institute for Information Technology, PO Box 9800, FIN-02015 TKK, Finland. E-mail: oriana.riva@hiit.fi.*
- *T. Nadeem is with Siemens Corporate Research, 755 College Road East, Princeton, NJ 08540. E-mail: tamer.nadeem@siemens.com.*
- *C. Borcea is with the Department of Computer Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102. E-mail: borcea@cs.njit.edu.*
- *L. Iftode is with the Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019. E-mail: iftode@cs.rutgers.edu.*

Manuscript received 20 June 2005; revised 6 Dec. 2005; accepted 13 Feb. 2007; published online 8 Mar. 2007.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-0179-0605.
Digital Object Identifier no. 10.1109/TMC.2007.1053.

available in these networks, thus precluding the use of Domain Name System (DNS) and well-known service discovery protocols such as Jini [2] and WS-Discovery [3]. Therefore, new naming and service discovery mechanisms are needed.

To address these issues, we propose a novel model of service interaction in ad hoc networks based on the concept of *context-aware migratory service*. Unlike a regular service that always executes on the same node, a context-aware migratory service is capable of migrating to different nodes in the network in order to effectively accomplish its task. The service migration is context aware as it is triggered by context changes of the nodes in the ad hoc network. The migration occurs transparently to the client application, which is constantly presented with a single service end point. Thereby, the interaction between a client application and a migratory service can continue uninterrupted, except for small delays generated by the migration process. Our model presents two advantages. First, when a node becomes unsuitable for hosting any longer a certain service, the client application does not need to perform any new service discovery because the current service can autonomously migrate to a node that is qualified for accomplishing the current task. Second, the migratory service incorporates all the state information necessary to resume the interaction with the client when the migration to a different node has completed.

We designed a framework that supports the development and execution of migratory services. This framework provides communication primitives, migratory service control, context management, and reliability support. The system support for this framework is supplied by the Smart Messages (SM) [4], [5] distributed computing platform, which provides support for naming, routing, and execution migration. The SM platform also defines the security architecture needed to protect against malicious services or nodes. We have built TJam, a proof-of-concept migratory service that dynamically predicts if traffic jams are likely to occur in a given region of a highway by using only car-to-car short-range wireless communication. Experimental results executed over mobile ad hoc networks of personal digital assistants (PDAs) show the feasibility and effectiveness of our approach. We have also simulated the same service in order to investigate its behavior in large-scale networks and to compare our model to a traditional static interaction paradigm.

The rest of the article is organized as follows: We start, in Section 2, with a brief overview of Smart Messages in order to make this article as self-contained as possible. Section 3 presents the migratory service model, and Section 4 presents the migratory service framework and application programming interface (API). Section 5 describes our migratory service prototype TJam. Experimental evaluation and simulation results for larger scale networks are presented in Section 6. The related work is discussed in Section 7. The article concludes in Section 8.

2 Smart Messages Overview

Smart Messages (SM) [4], [5] is a distributed computing platform for cooperative computing in highly volatile mobile ad hoc networks. An SM is an application whose execution is sequentially distributed over a series of nodes using execution migration. The nodes on which SMs execute are named by properties and discovered dynamically using application-controlled routing. To move between two nodes of interest, an SM explicitly calls for execution migration. Each node participating in the SM execution provides:

1. a *virtual machine* (VM) for execution over heterogeneous platforms,
2. a shared memory addressable by names, namely, the *tag space*, for inter-SM communication, synchronization, and interaction with the host,
3. an *admission manager* that prevents excessive use of resources by incoming SMs, and
4. a *code cache* for storing frequently executed code.

An admitted SM at a node generates a task that is scheduled for execution. During execution, an SM can interact with the node or other SMs through the tag space, which is local to each node. The tag space consists of *(name, data)* pairs, called tags, which are created by SMs and used for data exchange. The tag space also provides a simple update-based synchronization mechanism; an SM can block on a tag until another SM performs a write on that tag. Special I/O tags are predefined at nodes and used as an interface to the Operating System (OS) and I/O system (for example, battery lifetime, available memory, and location sensors). Tags also serve to name the destination of SM migrations.

Migration is the key operation in the SM programming model as it routes SMs across multiple hops to the nodes of interest. This high-level primitive for multihop migration is implemented using a low-level primitive for one-hop migration, namely, *sys_migrate*, and routing tables built in the tag space [6]. The *sys_migrate* primitive captures the execution context of the SM (data explicitly identified by the programmer and control state), packs it with the SM code, transfers the SM to the next hop, and resumes the execution with the following instruction in the code.

To illustrate the SM distributed computing model, we consider the example depicted in Fig. 1. An ad hoc network of intelligent cameras can be programmed to perform intruder tracking across the region of deployment. Each camera runs a preinstalled SM that periodically acquires images and performs a simple image analysis. Each time this SM detects a potential threat, it creates a "potential_threat" tag. A mobile user can track potential threats by injecting an *IntruderTracking* SM in the network. This SM takes as a parameter a set of features that define the object of interest and returns the motion path of the threat as a list of camera locations (that is, those cameras where this threat was recognized). At the user node, the IntruderTracking SM creates a "threat_path" tag, which is used to identify this node when the SM migrates back with the result. To accomplish its task, the injected IntruderTracking SM migrates to nodes named by "potential_threat" tags and performs threat recognition on each of them. If the threat is recognized on a certain node, the location of such a node is added to the threat motion path. This process continues recursively until no new nodes identified by a "potential_threat" tag can be found (that is, the migration times out).

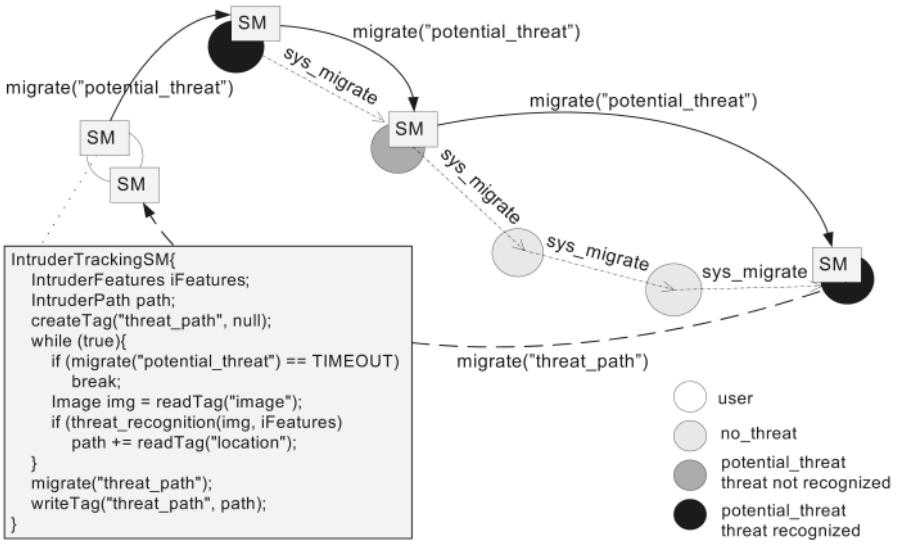

Fig. 1. Smart Messages example: IntruderTracking.

The IntruderTracking SM completes by migrating back to its user node and writing the threat path list to its corresponding tag. In this example, the set of features and the motion path list represent the SM data carried from node to node across migrations.

We have implemented the SM prototype in the Java programming environment over Linux. Specifically, we have modified Sun Microsystem's Kilobyte Virtual Machine (KVM) [7] because its source code is available and has a small memory footprint suitable for resource constrained devices. SMs are essentially Java programs that invoke an API encapsulated in two Java classes: 1) *SmartMessage*, which includes primitives for migration, new SM creation, and synchronization, and 2) *TagSpace*, which includes primitives to create, delete, read, and write tags. SMs can incorporate multiple Java classes, namely, *code bricks*, and multiple Java objects, namely, *data bricks*, which are explicitly specified by the programmer when an SM is instantiated. The data bricks contain the data that must be transferred during migrations. At runtime, SMs can create "child" SMs carrying a subset of their code bricks and data bricks.

When an SM calls explicitly for migration, its state needs to be captured and converted into a machine-independent representation that will be used to resume the SM execution at destination. Since the code bricks are already in the machine-independent Java class format, only the data bricks and execution control state are converted. Data bricks are converted using our own serialization format (we implemented it, as KVM does not support one). The execution control state of an SM is represented by the execution stack frames of its associated VM-level thread. Each stack frame is serialized into a tuple of six values: current offset of *instruction* and *operand stack* pointers, method name, signature name, class name, and a flag indicating whether the method is nonstatic. For nonstatic methods, we also encode the machine-independent identifier for the *this* self-reference. After the admission manager successfully received the code bricks, data bricks, and execution control information from a source node, a new VM-level thread and its associated SM structure are constructed. Additionally, the admission manager deserializes the data bricks and reconstructs the stack frames using the tuples sent from the source.

An SM is admitted at a node if enough resources can be provided to support its execution or migration (that is, the SM specifies the minimum amount of resources required). Based on the specified admission policy, the node may grant more resources to SMs that exceed the specified amount of resources during execution. If no more resources can be granted, the SM is requested to migrate to another node. To ensure that SMs do not interfere maliciously with each other, we defined five protection domains for controlling the access to tags. These domains (owner, SM with a common ancestor, SM with a common node of origin, SM with common code bricks, or unknown SM) define various relations between the creator of a tag (that is, the owner) and other SMs that attempt to access this tag. The owner of each tag specifies read and write permissions for each of the five protection domains. For each attempted tag space operation, the VM verifies the SM credentials and authorizes the access according to these credentials.

3 Context-Aware Migratory Services

A prominent interest behind the deployment of services in ad hoc networks originates from their capability of exploiting temporary and unstable network support to acquire real-time information in the proximity of geographical regions, entities, or activities of interest. We assume that nodes in these networks are willing to collaborate; some nodes offer services, others host client applications, and the rest cooperate to provide service discovery and routing of messages. To achieve its goal, in principle, a service of this type can contact other services, thus acting as a client for those services. In our work, however, we assume that services compute results by processing only data made available by their hosting nodes.

3.1 Motivating Scenarios

Deploying services of this type in ad hoc networks proves challenging due to the dynamism, in particular, mobility,

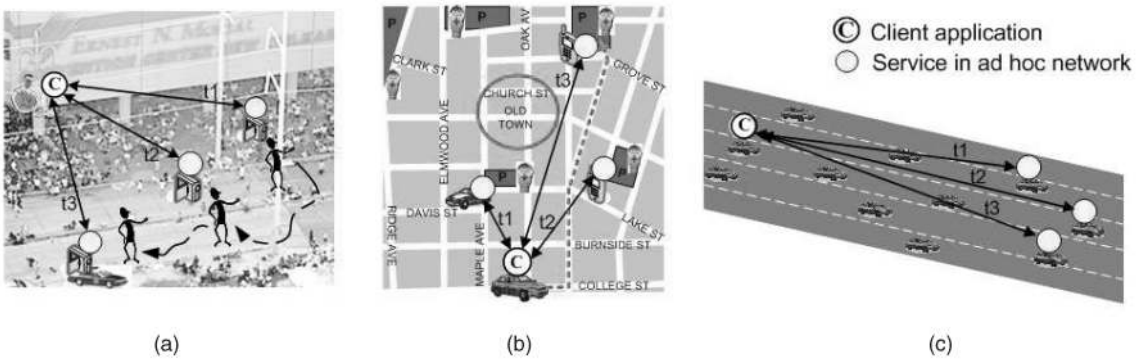

Fig. 2. Examples of client-service interactions in ad hoc networks. (a) Entity-tracking service. (b) Parking spot finder. (c) Driver-assistant service.

of the interacting entities. Highly dynamic operating contexts affect the client that generates the request, the service that processes the request, and, occasionally, the target of the service (for example, a moving object being tracked). Fig. 2 proposes three scenarios illustrating how a service interacting with a certain client can stop satisfying the client's request due to context changes affecting its hosting node or the environment. In these situations, under typical service models, the interaction between the client and the service ends. Hence, the client may attempt to discover new nodes hosting the same type of service and restart the interaction. It is possible, however, that no service satisfying the client's requirements exists. However, even if it does, these interruptions in the service interaction could lead to inefficient performance due to the cost of the discovery process and to the cost associated with the loss of the interaction state.

An *entity-tracking* client application can provide policemen with real-time images of certain suspicious entities (for example, people and cars) as they move across a given region, as well as with alerts every time a potential threat is recognized. This type of application particularly suits crowded events such as political conventions, conferences, and manifestations in which it is hard to quickly deploy wired networks of video cameras. A more feasible and cost-effective solution is to exploit a mobile ad hoc network of wireless video cameras that, for instance, can be installed on police patrols and policemen's helmets (both mechanisms have already been tested in real-life events). Tracking services execute on each video camera; they are capable of performing image recognition of entities specified by policemen and sending back images of those entities. There are two factors, however, that can force the client to interrupt its current interaction with a certain tracking service and start a new interaction with a different service: 1) the node where the service executes is mobile and might move away from the tracked entity and 2) likewise, the tracked entity is mobile and might move away from the sensing range of the service node. For example, Fig. 2a shows how the client needs to interact with three different tracking services over a short period of time. Besides the time necessary to carry out service discovery multiple times, the lack of service continuity precludes the service process from using advanced image recognition algorithms based on long-term learning, correlation, and history.

A *parking spot finder* client application can inform drivers about parking spot availability in the proximity of a specified destination. We assume that parking spot availability can be determined by services running on cars or smart phones that interact with wireless-enabled parking meters located in their transmission range. In his or her request, the user can specify the destination of interest and his or her current location, along with other preferences like cost and security of parking lots. The request is forwarded to a service located in the proximity of the destination using a spontaneously created network of wireless-equipped smart phones and cars. Upon receiving a request, a service checks if any parking spot is available. If so, it informs the client about the location of the most suitable parking spot (based on the request parameters) and keeps monitoring the parking meter associated with the free spot. If another driver takes this spot, the service replies to the client either with a new parking spot or with an "unavailable" response. Upon receiving an "unavailable" response, a user will need to discover another service in the destination area, as Fig. 2b shows. Furthermore, a user might be forced to contact a new service when the current service, executing on a mobile node, moves away from the monitored parking spot. Conversely, the user would like to submit his or her request only once and be informed about parking spot availability until the destination is reached.

A *driver-assistant* client application can inform drivers on highways about the traffic conditions of the road ahead of them. For instance, if a traffic jam is predicted at the next segment of the highway, the driver can decide to take an earlier exit. We assume that the driver requires to be continuously notified about traffic conditions at a constant distance ahead of her position (for example, 10 miles ahead). The client application communicates with services executing on some of the vehicles forming the mobile ad hoc network. Each service estimates the status of the road traffic by using locally available information such as the density of one-hop neighboring vehicles and their speeds. In such a scenario, we observe that 1) the service needs to constantly execute in the region of interest to the user, 2) such a user-defined region changes over time according to the user's movement/speed on the road, and 3) the set of cars located in the region of interest changes over time due to their mobility (for example, cars can leave the highway, stop, or slow down). As shown in Fig. 2c, due to these reasons, the client occasionally needs to reestablish the interaction with a new service that can meet the user's requirements.

3.2 Requirements for Services in Ad Hoc Networks

By analyzing these scenarios, we identified four requirements that services in ad hoc networks need to address:

1. *Context-awareness*. To be semantically correct and efficient, these services need to take into account their current operating context such as location, resources available on the node, and network connectivity. For example, in all described scenarios, the service must be location aware. Additionally, the service must occasionally monitor entities in its proximity (suspicious presences in the first scenarios and parking meters in the second scenario).
2. *User-driven adaptability*. To provide useful results, these services need to constantly adapt their execution according to current needs and operating context of the user (for example, location, activity, and terminal equipment). As the user operates in a highly dynamic environment, his or her request parameters are subject to frequent context-induced changes. For instance, the driver-assistant client must constantly communicate its location to the service, thus ensuring that results are computed in the region of interest.
3. *Service continuity*. Client applications can highly benefit from continuous service provisioning in many situations. Our scenarios show how, after a while, a node running a certain service can become inappropriate to host that service any longer. This can be due to mobility, limited resource availability, or network partitioning. In these cases, the client has to discover and restart its interaction with a new service, but the entire state of the old interaction is lost. Although this approach is acceptable for a stateless interaction, it can lead to a significant performance degradation for a stateful interaction. For instance, in the above scenarios, the entity-tracking and the traffic jam algorithms require history and learning support to provide accurate results; the parking spot finder needs to know a user's preferences and destination. Therefore, a mechanism for capturing and transferring the state of a service to a new node and resuming its execution using this state is necessary.
4. *On-demand code distribution*. It is unrealistic to assume that every node in an ad hoc network will possess the code for every type of service. For example, the code for running the parking spot finder might not be available on cars or smart phones in the proximity of the destination of interest. Therefore, a mechanism capable of dynamically transferring the code to certain nodes that are semantically and computationally suitable for running the service is necessary (that is, the code could be transferred from other nodes in the ad hoc network or even from the Internet if possible).

3.3 Migratory Service Model

To address the above requirements, we propose a novel service model based on the concept of *context-aware migratory service*, hereafter referred to as *migratory service*.

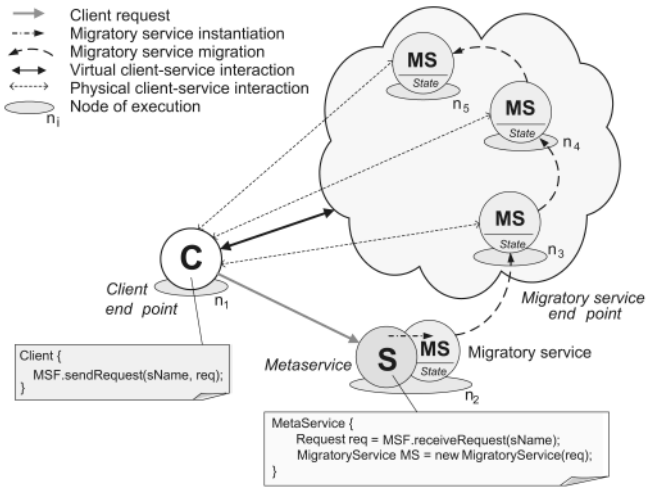

Fig. 3. Instantiation of a migratory service and example of interaction.

Intuitively, this kind of service is capable of migrating to different nodes in the network in order to effectively accomplish its functions. It executes on a certain node as long as it is able to provide semantically acceptable results using the available resources; when this is not possible anymore, it migrates through the network until it finds a new node where it can continue to satisfy the client request. The service migration occurs transparently to the client and, except for a certain delay, no service interruption is perceived by the client. Although a migratory service is physically located on different nodes over time, it constantly presents a single virtual end point to the client. Hence, a continuous client-service interaction can be maintained.

The migratory service model involves three main mechanisms. The first monitors the dynamism of interacting entities (client or service) by assessing context parameters characterizing their state of execution and available resource capabilities. The whole set of context parameters constitute the *context* of a certain entity. The second specifies, through *context rules*, how the service execution is influenced and should be modified based on variations of those context parameters. The third makes the service capable of migrating from node to node and of resuming its execution once migrated. We call this context-aware service migration since it is triggered by changes of the operating context, which occur on the service as well as on the user side.

Fig. 3 depicts a typical interaction in the migratory service model. As mentioned before, every node in the network is expected to possess the code for a limited number of services. To facilitate service code distribution in the network, every node provides a *metaservice* for each individual service code it owns. The role of a metaservice is to instantiate migratory services. Initially, the client discovers and contacts a metaservice that is capable of serving its request. The metaservice processes the request by instantiating a new migratory service that will take over the interaction with the client; the metaservice just spawns migratable instances of itself, but it does not migrate or send responses to clients. There is a *one-to-one mapping* between a migratory service and a client application. Upon the creation of a migratory service, the client application ceases

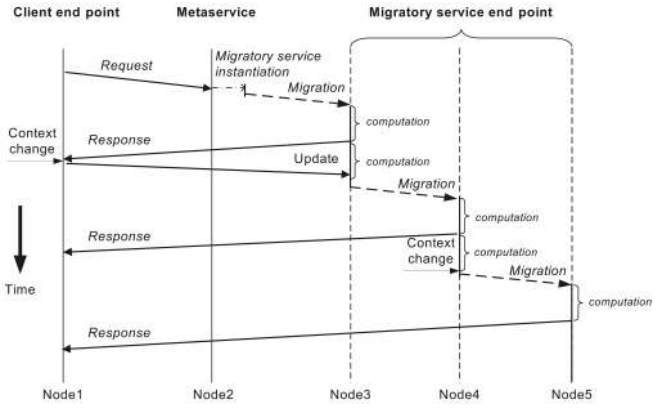

Fig. 4. Example of context-aware service migrations.

communication with the metaservice and continues to interact solely with its associated migratory service.

Our service model aims to support long-running queries and can be characterized as "one request, multiple responses." This is an appropriate model, especially for services that monitor entities or actions in real-time and report their observations periodically. Therefore, the service interaction consists of two main operations: 1) the migratory service sends responses to the client application, and 2) the client application sends "request updates" each time the user's context changes beyond relevant thresholds (that is, in fact, these updates are sent by the runtime system at the client node).

These concepts are exemplified in Fig. 4. Upon being instantiated on *Node2*, a driver-assistant migratory service changes the node of execution by migrating from *Node2* to *Node3*, which is located in the region to be monitored. Subsequently, the service migrates from *Node3* to *Node4* and from *Node4* to *Node5*. These two migrations are triggered by changes of the user's and service's context, respectively. For example, the first migration could be due to the fact that the user requests more accurate observations as she gets closer to the region of interest, whereas the second one could be due to the fact that *Node4* has left the region of interest.

4 Migratory Service Framework

To support the migratory service model, we designed a common framework that runs on all nodes willing to cooperate in the ad hoc network. A main challenge that hinders the practical development of applications and services in ad hoc network environments is the difficulty of writing software for distributed systems with such complex requirements. Although building applications and services from scratch could lead to a better performance for individual situations, we believe that a common software platform can provide a more functional set of primitives above the potentially heterogeneous operating systems. Thereby, application developers can program and deploy new applications and services more quickly.

Fig. 5 illustrates the system architecture for the migratory service framework. At the lower layer, the Smart Messages (SM) computing platform provides support for execution migration, naming, routing, and security. On top of the SM layer, we provided support for

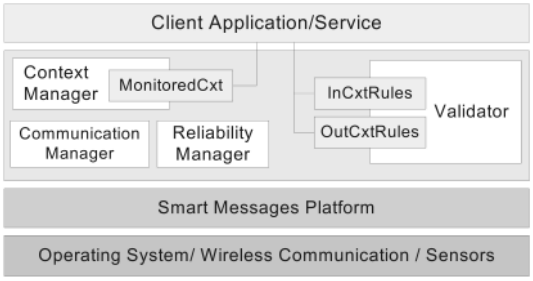

Fig. 5. Migratory service framework.

1. context provisioning and monitoring,
2. context rules creation and validation,
3. client-service communication, and
4. service reliability.

The migratory service framework can be logically divided into two functional planes: data plane and control plane. Data traffic, routing, and migration form the data plane and mostly build on basic functionalities offered by the SM platform. The Communication Manager is the corresponding module for the data plane. The control plane specifically targets issues related to unreliability and dynamism of ad hoc networks. The control plane is organized in a logical flow of three modules: the Context Manager monitors and stores the variability of the environment, the Validator evaluates the observed variability based on application/service requirements and decides how to act upon those changes through the Communication Manager, and the Reliability Manager allows migratory services to recover from node failures.

Practically, the framework consists of a set of Java classes that can be added to any SM platform. As it will be detailed in Section 4.5, clients, migratory services, and metaservices are Java programs that register with the framework and invoke simple message passing primitives provided by our API. The control plane execution is transparent to the application/service layer. Ultimately, the framework maps these programs onto lower level SMs.

4.1 Context Provisioning and Monitoring

The SM platform provides several types of context data accessible through specific I/O tags (see Section 2 for the definition of I/O tags). Commonly, the SM platform provides context information such as location, speed, and time by means of GPS and various system status information (for example, amount of available memory and remaining battery power). It also maintains and periodically updates a list of one-hop neighboring nodes. Additionally, the SM platform can perform reasoning of raw context data to infer higher layer context information and offer access to those through specific application tags.

The *Context Manager* supports storage and access to context data provided by the SM platform. Clients or services can specify, through the migratory service interface, which context parameters the Context Manager must monitor; these parameters constitute the *MonitoredCxt*. The Context Manager translates a certain MonitoredCxt identifier into an SM tag name according to the application-specific semantics. As the Context Manager can integrate different translators for different context ontologies, the application developer can ideally utilize any context

ontology to build the application logic. Upon the translation, the Context Manager provides access to the values corresponding to those context parameters by polling or blocking on the associated SM tags.

4.2 Context Rules Creation and Validation

The main task of the *Validator* module is to evaluate if a service computation can be "correctly" carried out on the current hosting node; the correctness of the execution is evaluated both in terms of the resources necessary to compute a result and in terms of the quality of the produced result. If the computation can no longer be correctly carried out, the Validator triggers a service migration.

The Validator validates incoming and outgoing data based on service-specific (or client-specific) context rules, referred to as *CxtRules*. These rules specify policies/filters/preferences ruling system behavior. For instance, they can define how the node's resources should be utilized, which type of incoming results should be accepted, or which level of security should be applied. CxtRules are of two types and are applied in different phases of the client-service interaction:

- *InCxtRules*. These are used to validate the correctness of incoming data. 1) A metaservice utilizes InCxtRules to decide whether to accept/refuse an incoming client request. This is done in collaboration with the Admission Manager at the SM level, which performs authentication and admission based on local security policies. 2) A client application utilizes InCxtRules to decide whether to accept/refuse a received service response. Based on the current user context, a response can be deemed irrelevant or even wrong. In such a case, the Validator instructs the Communication Manager to send a request update to the migratory service.
- *OutCxtRules*. These are used to validate the correctness of outgoing data at the service side and, implicitly, to trigger service migrations if necessary. Before computing a new response, the Communication Manager invokes the Validator to verify these context rules. For example, the Validator of a driver-assistant service verifies that the given node is still located in the region to be monitored; if the node has left this region, a service migration is triggered.

Context rules are expressed in the form of condition/action statements. Conditions are articulated as full binary trees of Boolean expressions. Each node of the tree can be a *comparisonNode* or a *combinationNode*. A comparisonNode is a triplet consisting of *contextName*, *comparisonOperator*, and *value*. ComparisonOperators currently supported are `equal`, `not-equal`, `morethan`, `lessthan`, `inRegion`, and `outRegion`. An example of a comparisonNode is `<batteryLevel, equal, low>`. A combinationNode combines two nodes by means of combination operators such as `and` and `or`. A common example of combinationNode is `<or, <batteryLevel, equal, low>,<responseLocation, outRegion, userRegion>>`. Therefore, `and` and `or` operators permit flexibly combining elementary conditions to build more complex ones. Actions currently supported are `migrate service`, `send update`, `accept/refuse response`, and `accept/refuse request`. The entire condition clause of a certain context rule is evaluated by verifying all the contained Boolean expressions based on the current values of user's and service's context parameters. Each rule is set as optional or mandatory, as the context parameters required by the rule may not always be available (for example, a node may not provide location information).

When defining context rules, applications or services may introduce ambiguities, contradictions, or logical inconsistencies. For instance, an application might have specified contradictory actions in response to similar context changes. Following the definitions presented in [8], our scenarios could involve both intraprofile and interprofile conflicts. Specifically, *intraprofile conflicts* emerge inside the specification of policies for applications or services, and they are local to a node. *Interprofile conflicts* involve only two entities on different nodes (that is, client and service). Conflict resolution can be performed partly statically and partly dynamically. The dynamic resolution selects the action that satisfies the largest number of conditions.

4.3 Client-Service Communication

We assume ad hoc networks without Internet connectivity and, consequently, without access to DNS or Internet-based service discovery mechanisms. Furthermore, we do not assume global addresses for the nodes in the network. Since migratory services can run on different nodes over time, we prefer to name the communicating programs directly. Due to the fact that these programs are ultimately converted into SMs, we enforce the naming conventions defined by the SM platform [5]. More exactly, tag names are used to uniquely identify the communication end points in the migratory service model. Each time a migratory service moves between two nodes, its name is removed from the old node and recreated on the new node.

In the SM platform, service discovery and routing are integrated in a single module that performs content-based routing [6]. To locate communication end points, our current implementation provides two basic SM routing algorithms: geographical routing and region-bound content-based routing. The geographical routing is similar to the Greedy Perimeter Stateless Routing (GPSR) [9]. At each node, the algorithm migrates the SM to the closest neighbor to the location of interest. The content-based on-demand routing (similar to Ad-Hoc On-Demand Distance Vector (AODV) [10]) is used to discover a node identified by a tag name within a given geographical region (reached using the geographical routing).

The *Communication Manager* is responsible for interacting with the SM layer to discover metaservices, route messages between communicating end points, and carry out service migration when necessary. Metaservices are identified through SM tags (learned by clients offline), and they are discovered by exploiting the two SM routing algorithms mentioned above. Identical identifiers for migratory services are generated independently on the framework at the client and metaservice nodes; each identifier is directly derived from a combination of the client name and the metaservice name. The metaservice also passes the client request to the migratory service. After these operations, the same two routing algorithms are used to enable the communication between the client and the migratory service. Note that the Communication Manager does not guarantee reordering of out of order messages or recovery

of lost messages. However, each exchanged message is identified by a sequence number that is accessible to the client/service; hence, it is up to the client/service to deal with losses or out of order messages.

If at any time the Validator deems the current service node unsuitable for hosting the service, the Communication Manager is invoked to perform service migration. The Communication Manager removes the SM tag identifying the service end point from the old node, uses the SM content-based migration to find a new node of execution, rebinds the service to this node (that is, creates its tag name on the node), and resumes the service execution. Although a client update could be lost during this process, the entire migration is transparent to the client that sees the same virtual end point. The lost of a client update can lead just to a slight decrease in performance, as the client will send a new one if an irrelevant or wrong response is received.

4.4 Service Reliability

In the basic framework implementation, a migratory service is lost if the node on which it executes fails. In this situation, a timeout expires at the client side, and a new request for a metaservice is generated. To improve upon this solution, we provide a mechanism that ensures fault tolerance to one failure. This mechanism is optional due to the extra load it induces in the network. The *Reliability Managers* at the nodes where the migratory service executes maintain an inactive version of the service on a secondary node. The secondary node is the first node where the migratory service executes. This version can take over the service provisioning if a failure in the interaction with the primary version of the service occurs. If no failures occur, at the end of the interaction with the client, the hosting node of the primary service generates a request to remove the inactive copy. To further improve the reliability, the secondary node could also be the client node itself. However, the overhead for such a solution could increase significantly.

The inactive version of the service is created before the first response is delivered to the client. Periodically, the Reliability Manager on the primary node (which changes as the service migrates) updates the service state on the secondary node. These updates are sent using the SM content-based routing and can be delivered over multiple hops. At the SM level, the creation of the secondary copy and the update deliveries are implemented by spawning a copy of the migratory service and by migrating it to the secondary node. Since frequently used SM code is cached at nodes, only the execution state is transferred. The Reliability Manager at the destination is responsible for maintaining only the latest version of the inactive service copy. Practically, the inactive service is an SM blocked on a tag. When a new update from the primary service arrives, the Reliability Manager unblocks the old SM, instructs it to terminate, and stores the new SM.

A client application that does not receive any answer for a given time period will time out and contact the inactive version of the service on the secondary node. The frameworks on all nodes share the same naming conventions; hence, the client is able to determine locally the name (that is, the SM tag) of the inactive version of the service. When a client request arrives at the secondary node, the Communication Manager invokes the Reliability Manager that will unblock the SM representing the inactive copy. This copy becomes active and starts its interaction with the client. If a response from the former active service is received after the client started its new interaction (that is, potentially, the former active copy was not lost), the response will be discarded. Furthermore, the framework at the client side will generate a request to terminate the old service.

4.5 Programming Migratory Services

With the perspective of a migratory service's designer in mind, we provide a Java API that offers all the functions that are common across migratory services, thus requiring the designer to only provide support for the service-specific functions. The API must also support the specification of client applications that interact with migratory services. This API shields the programmer from the underlying SM platform and the networking aspects.

Fig. 6 shows a typical client application, metaservice, and migratory service along with their interactions. To make the entire process of service migration transparent, each of these three entities has to register with the migratory service framework. At the registration invocation, the entities pass their class name, name of context parameters to be monitored, and context rules to be evaluated at runtime. These parameters are needed by the framework to create the underlying SMs associated with each entity and to initialize all the necessary framework-level SMs for multihop service migration, data delivery, and so forth. Moreover, at the first registration of a migratory service with a hosting node, the Reliability Manager is invoked to create the inactive copy of the service, which will take over if a failure occurs.

Since programmers are well versed with the message passing programming model, the framework provides a similar API. Note that all the communication primitives are synchronous. Metaservices run in a loop and block waiting for client requests (Fig. 6b, line 7). At the SM layer, this corresponds to SM blocking on a tag. When a client application sends a request to a metaservice (Fig. 6a, line 7), the SM tag is appended with the request parameters, and the metaservice is unblocked to receive the request. Then, the metaservice instantiates the migratory service.

Before sending any response to the client, the Communication Manager at the migratory service side invokes the Validator. The Validator, at its turn, invokes the Context Manager to get fresh context information and evaluates the OutCxtRules. If these are positively verified, the response is delivered (Fig. 6c, line 9). Subsequently, the Communication Manager invokes the Reliability Manager to update the state on the secondary node. If the validation fails, the Communication Manager deregisters the service (removes its unique SM tag) and invokes an SM migration to transfer the service to a suitable node. Once it arrived at the new node, the Communication Manager registers the service with the framework and returns the control to the service code. At the client side, the Communication Manager invokes the Validator for each newly received response (Fig. 6a, line 9); only responses that are positively validated by the InCxtRules are returned to the application.

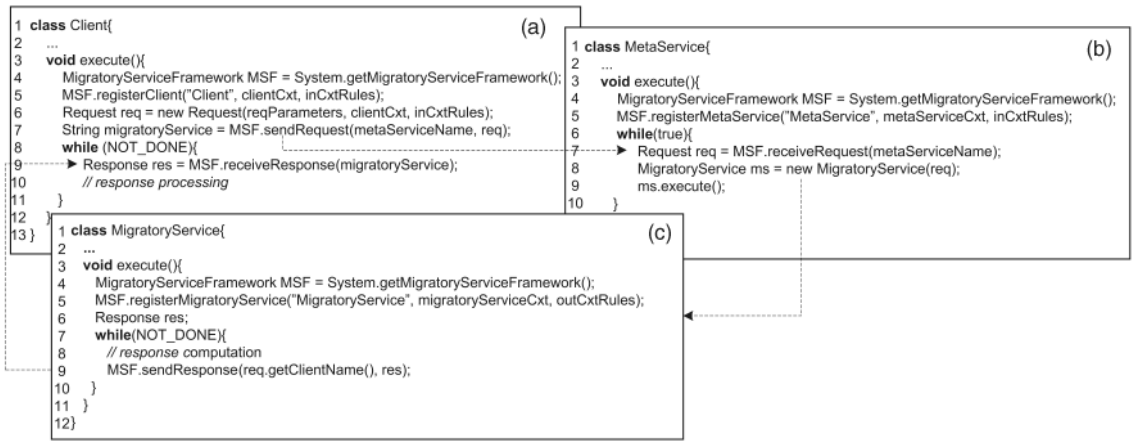

Fig. 6. Pseudocode for a typical client application, metaservice, and migratory service.

5 Case Study: TJam

We have built TJam, a proof-of-concept migratory service that dynamically predicts if traffic jams are likely to occur in a given region of a highway by using only car-to-car short-range wireless communication. For instance, a driver can use this system to decide which exit to take from a highway: If a traffic jam is likely to occur at the next exit, the driver can instead opt for the current exit. We assume that cars on highways communicate using short-range wireless networking (for example, IEEE 802.11) and they have GPS receivers for reporting location and speed. We also assume that all cars travel in the same direction. Typically, the driver instructs the service on which region to monitor by specifying the distance from her current position and the length of the region. Although other solutions can be envisioned to provide this service, a migratory service implementation is especially beneficial in this case due to 1) the highly dynamic operating contexts of ad hoc networks of vehicles, 2) the need to update the coordinates of the monitored region according to the user's location/speed, and 3) the need to transfer the execution state and maintain service history for accurately estimating the traffic jam probability.

In a typical example of interaction, the client first discovers the TJam metaservice. A TJam service request includes the name of the client, context parameters, and coordinates of the region to be monitored. If the metaservice accepts the request, it instantiates a TJam migratory service. This starts running on the metaservice node. If this node is not located in the region of interest, the TJam migratory service migrates to a node in the correct region. Then, it will continuously compute the traffic jam probability in such a region and send back results to the client. The framework at the migratory service node periodically recalculates the coordinates of the region based on the latest available location and speed of the client's car. Additionally, the framework at the client node, at any time, can decide to recalibrate this estimation by updating the request's parameters.

Traffic jams are locally congested phases in which cars travel at slow or zero velocity. To compute the traffic jam probability, TJam utilizes two types of information that every car in the network has locally available: 1) the number and 2) the speed of one-hop neighboring cars. The probability of a traffic jam P_{tjam} is computed using the following equations, where P_{number} is the probability of a traffic jam given the current number of neighboring cars and P_{speed} is the probability of a traffic jam given the current speed of neighboring cars:

$$P_{number} = maxP_{number} \cdot \frac{avg_{num} - min_{num}}{max_{num} - min_{num}}, \quad (1)$$

$$P_{speed} = maxP_{speed} \cdot \frac{avg_{speed} - max_{speed}}{min_{speed} - max_{speed}}, \quad (2)$$

$$P'_{tjam} = \alpha \cdot P_{number} + (1 - \alpha) \cdot P_{speed}, \quad (3)$$

$$P_{tjam} = P'_{tjam} \cdot \frac{N_{tjam}}{N_{total}}. \quad (4)$$

avg_{num} and avg_{speed} are computed by a low-pass filter with an exponential moving average with weight $w = 0.7$: $avg_{X_{n+1}} = (1 - w)avg_{X_n} + w \cdot X_n$. As avg_{num} varies from min_{num} to max_{num}, P_{number} varies linearly from 0 up to $maxP_{number}$, which is an upper bound of P_{number}. Below the lower boundary min_{num}, P_{number} is 0 (that is, the road is empty); above the upper boundary max_{num}, P_{number} is 1 (that is, the road is full). Likewise, similar parameters and boundaries are used to specify how P_{speed} varies with the avg_{speed}. The intermediate P_{tjam} probability, called P'_{tjam}, is computed by a weighted combination of P_{number} and P_{speed}, with $\alpha = 0.5$. Finally, P_{tjam} is the P'_{tjam} corrected by the ratio between the number of observations in which P_{tjam} was less than 0.5 (that is, N_{tjam}) and the total number of observations (that is, N_{total}).

To provide a more accurate computation of the probability, it is necessary to tune the employed parameters according to traffic variations and history. For the sake of brevity, we simply say that the minimum and maximum thresholds for the number and speed of neighboring vehicles are adjusted based on the minimum and maximum values observed during the period of observation. The probabilities $maxP_{number}$ and $maxP_{speed}$ are updated in such a way that, if it happens that avg_{num} is lower than min_{num}, then $maxP_{number}$ is decreased:

$$if(avg_{num} < min_{num}) \rightarrow maxP_{number} = maxP_{number} \cdot \beta,$$

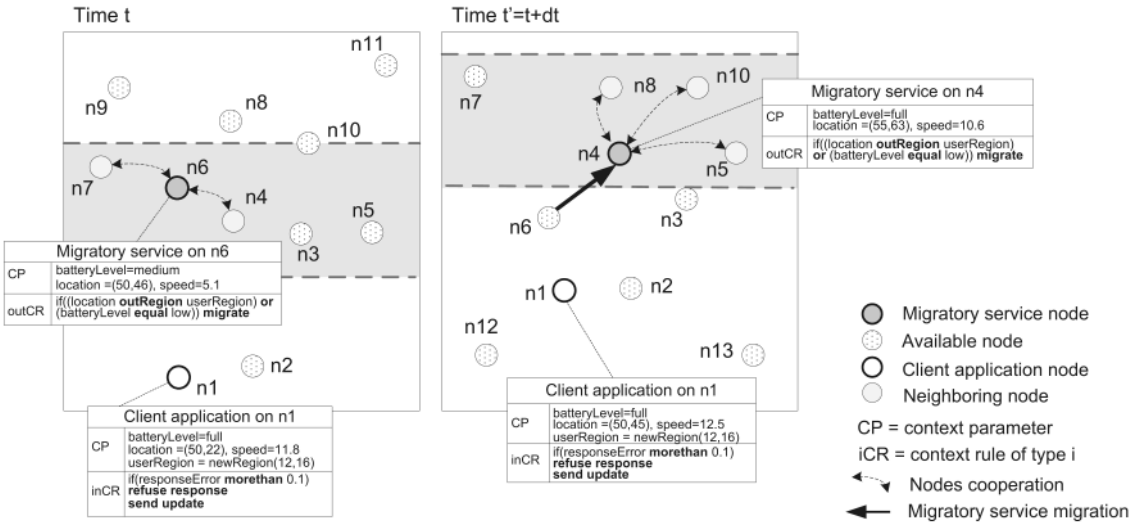

Fig. 7. Example of TJam's execution and migration over time.

where $\beta = 0.8$. If it happens that avg_{num} is greater than max_{num}, then $maxP_{number}$ is increased:

$$if(avg_{num} > max_{num}) \rightarrow maxP_{number} = maxP_{number} + \gamma,$$

where $\gamma = maxP^{old}_{number}/10$. The opposite applies to $maxP_{speed}$.

In order to illustrate an example of TJam migration along with some context parameters (CPs) and context rules (CRs), we refer to Fig. 7. The client application executes on node *n1*. At time t, the client interacts with the TJam migratory service that is located on node *n6*. TJam constantly computes the traffic jam probability and verifies the OutCxtRules (outCR). After dt, the position and the speed of nodes have changed. Therefore, the space of interaction is subject to a new computing environment. The new region of interest is either predicted by TJam based on the request's parameters and the user's context history or computed based on an update sent by the client node's framework. At this time, the migratory service framework at *node6* evaluates outCR and realizes that *n6* is out of the user-specified region. Therefore, TJam migrates to *n4*, which is located in the region of interest, and resumes the service computation.

While migrating, TJam carries its computation state consisting of all parameters necessary for predicting traffic jams (that is, min_{num}, max_{num}, min_{speed}, max_{speed}, $maxP_{number}$, $maxP_{speed}$, N_{tjam}, and N_{total}) and the history of the user's locations/speeds for updating the user-defined region. In this way, the service can provide a continuous and more efficient interaction with the client.

6 Evaluation

This section presents experimental and simulation results for TJam, our prototype migratory service. We ran experiments in a mobile ad hoc network testbed. The goal of the testbed study was to prove the feasibility and effectiveness of our model in dynamic environments. Given the rapid changes of the nodes' configuration and location, it is crucial to evaluate how the service interaction can adapt to such changes and recover from disconnections. Additionally, in order to investigate the scalability of our approach in larger scale networks, we simulated the prototype service. We simulated and compared the performance of two different versions of TJam, namely, *TJam-Smart* and *TJam-Base*. The first version implements our model of context-aware migration; the second one implements a baseline centralized approach.

6.1 Experimental Results

The ad hoc wireless network consists of 11 Hewlett-Packard Laboratories (HP) iPAQs that run Linux and use Orinoco's 802.11b PC cards. Since it has proved very difficult to run the experiments with the iPAQs moving at an adequate speed, we have emulated the mobility by instructing each node to periodically read from a file its position and speed on a two-lane highway. Each file contains the location coordinates and the speed of the node at time intervals of 5 seconds. The speed is a uniform variable between 5-10 m/s. In almost all of the experimental results, the service executes on a node that is at 2 hops away from the client node and has an average of 2-3 neighbors. The testbed configuration contains only one metaservice. We ran the same experiment 20 times, and each replication included 100 responses (that is, correct, wrong, or missing response).

In order to reduce the overhead due to the exchange of notifications carrying context changes, TJam predicts the new coordinates of the user's region of interest based on the past speeds/locations. Additionally, it includes this information in the response to the client. At the reception of such a response on the client side, the response is validated. If the predicted region's coordinates are incorrect compared to the current location, the answer is discarded, and an update is sent to the service. Otherwise, the answer is delivered to the client application, and no update is sent. This approach is feasible for TJam since the speed and direction of moving vehicles can be easily estimated on a highway. Furthermore, the client application sends an update if no answer is received within a certain time-out. In the experiments, the time-out is set to 7.5 seconds.

The main metrics employed in the experimental evaluation are given as follows:

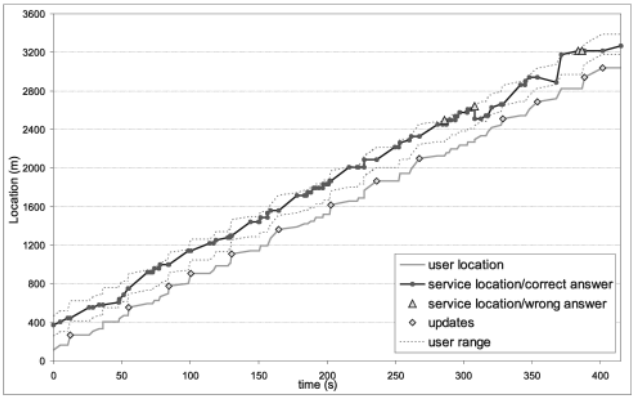

Fig. 8. The service location accurately follows the user movement.

- *Interresponse time*. This measures the elapsed time between consecutive correct answers. It also includes the time to update the user's request parameters at the service side.
- *Service discovery time.* This measures the elapsed time to discover a metaservice and receive the first correct response from the migratory service.
- *Service quality*. This is the percentage of correct answers out of the total number of received answers.
- *Response/update rate*. This is the average number of responses the client application receives per each update message that was sent.

Our main goal was to study how the migratory service allows the user to constantly receive correct observations in spite of disconnections and mobility. As Fig. 8 shows, the service migration successfully follows the movement of the user. Specifically, the graph compares the location of the user to that of the migratory service. If the service's position is out of the range of interest, the answer is labeled as wrong. When disconnections occur, the client application sends update messages. One update message is necessary to get almost six correct responses, thus reducing the communication overhead due to context changes. Table 1 summarizes these experimental results. It reports the average value and the 90 percent confidence interval for every metric. The reason why the interresponse time is rather high is that it includes the time to propagate and process the updates sent from the client side to recalibrate the query parameters. As nodes constantly move and end-to-end interactions usually involve 2-hop communication, high interresponse and service discovery times are mainly due to the routing overhead. However, these values are more than acceptable for this type of application.

6.2 Simulation Experiments

The objective of the simulation study was to evaluate the scalability of the migratory service model. The *TJam-Smart* and *TJam-Base* services have been simulated using the *ns-2* simulator [11], enhanced with the Carnegie Mellon University (CMU)-wireless extensions [12]. As wireless media, we used 802.11b with a data transmission rate of 11 Mbits and a transmission range of 250 meters. The received signal strength threshold is set to maintain information about neighbors within 200 meters only. TJam-Smart implements the migratory service model, whereas TJam-Base represents a baseline centralized approach. In TJam-Base, a few mobile nodes host the service in the network; upon receiving a client's request, the service node computes the traffic jam probability by directly querying nodes in the region of interest, sends back the results to the client, automatically updates the query parameters, and finally initiates a new query cycle.

TABLE 1
Results of Testbed Experiments

	avg [90% conf int]
Interresponse time (s)	5.46 [0.21]
Service discovery time (s)	5.19 [0.79]
Service quality (correct answers/total answers)	0.93 [0.01]
Response/update rate (responses per update)	5.88 [0.01]

Our simulation study is organized in three parts. First, we investigated the scalability of the migratory service model with respect to the number of clients (scenario 1) and, then, the effects of vehicular traffic variability with respect to the vehicles speed (scenario 2) and the vehicles density (scenario 3).

6.2.1 Vehicular Traffic Generator

We used a microscopic simulation model. Three types of simulation models for traffic analysis are generally available for usage: Macroscopic, Mesoscopic, and Microscopic simulation models [13]. Macroscopic simulation models such as those in [14] and [15] are based on the deterministic relationships of flow, speed, and density of the traffic stream. The simulation in a macroscopic model takes place on a section-by-section basis rather than by tracking individual vehicles. Microscopic models, such as those in [16] and [17], simulate the movement of individual vehicles. Typically, vehicles enter a transportation network using a statistical distribution of arrivals and are tracked through the network over small time intervals. Mesoscopic simulation models, such as that in [18], combine the properties of both microscopic and macroscopic models. As both macroscopic and mesoscopic models do not provide the flexibility to analyze traffic in as much detail as the microscopic models, we decided to employ a microscopic model.

As microscopic modeling tools are not freely available to the public or do not capture certain road characteristics, we have developed our own microscopic traffic generator tool, called *Micro-VTG*, which is based on the random-way point mobility model used by ns-2. The scenario generator accepts as parameters the simulation time, road length, number of lanes per road, average speed of the vehicles, average gap distance between vehicles on the same lane, number of service (or metaservice) nodes, and number of client nodes. More details about Micro-VTG can be found in [19], where it was used to generate highway traffic, and in [20], where it was used to generate city traffic. As far as the simulation study in this article is concerned, we focus on highway traffic, and we generate various scenarios that capture significant highway conditions.

6.2.2 Simulation Metrics

In the simulation study, we employed the following metrics:

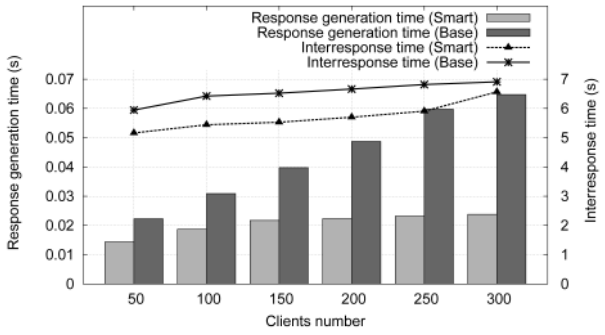

Fig. 9. Response generation time (bars) and interresponse time (lines) versus the number of clients.

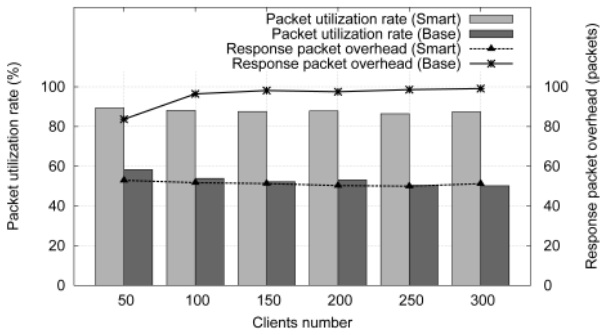

Fig. 10. Packet utilization rate (bars) and response packet overhead (lines) versus the number of clients.

- *Interresponse time*. This is the same metric previously defined for the testbed experiments. However, note that, in the simulation results, this time also includes the sleep interval between two consecutive result generation processes (in our tests, this was set to 5 seconds).
- *Response generation time*. This is the average time needed by a correct response to be computed and transmitted in the ideal case in which no responses are lost or wrongly computed and no updates are triggered from the client side. It measures the elapsed time between the time at which a query process is started (either by the latest node in the TJam-Smart or by the service node in the TJam-Base) and the time at which the response is delivered to the client. It differs from the interresponse time as it does not include the time needed by the update process or the time wasted due to lost responses.
- *Packet utilization rate*. This measures the average percentage of exchanged packets that were actually used in computing and delivering results out of the total number of exchanged packets.
- *Response packet overhead*. This is the average number of packets that needs to be exchanged for every correct result received by the client (that is, packets exchanged to process the query, to update request parameters, and to communicate the result).

6.2.3 Simulation Results

Service nodes and clients are selected uniformly among the vehicles. By "service" node, we mean a node hosting a service in TJam-Base and a node hosting a metaservice in TJam-Smart. When a client becomes active, a client request is generated after a warm-up period of 25 seconds. In TJam-Base, once a client request is assigned to a service node, the service node initiates a query process phase, which is periodically reinitiated every 5 seconds. In addition, if a client does not receive any result after three time periods, it resends the request.

Scenario 1—Effects of the number of clients. We first studied how TJam-Smart scales with an increasing number of clients. The scenario consists of a three-lane highway of length 25 km. The average vehicular speed is 30 m/s (that is, 108 km/h) with a gap of 150 meters. The total number of vehicles is 800, with 500 vehicles active. Fifty service nodes are randomly selected among all vehicles.

Fig. 9 plots the average time needed to generate new correct responses and the interresponse time experienced by TJam-Smart and TJam-Base users. Interresponse times include the sleep time of 5 seconds between subsequent querying phases. In TJam-Smart, the time needed to compute and transmit new correct responses is almost constant, whereas in TJam-Base, it increases as the number of clients grows. This is due to the capability of migratory services to follow the user motion and to the fact that no service is overloaded. In the statically centralized approach, the service is always running on the same node; as the number of clients grows, the average distances in hops between the client node and the service node and between the service node and the region of interest increase. On the other hand, with migratory services, those distances remain practically the same as the service migrates according to the user movement. In TJam-Base, if the packets need to traverse longer paths, their chance of getting lost increases due to geographic routing holes or collisions, thereby, the performance is negatively affected. This phenomenon partly also affects the interresponse times, where we observe lower interresponse times (of roughly 1 second) for TJam-Smart and slightly increasing interresponse times with the number of clients. Moreover, another reason why TJam-Smart performs better is that fewer packets (carrying responses, updates, intermediate results, and so forth) need to be exchanged.

As Fig. 10 shows, TJam-Smart performs more efficiently in terms of packet utilization rates and response packet overhead. In the migratory services approach, the packet utilization rate is over 86 percent, as the packet overhead consists only of update packets sent by the client to refresh its request parameters. In the centralized approach, as the service might execute on a node far from the region of interest, the additional overhead derives from packets used to propagate the service request to nodes located in such a region and to receive the collected data. Consequently, the total number of packets that need to be exchanged per each received correct response (that is, the response packet overhead) is higher in TJam-Base. Additionally, in TJam-Base, exchanged messages generally need to traverse longer paths as the monitored region moves further, thus generating a larger number of packets in the network.

Scenario 2—Effects of vehicles speed. We then investigated how the vehicular speed affects the performance of migratory services. The number of clients was fixed to 150. We varied the average speed of vehicles from 10 m/s to 30 m/s (that is, from 36 km/h to 108 km/h). The values

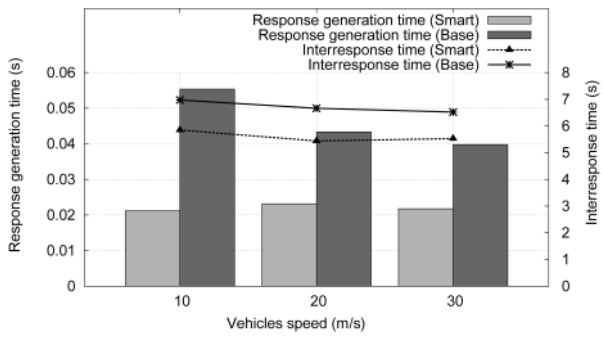

Fig. 11. Response generation time (bars) and interresponse time (lines) versus the average vehicular speed.

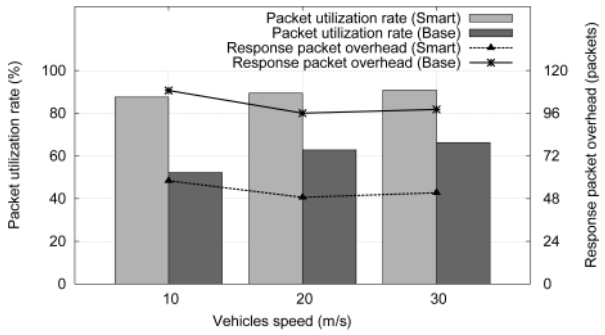

Fig. 12. Packet utilization rate (bars) and response packet overhead (lines) versus average vehicular speed.

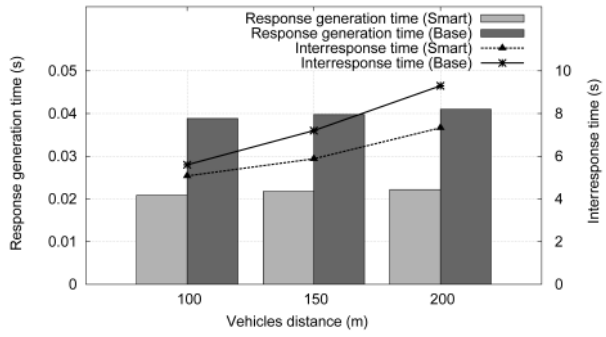

Fig. 13. Response generation time (bars) and interresponse time (lines) versus average vehicle distance

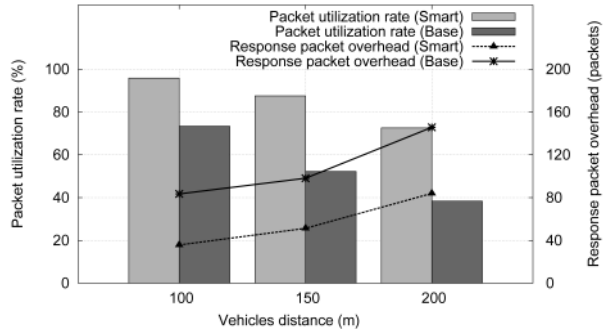

Fig. 14. Packet utilization rate (bars) and response packet overhead (lines) versus average vehicle distance.

of the rest of the parameters are as given in the previous scenario. Results for this scenario are reported in Figs. 11 and 12. TJam-Smart performs better with respect to all four metrics. As the vehicular speed increases, TJam-Smart guarantees almost constant response generation times and packet utilization rates, whereas TJam-Base performs slightly better with an increased vehicular speed. Moreover, with both mechanisms, as the average vehicular speed increases, the interresponse time and the response packet overhead decrease. The better performance for higher vehicles speed is due to the fact that a higher mobility increases network connectivity [21]. This phenomenon is more evident for TJam-Base because the packet overhead in it is almost double that in TJam-Smart.

Scenario 3—Effects of vehicles density. The settings for this scenario are the same as that for the previous one, with the average speed fixed to 20 m/s (that is, 72 km/h). The vehicles density is varied by increasing the average gap between each consecutive car from 100 to 200 meters. As Figs. 13 and 14 show, in both mechanisms, variations of the vehicles density have no effect on the time required by the service to generate correct responses. However, as the vehicle gap grows, the interresponse times observed by the client (which include the times to process updates, timeouts due to lost responses, and sleep time) increase since the chance of having holes in the geographic routing increases. The interresponse time is lower in TJam-Smart, and it increases more slowly compared to TJam-Base, as the packet overhead is higher in TJam-Base. In addition, larger gaps between vehicles negatively affect the communication and increase the number of packet losses and retransmission; with increasing vehicular distances, the packet utilization rate and the response packet overhead heavily decreases and increases, respectively.

From the above results, we can conclude that TJam-Smart performs better than TJam-Base in terms of response times, network utilization, and packet overhead. Moreover, these experiments showed that, besides increasing vehicles density, increasing the vehicles speed also helps reduce the communication overhead and provide a better performance. These results indicated how the service migration mechanism employed by TJam-Smart is more efficient and scalable than the traditional centralized mechanism used by TJam-Base. Essentially, with a higher number of clients and with different traffic conditions, migratory services guarantee better results than the statically centralized approach due to the smaller number of packets that need to be exchanged.

7 Related Work

In the near future, mobile users will expect to be provided with continuous access to personalized adaptive services [22]. Services become personalized when they are tailored to the user's context and adaptive when they are able to adapt to context changes observed in the environment. Several projects have investigated how to employ context awareness in carrying out service provisioning. In [23], an intelligent software agent transparently and constantly selects the most suitable network service (in terms of network quality of service (QoS) and connectivity) based on the user's profile. In [24], Lee and Helal demonstrate how service matching based on static descriptions of service characteristics provides minimal service discovery and filtering, whereas useful context information including service-specific selection logic, user characterization, and network conditions can be exploited to minimize user's manual selection and to make the

service discovery process more effective. In our work, we employ context awareness in a more extensive way. We propose a context-aware service provisioning model in which not only the framework supporting the client application monitors context changes and enables the application to react accordingly but the service itself can react to context changes occurring on the hosting node and in the surrounding environment. Additionally, the service is aware of the user's operating context.

The adoption of context awareness in mobile ad hoc networks enables interaction and coordination of entities in such networks. The Sentient Model [25] abstracts context-aware applications in pervasive ad hoc environments as a large collection of software components called Sentient Objects. These accept input via a variety of sensors and autonomously react by acting upon the environment through a variety of actuators. Sentient Objects were used to build sentient vehicles, which are context-aware vehicles cooperating over mobile ad hoc networks [26]. Julien and Roman propose the EgoSpaces model [27] and demonstrate how context awareness can be employed to abstract resources available in an ad hoc network into a data structure. EgoSpaces consists of logically mobile agents that operate over mobile nodes. The agents can specify which data have to be included in their operating context by means of declarative specifications containing properties of the data items, of the agents that own the data, of the hosts on which those agents are running, and of the attributes of the ad hoc network. The coordination model sees agents interacting with a dynamically changing environment through a set of views, which are custom-defined projections of the set of data objects present in the surrounding ad hoc network.

Similarly, the migratory service model addresses cooperation over mobile ad hoc networks, but it differs from these projects in the problem it solves. Context information is not only used to represent the operating environment of available entities but also to support the concrete deployment of real-world context-aware adaptive services that are built on top of those computing entities. The uniqueness of our solution consists of the context-aware service migration model and the framework that supports it. In our vision, the ad hoc network enables a completely new class of services. Few attempts have been made so far to deploy real services over mobile ad hoc networks. In [28], Doyle et al. propose a mobile context-aware narrative that allows the dynamic presentation of scenes and sequences based on the interaction of the user with the story itself. Behavior and actions of the user influence the narrative and make the story interactive. However, at the core of the system, there is a hybrid ad hoc network, which consists not only of mobile nodes associated with users, but also of fixed multimedia nodes embedded in the environment. In our scenarios, we consider only mobile and often resource-constrained entities.

To provide context-aware adaptive services, we use execution migration. Migratory services share ideas and leverage work done on process migration [29], [30], [31], [32], [33], virtual machine monitors (VMMs) [34], [35], [36], mobile agents [37], [38], active networks [39], [40], [41], and dynamic reconfiguration in distributed systems [42], [43]. Research work that can be seen as precursors of our model includes process migration for load balancing and service component offloading [44]. MobiDesk [45] allows a user's computing environment to be migrated transparently from one server to another by decoupling a user's desktop computing session from the underlying operating system and service instance. MobiDesk builds on Zap [46], a system for transparent migration of networked applications.

Fluid [47] offers a nomadic and resource-aware Web service framework that allows dynamically migrating a Web service to different nodes according to the available resources at nodes. However, this work addresses services running in relatively stable Internet-based networks and is similar to traditional process migration for load balancing. We share the concept of relocation to other hosts in the ad hoc network due to new context requirements on the original host, but we target services in mobile ad hoc networks. Moreover, in our approach, we consider a wider range of context information that characterizes both the user and service.

Another research project similar to migratory services proposes "follow-me" services [48]. In this proposal, as the user moves through the network, services can migrate from node to node to maintain a seamless interaction with the client application. As such, these services can be considered a particular example of migratory services where the migration is triggered by the lack of connectivity between client applications and services.

Work done on Transmission Control Protocol (TCP) connection migrations and continuations for Internet services could be employed to achieve the same goals as that of our model. Session-based mobility [49] provides continuations at the application level to support migration and load balancing. Service Continuations [50], [51] is an operating system mechanism that enables seamless dynamic migration of Internet service sessions between multiprocess cooperating servers. Fault-tolerant TCP (FT-TCP) [52] allows a faulty server to keep its TCP connections open until the server recovers or until they are moved to a backup server that will take over the interaction with the connected client processes. This process is transparent to the connected clients. There are several problems, however, that make these solutions hard, if not impossible, for ad hoc networks. First, the traditional end-to-end model of communication assumed by TCP does not work well in dynamic ad hoc networks [53]. Second, it is difficult to introduce changes in TCP as network providers deny them. Third, TCP may be relatively heavyweight for resource-constrained devices.

Trickles [54], a TCP-like transport protocol, provides service continuation at the packet level, thus enabling new functionalities within the network infrastructure. The novelty of Trickles is that the system state can be kept entirely on one side of a connection, thus allowing the other side of the connection (typically the server) to operate without any per-connection state. Trickles servers can be replicated and distributed while providing the same services as that of stateful servers. However, the applicability of this approach to support services in ad hoc networks may not always be feasible due to CPU limitations and dynamism that characterize nodes in ad hoc networks.

In recent years, there have been many research efforts to improve the programmability of sensor networks and ubiquitous computing environments. For example, Hood

[55] simplifies application development by providing high-level abstractions that group together nodes with similar properties and enable data sharing among neighboring nodes. MagnetOS [56] is a distributed operating system that supports the programming of ad hoc networks by making the entire network appear as a single VM. Applications are automatically and transparently partitioned into components that are placed on nodes within the network to reduce energy consumption and increase system longevity. Among many projects that target the programmability of ubiquitous computing environments, *one.world* [57] is similar to our work in the sense that both consider migration as a key mechanism to adapt to dynamic computing environments. Each application in *one.world* has at least one environment that contains tuples (similar to application tags in the SM platform), application's components, and other nested environments. When needed, a migration moves a checkpointed copy of an environment to another node.

8 Conclusions

In this paper, we presented a context-aware service model that allows ad hoc networks to provide services that quickly adapt to context changes while still guaranteeing service continuity to the client. A migratory service framework monitors these changes and reacts by triggering a service migration each time it renders the current hosting node unsuitable for supporting the service execution any longer. As a result, the service resumes its execution on a new node where it can effectively accomplish its task. Service migrations are transparent to client applications because the framework constantly presents a single virtual end point for every migratory service.

We used this framework to build a migratory service prototype, which was evaluated using an ad hoc network of PDAs, as well as large-scale simulations. The experimental results demonstrated the viability of our model in highly dynamic ad hoc networks such as cars moving on a highway. The simulation results showed that our migratory service performs better than a traditional implementation based on a centralized model in terms of response time, efficiency, communication overhead, and scalability.

Acknowledgments

This work was supported in part by US National Science Foundation Grants ANI-0121416, CNS-0520123, CNS-0520033, CNS-0454081, and IIS-0534520 and in part by the Nokia and Emil Aaltosen scholarships. The authors would also like to thank the anonymous reviewers who helped them improve this paper.

References

[1] N. Davies, A. Friday, S.P. Wade, and G.S. Blair, "L2imbo: A Distributed Systems Platform for Mobile Computing," *ACM Mobile Networks and Applications,* special issue on protocols and software paradigms of mobile networks, vol. 3, no. 2, pp. 143-156, 1998.

[2] *Jini Network Technology,* http://www.sun.com/software/jini, Sept. 2007.

[3] *Web Services Dynamic Discovery (WS-Discovery),* http://specs.xmlsoap.org/ws/2005/04/discovery/, Apr. 2005.

[4] C. Borcea, D. Iyer, P. Kang, A. Saxena, and L. Iftode, "Cooperative Computing for Distributed Embedded Systems," *Proc. 22nd Int'l Conf. Distributed Computing Systems (ICDCS '02),* pp. 227-236, July 2002.

[5] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode, "Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems," *Computer J.,* special issue on mobile and pervasive computing, pp. 475-494, 2004.

[6] C. Borcea, C. Intanagonwiwat, A. Saxena, and L. Iftode, "Self-Routing in Pervasive Computing Environments Using Smart Messages," *Proc. First IEEE Int'l Conf. Pervasive Computing and Comm. (PerCom '03),* pp. 87-96, Mar. 2003.

[7] *K Virtual Machine,* http://java.sun.com/products/cldc/, Sept. 2007.

[8] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications," *IEEE Trans. Software Eng.,* vol. 29, no. 10, pp. 929-945, Oct. 2003.

[9] B. Karp and H. Kung, "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks," *Proc. ACM MobiCom,* pp. 243-254, Aug. 2000.

[10] C. Perkins and E. Royer, "Ad-Hoc On-Demand Distance Vector Routing," *Proc. Second IEEE Workshop Mobile Computing Systems and Applications (WMCSA '99),* pp. 90-100, Feb. 1999.

[11] *The Network Simulator ns-2,* http://www.isi.edu/nsnam/ns/, Sept. 2007.

[12] *The Monarch Group at Rice University,* http://www.monarch.cs.rice.edu/, Sept. 2007.

[13] "Traffic Analysis Tools Primer," *Traffic Analysis Toolbox,* vol. 1, http://ops.fhwa.dot.gov/trafficanalysistools/tat_vol1, June 2004.

[14] *KRONOS,* http://street.umn.edu/, Sept. 2007.

[15] M. van den Berg, A. Hegyi, B. De Schutter, and J. Hellendoorn, "A Macroscopic Traffic Flow Model for Integrated Control of Freeway and Urban Traffic Networks," *Proc. 42nd IEEE Conf. Decision and Control,* pp. 2774-2779, Dec. 2003.

[16] *AIMSUN NG the Integrated Traffic Environment,* http://www.tss-bcn.com/aimsun.html, Sept. 2007.

[17] *DRACULA (Dynamic Route Assignment Combining User Learning and microsimulAtion),* http://www.its.leeds.ac.uk/software/dracula, Sept. 2007.

[18] *CONTRAM (CONtinuous TRaffic Assignment Model),* http://www.contram.com, Sept. 2007.

[19] T. Nadeem, S. Dashtinezhad, C. Liao, and L. Iftode, "TrafficView: Traffic Data Dissemination Using Car-to-Car Communication," *ACM Mobile Computing and Comm. Rev.,* vol. 8, no. 3, pp. 6-19, July 2003.

[20] P. Zhou, T. Nadeem, P. Kang, C. Borcea, and L. Iftode, "EZCab: A Cab Booking Application Using Short-Range Wireless Communication," *Proc. Third IEEE Int'l Conf. Pervasive Computing and Comm. (PerCom '05),* pp. 27-38, Mar. 2005.

[21] M. Grossglauser and D.N.C. Tse, "Mobility Increases the Capacity of Ad Hoc Wireless Networks," *IEEE/ACM Trans. Networking,* vol. 10, no. 4, pp. 477-486, 2002.

[22] *Technologies for the Wireless Future: Wireless World Research Forum (WWRF),* R. Tafazolli, ed., John Wiley & Sons, Oct. 2004.

[23] P.F.G. Lee, S. Bauer, and J. Wroclawski, "A User-Guided Cognitive Agent for Network Service Selection in Pervasive Computing Environments," *Proc. Second IEEE Ann. Conf. Pervasive Computing and Comm. (PerCom '04),* pp. 219-228, Mar. 2004.

[24] C. Lee and S. Helal, "Context Attributes: An Approach to Enable Context-Awareness for Service Discovery," *Proc. Third IEEE/IPSJ Symp. Applications and the Internet (SAINT '03),* pp. 22-30, Jan. 2003.

[25] M. Wu, A. Friday, G.S. Blair, T. Sivaharan, P. Okanda, H.D. Limon, C.-F. Sørensen, G. Biegel, and R. Meier, "Novel Component Middleware for Building Dependable Sentient Computing Applications," *Proc. ACM Workshop Component-Oriented Approaches to Context-Aware Computing (ECOOP '04),* June 2004.

[26] T. Sivaharan, G.S. Blair, A. Friday, M. Wu, H.A. Duran-Limon, P. Okanda, and C.-F. Sørensen, "Cooperating Sentient Vehicles for Next Generation Automobiles," *Proc. ACM/Usenix MobiSys Int'l Workshop Applications of Mobile Embedded Systems (WAMES '04),* June 2004.

[27] C. Julien and G. Roman, "Active Coordination in Ad Hoc Networks," *Proc. Sixth Int'l Conf. Coordination Models and Languages,* pp. 199-215, Feb. 2004.

[28] L. Doyle, G. Davenport, and D. O'Mahony, "Mobile Context-aware Stories," *Proc. IEEE Int'l Conf. Multimedia and Expo (ICME '02),* vol. 2, pp. 345-348, Aug. 2002.

[29] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," *ACM Computing Surveys,* vol. 32, no. 3, pp. 241-299, Sept. 2000.
[30] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for Unix Development," *Proc. Usenix Summer Conf.,* pp. 93-113, July 1986.
[31] A. Barak and R. Wheeler, "MOSIX: An Integrated Mulitprocessor Unix," *Mobility: Processes, Computers, and Agents,* pp. 41-53, 1999.
[32] D.R. Cheriton, "The V Distributed System," *Comm. ACM,* vol. 31, no. 3, pp. 314-333, 1988.
[33] P. Smith and N.C. Hutchinson, "Heterogeneous Process Migration: The Tui System," *Software—Practice and Experience,* vol. 28, no. 6, pp. 611-639, 1998.
[34] C.P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M.S. Lam, and M. Rosenblum, "Optimizing the Migration of Virtual Computers," *ACM SIGOPS Operating Systems Rev.,* vol. 36, pp. 377-390, 2002.
[35] *Internet Suspend/Resume Project,* http://www.pittsburgh.intel-research.net/projects/isr.html, Sept. 2007.
[36] *VMware VirtualCenter,* http://www.vmware.com/products, Sept. 2007.
[37] J. White, *Mobile Agents,* J.M. Bradshaw, ed., MIT Press, 1997.
[38] R. Gray, G. Cybenko, D. Kotz, and D. Rus, "Mobile Agents: Motivations and State of the Art," *Handbook of Agent Technology,* J. Bradshaw, ed., AAAI/MIT Press, 2002.
[39] D. Wetherall, "Active Network Vision Reality: Lessons from a Capsule-Based System," *Proc. 17th ACM Symp. Operating Systems Principles (SOSP '99),* pp. 64-79, Dec. 1999.
[40] J. Moore, M. Hicks, and S. Nettles, "Practical Programmable Packets," *Proc. IEEE INFOCOM,* pp. 41-50, Apr. 2001.
[41] B. Schwartz, A. Jackson, W. Strayer, W. Zhou, R. Rockwell, and C. Partridge, "Smart Packets: Applying Active Networks to Network Management," *ACM Trans. Computer Systems,* vol. 18, no. 1, pp. 67-88, 2000.
[42] J. Kramer and J. Magee, "Dynamic Configuration for Distributed Systems," *IEEE Trans. Software Eng.,* vol. 11, no. 4, pp. 424-436, 1985.
[43] J. Magee and M. Sloman, "Constructing Distributed Systems in Conic," *IEEE Trans. Software Eng.,* vol. 15, no. 6, pp. 663-675, June 1989.
[44] A. Messer, I. Greenberg, P. Bernadat, D.S. Milojicic, D. Chen, T.J. Giuli, and X. Gu, "Towards a Distributed Platform for Resource-Constrained Devices," *Proc. 22nd Int'l Conf. Distributed Computing Systems (ICDCS '02),* pp. 43-51, July 2002.
[45] R.A. Baratto, S. Potter, G. Su, and J. Nieh, "MobiDesk: Mobile Virtual Desktop Computing," *Proc. ACM MobiCom,* pp. 1-15, 2004.
[46] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," *ACM SIGOPS Operating Systems Rev.,* vol. 36, pp. 361-376, 2002.
[47] I. Pratistha and A. Zaslavsky, "Fluid: Supporting a Transportable and Adaptive Web Service," *Proc. ACM Symp. Applied Computing (SAC '04),* pp. 1600-1606, 2004.
[48] R. Handorean, R. Sen, G. Hackmann, and G.-C. Roman, "Context Aware Session Management for Services in Ad Hoc Networks," *Proc. IEEE Int'l Conf. Services Computing (SCC '05),* pp. 113-120, July 2005.
[49] A.C. Snoeren, "A Session-Based Approach to Internet Mobility," PhD dissertation, Dept. of Electrical Eng. and Computer Science, Massachusetts Inst. of Technology, Dec. 2002.
[50] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode, "Migratory TCP: Connection Migration for Service Continuity in the Internet," *Proc. 22nd Int'l Conf. Distributed Computing Systems (ICDCS '02),* pp. 469-470, July 2002.
[51] F. Sultan, A. Bohra, and L. Iftode, "Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions," *Proc. Symp. Reliable Distributed Systems (SRDS '03),* pp. 177-186, Oct. 2003.
[52] L. Alvisi, T.C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, "Wrapping Server-Side TCP to Mask Connection Failures," *Proc. IEEE INFOCOM,* pp. 329-337, 2001.
[53] C. Wan, A. Campbell, and L. Krishnamurthy, "PSFQ: A Reliable Transport Protocol for Wireless Sensor Networks," *Proc. First ACM Int'l Workshop Wireless Sensor Networks and Applications (WSNA '02),* pp. 1-11, Sept. 2002.
[54] A. Shieh, A. Myers, and E.G. Sirer, "Trickles: A Stateless Network Stack for Improved Scalability, Resilience and Flexibility," *Proc. Second Usenix Symp. Networked Systems Design and Implementation (NSDI '05),* pp. 175-188, May 2005.
[55] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: A Neighborhood Abstraction for Sensor Networks," *Proc. ACM MobiSys,* pp. 99-110, 2004.
[56] H. Liu, T. Roeder, K. Walsh, R. Barr, and E.G. Sirer, "Design and Implementation of a Single System Image Operating System for Ad Hoc Networks," *Proc. ACM MobiSys,* pp. 149-162, 2005.
[57] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall, "System Support for Pervasive Applications," *ACM Trans. Computer Systems,* vol. 22, no. 4, pp. 421-486, 2004.

Oriana Riva received the MSc degree in telecommunication engineering from Politecnico di Milano, Italy, in 2003. She is completing her dissertation on how to support mobile sensing applications in heterogenous pervasive environments for the doctoral degree in computer science at the University of Helsinki. She is a researcher at the Helsinki Institute for Information Technology. Her research interests include ubiquitous computing, ad hoc networking, and context-aware services for mobile users.

Tamer Nadeem received the PhD degree from the University of Maryland in 2006. He is a research scientist at Siemens Corporate Research. His research interests include wireless networks management, mobile ad hoc networks, vehicular networks, sensor networks, peer-to-peer systems, and pervasive computing. He is a member of the IEEE, the IEEE Computer Society, and the ACM. He is an elected member of the Phi Kappa Phi and Sigma Xi honor societies and a member of the Association of Egyptian American Scholars.

Cristian Borcea received the PhD degree in computer science from Rutgers University in 2004. He is an assistant professor in the Department of Computer Science, New Jersey Institute of Technology. His research interests include mobile computing, middleware for ubiquitous networked systems, vehicular networks, and sensor networks. He is a member of the IEEE, the IEEE Computer Society, the ACM, and Usenix.

Liviu Iftode received the PhD degree in computer science from Princeton University in 1998. He is an associate professor of computer science at Rutgers University, New Jersey. His research interests include distributed systems, operating systems, vehicular networks, and mobile and pervasive computing. He is a senior member of the IEEE and the IEEE Computer Society and a member of the ACM. He is the vice-chair of the IEEE Technical Committee on Operating Systems and a member of the editorial boards of *IEEE Pervasive Computing* and *IEEE Distributed Systems Online*.