Department of Computer Science
Series of Publications A
Report A-2007-6

Middleware for Mobile Sensing Applications in Urban Environments

Oriana Riva

Academic Dissertation

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium XIV, University Main Building, on November 2nd, 2007, at 12 o'clock noon.

University of Helsinki
Finland

ISSN 1238-8645
ISBN 978-952-10-4287-4 (paperback)
ISBN 978-952-10-4288-1 (PDF)
http://ethesis.helsinki.fi/

Computing Reviews (1998) Classification: C.2.4, D.2.11

Helsinki University Printing House
Helsinki, October 11 2007 (xvi + 195 pages)

Middleware for Mobile Sensing Applications in Urban Environments

Oriana Riva
Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
oriana.riva@cs.helsinki.fi
http://www.cs.helsinki.fi/u/riva/

Abstract

Sensor networks represent an attractive tool to observe the physical world. Networks of tiny sensors can be used to detect a fire in a forest, to monitor the level of pollution in a river, or to check on the structural integrity of a bridge. Application-specific deployments of static-sensor networks have been widely investigated. Commonly, these networks involve a centralized data-collection point and no sharing of data outside the organization that owns it. Although this approach can accommodate many application scenarios, it significantly deviates from the pervasive computing vision of *ubiquitous sensing* where user applications seamlessly access anytime, anywhere data produced by sensors embedded in the surroundings.

With the ubiquity and ever-increasing capabilities of mobile devices, urban environments can help give substance to the ubiquitous sensing vision through Urbanets, spontaneously created urban networks. Urbanets consist of mobile multi-sensor devices, such as smart phones and vehicular systems, public sensor networks deployed by municipalities, and individual sensors incorporated in buildings, roads, or daily artifacts. My thesis is that *"multi-sensor mobile devices can be successfully programmed to become the underpinning elements of an open, infrastructure-less, distributed sensing platform that can bring sensor data out of their traditional close-loop networks into everyday urban applications"*. Urbanets can support a variety of services ranging from emergency and surveillance to tourist guidance, shopping, and entertainment. For instance, cars can be used to provide traffic information services to alert drivers to upcoming traffic jams, and phones to provide shopping recommender services to inform users of special offers at the mall.

Urbanets cannot be programmed using traditional distributed computing models, which assume underlying networks with functionally homogeneous nodes, stable configurations, and known delays. Conversely, Urbanets have functionally heterogeneous nodes, volatile configurations, and unknown delays. More effectively, solutions developed for sensor networks and mobile ad hoc networks can be leveraged to provide novel architectures and models that address Urbanet-specific requirements, while providing useful abstractions that hide the network complexity from the programmer.

This dissertation presents two middleware architectures that can support people-centric mobile sensing applications in Urbanets. *Contory* offers a declarative programming model that views Urbanets as a distributed sensor database and exposes a simple SQL-like interface to application developers. *Context-aware Migratory Services* provides a client-server model, where services are capable of migrating to different nodes in the network in order to maintain a continuous and semantically correct interaction with clients. Compared to previous approaches to supporting mobile sensing applications in urban environments, our architectures are entirely distributed and do not assume constant availability of Internet connectivity. In addition, they allow on-demand collection of sensor data with the accuracy and at the frequency required by every single application.

These architectures have been implemented in Java and tested on smart phones. They have proved successful in supporting several prototype applications and experimental results obtained in ad hoc networks of phones have demonstrated their feasibility with reasonable performance in terms of latency, memory, and energy consumption.

Computing Reviews (1998) Categories and Subject Descriptors:

C.2.4 Computer-Communication Networks: Distributed Systems

D.2.11 Software Engineering: Software Architectures

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Middleware, Pervasive Computing, Context-awareness, Urban Environments, Mobile Devices, Sensors

Acknowledgements

At the end of this long journey—roughly 6003 km by bike—I'm happy to express my gratitude to many exceptional people without whom this dissertation would not have been possible.

First and foremost, my very special thanks go to my advisor Kimmo Raatikainen and my co-advisor Cristian Borcea. Kimmo gave me the freedom to undertake research in any place, in any way, and at any time I wished. I'm grateful to him for his generous support throughout my PhD work, for always trusting me, and for teaching me the importance of recognizing and carrying out "good" research. Cristian has been a true mentor as well as an amazing friend for me. I'm indebted to him for his constant encouragement, his patience, and invaluable guidance throughout this research. His critical feedback and insights had an essential impact on this dissertation.

I also wish to thank Jari Porras and Jukka Riekki for reviewing this dissertation and giving helpful comments to improve it.

The Department of Computer Science at the University of Helsinki and the Helsinki Institute for Information Technology provided an excellent working environment. My gratitude goes to Martti Mäntylä, Jukka Paakki, Hannu Toivonen, and Esko Ukkonen for making this work possible. A special thanks to Martti for his generosity with his time and advice. Office staff and system administrators are also acknowledged for keeping things running so smoothly.

I gratefully acknowledge the financial support of my hosting institutions through several projects I was involved in. In particular, I benefited from the DYNAMOS project funded by Tekes, ICT-Turku, Suunto, TeliaSonera and VTT under which a large part of the practical work of my PhD was carried out. I also express my gratitude to Santtu Toivonen for managing the project team.

During the last four years, I had the privilege to collaborate with many excellent researchers and visit several research institutes. My special thanks go to Liviu Iftode and to the Disco Lab at Rutgers University. I'm truly grateful to Liviu for believing in me since the very beginning of this work, for his generosity and vitality in pushing me to achieve the best, and for our lively discussions. My gratitude also goes to Valérie Issarny and to the ARLES group at INRIA-Rocquencourt. I'm grateful to Valérie for simply being the great woman she is, always available, always insightful, and so close to all her students. Finally, I wish to acknowledge the ESF-funded scientific program MiNEMA, which gave me the opportunity to interact with many experts in my field.

Life at the department would not have been so efficient and enjoyable without Tiina Niklander. I thank her for always finding the time to instill in me self-worth in those hard moments a PhD is made of. I thank my former colleagues Davide Astuti, Cristiano di Flora, and Simone Leggio who have always supported me with great enthusiasm. I thank my officemate Laila Daniel who has so warmly helped me on more occasions than I can remember. I'm grateful to the Fuego Core team and in particular to Jaakko Kangasharju for his expert advice and his patience while dealing with me and my bugs in many dark and cold Finnish afternoons. I owe a lot to Michael Przybilski for always willingly lending a hand, solving my electrical problems, and reminding me of "order and discipline". A special thanks to Evimaria Terzi for her true friendship, her constant advice, and our Friday sessions.

I would like to thank all my friends, particularly in Helsinki and Paris, for making my life during these years so memorable and enjoyable. Thanks to the "Sxxxxx" gang and especially to Sabria for always being so close to me. Thanks to my lively French community and in particular to Agathe for her patience and encouragement. Thanks to the "laiset" group and especially to Guido for those long bike rides—in many of which I sonorously complained.

Finally, I owe eternal gratitude to my parents, Adele and Francesco, for their love and understanding in these years far from home, and to my brother, Claudio, for his support and for always reminding me to close my laptop and get out of "that" office.

Helsinki, October 11th, 2007 *Oriana Riva*

Ai miei genitori.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

This dissertation proposes distributed middleware architectures for the support of people-centric, mobile sensing applications in everyday urban environments. In this chapter, we motivate our work, describe our problem statement, summarize our contributions, and present the organization of the dissertation.

1.1 Motivation

Advances in sensor technology, embedded computing, and wireless networking have increasingly enabled the deployment of sensor networks across large geographical areas to provide accurate real-time monitoring of the sensed environment. Sensor networks have been considered in many different domains such as civil engineering (e.g., structural integrity of buildings), environmental monitoring (e.g., water pollution in rivers), or tracking systems (e.g., intrusion detection in a restricted area).

With few exceptions, first-generation sensor networks have mainly focused on accomplishing application-specific deployments of networks of static sensors. The regular model of operation involves a centralized querying and data-collection point, and, typically, sensed data are not disclosed to anyone outside the organization that owns and controls the network. Although this operational model can accommodate many application scenarios, it significantly deviates from the pervasive computing vision of "ubiquitous

sensing" where people seamlessly access anytime, anywhere data produced by sensors embedded in the surroundings.

As wireless mobile devices become ubiquitous and computationally more powerful, besides providing email and web access on-the-go, they can begin to serve three new purposes:

- acting as collaborating, multi-sensor devices that provide sensing coverage across cities,
- becoming dynamic points for collecting and sharing data produced by individual sensors or public sensor networks, and
- ultimately enabling users to benefit from a sensor-rich world through novel mobile sensing applications (i.e., applications based on sensor data collection and processing).

My thesis is that *"multi-sensor mobile devices can be successfully programmed to become the underpinning elements of an open, infrastructure-less, distributed sensing platform that can bring sensor data out of their traditional close-loop networks into everyday urban applications"*.

In particular, smart phones and vehicular systems are becoming attractive, convenient mobile sensor platforms. Compared to tiny, energy-constrained sensors of regular sensor networks, smart phones can support more complex computations, provide reasonable data storage, and offer long-range communication. These phones already have audio and video sensing capabilities, integrate GPS receivers, and, in the near future, they will come equipped with other types of sensors too. Power remains, however, a major constraint for them. Vehicular systems, on the other hand, do not have energy restrictions, and offer powerful processors, significant memory, plenty of storage capacity, and a variety of sensors.

Urban environments represent a perfect ground to build spontaneously created networks consisting of mobile multi-sensor platforms, such as smart phones and vehicular systems, public sensor networks deployed by municipalities, and individual sensors incorporated in buildings, roads, or daily artifacts. Sensor networks and mobile ad hoc networks (MANETs) meet in the urban landscape to create rich and open sensing environments, where people, municipalities, and community organizations share their resources to

give mobile users real-time access to sensed data. We use the term *Urbanets* to refer to this new type of spontaneous urban networks.

We envision a variety of Urbanet applications that will run, for instance, on our personal mobile phones or computers embedded in our cars. For example, using a collaborative network of sensor-equipped cars, a personal driver assistant warns its driver about upcoming traffic jams and provides route directions customized to the present traffic conditions. In addition, the application detects traffic hazards such as fog patches or icy roads by interacting with environmental sensors such as humidity and temperature sensors along the road. Another scenario draws a municipal weather monitoring sensor network alerting cars passing by to a quickly-approaching tornado. Finally, a last scenario is a crowded political convention or manifestation, where policemen's surveillance applications track suspicious entities moving around by using a network of cameras installed on police patrols and policemen's helmets.

Although algorithms, protocols, architectures, and models developed for sensor networks and MANETs can be applied to Urbanets, Urbanets are intrinsically different from sensor networks and MANETs. Urbanets differ from the first-generation sensor networks in their goal to support concurrent people-centric sensing applications as well as in their hardware and software heterogeneity, high volatility, and very large scale. Furthermore, while Urbanets greatly enhance our ability to extend the sensing coverage and incorporate sensed data in a large spectrum of mobile applications, they are not expected to achieve the same level of sensing fidelity as in static sensor networks composed of nodes primarily dedicated to sensing. Urbanet applications are also different from traditional MANET applications such as file transfers. Urbanet applications focus on acquiring, processing, and distributing real-time sensing information provided by devices located in the proximity of geographical regions, entities, or activities of interest.

1.2 Problem statement

Research work on sensor networks and mobile ad hoc networks has been quite successful in designing device platforms, protocols, and network architectures that can be applied to Urbanets. However,

programming people-centric mobile sensing applications has so far received only marginal attention. As the domain of possible Urbanet applications diversifies, it will be almost impossible to program each application from scratch. Instead, application developers will require a common distributed computing platform that can support the development and execution of such applications.

Urbanets cannot be programmed using traditional distributed computing models, which assume underlying networks with functionally homogeneous nodes, static resources, stable configurations, and known delays. Conversely, Urbanets are composed of functionally heterogeneous nodes, have volatile configurations, and present unknown delays; they evolve unpredictably over time and space, making it hard to know the exact number or location of their resources. In addition, Urbanets are required to support large data traffic of concurrent user applications, they often consist of resource-impoverished environments, and they present devices that are not primarily dedicated to support sensing tasks (e.g., phones are primarily used to make phone calls).

The questions that drive our research are:

- How do we support the development and execution of mobile sensing applications in Urbanets? Instead of building every application from scratch, a middleware platform is necessary to provide common services, runtime mechanisms, and other application-specific functions.

- Which programming abstractions are appropriate for Urbanets? Which protocols and algorithms do these abstractions require to function correctly? Which part of the underlying systems (networks, hardware, subsystems) should be exposed to application developers? There exists a trade-off between the ease of programming that programming abstractions can provide, and the efficiency and flexibility that can be achieved.

- Which design strategies can allow resource-constrained mobile devices to control and reduce their resource consumption while running Urbanet applications? What should be the trade-off between the quality of the produced results and the utilization of network resources?

1.3 Contributions

This dissertation focuses on providing middleware support and appropriate programming abstractions to develop people-centric mobile sensing applications in dense urban environments. Two middleware platforms that can effectively support the development and execution of mobile sensing applications in our daily urban environments are proposed. Our solutions assume cooperation among Urbanet devices and can work without requiring any additional infrastructure (i.e., only the resources of the Urbanet devices are necessary). Therefore, they have the benefit of providing users with real-time sensed data even when Internet connectivity is not available or is too expensive. Additionally, they allow on-demand collection of sensor data with the accuracy and at the frequency required by every single application. These two middleware platforms incorporate two different programming models and offer support for a wide variety of mobile sensing applications. To cope with the resource constraints of mobile devices and the volatility of the execution environment they integrate adaptation strategies and support application-specific control policies.

The *Contory* (*Cont*ext fact*ory*) [Riva, 2006] middleware supports a declarative programming paradigm that views Urbanets as a distributed sensor database and exposes a simple SQL-like interface to programmers. While other projects demonstrated how declarative programming suits sensor networks well [Madden et al., 2005], Contory seeks to adapt this model to highly mobile and heterogeneous networks. In addition, we devised and incorporated in the Contory middleware reconfiguration strategies capable of coping with the dynamism of Urbanets. As a result, Contory allows sensing applications to continuously monitor their surroundings despite sensor failures and network volatility. Compared to other platforms supporting sensor data collection and, in particular, context-aware applications, Contory provides more flexibility and reliability. Moreover, Contory specifically addresses the requirements of resource-constrained mobile devices by dynamically adapting its execution based on resource availability and device status.

The *Context-aware Migratory Services* [Riva et al., 2007] framework provides a client-service model where services can migrate to different nodes in the network to maintain a continuous and se-

mantically correct interaction with clients. Although a migratory service is physically located on different nodes over time, it constantly presents a single virtual end point to the client. Compared to classical client-server architectures, our approach provides two advantages. First, when a node becomes unsuitable for hosting a certain service any longer, the client application does not need to perform any new service discovery because the current service can migrate autonomously to a new 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 after the migration to a different node has completed. In addition, Migratory Services provides fault-tolerance to service failures by integrating an adaptive primary backup mechanism.

We have implemented these middleware architectures and experimented with them on modern smart phones. Using them, we have prototyped several applications targeting real-life scenarios. TJam 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. WeatherWatcher allows users to retrieve weather information relative to a certain geographical region of interest. The DYNAMOS sailing application proactively provides mobile users with information about nearby services that are of interest based on the user's current context and needs. Our middleware platforms proved effective in easing application development. They decouple application development from several underlying communication modules, sensor technology, and hardware systems. Furthermore, our programming abstractions provide powerful and simple primitives to specify complex and dynamic concepts (e.g., geographical regions to be monitored) and distributed tasks to be executed (e.g., tracking moving entities).

We evaluated the performance of our platforms in terms of latency, memory, and energy consumption in small-scale ad hoc networks of phones. Our solutions proved to be feasible. Besides typical interference problems in places with high density of wireless devices, energy consumption turned out to be the most constraining technical factor. We reported on the difficulties, the lessons we have learned, and our recommendations on how to best develop middleware architectures on smart phones in [Riva and Kangasharju, 2007].

1.4 Research history

A large part of the results presented in this dissertation have already been published in international conferences and journal articles. The contributions presented in this dissertation were produced while working in two parallel projects.

The first part of the contributions has been produced in the context of the 2-year DYNAMOS [DYNAMOS, 2007] (Dynamic Composition and Sharing of Context-Aware Mobile Services) project at the Helsinki Institute for Information Technology (HIIT) in collaboration with VTT Technical Research Centre of Finland, ICT-Turku, Suunto and TeliaSonera. In the first year, the author designed, implemented, and tested in two field trials the DYNAMOS sailing application (see Section 4.2) running on mobile phones. An extensive description and evaluation of this application was published in [Riva and Toivonen, 2006, 2007]. Inspired by the experiences of the first year, in the second year of the project, the author designed and implemented Contory (see Chapter 4), a middleware for the provisioning of context information on smart phones. This work was published in [Riva, 2006; Riva and di Flora, 2006a].

The second part of the contributions for this dissertation has been produced in collaboration with Rutgers University and New Jersey Institute of Technology. The results of this collaboration were Migratory Services [Riva et al., 2007] (see Chapter 5) and the notion of people-centric mobile sensing applications in Urbanets [Riva and Borcea, 2007] (see Chapter 3). The contribution of the author has been designing and carrying out the entire implementation and experimentation of Migratory Services on mobile devices. The author also contributed to test the Smart Messages platform developed at Rutgers University on mobile phones (see Section 4.6.1). Simulation results described in [Riva et al., 2007] were produced by a co-author and have not been included in the dissertation.

1.5 Structure of the dissertation

The rest of this dissertation is organized as follows. Chapter 2 provides background information for this dissertation. In the first

part, it introduces the pervasive computing vision and its challenges, and gives a brief state-of-the-art on hardware, networking and middleware support to accomplish such a vision. In the second part, it presents the concept of ubiquitous sensing, and describes architectures and protocols proposed in the domains of sensor networks and mobile ad hoc networks that can be used to accomplish ubiquitous sensing. The chapter concludes by describing the Smart Messages computing platform that was used in the implementation of our prototype systems. Chapter 3 introduces Urbanets, spontaneous urban networks that can support mobile, people-centric sensing applications. It describes core characteristics of these environments, example applications, and programming challenges to be addressed. Our middleware architectures for the support of Urbanet applications are presented in the following two chapters. Chapter 4 presents Contory and Chapter 5 presents Context-aware Migratory Services. Each chapter describes requirements, design principles, software architectures, implementation, experimental evaluation, and prototype applications running on top of these middleware platforms. Finally, Chapter 6 concludes this dissertation by summarizing the contributions of this work, discussing open issues to fully accomplish our vision, and presenting future work.

CHAPTER 2

Ubiquitous sensing in pervasive environments

This chapter presents background research for the dissertation. It introduces the vision of pervasive computing and, in particular, the challenges its accomplishment involves. It presents the concept of *ubiquitous sensing* and the role it plays in pervasive computing environments. Then, it describes enabling technologies and state-of-the-art research in ubiquitous sensing, with a special focus on wireless sensor networks and mobile ad hoc networks. Finally, it concludes by describing the Smart Messages computing platform that was used in the implementation of our systems.

2.1 Pervasive computing

In 1991, M. Weiser described his vision of *ubiquitous computing* as the creation of environments saturated with a variety of computing and communication capabilities, which seamlessly integrate with the physical world [Weiser, 1991].

Technological advances, especially in the sector of microelectronics, have been essential to foster ubiquitous computing environments. Computers get smaller, cheaper, and more powerful. Therefore, computation gets ubiquitous. Likewise, networking technology gets smaller, cheaper, and more powerful. Therefore, more and more everyday artifacts are capable of autonomously networking together.

Weiser considers ubiquitous computing the third wave of computing [Weiser and Brown, 1997]. The first era of computing is that of *mainframe computing*, which can be described by the relationship "one computer, many people". The second era of computing is that of *personal computers*, characterized by the relationship "one computer, one user". Ubiquitous computing, the third era of computing, incarnates the relationship "one user, many computers". Embedding computation into the surrounding environments and everyday objects can enable people to exploit available computing capabilities in an unobtrusive manner, so that ubiquitous computing systems ultimately become an invisible technology and interactions with computers become invisible and natural [Weiser, 1994]. In Weiser's terms, the challenge is making technology "calm":

> *"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."* [Weiser, 1991]

Computing utilities of ubiquitous environments include traditional desktop devices, wireless mobile devices, digital assistants, game players, wrist watches, clothing, sensors, RFIDs, cars, consumer electronics (e.g., TV, microwave), etc. For example, an intelligent microwave can download recipes and automatically set the time, adjust the power, and do the roasting by the time the user is back home. Or an intelligent pen can automatically transmit a quote from a newspaper to the office with the swipe of it over the newspaper. In particular, in recent years, we have observed an enormous success of many portable devices such as mobile phones, personal digital assistants (PDAs), and laptop computers. Together with wireless connectivity, Internet, and environmental embedded sensors, they are expected to drive this major technological change.

Many terms and definitions exist to refer to the trend that Weiser initially called ubiquitous computing. L. Kleinrock spoke about *nomadic computing* [Bagrodia et al., 1995]. IBM coined the term *pervasive computing*. The European Commission spoke about *ambient intelligence* [Ducatel et al., 2001]. The Wireless World Research Forum (WWRF) defined the *I-centric communication model* where adaptable, personalized, and ambient-aware services are supported [Arbanowski et al., 2004; Tafazolli, 2004, 2006]. Although

these visions use different names and focus on different aspects, the underlying assumptions remain the same. In this dissertation, we choose to use the term "pervasive computing" to refer to this technological trend and all the challenges it subsumes. Even though the terms pervasive computing and ubiquitous computing are commonly used interchangeably, we share the thought of those who believe that pervasive computing incarnates a more practical approach than ubiquitous computing, less centered on the idea of calmness and invisibility and more focused on the enabling technologies to allow computation and interaction anytime, anywhere.

In the following, we start by summarizing main characteristics and challenges of pervasive environments, and we then briefly review existing hardware, networking and middleware support for pervasive applications.

2.1.1 Characteristics and challenges

M. Satyanarayana characterizes pervasive computing as a deciding step of an evolutionary process started in the mid-1970s [Satyanarayanan, 2001]. A preliminary step of this evolution was the design of distributed systems. *Distributed computing* [Couloris et al., 2001] permitted connecting remote users and resources in a transparent, open, fault-tolerant, scalable, and secure way. In the early 1990s, the advent of laptops and WLANs gave substance to distributed systems with mobile clients. The new field of *mobile computing* therefore emerged with the goal of providing mobile users with access to computing and communication capabilities anytime, anywhere. However, mobility complicated the design of distributed systems due to the unpredictable variability of network connectivity and reliability as well as several resource constraints of mobile devices, such as power consumption, reduced security, and low robustness [Kleinrock, 2003; Satyanarayanan, 1996].

Pervasive computing incorporates characteristics of distributed computing and mobile computing, but also presents several distinguishing characteristics [Bagrodia et al., 1995; Ducatel et al., 2001; Katz, 1994; Satyanarayanan, 2001; Wireless World Research Forum, 2001]. In the following, we list several core characteristics of pervasive environments and identify for each of them the challenges that they arise. Characteristics and challenges are also summarized

in Table 2.1.

Table 2.1: Characteristics and challenges of pervasive computing environments.

Characteristics	**Challenges**
Resource constraints (tiny sensors and mobile devices of small form factor)	Adaptability Resource management Energy-awareness
Mobility	Lossy wireless communication Disconnections Reconfigurability
User-centricity	Context-awareness Proactivity Scalability Calmness and invisibility Privacy
Heterogeneity	Interoperability Service discovery Portability
Spontaneous interactions	Resource naming Security Trust

Resource constraints

A user in a pervasive environment typically owns multiple computing devices, which over time and space need to interact with several other devices. User's devices can sense their environment and collect inputs that guide their actions. Specifically, they probe their surroundings to look for peripherals such as processors and repositories, or input/output devices such as displays, microphones, and video cameras. Every time a resource becomes unavailable or a new one appears, the device needs to adapt accordingly.

Pervasive environments are populated by a variety of devices with different resource constraints. These devices can be classified in three categories [Kehr et al., 1999]:

- *Sensors*: They represent the main input for pervasive applications. Typically, they offer limited computing power and wired-based or wireless, low-bandwidth communication. In recent years, sensor technology has greatly advanced in terms of size, power consumption, processing capabilities, and low cost. This has made possible the integration of sensors in the physical environment, in common everyday artifacts, and in handheld devices. Sensors range from positioning sensors such as GPS devices, to environmental sensors such as Mica motes [Hill and Culler, 2002], to biosensors on users, to Smart-Its artefacts [Smarts-Its, 2007] for augmenting daily objects with a digital presence (e.g., the MediaCup [Beigl et al., 2001] at the University of Kalsruhe).

- *Actuators*: They are complementary to sensors, in the sense that they are capable of receiving commands and data to control certain entities. Examples include ambient displays, tangible interfaces, home and office appliances, etc.

- *Information-processing devices*: Users of pervasive computing environments often act not only as regular information consumers, but also as active providers of information and services (i.e., they are "thin servers" [Weiser and Brown, 1997] capable of providing resources and hosting services for others). Their devices need to have reasonable computational and communication resources to accomplish their tasks autonomously or through interaction with available infrastructures. Examples of this kind of devices include smart phones and PDAs as well as more powerful vehicular computers. Yet, given the computation and communication load, mobile devices of small form factor need to take into account their resource constraints and constantly adapt to their executing environment (e.g., by saving local resources as much as possible and exploiting external resources whenever available).

It is important to emphasize the role played by smart phones among all information-processing devices. Smart phones have been

considered as "the first realistic platform for everyday pervasive computing applications" [Abowd et al., 2005]. These devices are already available to a large part of the global population and integrate several types of ubiquitous connectivity technologies, such as Bluetooth and WiFi. One of the most interesting of its potential uses is as an end point for information services such as navigators, recommenders, cognitive assistance, and health-care monitoring. [Roussos et al., 2005]

However, even though smart phones are becoming increasingly more powerful, in order to not comprise their portability and usability, they will always be constrained in terms of physical dimensions. This leads to constraints in their computing and communication capabilities, battery lifetime, screen and keyboard size. Processor capacity will continuously increase, but also applications will be more and more demanding in terms of processing and communication resources [Want et al., 2002]. A proof of this is the fact that smart phones are becoming more and more multi-functional with the integration of diverse services in a single general-purpose device.

While processing capability has followed Moore's law for the last 30 years, battery energy density is the slowest trend in mobile computing [Paradiso and Starner, 2005]. Although new technologies will improve the situation in the time to come, there will always be an even higher demand for "mobile energy". Mechanisms to control and reduce energy consumption on mobile devices are necessary at all levels of the system [Flinn and Satyanarayanan, 2004], including circuit design, hardware, and operating systems [Olsen and Narayanaswami, 2006]. Middleware and applications should be lightweight as much as possible and also be aware of the available resources (e.g., bandwidth, CPU cycles, battery power) in order to trade off application fidelity and energy [Noble et al., 1997]. Power management techniques need to be applied mostly to processing, storage, and communication components. All these components offer potentials for power optimization, but at the cost of an increased complexity of software interfaces. However, given the heterogeneity of pervasive systems it is essential to provide uniform interfaces that programmers can utilize everywhere. Power management techniques can also involve offloading of computing to servers available in the environment. This approach is called cyber foraging [Balan et al., 2002].

Mobility

User mobility or nomadicity [Kleinrock, 2003] is a fundamental source of dynamism for pervasive systems. As users move, applications have to cope with dynamically changing resource availability. Systems have to learn to exploit computing, storage, and communication opportunities whenever available, and survive when these resources are not available anymore. With portable wireless devices, disconnections and device failures have to be treated as part of normal operation. Disconnections can occur either accidently due to loss of wireless connectivity or voluntarily to save battery or reduce connection costs. In addition, wireless communication can be temporarily degraded due to signal interference and cause packet losses, variable bandwidth, and high error rates.

In order to cope with this dynamism, each time the execution environment changes, the change must be detected and if it is permanent enough to trigger a reconfiguration, then the behaviour of the application must change accordingly. There are several strategies for adaptation in pervasive computing [Satyanarayanan, 2001]. One possibility is to delegate reconfiguration and in particular resource management to the operating system or to underlying middleware layers thus making the application entirely transparent. Alternatively, policy-based approaches [Sloman, 1994; Wies, 1994] allow applications to specify the desired behaviour of the system, which is then translated in a set of reconfiguration rules to be enforced by the system at runtime [Bellavista et al., 2006]. Learning resource usage, predicting resource impoverishment, and anticipating changes of the application's requirements are all important challenges to accomplish efficient reconfigurability.

User-centricity

In pervasive environments, users focus on their activities and not on computers [Weiser, 1994]. The individual user has to be put at the center of the service provisioning process [Arbanowski et al., 2004]. Ideally, the system must determine which actions can help the user by monitoring the user's behaviour, by inferring the user's needs, and by predicting the user's intent.

Interactions in pervasive environments follow a proactive ap-

proach [Lyytinen and Yoo, 2002; Saha and Mukherjee, 2003] aimed to continuously and autonomously make information, resources, and services available while minimizing users' distraction and maximizing users' expectation. This is usually referred to as the *invisibility* or *calmness* property of pervasive environments. Specifically, calmness entails *calm timing* and *calm interaction* [Riekki et al., 2004]. Calm timing means that the application interacts with the user in the situation in which it is needed. Calm interaction means that during the interaction only relevant information is provided and only relevant inputs are required from the user.

Pervasive computing takes place in so-called *smart spaces*. Smart spaces are ordinary indoor or outdoor spaces, such as classrooms or parking areas, equipped with sensors, actuators, communicators, cameras, microphones, speakers, displays, RFID tags, etc. By embedding sensing, actuation, and computation capabilities in buildings and daily artifacts, spaces acquire autonomous sensing and actuation capabilities thus becoming "smart". Therefore, they are capable of perceiving and reacting to their inhabitants' behaviour. In such spaces, objects act as pervasive computing nodes while serving their conventional purposes [Kindberg and Fox, 2002]. For example, a smart coffee cup [Beigl et al., 2001] can integrate sensing, computing, and networking elements to communicate its state (e.g., full or empty). Or a smart meeting room [Abowd, 1999; Cooperstock et al., 1997] can be aware of users' presence and record actions. In addition, chairs, tables, whiteboards, and projectors in the room can also be augmented with smartness.

Compared to classical mobile applications, an essential requirement of pervasive applications is the need to *sense* their environment and *act* upon that. In other words, these systems must perform a continuous monitoring of the environment while consuming almost no power and promptly react when relevant events are detected [Estrin et al., 2002].

Context is the term generally used to define any available, detectable, and relevant piece of information that can be used to characterize the situation of an entity [Dey et al., 2001]. B.N. Schilit identifies three categories of context [Schilit et al., 1994]:

1. *User context*: user profile, location, people nearby, social situation, activity, health conditions, agenda setting, etc.

2. *Execution context*: network traffic, device status, resource availability, communication costs, etc.

3. *Environment context*: weather, light, noise level, temperature, time, etc.

Context provisioning is the process by which context information is acquired, processed, and made available for use. Finally, a system is said to be *context-aware* if it uses context information to provide relevant services to the user, where relevancy depends on the user's current task [Dey and Adowd, 1999].

Depending on the definition of context within a system, context information can be exploited at different levels of a system. At the OS and middleware level, context has a meaning in terms of processor, memory, power consumption, network resources, etc. [Capra et al., 2003]. At the application level, context awareness can highly enhance adaptive applications and attenuate the problem of service overload [Riekki et al., 2003] (i.e., users are overloaded with many services, but do not use them because they can hardly locate them, they do not know about their existence, or do not remember how to use them). At the user interface level, context enables a shift from explicit to implicit human-computer interaction, towards less visible user interfaces [Schmidt, 2000]. Among all possible context parameters, location has been considered the fundamental one for its outstanding richness in triggering system reconfiguration and selecting devices, resources, and information which are relevant to the user.

An additional requirement of smart spaces is that they need to accommodate an increasing number of interacting users, applications, and devices. As the smartness of an environment grows, the number of interacting devices and the frequency of human interactions increase. A pervasive environment needs to scale *(i)* in terms of physical extension and *(ii)* in terms of "computers per cubic centimeters". Specifically, interactions in pervasive environments are predominantly local, thereby they need to be closed after the device has moved away [Satyanarayanan, 2001]. Nevertheless, for wireless mobile users, scalability must take into account energy consumption, bandwidth, and signal interference problems.

Finally, pervasive systems have a significant impact on the social environments where they are deployed [Banavar and Bernstein,

2002]. For example, to control a certain environment, a pervasive system necessitates sensors in such an environment. Sensors embedded in the house can be used to automatically adjust the lighting system or to detect a fire. However, privacy concerns arise when storage, control, and access of such data are uncertain. In addition, the feeling of observability due to the constant presence of monitoring sensors can lead to undesirable psychological feelings. Even sensors that are highly desirable by all participants may become socially unacceptable [Grudin, 2002].

Heterogeneity

Assuming that uniform implementations of smart environments are not achievable, pervasive computing must devise mechanisms to mask this heterogeneity, also called "uneven conditioning" [Satyanarayanan, 2001], and smooth the "smartness gap‘" between different environments. Pervasive environments are heterogeneous both in terms of networking infrastructures and interaction protocols. Network heterogeneity arises from the use of various wireless technologies (e.g., cellular networks, WiFi, Bluetooth), from the adoption of different management models (e.g., infrastructure-based or ad hoc networks), and from the variable support for IP-level communication and configuration functions (e.g., IP multicast, DHCP). This heterogeneity leads to many independent networks being available to users at a location. One consequence of this is that as users can only be connected to a limited number of networks at the same time (often a single one), many services are often not accessible (i.e., not IP reachable). The heterogeneity of interaction protocols arises from the concurrent use of different middleware platforms (e.g., Jini, UPnP, Web Services). Due to incompatible data representation and communication formats, these protocols do not interoperate with each other. For example, users are able to discover only services that are advertised with the service discovery protocol(s) they support [Raverdy et al., 2006].

A core objective of pervasive computing is "serendipitous interoperability", interoperability under "unchoreographed" conditions, meaning that functionally heterogenous devices designed by different manufacturers and for different purposes can come together and interoperate. To understand and use the resources offered

by other devices it is necessary to agree on a common communication language capable of describing characteristics of such devices and their interfaces. The Composite Capability/Preference Profile (CC/PP) [Kiss, 2006] of W3C and the User Agent Profile (UAProf) [WAP Forum, 2001] of OMA's WAP Forum are RDF-based schemes [WRC, 2007] for representing devices characteristics. Moreover, several consensus ontologies have been proposed for pervasive environments. Examples include SOUPA [Chen et al., 2005], FIPA QoS ontology [Foundation for Intelligent Physical Agents, 2002b], and FIPA Device ontology [Foundation for Intelligent Physical Agents, 2002a].

Finally, heterogeneity must be handled at the application level too. Pervasive applications are typically developed for specific device types or platforms. Many separate versions of the same application exist, e.g., for handheld or desktop devices. As heterogeneity increases it becomes difficult to deploy portable applications that can run across all platforms and on new devices as they appear. Application development needs to be as much as possible independent on the device's technology and development platform. Even if an enterprise could generate new applications as fast as new devices appear, programming the application's logic only once would solve many application scalability problems. Likewise, as the number and type of devices increase, explicitly distributing, installing, and updating applications for each device will become unmanageable [Saha and Mukherjee, 2003].

Spontaneous interactions

Mobility and ubiquitous connectivity make pervasive environments very dynamic in the number and type of devices interacting at any point in time and location. Components of a pervasive system acting as clients, services or resources need to spontaneously interact in a volatile environment, where some components can change their properties, appear, or disappear [Kindberg and Fox, 2002]. New naming conventions are needed to uniquely identify resources and services, as IP addresses do not scale well to highly dynamic environments and adequate service discovery mechanisms are necessary.

Furthermore, in such environments, it becomes common for a node to encounter entities never met before. There is a need to

establish trustworthy relationships among them as well as to guarantee security. Moreover, since providing information, services, resources to others implies a cost, incentive mechanisms for cooperation are needed. A possible solution to this is to introduce service charge: every time a node requests a service to a node, it needs to reward it [Buttyán and Hubaux, 2003; Zhong et al., 2003]. However, interactions among devices might last only few seconds or minutes, therefore techniques to support trust, security, and cooperation should not affect performance by introducing delay and intensive computations on resource-constrained mobile devices. Privacy is another critical issue. Personal data such as context information, and in particular location, will need to be shared and it is therefore necessary to control the access to such data.

2.1.2 Hardware and networking support

When in 1991, M. Weiser articulated his vision of ubiquitous computing, this was a vision far ahead of its time. The hardware technology for its support did not exist. After years of hardware progress, the hardware technology needed is becoming a reality. The two most important enablers of such a vision are computing devices that are becoming increasingly more powerful, smaller, and affordable thus allowing a large deployment of them in everyday environments, and wireless networking technologies capable of supporting many different types of interactions occurring in such environments.

Advances in processing capabilities have followed Moore's law stating that "the number of active devices we can place on a given area of silicon doubles every 18 months". Hence, we can increase the number of components into a single chip and obtain compact and multi-functional devices operating at higher speed. In addition, storage capacity has showed a rate of improvement faster than Moore's law, thus becoming a much less critical obstacle for pervasive applications [Want et al., 2002]. However, battery energy density has advanced very slowly [Paradiso and Starner, 2005] and therefore represents a serious constraint for battery-powered devices such as smart phones.

Wireless communication is constantly advancing. With the emergence of WiMax (IEEE 802.16) and other IEEE 802 broadband

technologies (IEEE 802.20, IEEE 802.11n) as well as the forthcoming 3G/4G/next generation cellular networks and Unlicensed Mobile Access (UMA), broadband connectivity has quickly improved. ZigBee (IEEE 802.15.4) and the upcoming IPv6 over Low power Wireless Personal Area Networks (6LoWPAN), to send and receive IPv6 packets over IEEE 802.15.4 based networks, aim to allow low-rate wireless connectivity for devices with very limited form factor.

Wireless connectivity provides flexibility and is essential to support not only basic device-to-device communication, but also more advanced services. Moreover, wireless networking opens the way to two models of network management. The former is an infrastructure-based approach in which devices interact through an infrastructure, which offers several common facilities to multiple applications. The latter is an infrastructure-less approach in which devices spontaneously network together and interact autonomously, without any centralized control. Hence, this second model can provide more flexibility and higher availability. These two network management models are complementary and more or less convenient depending on the amount and type of resources available in the environment.

Other relevant advances in hardware support include sensor technology, high-quality displays, and wearable computers.

2.1.3 Middleware support

Middleware platforms that can abstract the underlying software and hardware complexity of a system and provide core services to application programmers can effectively facilitate the application development process.

Research on middleware for mobile computing seeks to address many issues that arise due to host mobility such as network disconnections, server unreachability, lower and inconstant availability of high-bandwidth networks, and inadequacy of synchronous communication. More discussion on middleware platforms for mobile computing can be found in [Davies et al., 1998; Mascolo et al., 2002; Noble and Satyanarayanan, 1999].

Middleware platforms have also been adopted also in pervasive computing. K. Raatikainen et al. [Raatikainen, 2005; Raatikainen et al., 2002] identify several requirements that middleware platforms for pervasive computing need to address, such as context-awareness,

personalization, adaptability to changes in the execution and communication capabilities, efficient use of communication resources, dynamic system configuration, robustness, high availability, and fault-tolerance. S.S. Yau et al. [Yau et al., 2002] highlight the importance of the following four requirements for pervasive middleware: *(i)* uniform and common development support despite the heterogeneity of operating systems, programming languages, and environments, *(ii)* application-specific context provisioning, *(iii)* context-driven execution, and *(iv)* transparent support for ad hoc communication, proactive resource discovery, and interoperability. Moreover, in contrast to the general tendency of traditional middleware for mobile computing that seeks to provide complete transparency to the application, with pervasive applications it is necessary to balance awareness and transparency. For example, L. Capra et al. [Capra et al., 2003] propose reflection and metadata to allow applications to inspect the system's execution context and adapt the middleware behaviour accordingly.

Many projects on middleware for pervasive computing have focused on resource management and resource discovery issues, and in particular on how to facilitate the on-the -fly integration of a device into a new environment. For example, to support efficient resource discovery in rapidly changing environments, the middleware needs to constantly monitor the type and quality of available resources and services [Bellavista et al., 2003; Chakraborty and Finin, 2006; Liu and Issarny, 2005]. In addition, middleware architectures have been devised to mask the heterogeneity of several resource discovery mechanisms and provide interoperability [Raverdy et al., 2006].

Due to its central role in pervasive computing environments, the support for context-awareness has been integrated in many middleware architectures. The core objective has been how to support acquisition, representation, and use of multiple fragments of context information gathered from multiple sources in a mobile environment. Examples of proposed approaches include the Reconfigurable Context-Sensitive Middleware (RCSM) [Yau and Karim, 2004; Yau et al., 2002], Gaia [Ranganathan and Campbell, 2003; Roman et al., 2002], Solar [Chen et al., 2004], the Sentient Model [Biegel and Cahill, 2004], MiddleWhere [Ranganathan et al., 2004], the Blackboard-based software framework [Korpipää, 2005], the Java Context Awareness Framework (JCAF) [Bardram, 2005],

and the ContextPhone [Raento et al., 2005]. An extensive list of references of works focusing on context-awareness support can be found in [Riva, 2007].

Security and privacy issues for pervasive systems are particularly exacerbated by the application's access to preferences, context information, and personal profiles of users [Campbell et al., 2002]. Confab [Hong and Landay, 2004] is an example of middleware architecture addressing this kind of issues.

A number of leading research institutes have focused on providing support for the deployment and execution of mobile and pervasive computing applications. Examples include Project Aura [Aura, 2007] at Carnegie Mellon University, Cooltown [Cooltwon, 2007] at Hewlett-Packard, EasyLiving [EasyLiving, 2007] at Microsoft Research, Endeavour [Endeavour, 2007] at the University of California at Berkeley (UC Berkeley), Oxygen [Oxygen, 2007] at the Massachusetts Institute of Technology (MIT), Portolano [Portolano, 2007] and OneWorld [Grimm et al., 2004; Grimm and et al, 2001] at the University of Washington.

Finally, software engineering tools to aid application development are extremely important for pervasive computing. J.A. Landay and G. Borriello [Landay and Borriello, 2003] pointed out the importance of design patterns as an effective way to share solutions of pervasive systems' design problems. They also proposed several interaction patterns for pervasive computing and user interfaces. This initial idea was further extended in [Chung et al., 2004], where the authors defined a pattern language of 45 pre-patterns describing application genres, physical-virtual spaces, interaction and systems techniques for managing privacy, and techniques for fluid interactions. The goal was to evaluate the effectiveness of design patterns in assisting designers developing applications for pervasive computing in terms of learning about a new domain, communicating with one another, evaluating existing designs, and generating designs. A complementary study [Riva and di Flora, 2006b] proposed a method to unearth design patterns from existing pervasive computing systems. In particular such a method was applied to identify GOF design patterns [Gamma et al., 1995] as well as novel patterns for the support of context-awareness.

In this dissertation, we focus on middleware platforms and the support that we can offer at such a level to develop and execute

pervasive applications.

2.2 Ubiquitous sensing

As discussed in the previous chapter, one of the key requirement to accomplish the pervasive computing vision lies in the ability of applications to seamlessly access anytime, anywhere data produced by sensors embedded in the surroundings. We refer to this capability with the term "ubiquitous sensing" and we call "sensing applications" those applications that make use of this capability. Ubiquitous sensing enables a large variety of sensing applications ranging from traditional context-aware applications (e.g., tourist guides, personal assistants, or gaming) to vehicular information systems (e.g., dynamic route planning or traffic jam detection), to social networking (e.g, recommenders or community-based interactions). This section discusses existing approaches to support ubiquitous sensing in pervasive environments, and, in particular, work done on wireless sensor networks and mobile ad hoc networks.

2.2.1 Overview

Ubiquitous sensing defines the capability of a system to access anytime, anywhere data produced by sensors carried by entities or embedded in the physical environment. An application that performs ubiquitous sensing is called *sensing application.*

The notion of ubiquitous sensing is conceptually close to that of context and context-awareness, but more focused on the aspects of provisioning and dissemination of sensor information than those of reasoning and classification of raw sensor data into higher-level context information. Moreover, ubiquitous sensing and context-awareness are overlapping fields depending on the definition of context in use. In most of the cases, the notion of context in ubiquitous sensing means low-level sensor data and it refers to entities, resources, and environments that can be close to the user's current location, but also relatively far (e.g., the number of free spots in the restaurant's parking area). In context-aware computing, the notion of context can refer to raw data as well as to semantically-represented information and very often concerns only the user's

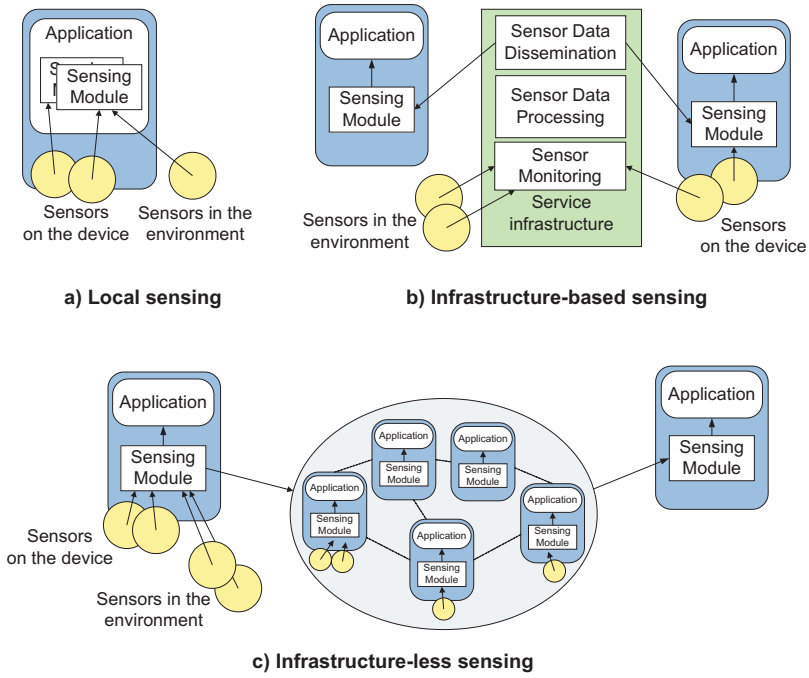

Figure 2.1: The three core approaches to supporting ubiquitous sensing.

close proximity (e.g., how many persons are in the same room where the user is). Hereafter, we will use the term context as understood by ubiquitous sensing.

When speaking about ubiquitous sensing, the term "sensor" can mean different things depending on the subject. If the subject is a sensor network, sensors are tiny sensor devices with wireless connectivity, such as Intel motes [Intel Mote, 2007]. If the subject is a mobile device, sensors can be either *local* such as GPS receivers embedded in a phone and on-board sensors for vehicle parameters (e.g., speed, gear position, throttle position, acceleration, or brake pressure), or *external* and connected to the actual device using, for example, Bluetooth (e.g., BT-enabled GPS receivers or environmental sensors).

To support ubiquitous sensing on a mobile device, we identify the three following strategies, also illustrated in Figure 2.1:

- *Local sensing*: this is the first basic strategy and consists of deploying specialized sensing modules to be installed on the device. As illustrated in Figure 2.1a, different types of sensing modules monitor the local sensors, process the acquired raw data, and make them available to the application. The direct integration of these sensing modules into the application may comport increased complexity, loss of generality and reuse, expensive and time-consuming application development. Therefore, these modules are more conveniently organized in libraries, toolkits [Dey et al., 2001], frameworks [Schmidt et al., 1999], middleware architectures [Yau and Karim, 2004]. However, a limitation of this approach is that, in many situations, it is unrealistic to assume that individual mobile devices will constantly carry any type of conceivable sensor or will be capable of interacting with any type of sensor embedded in the surrounding environment. The computational load on the mobile device might also be excessive for its limited capabilities.

- *Infrastructure-based sensing*: this strategy consists of deploying external service infrastructures, meaning autonomous sensing servers that run on remote devices and are accessible by multiple applications while being independent of each application logic. This type of approach is depicted in Figure 2.1b). The service infrastructure is in charge of discovering suitable sensor devices and processing, storing, and disseminating gathered sensor data. Multiple applications can contact (or subscribe with) the infrastructure to retrieve sensor information related to certain entities or geographical regions. This model has been employed to support sharing of context data among context-aware applications through many different service infrastructures [Bardram, 2005; Hong and Landay, 2001, 2004] and context servers [Hohl et al., 2002; Want et al., 1992] as well as to enable sharing of sensor data in urban environments [Hull et al., 2006; SenseWeb, 2007]. On the one hand, by sharing sensors and computing resources, this approach reduces the computational load on single devices and makes applications less tied to a specific sensor platform. On the other hand, relying on a centralized system presents scalabil-

ity, extensibility, and fault-tolerance issues. In addition, these approaches typically require Internet connectivity.

- *Infrastructure-less sensing*: in this approach, depicted in Figure 2.1c), the basic idea is to use a network of sensing devices to collect and disseminate sensor data available in the environment. Individual sensing devices as well as groups of sensor nodes can sense data, process them, and make them available to other entities. Counting on the cooperation of these entities, these networks can spontaneously form to provide a common service. Alternatively, they can be deployed to accomplish application-specific tasks over long periods of time. The advantages of this approach are distribution, flexibility, lower cost, and the capability to operate also when Internet connectivity is not available. Mobile ad hoc networks and sensor networks are two good examples of infrastructure-less sensing approaches.

While the first two types of sensing strategies are well-known and have been widely employed to support context-awareness in various types of pervasive applications, infrastructure-less sensing is more challenging due to several issues that its deployment involves. In the following sections, we will discuss existing technologies that can be used to achieve infrastructure-less sensing both in static and mobile environments. We start with reviewing characteristics, challenges, and envisaged applications of wireless sensor networks and then pass to mobile ad hoc networks. In the conclusion of this section, we will also summarize the main differences between wireless sensor networks and mobile ad hoc networks. Indeed, while in the literature these two terms are often used to indicate the same type of network, in our work they play two distinguishing roles in supporting ubiquitous sensing. Their roles will be further clarified in the next chapter too.

2.2.2 Wireless sensor networks

Wireless sensor networks (WSNs), or shortly *sensor networks*, are distributed, wireless networks of small, low-cost, tiny sensors, which are capable of collecting and disseminating observations across large and remote physical environments. The nodes in the network are

called *sensor nodes*. Due to their scarce resources, each sensor node is capable of only a limited amount of computation. However, sensor nodes can be coordinated to carry out specific tasks over wide geographical regions.

Sensor network applications

Deploying a network of sensors to monitor an environment is a common practice. For example, cameras in museums, supermarkets, or buildings are installed for surveillance purposes. Real-time images are gathered in a central system that can remotely control the entire environment. However, while a decade ago, most deployed sensor networks involved a limited number of sensors, wired to a central processing unit, nowadays, the focus is on wireless, distributed, sensing nodes.

D. Estrin and co-authors identified several reasons that make *distributed, wireless sensing* [Estrin et al., 2001] necessary. First, distribution is needed when the precise location of a signal of interest is unknown in a certain monitored region. Distribution guarantees higher SNR and improved opportunities for line of sight. Second, in many situations, the environment to be monitored does not have an existing infrastructure for communication or energy, therefore sensor nodes must rely on their local, finite sources of energy and wireless communication capabilities. Third, not only sensors, but also data processing needs to be distributed. Collecting measurements and transferring them to a central processing unit can turn out highly energy-consuming, for example for sensors that need to communicate over long distances.

WSNs enable a variety of applications categorized by D. Culler et al. [Culler et al., 2004] in the following way:

- *Monitoring physical spaces*: this includes environmental monitoring [Cardell-Oliver et al., 2005; Lundquist et al., 2003], habitat monitoring [Cerpa et al., 2001; Mainwaring et al., 2002; Szewczyk et al., 2004], precision agriculture [Burrell et al., 2004; Mayer et al., 2004; Zhang et al., 2004], indoor climate control [Ozdemir et al., 2005], seismic detection [Werner-Allen et al., 2006], surveillance [He et al., 2004], etc;

- *Monitoring things*: this includes structural monitoring [Chin-

talapudi et al., 2006; Xu et al., 2004], condition-based equipment maintenance [Krishnamurthy et al., 2005; Tiwari et al., 2007], medical diagnostics [Baldus et al., 2004], etc; and

- *Monitoring the interaction of things with each other and the encompassing space*: this includes monitoring of activities and wildlife habitats [Juang et al., 2002], disaster management, emergency response [Lorincz et al., 2004], rescue operations [Michahelles et al., 2003], healthcare, etc.

A more comprehensive comparison of several sensor network applications can be found in [Römer and Mattern, 2004].

Sensor network characteristics and challenges

Despite the diversity of sensor network applications, WSNs present one or more of the following characteristics [Estrin et al., 2001]:

- *Untethered for energy and communication*: there is no energy source, therefore energy resources must be carefully managed;
- *Ad hoc deployment*: there is no supporting infrastructure or predefined topology (e.g., sensors can be tossed from an airplane on the sea);
- *Unattended operation*: sensor networks are self-organizing, i.e., configuration occurs without human intervention; and
- *Dynamic operation*: the network must be capable of adapting over time to changing connectivity and environmental conditions.

Given these characteristics, as summarized in several survey papers [Akyildiz et al., 2002; Chong and Kumar, 2003; Culler et al., 2004; Hadim and Mohamed, 2006; Tubaishat and Madria, 2003], the deployment of WSNs needs to address the following challenges:

- *Low energy use*: many sensor network deployments are in remote regions where physical contact for replacement or maintenance may not be possible. The lifetime of a node is determined by its energy consumption and wireless communication is the only way for remote accessibility. Therefore, a sensor

node should accomplish sensing, data processing, and communication tasks while minimizing the energy consumption.

- *Data aggregation*: to save communication resources, intermediate nodes in the network can aggregate the sensed data, summarize the data (e.g., average or max/min) and then transmit the aggregated information.

- *Self-organization and fault-tolerance*: nodes in the network can physically deteriorate or exhaust their energy, or new nodes can appear in the network. As manual configuration of such networks is not feasible, nodes must be able to periodically reconfigure. Fault-tolerance is then defined as the capability to sustain a reasonable level of network functionality without any interruption due to sensor failures.

- *Querying ability*: a sensor network can be abstracted as a distributed database. Data are dynamically sensed and distributed across the nodes. Users need a simple interface to task and query the network. Challenges include query language design, query compilation, task allocation, and caching.

- *Scalability*: sensor networks might contain thousands of nodes. Therefore, scalability is a crucial issue as well as mechanisms to efficiently exploit high density of nodes to save network resources.

- *Security*: the deployment of sensor networks in hostile and harsh environments as well as the utilization of the wireless media increase the probability of malicious intrusions and attacks. Network techniques are needed to ensure survivability and security while consuming few network resources.

Sensor network architecture

A WSN consists of sensors communicating using wireless connectivity. Sensor nodes are battery-powered, wireless computers. As shown in Figure 2.2, they typically consist of four main components [Akyildiz et al., 2002]:

- *Sensing unit*: it consists of a group of sensors and analog-to-digital converters (ADCs). ADC converts the analog signals

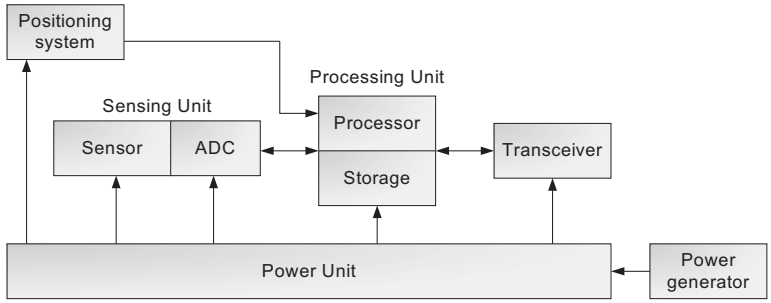

Figure 2.2: Example of sensor node components.

measured by the sensor to digital signals that are passed to the processing unit for further elaboration.

- *Processing unit*: it manages collaboration among sensors to carry out the assigned task. It also includes a storage unit.
- *Transceiver unit*: it connects the node to the network. Typically, three communication schemes are available for sensors: optical communication (laser), infrared, and radio frequency.
- *Power unit*: it consists of a battery that supplies power to the node. Optionally, the power unit can be connected to a *power generator*. Since this is rarely available, modern sensors are able to renew their energy from solar or vibration energy [Hill, 2003]. Additionally, a sensor node can be connected to a *positioning system* as many algorithms used in sensor networks require location-awareness.

Sensor nodes must be small (a few cubic centimeters), light, consume low power (a few tens of milliwatts instead of tens of watts for a common laptop computer), operate in high volumetric densities, and have low production cost.

Sensor network nodes (also called "motes") have been developed in numerous universities and companies. Pioneering work [Hill et al., 2000] in this field has been carried out at UC Berkeley in collaboration with Intel Research Berkeley laboratory. They designed Mica motes [Hill and Culler, 2002] along with TinyOS [Levis et al., 2004; TinyOS, 2007], an operating system specifically designed to

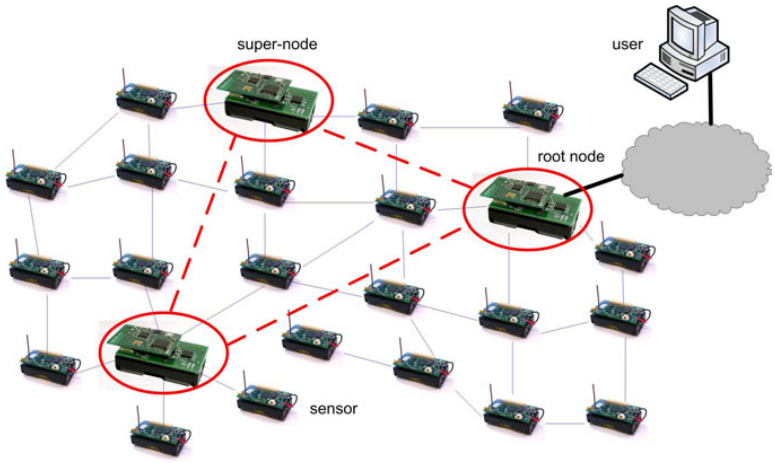

Figure 2.3: Example of sensor network.

run on highly resource-constrained motes. Intel motes [Intel Mote, 2007; Kling, 2003] constitute an enhanced generation of the original mote technology. Other sensor devices are Smart-Its [Smarts-Its, 2007], BTNodes [BTnodes, 2007], Ember [Ember, 2007], and MITes (MIT environmental sensors) [MITes, 2007].

WSNs are typically deployed across large, remote, and unattended geographical areas. Sensors are generally static and are either physically positioned or randomly distributed across the area of interest. As illustrated in Figure 2.3, the regular model of operation for a sensor network involves nodes communicating with each other and exchanging data. Data from individual nodes are typically aggregated in *super-nodes* with greater processing capabilities. Super-nodes can cache, process, filter the data to extract more meaningful information. These super-nodes are then in charge of delivering the collected data to a *root node* that acts as a gateway to an IP-network to which users connect.

In order to reduce energy consumption in the network and extend nodes' lifetime, nodes are turned off when possible. Through this technique, energy dissipation at the MAC-layer protocol can be reduced by employing protocols such as PAMAS [Singh and Raghavendra, 1998] and S-MAC [Ye et al., 2002]. Likewise, topology control protocols such as ASCENT [Cerpa and Estrin, 2004]

and STEM [Schurgers et al., 2002] turn off nodes while guaranteeing full connectivity in the network. In addition, as discussed in the next section, energy can be saved using energy-aware routing protocols.

Sensor data dissemination and collection

To accomplish dissemination and collection of data from the sensor network, flooding and gossiping are two basic and simple mechanisms that do not require any routing algorithm or topology maintenance.

To disseminate a piece of data (e.g., a task or an alarm) with the *flooding* protocol, the root node broadcasts the data to all of its neighbors; upon receiving a piece of data, each node stores and sends a copy of the data to all its neighbors, and so forth thus reaching more distant nodes. The protocol requires no state at any node, and can disseminates data rather quickly in a network where bandwidth is not scarce and links are not loss-prone. However, flooding presents several deficiencies [Heinzelman et al., 1999]: *(i) implosion*, when duplicated packets are sent to the same node; *(ii) overlap*, when two nodes monitoring the same region sense and transmit the same data, at the same time, and to the same node; and *(iii) resource blindness*, if the dissemination protocol does not take into account the available network resources. Dissemination can also be exploited to build network routes. During the dissemination process, each node records its neighbors closer to the root and can then use this information to build a tree for data collection (e.g., to route back data to the root).

A derivation of flooding is *gossiping* [Hedetniemi et al., 1988] in which nodes do not broadcast data, but send the data to a randomly selected neighbor. Through this random selection (rather than broadcasting), gossiping does not incur in the problem of implosion, but can cause data propagation delays.

Numerous routing protocols have been proposed for sensor networks [Akyildiz et al., 2002; Culler et al., 2004] and most of them can be classified as data-centric, hierarchical or location-based together with few distinct ones based on network flow or QoS awareness [Akkaya and Younis, 2005]. In the following, we briefly discuss some of these approaches.

Unlike typical Internet architectures, in sensor networks, communication is not address-based, but *data-centric* because it is very hard to maintain global identifiers in the network. In data-centric routing, each sink sends queries to certain regions and waits for data from the sensors located in the selected regions. Attribute-based naming is used to specify the properties of the requested data. For example, in *Directed Diffusion* [Intanagonwiwat et al., 2000], one of the most important data-centric protocols, nodes are identified by properties such as their location or sensor features. Data generated by the nodes are named by attribute-value pairs and an "interest" (i.e., a list of attribute-value pairs describing the task) is propagated in the network for named data. Data that match this interest are routed towards the source. SPIN [Heinzelman et al., 1999] is the first data-centric protocol that has been proposed. Many other protocols [Braginsky and Estrin, 2002; Chu et al., 2002; Sadagopan et al., 2005] have been then developed following these or similar principles.

Hierarchical protocols aim at grouping the nodes in clusters, where each cluster head performs data aggregation and fusion in order to reduce the number of transmitted packets. LEACH [Heinzelman et al., 2000] is one of the first hierarchical protocol for sensor networks that has been proposed.

Location-based protocols use the location information to propagate queries only across regions of interest and hence save energy. For example, GEAR (Geographical Energy-Aware Routing) [Yu et al., 2001] uses the location and energy information to select its neighbors and balance the energy consumption among them. GAF (Geographical Adaptive Fidelity) [Xu et al., 2001] uses the location information to build a geographical grid such that only one node is turned on in each cell while maintaining an equivalent routing tree in the network.

Query processing in sensor networks

Sensor networks can be abstracted as large distributed databases for the environment where they are deployed [Bonnet et al., 2001]. As we will discuss later in this section, this observation led many research groups to utilize a declarative paradigm for programming sensor networks. TinyDB [Madden et al., 2002, 2003, 2005] and

Cougar [Fung et al., 2002; Yao and Gehrke, 2002] are two well-known query-processing systems for extracting data from a network of sensor devices. Specifically, applications generate queries to input into the network through a simple interface, typically an SQL-like interface, such that the network can then extract and return the data of interest. In the query, the application specifies which data should be collected and how they should be combined and summarized.

According to TinyDB's database abstraction, sensor values belong to a table called `sensors` which, virtually, has one row per each node per instant in time, with one column per each attribute sensed by the sensor device (e.g., temperature, light, etc.). Conceptually, the `sensors` table is an unbounded, continuous data stream of values, where reading out the sensors at a node can be considered as adding a row to `sensors`. Note that, as in classical database systems, the application specifies the type of data is interested in, but does not select the software algorithm that the system must use to collect the results.

Let us consider an example of query in TinyDB taken from [Madden et al., 2005]. A user wants to monitor the occupancy of a room at the 6th floor of the building. She uses microphone sensors attached to motes and looks for rooms where the average volume is over a certain threshold. The corresponding query is the following:

```
SELECT AVG(volume), room
FROM sensors
WHERE floor = 6
GROUP BY room
HAVING AVG(volume) > threshold
SAMPLE PERIOD 30 s
```

This query evaluates the values of all motes on the 6th floor, room by room, and reports all rooms where the average volume is over the specified threshold. Results are refreshed every 30 seconds. Therefore, unlike traditional SQL queries, the result of sensor queries of this type is a stream of sensor tuples.

Once a query is generated, it is optimized. Query optimization means selecting the best execution plan for a certain query given the network's conditions. The cost of a plan is computed based on the estimated costs of each operator, of sensors reading, etc. The query plan might need to be modified at runtime. In addition, multi-query

optimization is employed to avoid to run multiple copies of the same or similar queries.

Upon optimization, the system disseminates the query. The system maintains a routing tree (spanning tree) rooted at the user end point. Every node in the network has its own query processor that processes sensor values and maintains routing information. Starting from the root of the network, each node decides if the query has to be executed locally or broadcasted to its children in the rooting tree. In order to limit the scope of a certain query, a node can maintain a record of attribute values characterizing its children (e.g., node resources, location, etc.). For example, in TinyDB this is called "semantic routing tree" [Madden et al., 2005]. If a node does not offer adequate resources the parent node does not forward the query to it. Otherwise, the parent node agrees with its children on a time interval for listening for results from them. The process repeats for every period and query.

Once the query is disseminated, the nodes process it. An acquisition operator at each node periodically performs the sensor measurement. Subsequently, it routes the sensed values, or tuple, through the query plan. The plan consists of a number of operators to be applied in a predefined order. If the tuple fails to pass a join or selection operator, it is rejected and the query processor enters a power-down phase. Otherwise, the tuple is passed to the next operator or combined with other tuples. Tuples are passed in the routing tree up to the root node which can further combine them with other data collected from other children.

Data aggregation is a powerful mechanism to save resources in the network. In the above query example, the system aggregates microphone readings over time and reports periodically to the user the average value. Besides traditional aggregates such as `MAX, MIN, COUNT, SUM, AVERAGE, MEDIAN`, etc., TinyDB also allows users to define aggregates and Cougar supports advanced aggregates such as the object tracking operator [Gehrke and Madden, 2004]. Broadly speaking, we can say that the most efficient mechanisms to save network resources are those that push computing into the network, towards the origin of the data to be processed.

Mobile sensor networks

Traditional sensor networks usually consist of static nodes. However, static sensor networks are not feasible in every type of environment due to deployment costs over large areas. Several projects have focused on *mobile sensor networks*. Instead of treating mobility as a hostile factor, it is exploited to improve network performance [Kansal et al., 2004]. Examples include Networked Infomechanical Systems (NIMS) [Pon et al., 2005] for characterization of environmental phenomena, underwater sensing [Vasilescu et al., 2005] for monitoring coral reefs and fisheries, ZebraNet [Liu et al., 2004] with sensors positioned on animals to observe animals' movement and habits, and Cartel [Hull et al., 2006] where cars are used as data mules to collect information about road conditions.

2.2.3 Mobile ad hoc networks

Mobile ad hoc networks (MANETs) originated from the Defence Advanced Reserach Project Agency (DARPA) Packet Radio Network (PRNet) in the early 1970s. A MANET consists of a collection of mobile nodes connected by wireless links that can autonomously form network topologies without relying on any centralized access point or infrastructure. MANETs are self-creating, self-organizing, and self-administering networks. In the following, we will present some typical MANET applications and illustrate core characteristics and challenges of these networks, with a special focus on their routing protocols.

MANET applications

In situations in which there is no existing infrastructure or the existing infrastructure is too expensive or inconvenient, ad hoc networking technology can be employed. With the proliferation of mobile devices such as smart phones and advances in wireless communication, ad hoc networking has become an accessible technology in everyday life. Typical envisioned scenarios for ad hoc networks include:

- *Military battlefield*: this was the very first domain where ad hoc networking was considered in 1970s. Soldiers, military

bases, and vehicles can set up private ad hoc networks to exchange information.

- *Emergency services*: MANET technology can provide a flexible method to establish communications for fire/safety/rescue operations or in other situations requiring rapidly-deployable communications with survivable, efficient dynamic networking [Corson and Macker, 1999]. Ad hoc networking can be essential because the preexisting infrastructure might be damaged or destroyed.

- *Vehicular networks*: there is an emerging interest in equipping vehicles with portable computers to provide services such as dynamic route planning, route recommendations, traffic jam detection, collision avoidance, road diagnostics, or content sharing.

- *Social networking*: people are interested in sharing information such as news, restaurant recommendations, driving directions, experiences or simply in getting to know each other. Ad hoc networks can be used to exchange information in convention centers, conference venues, smart classrooms, offices, shopping malls, and many other civilian forums.

- *Personal Area Network (PAN)*: ad hoc networks can facilitate the exchange of data among personal devices, such as PDAs, laptops, smart phones. In particular, ad hoc networks can serve to extend the access to the Internet or other networks.

MANET characteristics and challenges

MANETs present unique characteristics and challenges [Chlamtac et al., 2003; Corson and Macker, 1999; Sun, 2001]. Core characteristics of MANETs are the following:

- *Autonomous and resource-constrained terminals*: in MANETs, each mobile terminal is an autonomous node that can act as both a host and a router. These nodes are generally mobile, battery-powered and offer limited computational capabilities and small storage space.

- *Distributed network management*: there is no central authority, hence the control and management of the network, such as routing and security, rely on the cooperation of all nodes.

- *Bandwidth-constrained, variable-capacity links*: wireless links have significantly lower bandwidth than wired links and their capacity constantly varies due to noise, fading, and interference. In addition, nodes in MANETs are mobile and constantly change their position within the network, thus rendering symmetric links hard to maintain. As links come and go depending on the transmission characteristics, one transmission can interfere with another one and nodes can overhear transmissions of other nodes and corrupt the entire transmission. As such, a MANET presents low bandwidth links, high bit error rates, and unstable and asymmetric links. This is in contrast to wired networks characterized by high bandwidth links, low bit error rates and stable and symmetric links.

- *Multi-hop communication*: as nodes have limited transmission ranges, some nodes cannot communicate directly with each other, but need intermediate nodes that can forward packets on their behalf.

- *Dynamic network topology*: as nodes are mobile and links are prone to failures, the network topology can change quickly and in an unpredictable manner, thus varying the connectivity among the terminals and causing network partitions.

Core challenges for MANETs are the following:

- *Routing*: since the network is dynamic, it is hard to maintain stable connectivity between pairs of nodes, especially if these are multi-hop away. Multicast routing is also challenging because the multicast tree is no longer static and hard to maintain.

- *Energy-aware operation*: as most devices in ad hoc networks are battery-operated, conservation of energy and energy-aware routing mechanisms are important design challenges in order to extend battery lifetime.

- *Quality of service*: supporting quality of service in such dynamic conditions is hard and needs to constantly adapt to the network dynamics.

- *Security, reliability, and availability*: mobile wireless networks are more prone to physical security threats of eavesdropping, interception, denial-of-service, and routing attacks compared to fixed wired networks [Corson and Macker, 1999]. Reliability is affected by the limited transmission range, mobility, node failures, and data transmission errors. Security and fault-tolerance mechanisms need to be applied to handle these problems. Key management, authentication, and authorization mechanisms need to be implemented in a distributed manner, without relying on any trusted third-party authority. On the other hand, the decentralized nature of MANETs adds robustness against single points of failures of centralized networks.

- *Location-awareness*: the node's location is an essential information in MANETs. In general, a node is interesting for a certain task depending on the resources that it can offer at certain time and location. Location represents the physical structure of the network and it is required not only by the application for expressiveness purposes (e.g., location-based queries), but also by the lower system layers to route data and distribute computation in the network [Ni, 2006].

- *Scalability*: in some scenarios (e.g, large battlefield deployments, urban vehicular systems), MANETs can grow to incorporate several thousands of nodes. Hierarchical structures are usually employed to handle scalability, but due to mobility and lack of fixed references, pure ad hoc networks do not easily tolerate hierarchical structures [Gerla, 2005].

Routing protocols

Due to the central role played by routing protocols for MANETs' operation, in the last decade, numerous routing protocols have been investigated. Moreover, a working group for MANETs [manet IETF WG, 2007] within the Internet Engineering Task Force (IETF) has

been formed with the goal to standardize IP routing protocol functionality suitable for MANETs.

Routing protocols traditionally used in wired networks, such as Distance Vector routing [Hedrick, 1988] and Link State routing [McQuillan et al., 1978], cannot be used in mobile ad hoc networks due to the high control overhead they would require to work. In wired networks, these algorithms perform well because of the predictable network properties, such as static link quality and network topology. In dynamic networks, these assumptions are not valid, and these algorithms would also cause route loops and routing inconsistency [Liu and Kaiser, 2005].

Routing protocols in MANETs are usually classified based on the network structure and on the routing strategy (i.e., how routing information is acquired and maintained by mobile nodes). Based on the network structure, they are classified as uniform and non-uniform routing protocols. Uniform routing protocols adopt a flat addressing scheme. Each node participating in routing plays an equal role. Flat routing protocols can be further classified based on their routing strategy in proactive (or table-driven) and reactive (or source-initiated) routing protocols [Hong et al., 2002; Liu and Kaiser, 2005; Royer and Toh, 1999]. In non-uniform routing protocols, each node participating in the routing plays distinct management and/or routing functions. Two additional categories of routing protocols that we consider are geographical location assisted routing and energy-efficient routing protocols. These two categories partially overlap with the previous ones, but present a clear focus respectively on the use of location information and on energy-saving strategies. In the following, we review some examples of routing protocols for each category.

Proactive routing protocols

They attempt to maintain a consistent, up-to-date view of the network topology. Each node in the network needs to maintain a routing table where storing the routing information. Every time a change in the network topology occurs, a corresponding update is propagated throughout the network. Regardless of whether there is a data traffic in the network, the network state is proactively updated, thus possibly generating high overhead.

A well-known example of proactive routing protocol is the Desti-

nation Distance Vector (DSDV) [Perkins and Bhagwat, 1994]. This is based on the classical Bellman-Ford routing algorithm [Ford and Fulkerson, 1962]. Each mobile node maintains a routing table listing all available destinations within the network and the number of hops to reach each destination. Each route table entry is identified with a sequence number assigned by the destination. The sequence number permits distinguishing state routes from new ones thus avoiding routing loops. To maintain consistency of the routing tables, each node periodically transmits updates to its neighbors and transmits updates every time significant new routing information becomes available. Therefore, the update is both time-driven and event-driven. To limit the potentially large traffic of updates, routing updates can be sent as "full dump" or incremental update. The former carries all the available routing information, whereas the latter carries only information about those routing entries that have changed since the last full dump. When nodes do not move, incremental updates are transmitted to reduce the network traffic. When nodes' movements increase, full dumps are more frequent. Every routing packet contains the unique sequence number assigned by the transmitter. Any route with a more recent sequence number is used. Routes with older sequence numbers are discarded. A route with a sequence number equal to an existing route is selected if it can provide better performance (i.e., shorter route).

Other examples of proactive routing protocols include the Wireless Routing Protocol (WRP) [Murthy and Garcia-Luna-Aceves, 1996] and the Fisheye State Routing (FSR) [Iwata et al., 1999].

Reactive routing protocols

They create routes only when requested by the source node. When a node requires a route to a destination, it initiates a route discovery process across the network. This process terminates once a route is found or no route is available. Upon a route has been established, this needs to be maintained by a route maintenance mechanism until either the destination becomes unreachable or the route is no longer needed by the source. Compared to proactive protocols, reactive protocols require less control overhead, thus ensuring better scalability. On the other hand, reactive protocols might induce long delays for route discovery before packets can be forwarded. The Ad hoc On-demand Distance Vector Routing

(AODV) [Perkins and Royer, 1999] and the Dynamic Source Routing Protocol (DSR) [Johnson and Maltz, 1996] are two of the most famous reactive protocols for MANETs.

AODV builds on the DSDV protocol previously described, but unlike DSDV, it minimizes the amount of broadcasts by discovering routes on-demand, instead of maintaining the list of all available routes. To find a path to a certain destination, the source initiates a route discovery process. It broadcasts route request (RREQ) packets to its neighbors, which in turn forward the request to their neighbors, and so forth, until the destination, or an intermediate node with recent routing information to the destination, is found. Neighbor nodes receiving the RREQ either send back to the source a route reply (RREP) or re-broadcast the RREQ with an increased hop count (stored in the RREQ). Each node maintains a cache to keep track of the received RREQs, and when it forwards an RREQ to its neighbors, it also stores in its tables the node from which the first copy of the RREQ came. This information is used to build the reverse path to route back the reply. As the RREP is routed back, nodes along the reverse path set up forward route entries in their routing tables which point to the node from which the RREP came. AODV uses only symmetric links, meaning that reply packet follows the reverse path of the RREQ packet. Route maintenance in AODV is such that if the source moves then it can re-initiate route discovery to the destination. If a node along the route moves then its upstream neighbor detects the move and propagates a link failure notification to all upstream neighbors and so forth till it reaches the source. Finally, to maintain information about the local connectivity of a node, AODV uses "hello" messages that are periodically broadcasted by a node to its neighborhood.

DSR is a source-routed routing protocol. Each mobile node maintains route caches containing the source routes of which it is aware. The node updates entries in the route cache as new routes are learned. If the source wants to send a packet to a certain destination, it first consults its route cache to determine if a valid route is present. If no unexpired route is found, it initiates the route discovery process by broadcasting a route request packet. Upon receiving a route request packet, a node checks its route cache. If the node does not have the required routing information, it appends its own address to the route record field of the route request packet

and forwards the request packet to its neighbors. If the route request packet reaches the destination or an intermediate node that has routing information to the destination, a route reply packet is generated. To route back the reply back there are three possibilities to get a backward route: *(i)* the node already has a route to the source, *(ii)* the network has symmetric links such that the reply packet can be routed back using the information stored in the route record of the request, or *(iii)* the node initiates a route discovery process. Route maintenance is accomplished through the use of route error packets and acknowledgements. When the data link layer detects a link disconnection, a route error packet is sent backward to the source so that the source initiates a new route discovery and all routes containing the broken link are removed from the cache.

Although similar, AODV and DSR present some important differences [Liu and Kaiser, 2005; Royer and Toh, 1999]. The overhead of DSR is potentially larger than that of AODV since each DSR packet carries the routing information, whereas AODV stores the next hop routing information in the routing tables at nodes. Likewise, the route replies in DSR are larger because they contain the address of every node along the route, whereas in AODV route replies only carry the destination addresses and the sequence number. DSR assumes small diameter networks, whereas AODV can scale much better. On the other hand, AODV requires symmetric links, while DSR can also utilize asymmetric links if symmetric ones are not available. DSR suits better networks in which nodes move at moderate speed with respect to the packet transmission latency. DSR does not make use of periodic routing advertisements and can thus save bandwidth and reduce power consumption. In particular, when there are no changes in the network topology, DSR does not consume any resource. Moreover, by caching multiple routes to a destination, when a link breaks, the source can check for another valid route in its cache and route recovery is very fast. Performance of DSR and AODV are compared through simulations in [Broch et al., 1998; Das et al., 2000].

Other examples of reactive protocols include Temporally Ordered Routing Algorithm (TORA) [Park and Corson, 1997], Associativity Based Routing (ABR) [Toh, 1996], and Signal Stability Routing (SSR) [Dube et al., 1997].

Non-uniform routing protocols

They seek to overcome the limitations of uniform routing protocols. The problem with flat approaches is that they do not scale well and they do not allow for route aggregation of updates. With a non-uniform approach better scalability can be guaranteed. Non-uniform routing protocols can perform zone based hierarchical routing, cluster based hierarchical routing, and core-node based routing [Liu and Kaiser, 2005].

In zone based hierarchical routing, different zones are created and each one employs different algorithms to accomplish routing in the network. Nodes in one zone know how to reach each other with a smaller cost compared to that of maintaining routing information for the whole network. For example, in the Zone Routing Protocol (ZRP) [Haas and Pearlman, 2001], the network is divided into routing zones based on the distances between mobile nodes. For nodes within the same zone a proactive mechanism is used to maintain routing information because it is more efficient in small-scale networks. For inter-zone communication a reactive mechanism is employed, thus ensuring better scalability.

Cluster based hierarchical routing is based on the same principle. Nodes geographically close to each other are grouped into clusters. Each cluster has a cluster head that acts as a gateway to other clusters. Within the cluster, a proactive routing protocol is used and for inter-cluster communication a reactive approach is used. The organization into clusters can be multi-level meaning that clusterheads of low-level clusters organize themselves into upper level clusters and so forth. For example, in the Hierarchical State Routing (HSR) [Iwata et al., 1999], a hierarchical address is assigned to each node and this reflects the network topology as a basis to route packets in the network.

In core-node based routing, some nodes in the network are selected to carry our special functions and compose a sort of backbone for the entire network.

Geographical location assisted routing protocols

These protocols employ location information to improve routing performance. They assume that nodes can determine their position using GPS or other positioning techniques [Hightower and Borriello, 2001] and can learn the position of their direct neighbors through

one-hop broadcasts (i.e., all nodes periodically send beacons containing the position of the sending node). The source specifies the location of the destination in the packet and the packet is forwarded hop by hop based on the location information of its neighbors and of the destination. Compared to topology-based routing, geographical routing mechanisms do not require the establishment or maintenance of routes. Moreover, location-based routing provides the delivery of packets to all nodes in a certain geographical region in a natural way. This is usually referred as geocasting [Navas and Imielinski, 1997].

M. Mauve et al. distinguish three main packet-forwarding strategies for geographical routing [Mauve et al., 2001]: greedy-forwarding, restricted directional flooding, and hierarchical approaches.

In greedy-forwarding, when an intermediate node receives a packet, it forwards it to a neighbor lying in the direction of the destination specified in the packet itself. The process is repeated until the destination is reached. To select to which neighbor a given packet should be forwarded, a natural approach is to forward the packet to the node that makes the most progress towards the destination in order to minimize the number of hops to traverse. This strategy is called the "most forward within r" (MFR) [Takagi and Kleinrock, 1984]. Other possible approaches are the "nearest with forward progress" (NFP) [Hou and Li, 1986], where the packet is forwarded to the nearest neighbor of the sender which is closer to the destination, compass routing [Kranakis et al., 1999], which attempts to minimize the spatial distance to the destination by selecting the neighbor closest to the straight line between sender and destination, or random selection of a node closer to the destination than the forwarding node [Nelson and Kleinrock, 1984]. In some cases, greedy routing can fail to find a path to the destination and reach a dead end (i.e., a node whose neighbors are all farther away from the destination than itself). Recovery strategies can then be employed such as in the Greedy Perimeter Stateless Routing (GPSR) [Karp and Kung, 2000]. Its recovery strategy is based on planar-graph traversal. A planar sub-graph (i.e., a sub-graph with no intersecting edges) is built and based on this a planar-graph traversal is used to find a path towards the destination.

In restricted directional flooding, a node forwards the packet not only to one (as in greedy forwarding) but to multiple direct neigh-

bors. For example, in the Distance Routing Effect Algorithm for Mobility (DREAM) [Basagni et al., 1998], the source of a packet forwards the packet to all one-hop neighbors that lie in the direction of the destination. The expected region containing the destination is computed as a circle around the position of the destination. To limit the flooding of data packets to a small region in the network, DREAM requires that each node regularly floods packets to update the location information maintained by the other nodes. Another protocol that belongs to this category is the Location-Aided Routing (LAR) [Ko and Vaidya, 1998] protocol. LAR is a reactive unicast routing mechanism that uses the location information to limit the scope of the route request flooding when performing route discovery. During route discovery, the route request is flooded across a request zone that contains the zone where the destination is expected to be located and the location of the source node.

Position-based, hierarchical strategies attempt to establish some form of hierarchy to scale to a large number of nodes while still exploiting the location information. In Terminodes routing [Blazevic et al., 2005], packets are routed according to a proactive distance vector algorithm if the destination is close to the source node. Otherwise, for long-distance routing, a greedy position-based approach is used and once the packet has reached the area close to the destination, it continues to be forwarded using a local routing protocols. Grid routing [De Couto and Morris, 2001] is very similar to Terminodes routing, with the addition that the hierarchy scheme is exploited not only to improve scalability, but also to enable nodes that are not aware of their location to participate in the routing process. Basically, in each area it is necessary to have at least one location-aware node that can be used as proxy for location-unaware nodes (i.e., the location-unaware node uses the location of the location-aware node as its own location). Packets directed to a location-unaware node arrive at a location-aware proxy and are then forwarded to the other interested nodes.

Energy-efficient routing protocols

They seek to extend the network lifetime by minimizing the energy consumed by the nodes to support routing [Yu et al., 2003]. Transmission power control and load distribution are two strategies aimed to minimize the communication energy during active periods,

while sleep/power-down mode can be used to minimize energy consumption during idle periods.

In transmission power control protocols [Chang and Tassiulas, 2000; Stojmenovic and Lin, 2001], the transmission power is adjusted in such a way to maintain a connected topology while consuming the minimal power. These protocols need to know the transmission energy over the wireless link and the remaining battery lifetime of the node.

In load distribution approaches, the goal is to balance the energy consumption among all mobile nodes by selecting routes with under-utilized or powerful nodes rather than shortest and overloaded routes. In this way, the protocol can guarantee longer network lifetime and a more uniform utilization of the nodes in the network. An example of this type is the Localized Energy-Aware Routing (LEAR) [Woo et al., 2001] protocol. This is DSR-like protocol that performs route discovery in an energy-aware manner. Any intermediate node decides to forward the route request packet depending on its residual battery power. This ensures that the destination node receives a route request packet only when all intermediate nodes in the route own sufficient battery power to perform the routing.

Finally, the sleep/power-down mode approach focuses on turning the radio subsystems at nodes into sleep mode or off during a period of inactivity. In most of the existing proposals [Chen et al., 2001; Xu et al., 2001], a master node is elected in the network to coordinate the communication while all its neighboring slaves sleep. Each slave node periodically wakes up and communicates with the master node to find out whether it has data to receive. A core challenge of this kind of architecture is how to select and maintain the master-slave architecture in dynamic operating conditions.

To conclude this overview on routing protocols for MANETs, we observe that a serious problem in ongoing ad hoc networking research is that, in many cases, only network simulations are used to prove the feasibility of a certain solution. Unfortunately, simulations can only be a first step in evaluating protocols and algorithms for MANETs [Kiess and Mauve, 2007]. This is because simulations require assumptions about the real world that might not be true. In addition, characteristics of MANETs such as energy consumption,

mobility, and transmission range are hard to model in a realistic enough manner in simulators.

2.2.4 Differences between wireless sensor networks and mobile ad hoc networks

MANETs and WSNs present many similarities but also fundamental differences. Both are distributed, self-configuring, self-managing wireless networks, involve multi-hop routing, and consist of battery-powered mobile devices. These characteristics have a different impact and different emphasis in each type of network.

Except for few cases, sensor devices in sensor networks do not move after deployment, while phones or cars in ad hoc networks constantly move. On the other hand, nodes of remote sensor networks, like those deployed on active volcanos, in the middle of the ocean, or in a forest, can be damaged and fail, thus requiring a change of the network topology. In addition, topology changes in sensor networks can be often needed to accomplish energy savings.

Sensors of large and remote WSN deployments cannot be easily recharged. Sensor nodes can be left unattended for long periods of time (e.g., months or even years), their range of communication is within few meters and at low rates, few kilobytes of memory are available, and processor speeds can be very limited. On the contrary, devices in ad hoc networks are generally more powerful in terms of computation, communication, and storage resources [Ni, 2006].

In addition, MANETs are usually "close" to humans, in the sense that nodes in the network are devices used by human beings, while in sensor networks the focus is not on human interactions, but on the physical environment [Macias and y, 2006]. Sensor networks are usually application-specific, while MANETs are general-purpose networks.

For these and other reasons, even though WSNs can be seen as a special case of MANETs, the protocols, algorithms and design choices of one type of network cannot be always directly applied to the other type.

2.3 Programming ubiquitous sensing applications

Middleware architecture are usually employed to bridge the gap between applications and operating system and provide useful abstractions to the application developer. Y. Ni discusses in his PhD Thesis [Ni, 2006] fundamental reasons why traditional programming models for distributed systems, such as the remote procedure call (RPC) or the remote invocation method (RMI), cannot work effectively in MANETs. Even though also MANETs consist of many separate processing elements that need to communicate, they are highly dynamic networks where nodes constantly join and leave the network. There is no predefined topology. IP-addresses as well as resources offered by the nodes are not known a priori and resources in the network need to be classified also based on their location. In addition, unlike reliable message passing systems, these networks consist of lossy, unstable links. Therefore, if the server is not reachable and no reliability can be guaranteed traditional client-server models cannot work. Finally, instead of powerful clusters of servers, MANETs consist of battery-powered devices that need to optimize the execution of their programs to save energy resources.

For similar reasons, middleware architectures designed for traditional distributed systems can hardly be applied also to sensor networks mostly due to energy restrictions, network dynamics, and data aggregation requirements of sensor networks [Heinzelman et al., 2004].

Most programming platforms designed for pervasive computing (such as those presented in Section 2.1.3) cannot be used directly in MANETs and WSNs either, because they lack the flexibility to work in highly volatile, data-centric environments without powerful servers and Internet connectivity.

Therefore, to address the specific requirements of MANETs and WSNs, new programming models and platforms have been devised. In the following, we review existing programming support and programming abstractions specifically designed to address the characteristics of MANETs and WSNs. Programming support focuses on providing systems, services, and runtime mechanisms. Programming abstractions focus on providing useful concepts and abstrac-

tions of network nodes and sensor data [Hadim and Mohamed, 2006].

2.3.1 Programming support

We distinguish approaches based on database abstraction, virtual machines, modular programming, and event-based and message-oriented middleware.

Database approaches have been largely applied to sensor networks. Their core idea is to abstract the network as a distributed database where applications can issue SQL-like queries to have the network perform a certain sensing task. TinyDB and Cougar, previously described, are two main representatives of this class. Other approaches of this type are SINA (System Information Networking Architecture) [Shen et al., 2001] and DsWare (Data Service Middleware) [Li et al., 2004].

Virtual machine (VM) approaches let application programmers write their applications in single modules. The system distributes the modules through the network and these modules are then interpreted and scheduled for execution by the VM at the nodes. The advantage of this method is that modules can be distributed across the network using different strategies, such as reduction of energy consumption or load balancing.

An example of virtual machine approach is Maté [Levis and Culler, 2002]. Maté is a bytecode interpreter running over TinyOS [Levis et al., 2004], an operating system specifically designed to run on highly resource-constrained motes. It consists of a single TinyOS component running on top of several system components, including sensors, network stack, and nonvolatile storage. Maté's programs are broken into capsules up to 24 instructions, which is the limit to fit into a single TinyOS packet. Each code capsule includes the type and version information to disambiguate between different network updates. Maté's core components are the VM, the network, the logger, the hardware, and the scheduler. It uses a synchronous model in which execution begins in response to an event (a timeout, a packet being received or being sent) and avoids message buffering. Maté particularly suits applications requiring frequent reprogramming in large-scale networks. On-line software upgrading is enabled by capsules that can forward themselves through the network.

MagnetOS [Barr et al., 2002] also belongs to this category. MagnetOS is a distributed, adaptive, power-aware operating system for ad hoc and sensor networks. MagnetOS provides a single system image of a unified Java virtual machine across nodes in the network. Applications are partitioned into components that are dynamically placed on nodes in the network. MagnetOS employs two online power-aware algorithms, called NetPull and NetCenter, to reduce application energy consumption and maximize system longevity. These algorithms can automatically move communicating objects onto nodes that are topologically closer in order to shorten the mean path length of data packets (e.g., by moving objects closer to data sources). NetPull operates at the physical link level, whereas NetCenter operates at the network level.

Other virtual machine approaches are SensorWare [Boulis et al., 2003], Smart Messages [Kang et al., 2004], and Agilla [Fok et al., 2005] (based on Maté). We used the Smart Messages computing platform for our system development. This platform will be described in detail in the next section.

Modular approaches seek to develop applications modular enough to be easily distributed through the network using mobile code and mobile agents. Transferring small modules is more efficient than transmitting the entire application. For instance, Impala [Liu and Martonosi, 2003], implemented on ZebraNet [Juang et al., 2002] hardware nodes, is an asynchronous, event-based middleware architecture capable of supporting multiple applications in WSNs. Impala performs dynamic application adaptation based on device failures to improve fault-tolerance, energy consumption, and performance. In addition, it allows software updates to be received and applied to the running system on-the-fly. Mobile Gaia [Shankar et al., 2005] adopts a component-based approach to program ad hoc environments. It views a MANET as a collection of clusters (or active spaces) where each cluster has a cluster head and several client devices. Available services are classified in coordinator services, when they are provided by the cluster coordinator, and client services, when they are provided by the client device. The middleware supports formation of device clusters, resource sharing among devices, and service interactions. Applications can be decomposed into smaller components that can run on different devices inside the cluster. The communication model is event-based.

The last category that we consider is that of event-based and message-oriented middleware. A publish-subscribe paradigm can facilitate message exchange between nodes and sink nodes and particularly fits MANET's monitoring applications. STEAM [Meier and Cahill, 2002] exploits a proximity-based group communication service to enable interaction among entities. Event filers are used to control the propagation of events. STEAM supports three types of filters, which are subject, proximity, and content filters. Mires [Souto et al., 2005] aims to adapt traditional message-oriented middleware for fixed distributed systems to WSNs. It is built on TinyOS using NesC. EMMA [Musolesi et al., 2005] seeks to adapt the Java Message Service (JMS) to MANETs by modifying the message passing used in JMS and adding an epidemic routing mechanism.

Other types of programming support include tuple space based middleware such as LIME [Murphy et al., 2001], peer-to-peer based middleware such as Expeerience [Bisignano et al., 2003], and data sharing based middleware such as XMIDDLE [Mascolo et al., 2002]. More references on this subject can be found in [Hadim and Mohamed, 2006; Salem Hadim and Jameela Al-Jaroodi and Nader Mohamed, 2006].

2.3.2 Programming abstractions

In their survey on middleware for WSNs [Hadim and Mohamed, 2006], S. Hadim and N. Mohamed distinguish between those programming abstractions that focus on the global behavior and hide the network details from programmers, and those that focus on localized behaviour and expose some network details to programmers.

The first category, also called macroprogramming, proposes to program the network as a whole, at high-level so that the programmer can focus on the higher-level algorithms of an application. Then the compiler automatically generates the node-level programs that implement the application logic on the network. An example of macroprogramming is Kairos [Gummadi et al., 2005]. This views the sensor network as a group of nodes that can be instructed together in a single program. Kairos extends C with functions to manipulate named nodes and lists of nodes in the network and to provide remote data access. Low-level behaviours such ad inter-

node communication and node-level resource management remain transparent to the programmer. The Kairos compiler translates Kairos programs into node-level nesC programs that can be directly linked with standard TinyOS components and the Kairos runtime system, and then run over a sensor network. Other examples of macroprogramming are declarative approaches [Gehrke and Madden, 2004] such as TinyDB [Madden et al., 2005] and Cougar [Yao and Gehrke, 2003].

The second category of programming abstractions allows programmers to focus on single nodes in the network. Moreover, the interest is often on a specific location rather than an individual sensor measurement. For example, Hood [Whitehouse et al., 2004] allows a node to identify a set of nodes around itself (i.e., neighborhood) based on a certain membership criteria and then decide which attributes are to be shared among the identified neighbors. Similar to Hood, Abstract Regions [Welsh and Mainland, 2004] is motivated by the need for "spatial processing" in several types of sensor network applications. Spatial operations are performed by sharing data and coordinating tasks among nodes in a certain neighborhood. An abstract region specifies a neighborhood relationship between a certain node and other nodes in the network. For example, the set of nodes positioned within a certain distance, or the set of nodes offering certain properties. Each node can define several abstract regions. EnviroTrack [Abdelzaher et al., 2004] proposes a new address space for sensor networks that is particularly useful for applications that need to monitor environmental events. Events in the environments become addressable entities and their state can be accessed as local variables. In addition, computation and actuation can be attached at these logical network addresses as in any IP-based network. For example, a camera could be turned on to track a certain event inside the network and instructed to deliver images to the user; when the event moves away from the camera's range, the middleware will turn on an appropriate nearby camera so that a continuous stream of images is provided. State-centric programming [Liu et al., 2003], TinyGALS [Cheong et al., 2003], and Semantic Streams [Whitehouse et al., 2006] also belong to this category.

A last proposal that we discuss is Spatial Programming [Borcea et al., 2004]. Spatial Programming is an imperative programming

model that views an ad hoc network as a single virtual name space, used by applications to access individual resources at nodes. Compared to declarative programming, imperative programming can be more appropriate for complex tasks that go beyond data collection, especially tasks whereby algorithmic details cannot be left to a common middleware. An application written under the Spatial Programming model is a sequential program that transparently reads and writes virtual names mapped to network resources as if they were local variables. In this way, Spatial Programming shields the distributed and volatile nature of mobile ad hoc networks from application programmers. Although Spatial Programming can be classified among the macroprogramming models, Spatial Programming applications can often exhibit localized behavior. Spatial Programming was implemented on top of the Smart Messages [Kang et al., 2004] platform (see next section).

2.4 Smart Messages platform

Our middleware architectures have been implemented using the Smart Messages (SM) computing platform [Borcea et al., 2002; Kang et al., 2004]. Smart Messages is a distributed computing platform for cooperative computing in highly volatile mobile ad hoc networks. In the following, we briefly describe the Smart Messages platform and an example of operation.

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 calls explicitly for execution migration. Each node participating to the SM execution provides:

- a *virtual machine* (VM) for execution over heterogeneous platforms;
- a shared memory addressable by names, namely the *tag space*, for inter-SM communication, synchronization, and interaction with the host;
- an *admission manager* that prevents excessive use of resources by incoming SMs; and

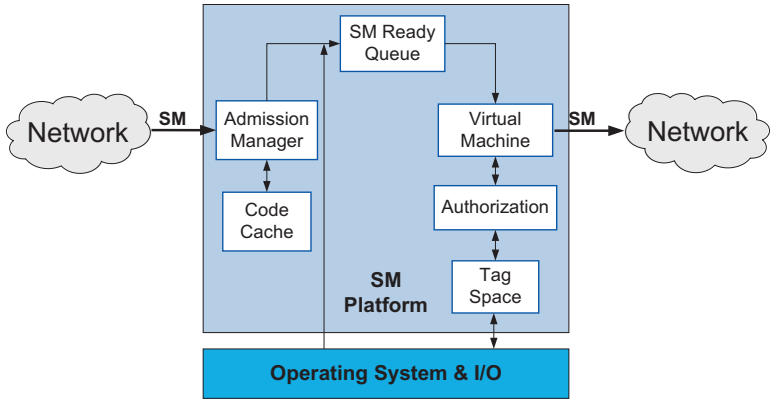

Figure 2.4: The Smart Messages architecture.

- a *code cache* for storing frequently executed code.

The SM architecture is depicted in Figure 2.4. An admitted SM at a node generates a task which is scheduled for execution. During execution, an SM can interact with the host or other SMs through the tag space, which is local to each node. Essentially, the tag space consists of *(name, data)* pairs, called tags, which are created by SMs and used for data exchange and synchronization among SMs. Special I/O tags are predefined at nodes and used as an interface to the OS and I/O system (e.g., battery lifetime, available memory, location sensors). Tags serve also to name the destination of the 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 multi-hop migration is implemented using a low-level primitive for one-hop migration, namely *sys_migrate*, and routing tables built in the tag space [Borcea et al., 2003]. The *sys_migrate* primitive captures the execution context of the SM (data explicitly identified by the programmers and control state), packs it together 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 Figure 2.5. This example assumes an ad hoc network of intelligent cameras that can be programmed to per-

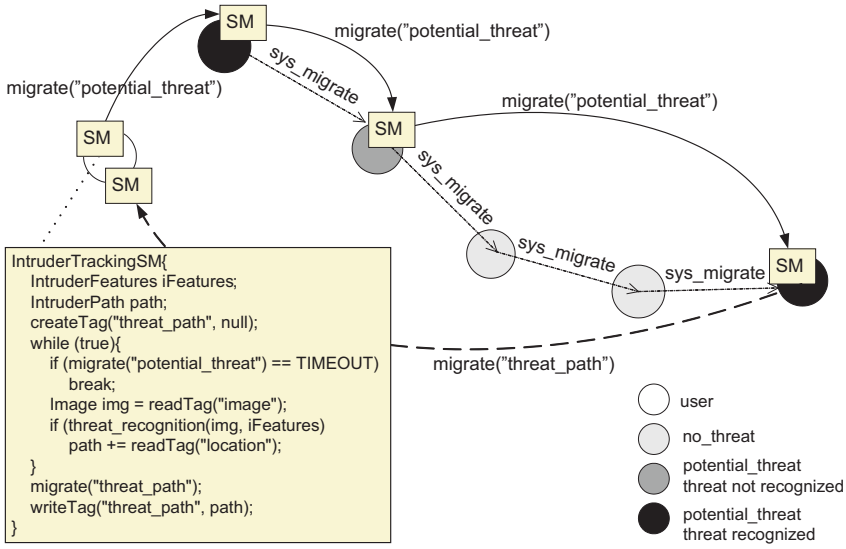

Figure 2.5: Smart Messages example: the `IntruderTrackingSM` provides the motion path of a user-specified threat.

form intruder tracking across the region of deployment. Each camera runs a pre-installed SM that periodically acquires images and performs 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 `IntruderTrackingSM` in the network. This SM takes as 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 (i.e., those cameras where this threat was recognized). At the user node, the `IntruderTrackingSM` creates a tag, "threat_path", that is used to identify this node when the SM migrates back with the result. Then, it migrates to nodes named by "potential_threat" tags (in the figure, this corresponds to the action *migrate("potential_threat")*) 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 (i.e., the migration times out). The `IntruderTrackingSM` 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.

The SM architecture was implemented in the Java programming environment over Linux by modifying the Sun Microsystem's KVM (Kilobyte Virtual Machine) [KVM, 2007] because its source code is available and has a small memory footprint suitable for resource-constrained devices. The entire software architecture needed for SM execution was implemented inside the VM due to the need for VM support to efficiently capture and restore the execution state upon migration to a new node.

SMs are essentially Java programs that invoke an API encapsulated in two Java classes: *(i) SmartMessage* which includes primitives for migration, new SM creation, and synchronization; and *(ii) 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 the SM-specific serialization format. 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 non-static. For non-static methods, also the machine-independent identifier for the *this* self-reference is encoded. After the admission manager successfully receives the code bricks, the data bricks, and the execution control information from a source node, then a new VM-level thread and its associated SM structure are constructed. Additionally, the admission manager de-serializes 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 (i.e., 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, five protection domains for controlling the access to tags are defined. 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 (i.e., 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.

Because of the limited portability of this implementation and the impossibility (or unwillingness) for users to modify the system software on their mobile devices, the original SM architecture was re-implemented in such a way that it can run on top of unmodified Java virtual machines. This second version of SM is called Portable Smart Messages (Portable SM) [Ravi et al., 2004]. One main issue that needed to be addressed in this implementation was how to accomplish migration without having access to the VM and, in particular, to the execution state. The solution was to design a lightweight migration based on Java bytecode instrumentation. The bytecode is instrumented in such a way that Portable SM can save its state before migrating and restoring it upon migration. The state is encoded in the data bricks and no explicit state information is shipped. Soot 1.2.5 [Soot, 2007] was used to do the off-line bytecode instrumentation. The performance evaluation demonstrated that the overhead generated by the increase in Java bytecode size due to the bytecode instrumentation is only 3%. More details on the instrumentation mechanism are provided in [Ravi et al., 2004]. Portable SM was tested on J2ME CDC platforms.

In implementing our middleware architectures, we used both SM versions. Contory was implemented using Portable SM running on J2ME CDC platform and it was tested on Nokia Series 80 phones.

The Migratory Services framework was first implemented on top of the original SM version and tested on HP iPAQs, and then extended using Portable SM and tested on Nokia Series 80 phones.

2.5 Concluding remarks

Ubiquitous sensing is an essential requirement for the deployment of pervasive applications. Ubiquitous sensing defines the ability of applications to seamlessly access anytime, anywhere data produced by sensors embedded in the surroundings. This capability enables a wide variety of "sensing applications" ranging from traditional context-aware applications to vehicular information system, to social networking.

This chapter presented research work that has been done so far to support ubiquitous sensing, and, in particular, sensor networks and mobile ad hoc networks. Algorithms, protocols, and architectures that have been proposed in these domains build the background knowledge to understand the middleware solutions we will present in this dissertation. In particular, the Smart Messages platform, which was specifically designed to cope with highly volatile ad hoc environments, has been used as system platform for implementing our middleware services.

Before we present the design, implementation, and evaluation of our middleware architectures, in the next chapter, we define our problem space, called Urbanet. In Urbanets, sensor networks and mobile ad hoc networks meet and generate open sensing environments potentially capable of supporting a large variety of mobile sensing applications.

CHAPTER 3

Programming challenges in Urbanets

This chapter discusses core challenges to be addressed to successfully program people-centric mobile sensing applications in Urbanets, spontaneously created networks in urban environments. We first describe Urbanets and their characteristics, then illustrate several examples of mobile sensing applications, and finally discuss the challenges that middleware architectures have to address to support the envisioned applications. The last part of the chapter discusses related work in this field.

3.1 Urbanets

Sensor data of different types and from different sources will be more and more available in the urban landscape. Installations of sensor networks as well as single sensors embedded in buildings, roads, or fields will become more and more common in city environments. Vehicles will come more and more often equipped with sensor technology to monitor the road conditions or exchange information about the traffic situation. Smart phones carried by people already offer several sensing possibilities (i.e., capability of acquiring sensor data) and in the future they will be capable of supporting sensors of various types. In addition, the ubiquity of wireless connectivity in urban settings allows private inhabitants as well as public municipalities to publish and share their sensors data. This,

in turn, will allow people to access a wide variety of sensor data, and spatially and temporally query them to support their personal tasks.

We use the term *Urbanets* to define spontaneous urban networks composed of heterogeneous and mobile multi-sensor platforms, such as smart phones and embedded vehicular systems, public sensor networks deployed by municipalities, and individual sensors incorporated in buildings, roads, or daily artifacts. Sensor networks and mobile ad hoc networks meet in Urbanets to create rich and open sensing environments, where people, municipalities, and community organizations share their resources to allow mobile users real-time access to sensor data. Much of these data will be incorporated in novel "sensing" applications running on our personal mobile devices. Sensing applications are therefore defined as applications capable of accessing data produced by sensors integrated in the device, provided by external devices, or embedded in the physical environment anytime, anywhere.

Urbanets present distinguishing characteristics compared to sensor networks and mobile ad hoc networks. Urbanets differ from the first-generation sensor networks in their goal to support concurrent people-centric sensing applications as well as in their hardware and software heterogeneity, high volatility, and very large scale. More specifically, in Urbanet applications, the focus is on the user, on her requirements, and on her resource capabilities. Urbanet users are not only static consumers of information, but also active producers of information from which other users may benefit. Users are mobile and their requirements and resources varying over time and location need to be constantly evaluated. In addition, users can be many and different, thus leading to software and hardware heterogeneity, scalability, and load-balancing issues. Finally, unlike sensor networks, in Urbanets, networks of sensing devices are not anymore fully dedicated to support one single task; more likely, they will need to satisfy different concurrent requests. This also means that while Urbanets greatly enhance our ability to extend the sensing coverage and incorporate sensed data in a large spectrum of mobile applications, they are not expected to achieve the same level of sensing fidelity as in static sensor networks composed of nodes primarily dedicated to sensing.

Urbanet applications are also different from traditional MANET

applications such as file transfers. Urbanets aim to provide support for applications that acquire, process, and distribute real-time sensing information from devices located in the proximity of geographical regions, entities, or activities of interest.

In Urbanet environments, with mobile sensor platforms of various types and with different service infrastructures available at location, different interactions can be established among sensors, sensors' owners, physical surroundings, sensor network gateways, service infrastructures, and Internet services. In the following, we classify the main types of sensor devices that can be encountered in Urbanets:

- *Static sensors*: sensor devices can be placed almost everywhere: in buildings, streets, bridges, offices, homes, train stations, etc. Static deployments of sensors are likely to be long-term. There are already many sensors deployed in cities, such as CCTV cameras, parking meters, traffic signals, air-sniffing sensors, and pedestrian crossing [Yoneki, 2005].

- *Mobile sensors:* sensors can be carried by people or be integrated in moving objects such as phones, cars, buses, or bikes. They can serve numerous purposes such as blindspot obstacle detection, rain detection for automatic windscreen wipers, or surround imaging for parking finding. In particular, a lot of research has focused on sensing systems for cars such as accelerometers for airbag control and emergency door unlocking, GPS for route navigation, and on-board sensors for speed, steering wheel angle, yaw rate, gear position, throttle position, acceleration, and brake pressure [Cheng and Trivedi, 2006; Waldo, 2005].

- *Static sensor collection points:* these are usually sensor networks' root notes (or sinks), which are responsible for aggregating sensor data provided by multiple homogenous, tiny sensor devices. For example, they can be data loggers and processing units for structural monitoring of highway overpasses, roads, retaining walls, bridges, buildings, amusement park rides, etc. In addition, static sensor collection points can also be devices capable of aggregating sensor data provided by heterogenous mobile (or static) sensors. For example, in

the Cartel [Cartel, 2007] project, cars (the mobile sensors) moving on the highway are used to collect various types of highway information and opportunistically transfer them to fixed base stations (the static sensor collection points) deployed along the road [Hull et al., 2006]. Interactions with static sensor collection points are usually based on physical proximity or through Internet connectivity [Campbell et al., 2006]. For example, core aggregation points in sensors networks have usually Internet connectivity to store the sensed data remotely and make them accessible to the end user.

- *Mobile sensor collection points:* in mobile sensor networks, a set of mobile nodes are used as data mules [Hull et al., 2006] to collect data provided by sensors encountered while moving. Ultimately, this permits achieving high-density data sampling over wide areas. Mobile sensor collection points, also called Mobiscopes [Abdelzaher et al., 2007], can be particularly beneficial to allow isolated sensor networks to occasionally connect to the Internet and store their sensed data remotely. For example, cars can act as collection points of data provided by sensors embedded along the road, in buildings, or in other cars that are occasionally encountered while moving across the city. Cars can then opportunistically transfer the gathered data to (more powerful) static collection points present in the city infrastructure. Likewise, human interactions also enable sensor sharing among carried devices [Fall, 2003].

Once deployed, these kinds of Urbanet sensor platforms may be private, publicly available, or belong to different administrative domains, they may require a priori registration and authentication, and they may be or not be cost-free. Therefore, interactions in Urbanets might be affected by constraints beyond those imposed by physical proximity, resource availability, and network connectivity [Campbell et al., 2006].

3.2 Mobile sensing applications in Urbanets

The capability to acquire sensor information characterizing local and remote urban environments enables a variety of novel sensing

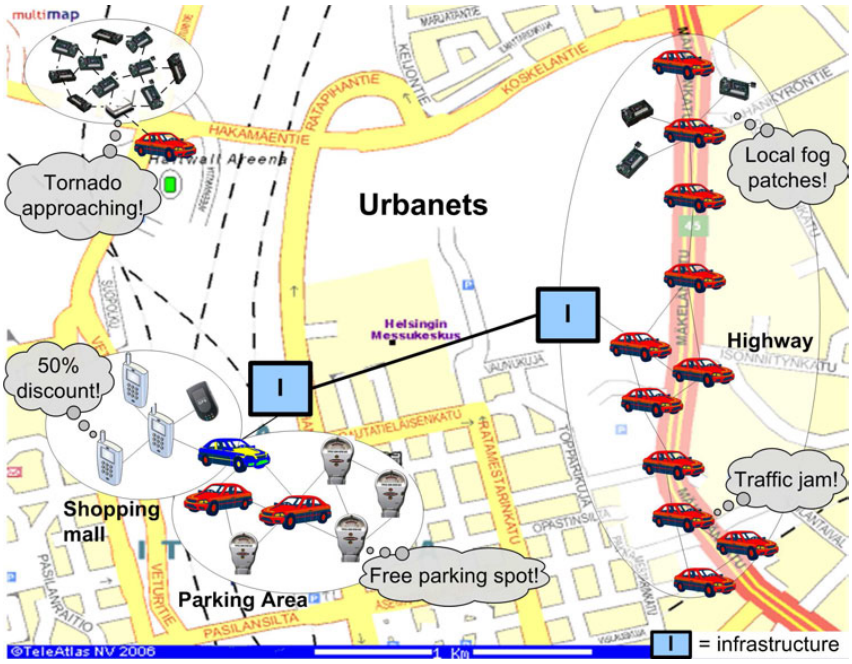

Figure 3.1: Example Urbanet scenarios in a city.

applications. Figure 3.1 depicts various sites of daily life in which we envision mobile sensing applications.

A driver assistant application informs drivers about traffic and road conditions further ahead. Drivers are promptly warned about unexpected traffic jams and can receive real-time route recommendations customized to their progress on the road. To collect the necessary sensor information, the application uses a network of cars and available sensors. For example, to detect and predict traffic jams, the application needs information such as density and speed of vehicles moving in the local proximity or in the proximity of a destination of interest (e.g., next exit on the highway). To detect hazards such as fog patches or icy roads and quickly propagate alerts to upcoming drivers, the application uses environmental sensors embedded in other cars as well as in the physical surroundings (e.g., a weather network of humidity and temperature sensors along the road).

When driving downtown, the driver uses a parking finder appli-

cation to receive driving directions to free parking spots close to the destination of interest (e.g., a shopping mall). This application uses a network of smart phones and vehicular systems to query wireless-enabled parking meters placed on the adjacent streets or parking areas.

Smart phones are protagonists in numerous Urbanet spots such as shopping malls, offices, conference centers, and highways. In the shopping mall, a recommender system running on the phone reads product RFIDs and exchanges product information with other phones in order to provide users with personalized recommendations and notify them about special offers.

Farther away, a municipal sensor network for weather monitoring measures twister's wind speed, temperature, and atmospheric pressure to detect a tornado quickly approaching from north-west and alert cars passing by. The real potential in this example is that cars moving across the city can act as carriers to deliver and propagate information collected by (isolated) sensors met on the way.

3.3 Middleware challenges

As previously discussed in Section 2.3, research on MANETs and WSNs has been quite successful in designing device platforms, protocols, and network architectures that can be applied to Urbanets. However, programming people-centric, mobile sensing applications, such as those envisioned in Urbanets, has received only marginal attention so far. As the domain of possible Urbanet applications diversifies, it will be almost impossible to program each application from scratch. Therefore, we expect an increasing demand for a common distributed middleware platform to support the development and execution of such applications.

This middleware platform should present application developers with programmable environments by offering a set of services that allow multiple software modules to interact across the network. These services should provide a more functional set of application programming interfaces than those that operating systems and network services provide. The characteristics of Urbanets and the specific requirements of people-centric, mobile sensing applica-

tions place new challenges on middleware developers. Among the several challenges, in the following we discuss some crucial ones.

3.3.1 Network volatility

Urbanets consist of functionally heterogeneous nodes, have volatile configurations, and present unknown delays. Therefore, traditional distributed computing models that assume underlying networks composed of functionally homogeneous nodes, with stable configurations, and known delays, cannot be used directly in Urbanets. In addition, Urbanets evolve unpredictably over time and space, and it is impossible to know the exact number or location of their resources. For example, many traditional client-server interaction models are connection-oriented. Clients select servers, bind to their interfaces, and then invoke operations on these interfaces. When these approaches are applied to Urbanets, as the environment and network connectivity change, the connection can drop and force the client to discover a new server. The client may then discover another server that offers a similar service, but that uses a different, unknown data format, thus making the interaction impossible.

Urbanet applications should not fail each time something goes wrong in the network. Applications should be tolerant to node failures and network partitions. Compared to traditional distributed computing systems where servers work correctly for long periods of time, Urbanets are more challenging in terms of reliability because communication, software, and hardware faults occur frequently and can thus render long-term, stateful interactions unfeasible.

Therefore, to cope with the volatility and unreliability of Urbanets, more flexible and adaptive middleware approaches are required.

3.3.2 Sensor variability and fidelity

Urbanet applications should work even when confronted with highly variable sensor data fidelity. For example, uncoordinated mobile sensors such as cars may sense certain parts of the city several times (busy city streets) and rarely visit other parts (garden suburbs). Or also the sensing quality may be accurate in rush hours, but not during the night. This means that the sensing coverage

is highly heterogenous in space and time [Abdelzaher et al., 2007]. "Best effort" semantics that tolerates the network and sensor dynamics while providing a certain quality of result to applications is desirable [Ni et al., 2005]. An Urbanet middleware platform should allow the application to specify the desired quality of result and trade-off quality of the produced results for network resources.

Moreover, applications should be able to verify sensor data's correctness. An Urbanet service may return a set of "correct" results with qualifying properties. Applications can then order these results by using different criteria. Qualifying properties describe the physical context in which the measurement was carried out. Location and time may be easily available. Other parameters might be application-specific. Ideally, applications should be able to validate the received data without violating the source's privacy.

3.3.3 Naming

Traditional distributed computing models assume fixed bindings between names and node addresses. This naming is too rigid for Urbanets, where nodes of interest are determined based on their contextual properties, such as available sensors, location, computational resources, or energy. From an application's point of view, nodes with the same properties located in the same region may be interchangeable. For example, an application for detecting traffic jams in a certain region of the city needs to interact with cars that are constantly located in such a region and are equipped with speed sensors. Rather than binding to physical nodes (identified by static IP addresses), the application interacts with virtual nodes presenting the required properties. Fixed naming schemes are not appropriate for these environments; more likely, naming schemes will be based both on space and content. Therefore, data-centric or property-based naming, like in sensor networks, are necessary.

3.3.4 Limited resources

In Urbanets, nodes are not fully dedicated to support sensing applications such as in sensor networks. These nodes are primarily involved in other tasks and only secondarily in supporting Urbanet applications. Hence, resources on hosting devices must be used

without compromising other primary processes running on the device. For example, phones are in the first place used for making and receiving phone calls. During a phone conversation, sensing tasks should be suspended. When the battery level is low, the sensing activity should be decreased, for example, by reducing the sampling frequency of the connected sensors. Therefore, it is important to dynamically optimize resource utilization given the sensing activity (e.g., physical measurements), the network conditions (e.g., neighborhood connectivity, link states, or available bandwidth), the local resources (e.g., remaining battery power), and other active tasks running on the device (e.g., e-mail synchronization).

On the other hand, it is also important to make an opportunistic utilization of available communication, computation, and sensing resources. The ultimate goal is to optimize the global network resource utilization. Mobile collection points should transfer the collected data to storage points as soon as they are available. When numerous mobile nodes are positioned in the same region (e.g., morning rush hours), depending on the application's requirements, only selected subsets of the available nodes should sense and transfer data. Moreover, delegation of sensing tasks among nodes can help carry out the assigned sensing tasks more efficiently. For example, static collection points can delegate mobile sensors to execute the sensing and report back the results. This could be the case of city buses that collect data on their regular routes and report data to processing units at the central bus station.

3.3.5 Large data traffic

Finally, another distinguishing feature of Urbanets compared to regular sensor networks is that they must support concurrent user applications. Managing simultaneous user applications can generate large data traffic in often resource-impoverished environments. For example, applications may generate several queries, each for different sensor types, with different frequencies, and in different locations. An Urbanet middleware should balance resource utilization across multiple applications and limit the geographical scope of the control updates. Task coordination can help avoid redundancy. Data aggregation can largely reduce the traffic load. Task prioritization is necessary to support emergency and critical applications.

3.4 Related research on people-centric urban networks

Urbanets share the goal of building large-scale, people-centric, mobile, urban sensor networks with several recently-initiated projects.

The SenseWeb [SenseWeb, 2007] project at Microsoft Research aims at providing a web-based platform and tools that allow people to easily publish and query sensor data. The SenseWeb architecture includes four main components: *(i)* GeoDB which is a geo-indexed database storing descriptions of (non real-time) sensor data; *(ii)* DataHub which is a web service that allows data publishers to register sensor descriptions and also offers real-time data storage; *(iii)* IconD which queries GeoDB or DataHub, depending on the submitted user's query and generates icons to be displayed at the client interface; and *(iv)* SenseWeb Client which allows users to specify their queries in terms of locations of interest, sensor types, and filtering and aggregation operators. The SenseWeb portal, called SensorMap, has been made available worldwide to allow publishers to publish their sensor data.

The CarTel [Cartel, 2007] project at MIT focuses on building a delay-tolerant mobile sensing architecture based on opportunistic communication to collect, process, deliver, and visualize data from mobile sensor nodes, such as cars [Hull et al., 2006]. The system includes three core components: *(i)* a central portal which hosts CarTel applications, hosts a continuous query processor, and acts as sink for all data sent by mobile sensor nodes, *(ii)* ICEDB (intermittently connected database) which supports user queries specification, and *(iii)* CafNet (carry-and-forward network) which delivers data in intermittently connected networks. Applications issue SQL queries using the ICEDB interface. Mobile nodes collect data, store them in their local ICEDB databases and when connectivity is available, they send data to the portal using CafNet's data delivery mechanisms. Users can then browse the results using web-based applications running on the portal. Query processing is distributed across the mobile units, but collection of sensor data and access to the results are centralized in the portal.

The MetroSense [MetroSense, 2007] project at Dartmouth University proposes a new paradigm of people-centric sensing at scale

to support the execution of multiple sensing applications in parallel. This project proposes a three-tier architecture consisting of sensors at the first tier, sensor access points that collect and aggregate data at the second tier, and servers at the third tier. Sensors can be both static and mobile. Sensor access points provide secure and trusted collection of sensor data by tasking available sensors for a given application. Given the uncontrolled mobility of the sensor nodes, sensor tasking, data sensing, and data collection occur in an opportunistic manner. Servers are responsible for providing support to the collection of sensor data, for storing data, and for running applications related to a certain MetroSense administrative domain. Sample applications under study are BikeNet, which allows cyclists to share information such as bike paths with other cyclists, and SkiScape, which provides skiers with real-time information such as trail conditions or safety/emergency alerts.

The Urban Sensing [UrbanSensing, 2007] project at UCLA seeks to build short-term, community-oriented urban sensor networks. Its network architecture provides features such as resource discovery and application-level support for data gathering campaigns and context attestation. In particular, the technical design aims to address three issues of urban applications: verification and authentication of context data provided by the network or the application, privacy protection, and dissemination rules. The architecture consists of four types of entities: *(i)* sensors which provide sensor data; *(ii)* subscribers which receive, store, and process sensor data; *(iii)* registries which help subscribers discover and connect to sensor data streams based on query attributes such as location and type of sensor data; *(iv)* mediators which perform specific in-network services such as verifications, data anonymization, and fault-tolerance.

A common feature of all these projects is that they assume central collection points across the Internet that perform data and task management and act as mediators between users and the network. Since there can be a large number of active users querying the network, each with unique requirements in terms of frequency, accuracy, and location of the sensor measurements, the challenge is how to accommodate an increasing number of diversified queries while providing accurate and real-time data. In addition, given that a large part of users will be interested in acquiring information about their local surroundings, it is important to optimize this system to

reduce network bandwidth utilization. Finally, many mobile users might not have continuous Internet connectivity or this might be expensive.

With our middleware architectures for Urbanet environments we propose a complementary, decentralized view for programming distributed sensing applications. Distribution allows us to address the above challenges and, in particular, our solutions do not require servers or Internet connectivity.

Finally, it is worth to mention another initiative relevant to urban sensing projects like ours that is the Nokia's SensorPlanet [SensorPlanet, 2007] project. This is a Nokia-initiated cooperation on large-scale sensor networks. The objective of SensorPlanet is to build a test platform for researchers on mobile-centric wireless sensor networks.

3.5 Concluding remarks

This chapter presented Urbanets and the challenges that middleware platforms for the support of mobile sensing applications need to address. We gave examples of potential mobile sensing applications and we surveyed ongoing research projects on people-centric urban networks with a focus on their architectural approaches. Generally, current work on mobile sensing applications has taken centralized and infrastructure-based approaches. Centralized approaches can limit the scalability, flexibility, and guarantees of real-time data delivery that are essential requirements of future mobile sensing applications. Moreover, mobile users might not always have available Internet connectivity or this might be expensive. These observations lead us to develop distributed, infrastructure-less architectures that will be presented in the next two chapters.

CHAPTER 4

Contory

This chapter presents *Contory* (*Cont*ext fact*ory*). Contory is an Urbanet middleware that provides a database abstraction of the environment. Contory offers an SQL-like interface to issue context queries, in which application developers can specify the type and quality of the desired context items, sensor data sources, push or pull mode of interaction, and other qualifying properties. The key feature of Contory is that it executes context queries and collects sensor data by employing multiple strategies for sensor data collection: *(i)* internal sensor-based, *(ii)* external infrastructure-based, and *(iii)* distributed provisioning in ad hoc networks. The advantages of this approach are twofold. First, arranging different strategies for sensor data collection permits compensating for the temporary unavailability of one mechanism and coping with the dynamic resource availability. Second, combining results collected using different mechanisms allows applications to partly relieve the uncertainty of a single sensor source and to more accurately infer higher-level context information.

This chapter describes requirements study, design principles, query model, software architecture, and programming interface of Contory. We give insights into its implementation on smart phones, present experimental results in WiFi networks of smart phones, and describe prototype applications using it.

4.1 Motivating scenarios

Contory aims to support mobile applications that need to be aware of both "local" and "remote" context. Specifically, we deal with situations in which context consists of low-level sensor data characterizing entities, resources, and physical environments that can be local or remote with respect to the user location. Furthermore, as our focus is on low-level context information, we will use the terms "context data" and "sensor data" interchangeably, unless specified otherwise.

To understand how Contory can be useful, let us consider the Urbanet scenarios depicted in Figure 3.1 of the previous chapter. Contory can be used, for example, to support detection of local fog patches or icy roads along the highway. Contory interacts with environmental sensors located in the region of interest, in this case a portion of the highway, and periodically acquires sensor measurements such as temperature, humidity, and pressure. The collected sensor data are transferred to the user node using a network of collaborating nodes such as cars. The user node can be a car's embedded computer or a smart phone that can process the raw data and return results to the application. This is an example of local context. Then, computed results can also be broadcasted to neighboring devices or stored in remote repositories so that remote users can access such data using, for example, Internet connectivity. This is an example of remote context.

The possibility of gathering different types of information from a network of mobile devices in the same proximity offers a way to characterize the status of several types of services such as restaurants, shopping malls, or guest harbors, in real-time and in an accurate manner. For example, Contory can be used to find out the number of free tables and estimated waiting time in a restaurant. An ad hoc network of mobile phones owned by people who are already in the restaurant can be used to infer how many empty places are left and for how long the present people have already been in the restaurant.

Although most of the implemented context-aware applications have used mainly location information, there exist many other types of context information that can be considered depending on whether the sensor technology necessary for its support is already available.

4.2 Requirements study

Requirements for the development of Contory were gathered through experiences with a context-based application developed in the context of the DYNAMOS [DYNAMOS, 2007](Dynamic Composition and Sharing of Context-Aware Mobile Services) project. This DYNAMOS application, described in [Riva and Toivonen, 2006, 2007], aims to proactively provide mobile users with nearby services that are of interest based on the user's current context and needs. The application prototype runs on smart phones and was specifically designed to fulfill the expectations of a community of recreational sailboaters. Figure 4.1 shows some examples of how the DYNAMOS application can be used in a sailing scenario. First of all, the application proactively provides mobile users with an updated list of services of interest (i.e., "MatchedServices") available in the surrounding environment (see the top-left part of the figure). The user can also annotate these service descriptions (i.e., generate "service annotations") and share them with other users that might be interested in those services too (see the top-center part of the figure). Finally, the user can generate "user notes" that can be routine messages, safety warnings, or emergency alarms (see the bottom-right part of the figure).

In June and August 2005, we organized two field trials with sailboats. The first trial was a short excursion by sailboat in which the application was in use for 5 hours by 2 users. In the second trial, 28 persons on 9 sailboats participated in a one-day regatta. Each sailboat was equipped with a Nokia 6630 phone and a GPS device with Bluetooth (BT) interface.

During the regatta, location-awareness was accomplished using GPS devices connected through BT to the phone. Location updates were encapsulated in events and constantly transmitted (using 2G/3G networks) to a remote repository. Collected location traces were fairly discontinuous due to several disconnection problems. First, the BT connection to the GPS device went down several times, typically once per hour. Second, whenever the 3G connection was active and the phone had to make a handover to 2G, it switched itself off (this did not occur if the phone was set to operate only in 2G mode). Additionally, the traffic of events carrying context updates and going from the phone to the remote repository

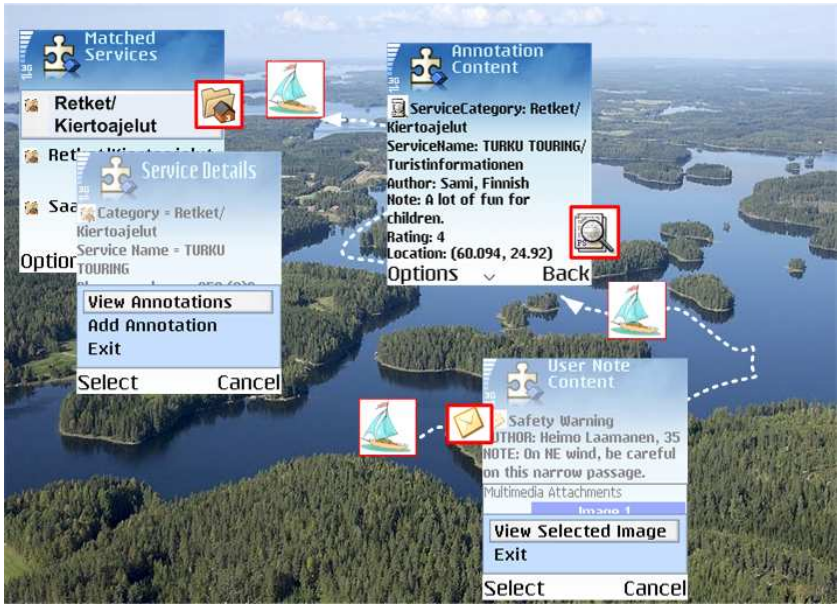

Figure 4.1: Examples of context-based services provided by the DYNAMOS platform in a sailing scenario.

had to be optimized and largely reduced, in order to keep peak power consumption below the limit over which the phone would switch itself off.

These experiences in the real field of action helped discover technical problems regarding context sensing and context management. We found that *(i)* context provisioning based exclusively on local sensors is often not reliable enough; *(ii)* sharing of context information owned by multiple users can provide useful services to the end user, enlarge the spatial range of context monitoring, reduce the global resource utilization, and permit coping with the sensors' unreliability; *(iii)* external infrastructures should be ready to cope with frequent users' disconnections (e.g., by incorporating prediction or learning algorithms); and *(iv)* the client application should be ready to cope with frequent disconnections from remote repositories.

4.3 Design principles

Both the practical experiences gained in the DYNAMOS project and the analysis of existing context-aware systems permitted identifying core issues for the support of context provisioning. The analysis of existing solutions also allowed to either reveal unsolved problems or find out solutions to well-known problems. In some cases, we also characterized recurrent design solutions through well-known or novel design patterns [Riva and di Flora, 2006b]. Systems to be analyzed were selected based on the extent of their completeness and utilization in supporting various context-aware applications. A list of analyzed systems can be found in [Riva, 2007].

Based on existing solutions to well-known issues and with the aim to address unsolved issues, we derived a list of principles for the design of Contory:

- *Flexible and reliable context provisioning*: Ideally, context provisioning should take place without any interruption, e.g., due to hardware faults or temporary disconnections from sensor devices. In Contory, multiple context provisioning strategies are made available and can be interchanged dynamically and transparently based on sensor availability and resource consumption.

- *Common querying interface*: To formulate requests of heterogeneous context items, Contory supports an SQL-like context query language. This common interface allows applications to specify the type and qualifying properties of the required context data.

- *On-demand, event-based, periodic queries*: Applications can interact with Contory by issuing on-demand queries or long-running queries, in which results are received either periodically or when certain event conditions are verified.

- *Modularity and extensibility*: Contory glues together several modular components for context provisioning thus enhancing the ability of the middleware to assume variable configurations. New sources of sensor data as well as sensor processing algorithms, which will be developed in the forthcoming years,

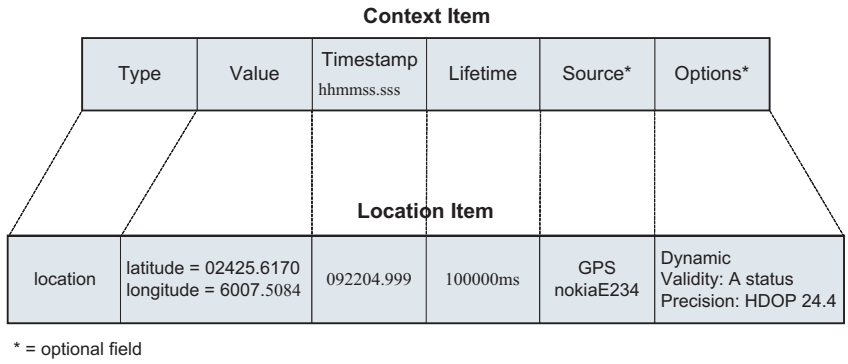

Figure 4.2: The context item format and an example of location item.

will also need to be easily accommodated in the existing architecture.

4.4 Middleware architecture

Contory seeks to provide specialized and transparent support for retrieving context items of different types and quality. This section describes core concepts for the design of Contory, its software architecture, and programming interface.

4.4.1 Context items and context metadata

Generally, the context associated with a certain situation can be expressed as a set of *context items*, each describing a specific element of the situation. For instance, the situation *walking outside* can be represented by the triplet of context items <*noise=medium, light=natural, activity=walking*>. Context items can describe spatial information (location, speed), temporal information (time, duration), user status (activity, mood), environmental information (temperature, light, noise), and resource availability (nearby devices, device power). In Contory, context data are exchanged by means of `cxtItem` objects (see Figure 4.2). Each `cxtItem` consists of `type` (context category), `value` (current value(s) of the item),

`timestamp` (the time at which the context item had such a value), and `lifetime` (validity duration). Optionally, the item can contain a `source` identifier (e.g., sensor, infrastructure, or device addresses) and other `metadata` information in the `options` field. Types of metadata information include correctness (i.e, closeness to the true state), precision, accuracy, completeness (if any or no part of the described information remains unknown), and level of privacy and trust.

4.4.2 Context query language

From an application's point of view, Contory acts mainly as a data-retrieval system to which applications submit context queries. Although similar to a database system, the dynamism and fuzziness of context data lead to distinguishing characteristics of the SQL-like interface. For example, sources of sensor data can provide large amounts of data, hence aggregation and filtering are required. Furthermore, monitoring sensor data is a continuous process, hence not only on-demand queries but also long-running queries have to be supported. Although different applications usually pose different requirements, rather than developing application-specific interfaces, we abstracted the functionality of several applications into one common SQL-like *context query language*. Figure 4.3 shows the query template.

```
SELECT <context name> [*]
FROM <source>
WHERE <predicate clause>
FRESHNESS <time>
DURATION <duration> [*]
EVERY <time> | EVENT <predicate clause>
```

Figure 4.3: The Contory's context query template.

The `SELECT` and `DURATION` clauses (marked with `[*]`) are mandatory. `SELECT` specifies the type of the requested context item. `DURATION` specifies the query lifetime as time (e.g, `1 hour`) or as the number of samples that must be collected in each round (e.g., `50 samples`).

In carrying out context provisioning, Contory aims to offer different levels of transparency to application developers. Essentially, depending on the application semantics, application developers can require to have low (maximum transparency) or high (minimum transparency) control over the execution of the context provisioning task. The maximum transparency is achieved when the `FROM` clause is unspecified and the middleware can dynamically and autonomously select which mechanism to adopt for collecting sensor data. Alternatively, the `FROM` clause permits specifying type and characteristics of the sources of sensor data and of the mechanisms to be employed for collecting the data.

As summarized in Table 4.1, three mechanisms for context provisioning are currently supported: internal sensor-based (`intSensor`), external infrastructure-based (`extInfra`), and distributed context provisioning in ad hoc networks (`adHocNetwork`). In the case of `adHocNetwork` provisioning, the `FROM` clause also tells the multiplicity of the sensor nodes to be utilized (`numNodes`) and their distance from the source (`numHops`) or the geographical region in which they should be located (`region`). For example, the search of suitable context items can involve all nodes that can be discovered (`numNodes=all`) in a certain region (`region=(60.21,24.81)`), or the first k nodes found within a distance lower than j hops (`numNodes=k`, `numHops=j`). Geographical regions are expressed using latitude and longitude coordinates, but, in principle, they could also be landmarks created by the end user, such as the user's home or the next exit on the highway. Alternatively, the programmer can also submit the specific destination (`destinationID`) to which the query has to be sent, that is usually the identifier of an entity (e.g., to know when a friend is nearby).

Table 4.1: Example values for query clauses.

Clause type	**Values**
SELECT	`location,light,noise,temperature,speed`
FROM	`intSensor,extInfra,adHocNetwork(numNodes,numHops),` `adHocNetwork(numNodes,region), {destinationID}`
WHERE	`accuracy,precision,correctness,levelOfTrust`

`WHERE` contains filtering predicates built using the context item's `metadata`. `FRESHNESS` specifies how recent the context data must be. Finally, our query language provides support for long running queries by means of `EVERY` and `EVENT` clauses. These clauses are mutually exclusive. The `EVERY` clause allows the application developer to specify the rate at which context data must be collected (periodic query). The `EVENT` clause determines the set of conditions that must be met at the context provider's node before a new result is returned (event-based query).

If we consider the example of an application for traffic jam prediction, the query presented in Figure 4.4 is sent to collect accurate `speed` values of all nodes (e.g., cars) found in `remote_region` (e.g., the next exit on the highway):

```
SELECT speed
FROM adHocNetwork(all,remote_region)
WHERE accuracy = true
DURATION 1 hour
EVENT AVG(speed)<min_speed
```

Figure 4.4: Example of context query requesting speed measurements from a remote region.

Speed measurements are returned when the average speed drops below `min_speed`. To compute the probability of traffic jam, the application can combine these data with the number of collected samples (i.e., density of cars), past observations, and, possibly, knowledge of the road topology.

4.4.3 Contory software architecture

Figure 4.5 depicts the conceptual architecture of Contory. *ContextFactory* is the core component of the overall architecture. One *ContextFactory* runs on each device and is supposed to manage multiple applications simultaneously. Based on the *Factory Method* design pattern [Gamma et al., 1995], this design model aims to define an interface for creating objects, but let subclasses decide which class to instantiate. In our case, the *ContextFactory* offers an interface to submit context queries, but lets *Facade* components (subclasses) decide which *CxtProvider* components (classes) to instanti-

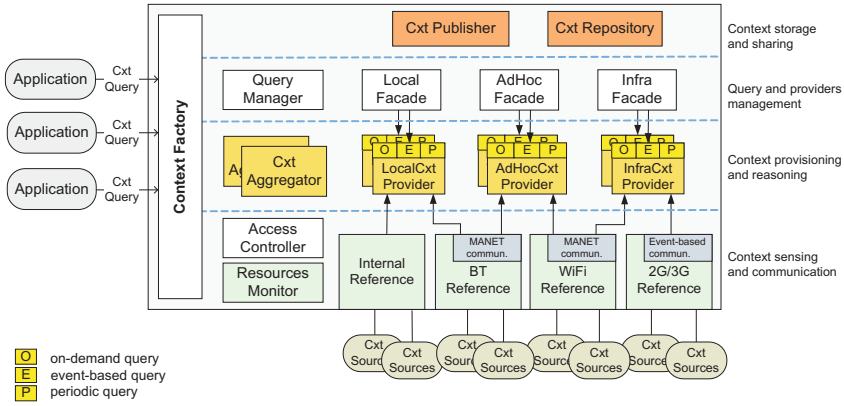

Figure 4.5: The Contory middleware architecture.

ate. The alternatives are *LocalCxtProvider*, *AdHocCxtProvider* and *InfraCxtProvider*.

The *ContextFactory* is responsible for:

- *context sensing and communication*,
- *context provisioning and sharing*, and
- *queries and providers management*.

In the following, we describe each functionality together with their core architectural components.

Context sensing and communication

Context data can be sensed from a large variety of *CxtSources* such as external sensors (e.g., a GPS device), integrated monitors (e.g., a power management framework), or external servers (e.g., a weather station). To provide discovery of *CxtSources* as well as to support communication with them, different types of *Reference* modules are provided. Typically, a *Reference* is responsible for mediating the access to a certain communication module by offering useful programming abstractions. As shown in Figure 4.5, Contory includes four types of *References*. The *InternalReference* is specialized to

support communication with sensors integrated in the device. The *BTReference* provides support to discover BT devices and services, and to communicate with them. The *WiFiReference* manages communication in WiFi networks and, for instance, provides abstractions for content-based routing and geographical routing in mobile ad hoc networks. The *2G/3GReference* manages interactions with remote entities over the corresponding network standards and offers an event-based interface to establish and control such interactions.

Mobile systems can undergo unexpected changes in the level of available resources, for example, when a new application is started on the device or when the host moves to a different network. Furthermore, in wireless environments, disconnections and bandwidth fluctuations are common. These issues make necessary the adoption of dynamic resource allocation mechanisms. The *ResourcesMonitor* component is in charge of maintaining an updated view on the status of several hardware items (e.g., device drivers), on the device's overall power state, and on the available memory space. Each time, network, sensors, or device failures affect the functioning of a communication module, the corresponding *Reference* notifies the *ResourcesMonitor* module. This, in turn, informs the *ContextFactory* which can then enforce a reconfiguration strategy to take over. For example, if the BT connection to a GPS device suddenly goes down, the task of location provisioning can be moved from a *LocalLocationProvider*, which is currently using the *BTReference* to connect to the GPS device, to an *AdHocLocationProvider*, which can use the *WiFiReference* to discover an active location provider in the nearby.

The *AccessController* module is responsible for controlling the interaction with external sources and requesters of context items. The *AccessController* keeps track of previously connected context sources, such as simple sensors or other sensor-equipped devices, and also of "blocked" context sources. Blocked context sources are those that due to previous unsuccessful interactions or due to current operating conditions should not be connected to the device. This list of devices is refreshed continuously so that only the most recent and the most often accessed sources are kept in memory. To decide to which devices Contory should grant access, if the application requires high-security operating mode, each time a new context source is encountered, it is blocked or admitted based on explicit

validation from the application. In low-security mode, every new encountered device is trusted.

Context provisioning and sharing

CxtProviders (i.e., context providers) are responsible for accomplishing context provisioning (or sensor data provisioning). Optionally, they can also incorporate reasoning mechanisms for inferring higher-level context data. A *CxtAggregator* can be used to combine context items collected from single or multiple *CxtProviders*. Alternatively, advanced context processing mechanisms can be performed by external context infrastructures or distributed across remote components [Flinn et al., 2002] in order to save energy on energy-constrained devices. Reasoning mechanisms strongly dependent on the application's semantics may be incorporated in the application itself.

CxtProviders are of three types. *LocalCxtProviders* manage the access to local sensors which can be embedded in the device or be accessible via BT. These providers periodically pull sensor devices and report values that match `WHERE` and `FRESHNESS` requirements. *InfraCxtProviders* are responsible for retrieving context data from remote context infrastructures. *AdHocCxtProviders* are responsible for supporting distributed context provisioning in ad hoc networks; to gather context data from nodes in a MANET, these providers utilize the *BTReference* (only for one-hop communication) or the *WiFiReference* (for multi-hop communication).

By supporting the specification of `EVENT` and `EVERY` clauses, context providers offer three modes of interaction: *on-demand query*, *periodic query*, and *event-based query*. An on-demand query returns results once. The query is executed for the first time. If results are available, these are returned and the query is removed from the nodes' cache. If results are not currently available, the query is executed repeatedly until valid results are found or the query's `DURATION` expires. A periodic query is executed for the desired `DURATION` and results are computed at the frequency specified in the `EVERY` clause. Finally, with an event-based query, each node in the network is instructed to return results each time the `EVENT` condition is verified, for the entire duration of the query.

The *CxtRepository* module is responsible for storing gathered

context information, locally or remotely. Only a few recent context data are stored locally, while complete logs can be stored in remote repositories of external infrastructures.

The *CxtPublisher* permits publishing context information in ad hoc networks by means of the *BTReference* or the *WiFiReference*. Each time a context item has to be published, two access modalities can be associated with it: "public access" allows any external entity to access the item, and "authenticated access" locks the item with a key that must be known by the requester.

Queries and providers management

The *QueryManager* is responsible for maintaining an updated list of all active queries and for assigning queries to the appropriate *Facade* components. For each of the three types of context provisioning mechanisms supported, a corresponding *Facade* module offers a unified interface for managing *CxtProviders* of that specific type. The purpose of utilizing the *Facade* design pattern [Gamma et al., 1995] is to abstract the subsystem of *CxtProviders* to offer a more convenient (unidirectional) interface to the *ContextFactory*. The *Facade* knows which subsystem classes (i.e., *CxtProviders*) are responsible for a certain query and can direct actions or requests of the *ContextFactory* to the suitable component.

The *QueryManager* invokes the `processCxtQuery` factory method of the *ContextFactory* interface to assign the query to one or multiple *Facade* modules. The assignment is done based on the requirements specified in the query's `FROM` clause, based on the sensor availability, and in the respect of the active control policies. For instance, a control policy can specify the maximum amount of memory and power consumption that can be allocated at runtime to a certain application.

Control policies are formulated as *contextRules*. Each contextRule consists of a condition and an action statement. 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`, and `lessthan`. An example of com-

parisonNode is

```
<batteryLevel,equal,low>
```

A combinationNode combines two nodes by means of *combinationOperators* such as `and` and `or`. An example of combinationNode is

```
<or,<batteryLevel,equal,low>,<memoryFree,lessthan,K>>
```

Therefore, `and` and `or` operators permit flexibly combining elementary conditions to build more complex ones.

The entire condition clause of a certain contextRule is evaluated by verifying all the contained boolean expressions based on the current values of the device's context parameters. Each rule is set as optional or mandatory, as the context parameters required by the rule may not be always available. Whenever a condition clause is positively verified at runtime, the associated action becomes active and it is enforced by the *ContextFactory*. Actions currently supported are `reducePower`, `reduceMemory`, and `reduceLoad`.

The enforcement of these reconfiguration actions can have different effects such as the switch from a certain provisioning mechanism to another one or the interruption of a query execution. For example, the activation of the `reducePower` action can cause the suspension or termination of high energy-consuming queries (e.g., those using the *2G/3GReference*) or the replacement of WiFi-based multi-hop provisioning with BT-based one-hop provisioning. In the case of `intSensor` and `extInfra` provisioning, interrupting a query execution simply requires a cancel message to be sent to the sensor devices or to the external infrastructure respectively. In the case of `adHocNetwork` provisioning, interrupting a query execution might be an expensive (or almost impossible) process, especially in a highly mobile ad hoc network. Therefore, the middleware can locally interrupt the query execution and optionally disseminate cancel messages into the network.

Once the query has been assigned to a *Facade*, in order to avoid redundancy and keep the number of active queries minimal, the *Facade* performs query aggregation. This process consists of two sub-processes: query merging and post-extraction. The *Facade* first verifies whether the new submitted query `q1` can be merged with

any other active query `q2`. If `q3 = merge(q1,q2)` can be computed, `q3` is the new query to be processed. The post-extraction sub-process is applied to the received results for `q3` in order to extract the data matching the original queries `q1` and `q2`.

The `merge` function implements a very simplified version of the clustering algorithm defined in [Crespo et al., 2003]. This algorithm builds on the definition of a "distance" metric between queries. The algorithm computes the distance between each pair of queries and if this distance is below a certain threshold, the two queries are put in the same cluster. Our design is based on a similar principle. We first group queries with the same `SELECT` clause and then perform the merging by applying clause-specific merging rules, as exemplified in Figure 4.6.

```
q1: SELECT temperature
    FROM adHocNetwork(all, 3)
    FRESHNESS 10 s
    DURATION 1 hour
    EVERY 15 s

q2: SELECT temperature
    FROM adHocNetwork(all, 1)
    FRESHNESS 20 s
    DURATION 2 hour
    EVERY 30 s

q3 = merge(q1, q2): SELECT temperature
                    FROM adHocNetwork(all, 3)
                    FRESHNESS 20 s
                    DURATION 2 hour
                    EVERY 15 s
```

Figure 4.6: Example of merging of two context queries.

Upon the aggregation process has completed and the merged query is obtained, the *Facade* module either instantiates a new *CxtProvider* or updates the query parameters of an existing *CxtProvider* (e.g., in case the new query has been merged with an already active query). *CxtProviders* of different *Facades* can be assigned to the same query, but a *CxtProvider* is assigned only to one (single or merged) query at time.

4.4.4 Contory programming interface

The Contory API shields the programmer from the underlying communication platforms and context provisioning aspects. To interact with Contory, an application needs to implement a `Client` interface and implements the following methods:

- `receiveCxtItem(CxtItem cxtItem)` in order to handle the reception of the collected context items;
- `informError(String msg)` to be called by several Contory modules in case of malfunctioning or failure;
- `makeDecision(String msg)` to be invoked by the *AccessController* to grant or block the interaction with external entities.

Applications using Contory can invoke its services using the `ContextFactory` interface, shown in Figure 4.7. First of all, the interface offers methods for submitting and erasing context queries (see line 2 and line 3). When the application invokes the method `processCxtQuery`, the middleware decides to which context provider assign the query. The application can later reference a submitted context query by means of the `queryID`, which is automatically generated by the system at the query's instantiation time (i.e., through the constructor `createCxtQuery`).

```
1 public interface ContextFactory{
2   boolean processCxtQuery(CxtQuery query);
3   void cancelCxtQuery(String queryID);
4   void registerCxtServer(CxtServer publisher);
5   void deregisterCxtServer(CxtServer publisher);
6   boolean publishCxtItem(String cxtItem, boolean pub);
7   void storeCxtItem(CxtItem cxtItem);
8 }
```

Figure 4.7: The `ContextFactory` interface.

In order to be eligible to publish context items and make them accessible to other context requestors, the publishing application must register with the middleware and be authenticated (line 4). After registration, using the method `publishCxtItem`, the application can publish (`pub=true`) or erase (`pub=false`) specific context

items (see line 6). The publishing method invokes the `CxtPublisher` that, in turn, contacts the `CxtProvider` in charge of providing context items of that specific type (the type is stored in the `cxtItem` object passed by the application). The `CxtProvider` stores the `CxtPublisher`'s subscription and, each time the `CxtSources` make a new context value available, it communicates the new `cxtItem` to the `CxtPublisher`. The `publishCxtItem` function returns `true` if the publishing action succeeded or `false` otherwise. For example, the action does not succeed when the application tries to publish an item for which no corresponding `CxtProvider` currently exists or when it attempts to erase a non existing item. When the application completes or the publishing task needs to be interrupted for some reason, the application deregisters (line 5).

Finally, the interface permits storing context items on remote repositories (see line 7) that have registered with the middleware. The `CxtRepository` object is in charge of managing the interaction with the repository.

To support the generation of context queries, the following vocabularies are available to the application developer:

- the `CxtVocabulary` contains context types, context values, and metadata types for specifying context items and device resources;
- the `QueryVocabulary` contains parameters for specifying context queries; and
- the `CxtRulesVocabulary` contains operators and actions for specifying control policies.

4.5 Implementation

Contory has been implemented using Java 2 Platform Micro Edition (J2ME). Currently, two separate implementations exist: one for Connected Limited Device Configuration (CLDC) 1.0 and Mobile Information Device Profile (MIDP) 2.0 APIs, and one for Connected Device Configuration (CDC) 1.0. Contory's implementation is available under an open source license [Contory, 2007].

The J2ME platform was selected because it currently represents the most widespread and standard computing platform for personal mobile devices. However, as we discuss in [Riva and Kangasharju, 2007], disadvantages of this platform are limited debugging support, limited programming environment, slow storage access, and not yet available sensor API to manage and control sensors connected to the phone. For Java ME, the Mobile Sensor API [JSR 256, 2007], is currently under development, but it will still take time for this work to be completed and become available on actual devices.

It has proven very difficult experimenting with real-world sensors that can be connected to the phone. Proper support for context-aware applications requires sensor devices to be available to the phone and also to be able to produce real-time measurements. Currently, there are very few phones that integrate sensors. The Nokia N95, the first smart phone with GPS integrated, was announced only in late 2006. The more common alternative are external sensors, usually connected through BT with a BT-based programming API. Yet, apart from BT-GPS devices, most BT-based sensors are capable of providing only logs of sensor data, but not real-time measurements.

All software development was done using Nokia Series 60 and Nokia Series 80 phones running Symbian OS. We decided not to implement our platforms in Symbian mainly because many of the external software systems that we intended to integrate in our platforms, such as the Fuego Core's event system and the SM platform, are implemented in Java. Moreover, available native Symbian implementations often require different builds because different Symbian OS versions do not always interoperate.

In the following, we provide specific insights into the implementation of the *Reference* modules and distributed context provisioning in ad hoc networks. The *InternalReference* module has not been implemented yet, because no sensors integrated in the phone platform that we used for the development were available at deployment time.

4.5.1 Network communication modules

The *BTReference* uses the Java Specification Request 82 (JSR 82) [JSR 82, 2007] available for CLDC. This specification defines a

standard set of APIs for BT wireless technology and in particular targets devices that suffer from constraints in processing power and memory. The specification includes support for *(i)* device discovery, service discovery, and service registration, *(ii)* establishing connections between BT devices and using those connections for BT communication, and *(iii)* managing and controlling active BT connections.

The *WiFiReference* uses the Smart Messages (SM) [Borcea et al., 2002] distributed computing platform, described in Section 2.4, to provide device and service discovery, content-based routing, and multi-hop communication in ad hoc networks. We utilized Portable SM [Ravi et al., 2004] implemented for the J2ME CDC platform.

The *2G/3GReference* offers support for event-based communication by using the Fuego Core middleware [Fuego Core, 2007; Tarkoma et al., 2006]. This middleware is implemented in Java and provides a scalable distributed event framework and XML-based messaging service [Kangasharju et al., 2005]. This middleware also runs on mobile phones supporting Java MIDP 1.0.

4.5.2 Distributed context provisioning

Distributed context provisioning is accomplished through the collaboration of Urbanet nodes willing to use their resources to support the execution of Contory. A Contory mobile ad hoc network consists of nodes running Contory and communicating using wireless connectivity. In our implementation, the *WiFiReference* is used for managing the communication in multi-hop networks and the *BTReference* is used only in one-hop networks.

Distributed context provisioning is performed in three phases:

1. *Initialization*: nodes willing to cooperate initialize their *Reference* modules, especially to support routing of Contory messages.

2. *Publishing*: through the *CxtPublisher* module, Contory nodes can publish their context information for public access; we will call these nodes "publishers".

3. *Execution*: *AdHocCxtProviders* running on Contory nodes can issue context queries, the requested context items are

searched in the network of publishers, and the results are returned to the requestor nodes, called "consumers".

Figure 4.8 shows an example of Contory network where nodes 2,4,5,8,9 and 10 are publishers, node 1 is a consumer, and the remaining nodes simply support routing of Contory messages (i.e., they are initialized).

In the WiFi-based implementation, the Smart Messages computing platform was used. In the initialization phase, the *WiFiReference* running on each node expresses its willingness to participate in the Contory ad hoc network by exposing the SM tag "contory". Essentially, this means that to perform multi-hop routing, all nodes exposing the tag "contory" can be used to route packets from one source to the destination. In the network in the figure, we assume that all nodes are exposing the tag "contory".

To publish a context item in the network, the *CxtPublisher* module of the publisher node creates and exposes an SM tag whose name is the context type (`cxtTagName`) and whose value contains the value (`cxtTagValue`) of the context item together with the metadata information qualifying the item. An example of tag is the following:

$$tempTag = \begin{cases} cxtTagName = temperature \\ cxtTagValue = 14^oC, 1^oC, trusted \end{cases}$$

To discover context items of interest, the context query is routed towards publisher nodes exposing the desired `cxtTagName` (i.e., the tag whose name matches the `SELECT` clause of the carried query). This is accomplished by using the SM's geographical routing and content-based routing functions. To disambiguate between multiple messages, a unique identifier is associated with each query (`queryID`) and with each result (`resultID`). If no valid result is received within the specified duration, the query execution is terminated. Otherwise, if nodes exposing context items of the type of interest are discovered, `WHERE`, `FRESHNESS` and `EVENT` requirements specified in the query are evaluated. If positively verified, the value of the context item along with additional metadata properties are returned to the consumer node that issued the query. In the example in the figure, only node 8 and node 10 send back

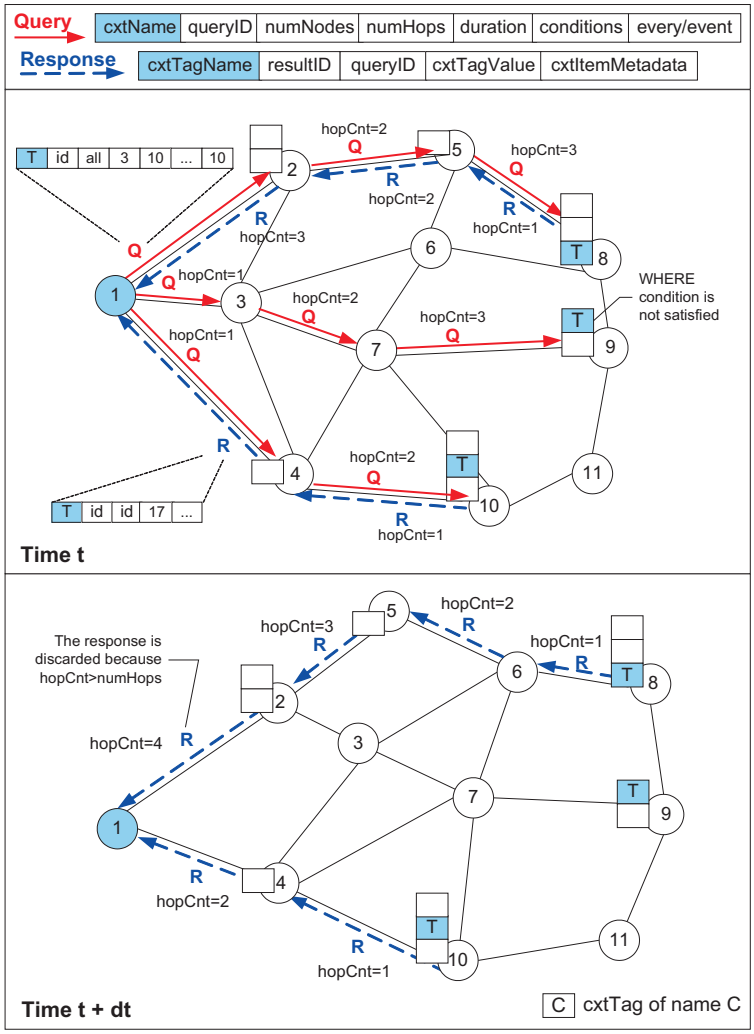

Figure 4.8: Example of distributed context provisioning in a Contory mobile ad hoc network.

responses, whereas node 9, although located in the region of interest (i.e., within a distance of 3 hops), does not satisfy all query's requirements.

At the SM level, queries and responses are encapsulated in SMs called `SM-FINDER`s. Each `SM-FINDER` maintains a `hopCnt` to perform routing of both queries and responses. `hopCnt` indicates how many hops the message has traversed until that moment. As shown in the figure, in the query dissemination, `hopCnt` is used to control the propagation of the query in the region of interest. In the query execution, `hopCnt` is constantly compared with the `numHops` specified in the context query. When the response is delivered to the consumer, if `hopCnt`>`numHops`, the consumer discards the result because the publisher that provided such a result is out of the range of interest. In the figure in the bottom, after nodes have moved, while node 10 is still located in the region of interest, node 8 has moved out of the region of interest (i.e., it is now 4 hops away) and its responses are therefore discarded.

In the BT-based implementation, the initialization phase places the BT device into inquiry mode and specifies an event listener that will respond to inquiry-related events. The *CxtPublisher* module running on the publisher node can publish a context item by advertising a context service on the BT server (service registration). The server creates a service record describing the offered context service and adds it to the server's Service Discovery Database (SDDB). This is visible and available to external BT entities. The *AdHocCxtProvider* running on the consumer node first discovers accessible BT devices (in some cases a list of pre-known devices can be used) and then checks the services available on the discovered devices.

4.6 Experimental evaluation

To assess the performance of Contory, we built an experimental testbed of smart phones and measured response times and energy consumption for different context operations. The objective of this experimental analysis was to demonstrate the practical feasibility of the proposed approach, give an insight on the performance of our prototype implementation, and quantify its cost mostly in terms of energy consumption. Our experimental testbed consists of a Nokia

6630 phone, a Nokia 7610 phone, 3 Nokia 9500 communicators (see Table 4.2 for their core technical features), and a Bluetooth GPS Receiver InsSirf III. This section presents the experimental results obtained in such a network testbed.

Table 4.2: Technical specifications of the phones used in the experiments.

Phone	Symbian OS	Processor	RAM	Connectivity
Nokia 7610	v7.0s	123 MHz	8 MB	GPRS
Nokia 6630	v8.0s	220 MHz	10 MB	GPRS/EDGE/WCDMA
Nokia 9500	v7.0s	150 MHz	76 MB	GPRS/EDGE/WiFi

4.6.1 SM experiments

We first ran some preliminary tests to assess the performance of the Portable SM platform [Ravi et al., 2004] on Nokia 9500 phones. We first measured the cost of some basic `TagSpace` operations, reported in Table 4.3. As explained in Section 2.4, the tag space consists of *(name, data)* pairs, called tags, which are created by SMs and used for inter-SM communication, synchronization, and interaction with the host.

Then we measured the round-trip times obtained for different data brick sizes in a one-hop communication network and with code cache at nodes. Results are reported in Table 4.4. These results can be directly compared with analogous experiments ran for Portable SM and the original SM platform on HP iPAQs; these can be found in [Ravi et al., 2004]. We observe that the overall performance obtained on Nokia Series 80 phones is worse compared to HP iPAQs. The additional overhead on Nokia Series 80 phones can be partially explained by the higher computation power of iPAQs and by the two different Java virtual machine implementations.

The break-up analysis for the RTT experiments shows that the cost of connection establishment accounts for less than 20 ms and the thread switching for less than 50 ms. The data transfer time varies linearly from 187 ms (with 1044 bytes) up to 278 ms (with 16010 bytes). In particular, Java serialization, which is based on

Table 4.3: Portable SM on Nokia 9500 phones: average latency of basic `TagSpace` operations.

TagSpace operation	Avg latency (μs)
readTag	88.33
createTag	127.44
writeTag	145.33
deleteTag	215.37

Table 4.4: Portable SM on Nokia 9500 phones: average round-trip time for different databrick sizes with cached code.

Databrick size (bytes)	Avg round-trip time (ms)
1044	343.36
2088	371.84
4056	411.02
8010	509.85
16010	679.21

slow reflection, is very heavyweight. For example, it consumes over a quarter of the total transfer time with 1044 bytes. Using XML instead of Java serialization would have allowed a similar flexibility without sacrificing efficiency [World Wide Web Consortium, 2007].

4.6.2 Latency experiments

Table 4.5 reports latency times for four main Contory operations, namely `createCxtItem`, `publishCxtItem`, `createCxtQuery`, and `getCxtItem`. The size of a context query object is 205 bytes, while the size of a context item varies from 53 bytes (a wind item) to 136 bytes (a location item). For these experiments, we used a `lightItem` whose size is 136 bytes. `CxtItem` and `cxtQuery` objects that are transmitted over UMTS network using the event-based platform are encapsulated in event notifications whose size is 1696 bytes.

Table 4.5: The average elapsed time of basic Contory operations of both publisher and consumer nodes.

Entity acts as:	**Operation**	**Elapsed time (ms) Avg [90% Conf. Interval]**
Context Publisher	createCxtItem	0.078 [0.001]
	adHocNetwork, BT-based - publishCxtItem	140.359 [0.337]
	adHocNetwork, WiFi-based - publishCxtItem	0.130 [0.006]
	extInfra, UMTS-based - publishCxtItem	772.728 [158.924]
Context Consumer	createCxtQuery	0.219 [0.001]
	adHocNetwork, BT-based, one hop - getCxtItem	31.830 [0.151]
	adHocNetwork, WiFi-based, one hop - getCxtItem	761.280 [28.940]
	adHocNetwork, WiFi-based, two hops - getCxtItem	1422.500 [60.001]
	extInfra, UMTS-based - getCxtItem	1473.000 [275.000]

On the context publisher side, publishing a context item using the BT-based implementation takes much longer than with the WiFi-based implementation. The reason behind this stems from the BT registering process. With BT, to make an item accessible, this needs to be encapsulated in a `DataElement` and registered into the BT `ServiceRecord`. With SM, this operation corresponds to simply creating a new SM tag and storing its name and value in the `TagSpace` hashtable. The variability of latency times for publishing a context item in the remote infrastructure is high and mainly due to the large delay variability in UMTS networks.

On the context consumer side, `adHocNetwork` provisioning can be BT-based or WiFi-based. For the BT case, the latency time reported in the table represents the time needed to receive a context item, once device discovery and service discovery have occurred.

The BT device discovery takes approximately 13 seconds and BT service discovery takes approximately 1.1 seconds. For the WiFi case, we ran experiments using a 2-hop topology with three communicators arranged in a line. The two latency times reported in the table represent the time needed to retrieve one context item located at a distance of one or two hops, once the route has been built. The additional time required to build the route is approximately twice the corresponding latency value in the table. The break-up analysis for SM experiments shows that connection establishment accounts for 4–5% of the total latency time, serialization for 26–33%, thread switching for 12–14%, and transfer time for 51–54%. Finally, measured latency times for `extInfra` vary quite a lot by ranging from 703 ms up to 2766 ms.

4.6.3 Energy consumption experiments

Energy consumption remains one of the most critical issues that needs to be addressed in application development on mobile phones. While CPU speed and storage capacity have increased over the last 10 years, battery energy shows the slowest trend in mobile computing [Paradiso and Starner, 2005].

To measure energy consumption on phones, we inserted a multimeter in series between the phone and its battery. Figure 4.9 shows the testbed setup. We used a Fluke 189 multimeter, which was connected to a PC to record the readings. The meter read current inputs approximately every 500 ms. The precision of our measurements depends mostly on the precision of the multimeter and the stability of the voltage on the phone battery (the resistance of the wires was found to be negligible). The multimeter has an accuracy of 0.75% and a precision of 0.15%.

The stability of the voltage is important since this is used to compute the power consumption based on Ohm's law. We did some preliminary experiment to measure the voltage on the phone while performing different operations. We found out that under high load the battery deviated less than 2% from 4.0965 V for the first hour at least. To minimize the impact of the voltage variance, we ran short experiments and always with a full battery. Given that the shunt voltage of the meter is 1.8 mV/mA, we calculated that the maximum inaccuracy of our experiments was approximately 8%.

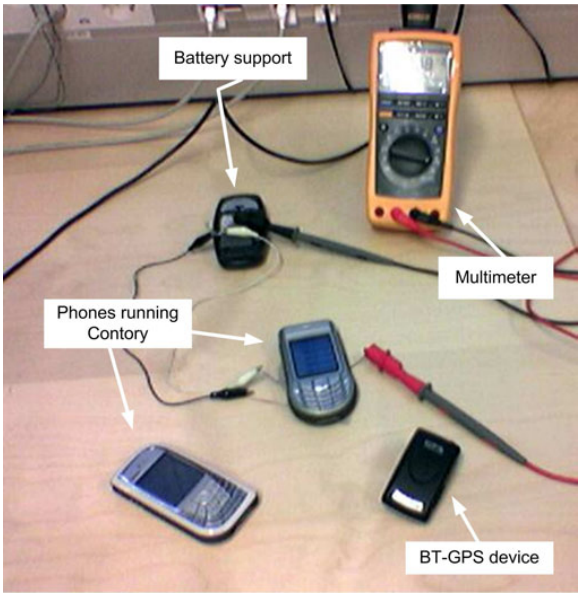

Figure 4.9: The power measurements testbed.

We ran the experiments in an office environment with background noise due to other mobile phones, wireless LANs, BT, etc. Even though a noise-free environment would have been desirable, we ran all experiments in the same spot, thus emulating a daily life scenario with an almost constant level of background noise.

All experiments were performed from five to ten times. High energy consuming experiments were set to last no longer than 10 minutes. All numbers hereafter reported were collected on a Nokia 6630 phone and a Nokia 9500 communicator (only when WiFi was used).

Initially, we measured the cost of different operating modes when the GSM radio was turned off. The relative average power consumptions are reported in Table 4.6. Power consumption varies from 5.75 mW, corresponding to the case in which BT is turned off, back-light is switched off, the display is switched off, and Contory is not running, up to 76.20 mW, corresponding to the case in which back-light and display are turned on.

We ran all Contory experiments, except those in which the UMTS

Table 4.6: The average power consumption of five reference tests on a Nokia 6630 phone.

Test	Avg power consumption (mW)
BT off, back-light off, display off, Contory off	5.75
BT off, back-light off, display on, Contory off	14.35
BT off, back-light on, display on, Contory off	76.20
BT on, back-light off, display off, Contory off	8.47
BT on, back-light off, display off, Contory on	10.11

radio was used, with the GSM radio off, back-light off, display off, and BT in page and inquiry scan state. This represents our "idle" reference case in which the power consumption is approximately 8.47 mW.

Table 4.7 reports energy consumption results for all three context provisioning mechanisms. On the publisher side, the energy consumption for processing a request for a published context item is relatively contained.

On the consumer side, we distinguish three cases. With BT-based mechanisms, the cost of executing a context query is mainly due to the device discovery phase which lasts approximately 13 seconds. Once the BT device is discovered, being periodically notified with context data is fast and the energy cost is definitely low. Results for `intSensor` were gathered by connecting the BT-GPS device to the phone. While the BT discovery cost is the same with both `intSensor` and `adHocNetwork` mechanism, the cost of maintaining a periodic exchange of data is higher with `intSensor`. This is due to the larger size of the exchanged data (GPS-NMEA data are 340 bytes big) and to BT's packet segmentation.

With WiFi-based provisioning, energy costs are much higher than in the BT cases. We encountered several problems in running these experiments. Each time a WiFi connection was established on the communicator inserted in the circuit, the communicator switched itself off after less than 30 seconds. New smart phones are low-voltage devices operating from a single Lithium-Ion cell. During the startup phase, the high in-rush current causes the phone's voltage supply to drop due to the multimeter's internal re-

Table 4.7: The average energy consumption of the three context provisioning mechanisms supported by Contory.

Context provisioning method: operation	**Energy consumption per cxtItem (J) Avg [90% Conf Interval]**
adHocNetwork, BT-based: provideCxtItem	0.133 [0.002]
adHocNetwork, BT-based: getCxtItem (one-hop and on-demand query, including discovery)	5.270 [0.010]
adHocNetwork, BT-based: getCxtItem (one hop and periodic query, without discovery)	0.099 [0.007]
intSensor, BT-based: getCxtItem (periodic query, without discovery)	0.422 [0.084]
adHocNetwork, WiFi-based: getCxtItem (one hop and periodic query)	$>$ 0.906 [a]
adHocNetwork, WiFi-based: getCxtItem (two hops and periodic query)	$>$ 1.693 [a]
extInfra, UMTS-based: getCxtItem (on-demand query)	14.076 [0.496]

[a] includes the cost of having back-light switched on

sistance; hence, this drop triggers the internal power management protection circuit to turn off the phone. However, based on the logs we gathered, having WiFi connected at full signal (with back light on) drains a constant current of 300 mA, which leads to an average power consumption of 1190 mW. This also means that having WiFi connected is more than 100 times more energy-consuming than having BT in inquiry mode.

In the tests using `extInfra` provisioning, turning on the GSM radio produces an additional power consumption, which comes in peaks of 450–481 mW and every 50–60 seconds. Figure 4.10 shows the power consumption measured in a test in which 5 queries were sent to the infrastructure over UMTS network, every 60 seconds. The high energy consumption is due especially to the cost for initializing the radio channel which causes 1 W peaks of consumption for several seconds.

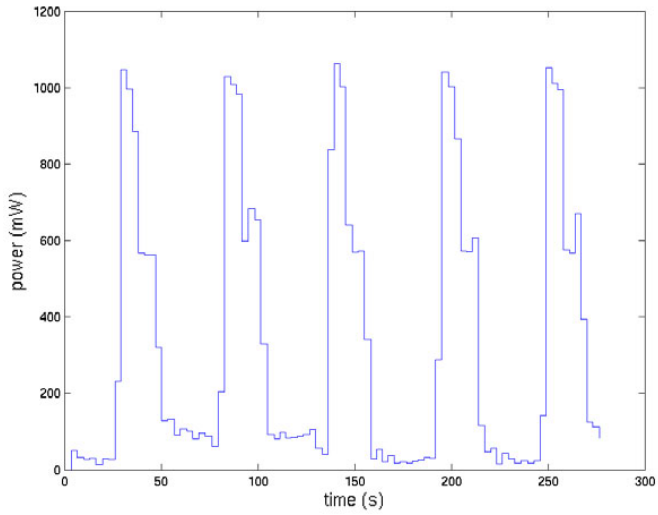

Figure 4.10: The power consumption of the `extInfra` provisioning strategy in a test with 5 queries.

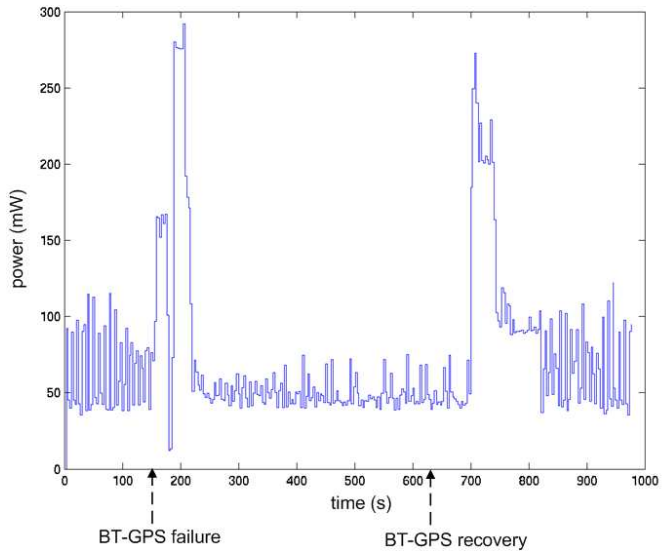

Figure 4.11: The power consumption of the BT-based `intSensor` and `adHocNetwork` strategies in the presence of a GPS failure.

Finally, we ran a test aimed to investigate how Contory is able to recover from sensor failures by dynamically switching from one context provisioning mechanism to another. In the test, the phone is initially connected to a GPS device via BT and receives location items every 10 seconds. After 155 seconds, we cause a GPS failure by manually switching off the GPS device. As a reaction, Contory switches from sensor-based provisioning to ad hoc provisioning and starts collecting location data from a neighboring phone, which is discovered and connected via BT. As Figure 4.11 shows, the switch occurs in correspondence of the first pair of power peaks due to the BT discovery of the nearby phone. Later on, we switch on the GPS device. Once Contory has realized that the GPS device is again available, it switches back to sensor-based provisioning. This occurs in correspondence of the second pair of power peaks. The power cost of the BT's device and service discovery varies approximately from 163 mW up to 292 mW.

4.6.4 Experiments summary

These experimental results confirmed the practical feasibility of our approach. The combined use of different context provisioning strategies can bring several benefits. First, it allows to cope with failures of sensing devices by dynamically replacing one context strategy with another. Second, as each context provisioning strategy guarantees different performance at different costs, the possibility of flexibly switching from one mechanism to another permits optimizing the utilization of computing and communication resources at run time.

The power experiments demonstrated how wireless communication can be very demanding in terms of power consumption. This leads to two important observations. First, communication connections should be kept open to the extent feasible and reused for further communication. For example, in the tests using UMTS networks, the power cost due to the radio channel initialization is very high. Second, the middleware should aggregate application messages to be sent simultaneously thus largely reducing the energy consumed per message.

Figure 4.12: Two screenshots of the WeatherWatcher application.

4.7 Application prototypes

Using Contory, we implemented several context-based applications. The use of Contory permitted decoupling the application implementation from the underlying communication modules (e.g., the BT JSR 82, the Fuego Core event-based framework, the SM platform), from the repository system, and from specific sensor interfaces. The implementation of common services such as connecting BT sensors or communicating with the remote repository was accomplished by simply instantiating context query objects in a few lines of code.

Moreover, Contory allowed to: *(i)* extend the application's context monitoring range by collecting region-specific observations using ad hoc networks and by making the sensor data available to remote clients through the infrastructure support; *(ii)* share context information about multiple entities and across multiple devices; and *(iii)* combine information from multiple context sources to enhance context estimation.

Using Contory, we re-implemented the DYNAMOS sailing application [Riva and Toivonen, 2007] described in Section 4.2. In addition, in the following, we present two other DYNAMOS-related services that we implemented and we give insights into their implementation.

The *WeatherWatcher* allows users to retrieve weather information relative to a certain geographical region of interest (e.g., the user wants to know the weather in the proximity of a guest harbor to visit). Figure 4.12 shows the screen interface for this applica-

```
 1 public void createWeatherQuery(Location location){

 2   FromCondition from = new FromCondition();
 3   from.setRangeKm(100);
 4   from.setLocationDestination(location);
 5   WhereCondition where = new WhereCondition();
 6   where.setAccuracy(false);
 7   where.setPrecision(false);

 8   CxtQuery q1 = new CxtQuery(CxtNames.HUMIDITY);
 9   q1.setSource(this);
10   q1.setFROM(from);
11   q1.setWHERE(where);
12   q1.setDURATION(10);
13   q1.setFRESHNESS(240);
14   q1.setEVERY(15);
15   Contory.getProviderFactory().processCxtQuery(q1);
16   issuedQueries.put(q1.getQueryID(),q1);

17   CxtQuery q2 = new CxtQuery(CxtNames.TEMPERATURE);
18   q2.setSource(this);
19   q2.setFROM(from);
20   q2.setWHERE(where);
21   q2.setDURATION(10);
22   q2.setFRESHNESS(240);
23   q2.setEVERY(15);
24   Contory.getProviderFactory().processCxtQuery(q2);
25   issuedQueries.put(q2.getQueryID(),q2);

26   CxtQuery q3 = new CxtQuery(CxtNames.LIGHT);
27   q3.setSource(this);
28   q3.setFROM(from);
29   q3.setWHERE(where);
30   q3.setDURATION(10);
31   q3.setFRESHNESS(240);
32   q3.setEVERY(15);
33   Contory.getProviderFactory().processCxtQuery(q3);
34   issuedQueries.put(q3.getQueryID(),q3);

35   CxtQuery q4 = new CxtQuery(CxtNames.WIND);
36   q4.setSource(this);
37   q4.setFROM(from);
38   q4.setWHERE(where);
39   q4.setDURATION(10);
40   q4.setFRESHNESS(240);
41   q4.setEVERY(15);
42   Contory.getProviderFactory().processCxtQuery(q4);
43   issuedQueries.put(q4.getQueryID(),q4);
44 }
```

Figure 4.13: Pseudo-code of the WeatherWatcher application.

Figure 4.14: Two screenshots of the RegattaClassifier application.

tion. Weather information consists of temperature, wind, speed, humidity, atmospheric pressure, etc. In a sailing scenario, weather conditions represent an important element for selecting the sailing route, but as this type of information can change very quickly, the information owned by boats currently sailing in such a region is often more reliable than the one provided by official weather stations. Once the user has issued a weather request, if the target region is not dense enough or too far away to support multi-hop ad hoc network communication, the query is sent to the remote infrastructure. The infrastructure checks if any WeatherWatcher of users currently sailing in that region has recently provided weather information and returns this information to the requester.

To illustrate how Contory can be used to program sensing applications, Figure 4.13 shows code excerpts of the implementation of the `createWeatherQuery` method of the WeatherWatcher application. This method takes in input the location where the weather conditions must be monitored. Four queries are specified using the Contory API and submitted to the `ContextFactory` (e.g., see line 15).

To execute these queries, Contory first compares the location specified in the query (`location`) and the current location of the user. If weather conditions must be monitored in the proximity of the user, the `adHocNetwork` mechanism is preferred. If the query must be executed in regions at a great distance, `extInfra` is the preferred method. Once the query results are collected by the application they are combined to estimate the weather conditions.

The other service that we implemented using Contory is the *RegattaClassifier*. During a regatta competition, this service constantly provides an updated classification of the current winner of the regatta. Virtual checkpoints can be arranged along the route that the boats will take during the competition. Each time a boat reaches a checkpoint, the RegattaClassifier running on the phone's participant (see Figure 4.14) communicates location and speed of the boat (collected using GPS sensors) to the infrastructure. The infrastructure processes this information and provides each participant with an updated classification and additional statistics about the competition.

4.8 Concluding remarks

This chapter described Contory, a middleware specifically designed to enable simple and quick development of sensing applications on smart phones. Contory provides flexibility in supporting context provisioning by integrating several context strategies, namely internal sensor-based, infrastructure-based, and distributed provisioning in ad hoc networks. Additionally, Contory offers a unified SQL-like interface for specifying context queries. Contory allows sensing applications to collect context information from different sources without the need to uniquely and continuously rely on their own sensors or on the presence of external infrastructures. We implemented Contory on smart phone platforms. To assess system performance and quantify the energy consumption on smart phones, we ran experiments in a testbed of Nokia Series 60 and Nokia Series 80 phones. Furthermore, to evaluate the practical feasibility of the proposed approach, we built prototype applications using Contory.

CHAPTER 5

Context-aware Migratory Services

A well-understood and simple programming model is the client-service model. Having a model like this available to program Urbanet environments could definitely facilitate the application development process. Services running in Urbanets can exploit the temporary and unstable network support to acquire real-time information from nodes located in the immediate proximity of geographical regions, entities, or activities of interest. A client interested in such a type of information could, for example, issue a request to the adequate service and receive period observations over a certain period of time. Building services in highly volatile environments like Urbanets and maintaining long-running client-service interactions is, however, challenging especially due to the rapidly changing operating contexts, which often lead to situations where a node hosting a certain service becomes unsuitable for hosting any longer the service execution.

This chapter presents a novel model of client-service interaction based on the concept of context-aware migratory service. Unlike a regular service that executes always on the same node, a context-aware migratory service can migrate to different nodes in the network in order to accomplish its task. We describe requirements, design, and implementation of the Context-aware Migratory Services framework together with its performance evaluation and prototype application.

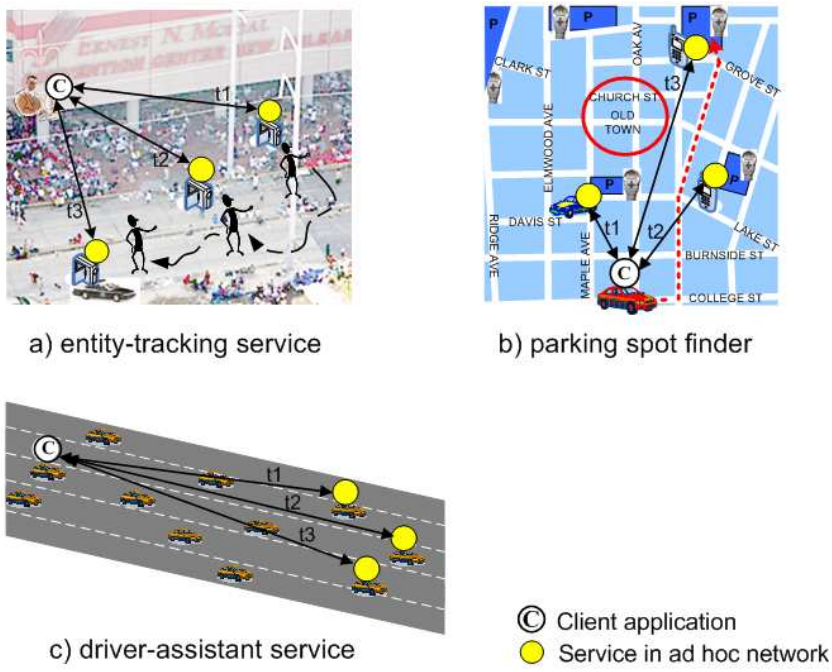

Figure 5.1: Examples of client-service interactions in Urbanets.

5.1 Motivating scenarios

We assume that nodes in Urbanets are willing to collaborate. Some nodes offer services, others host client applications, and the rest cooperate to provide service discovery and routing of messages. Deploying services in Urbanets proves challenging due to the dynamism, in particular mobility, 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 (e.g., a moving object being tracked).

Partly based on the examples of Urbanet applications presented in Chapter 3, Figure 5.1 proposes three scenarios illustrating how nodes hosting a service answering a specific client request can over time stop satisfying the client request due to certain context changes. 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 requirements of the client exists. But 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 (e.g., people, 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: *(i)* the node where the service executes is mobile and might move away from the tracked entity, *(ii)* likewise, the tracked entity is mobile and might move away from the sensing range of the service node. For example, Figure 5.1a shows how the client needs to interact with three different tracking services over a short period of time. Besides the time spent on carrying out service discovery several 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 spots 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 her request, the user can specify the destination of interest and her current location, along with other preferences like cost and security of parking spots. 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 spot" response. Upon receiving an "unavailable spot" response, a user will need to discover another service in the destination area, as Figure 5.1b 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 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 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 (e.g., 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 *(i)* the service needs to constantly execute in the region of interest to the user; *(ii)* such a user-defined region changes over time according to the user's movement and speed on the road; *(iii)* the cars located in the region of interest change over time due to their mobility (e.g., cars can leave the highway, stop, slow down). As shown in Figure 5.1c, due to these reasons, the client occasionally needs to re-establish the interaction with a new service that can meet the user's requirements.

5.2 Requirements for services in ad hoc networks

By analyzing these scenarios, we identified four requirements that services running in Urbanet environments 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 (e.g, location, activity, terminal equipment). As the user operates in a highly dynamic environment, 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 largely 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 re-start its interaction with a new service, but the entire state of the old interaction is lost. While this approach is acceptable for a stateless interaction, it can lead to 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 preferences and destination of the user. 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 Urbanet node will possess the code for every type of service. For example, the code for 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 (i.e., the code could be transferred from other nodes in the ad hoc network or even from the Internet if possible).

Even though all these requirements may not be necessary for every single Urbanet application, since we want to develop a framework that can support a large variety of applications, we need to address all of them.

5.3 Migratory Services 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*. Intuitively, migratory services are capable of migrating to different nodes in the network in order to effectively accomplish their function. They execute on a certain node as long as they are able to provide semantically acceptable results using the available resources; when this is not possible anymore, they migrate through the network until they find a new node where they 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 Services 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 entire 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

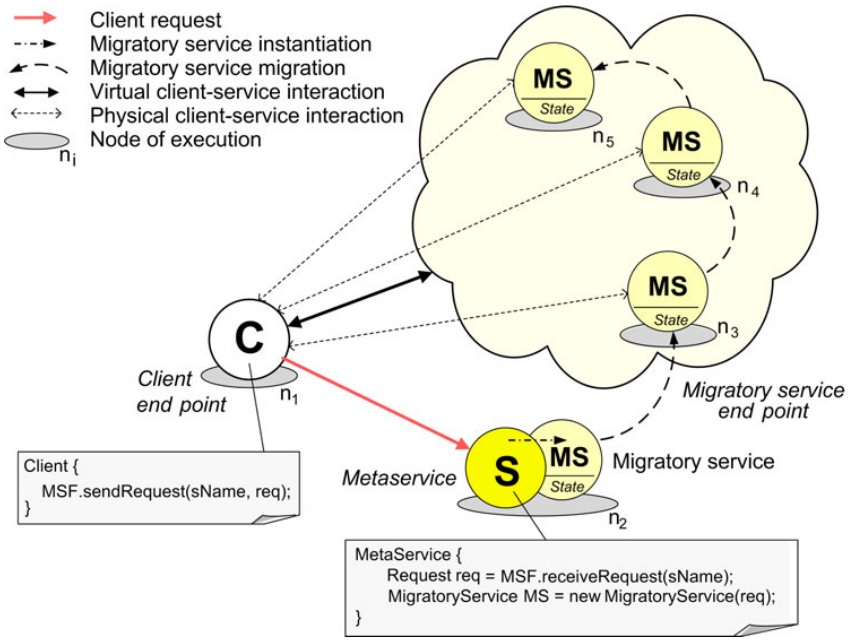

Figure 5.2: Example of Migratory Services execution: a metaservice instantiates a migratory service that migrates in the network to satisfy the client request.

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.

Figure 5.2 depicts a typical interaction in the Migratory Services 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 only 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

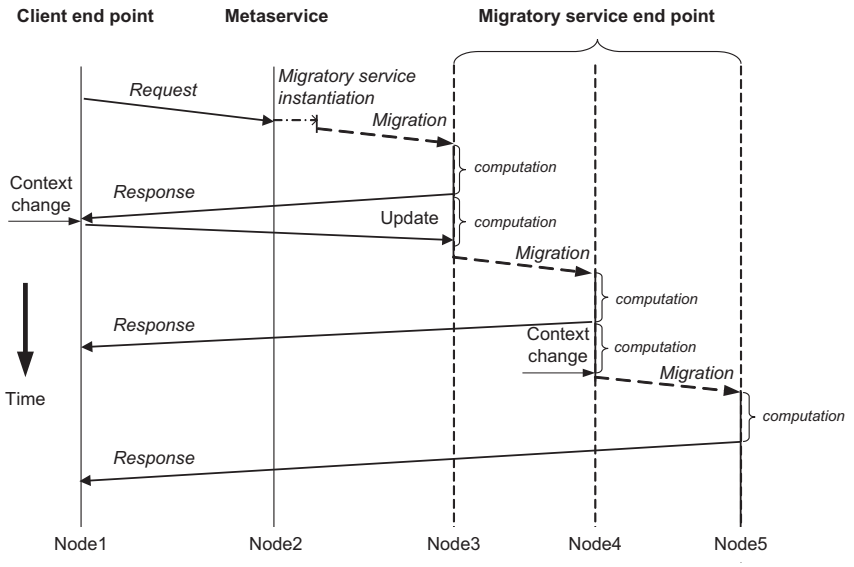

Figure 5.3: Sequence diagram of Migratory Services illustrating three context-aware service migrations.

application. Upon the creation of a migratory service, the client application ceases communication with the metaservice and continues to interact solely with its associated migratory service.

In the extreme case, client and meta service can also run on the same node. This corresponds to the case in which the client holds the service specification and can autonomously generate the migratory service that will flood the network.

Our client-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 need to monitor entities or activities in real-time and report their observations periodically. Therefore, the service interaction consists of two main operations: *(i)* the migratory service sends responses to the client application, and *(ii)* the client application sends "request updates" each time the user's context changes beyond relevant thresholds (i.e., in fact, these updates are sent by the runtime system at the client node).

These concepts are exemplified in Figure 5.3. Upon being instantiated on *Node2*, a driver-assistant migratory service changes node

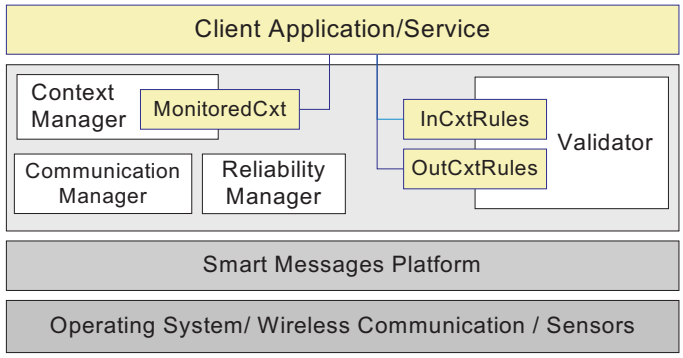

Figure 5.4: The Migratory Services framework.

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, while the second one could be due to the fact that *Node4* has left the region of interest.

5.4 Migratory Services framework

To support the Migratory Services model, we designed a common framework which runs on all Urbanet nodes willing to cooperate. A main challenge that hinders the practical development of applications and services in Urbanet environments is the difficulty of writing software for distributed systems with such complex requirements. Although building applications and services from scratch could lead to 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 new applications and services more quickly.

Figure 5.4 illustrates the system architecture for the Migratory Services framework. At the lower layer, the Smart Messages (SM) computing platform provides support for execution migration, nam-

ing, routing, and security. On top of the SM layer, we built a higher layer providing support for

- context provisioning and monitoring,
- context rules creation and validation,
- client-service communication, and
- service reliability.

The Migratory Services framework can be logically divided into two functional planes: data plane and control plane. Data traffic, routing, and migration form the data plane and build mainly on basic functions offered by SM. The Communication Manager is the corresponding module for 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 the requirements of the application and the service and decides how to act upon those changes through the Communication Manager. 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 described in Section 5.4.5, 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 and the service layer. Ultimately, the framework maps these programs onto lower-level SMs.

5.4.1 Context provisioning and monitoring

The SM platform provides several types of context data accessible through specific I/O tags (see Section 2.4 for the definition of I/O tags). Commonly, the SM platform provides context information such as location, speed, and time by means of GPS devices, and various system status information such as amount of available

memory or 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-level 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 Services 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 for those context parameters by polling or blocking on the corresponding SM tags.

5.4.2 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 as *CxtRules*. These rules specify policies, filters, and preferences ruling the system's behaviour. For instance, they can define how the node 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* are used to validate the correctness of incoming data. *(i)* A metaservice utilizes InCxtRules to decide whether to accept or 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. *(ii)* A client application utilizes InCxtRules to decide whether to accept or 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* 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.

Similarly to the context rules defined for the Contory middleware (see Section 4.4.3 for more details), context rules are expressed in the form of condition and 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`. A combinationNode combines two nodes by means of combinationOperators such as `and` and `or`. Actions currently supported are: `migrate`, `send update`, `accept response`, `refuse response`, `accept request`, and `refuse request`.

The entire condition clause of a certain 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 be always available (e.g., 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 [Capra et al., 2003], 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 (i.e., 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.

5.4.3 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 directly the communicating programs. Due to the fact that these programs are ultimately converted into SMs, we enforce the naming conventions defined by the SM platform [Kang et al., 2004]. More exactly, tag names are used to uniquely identify the communication end points in the Migratory Services model. Each time a migratory service moves between two nodes, its name is removed from the old node and re-created on the new node.

In the SM platform, service discovery and routing are integrated in a single module that performs content-based routing [Borcea et al., 2003]. 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 GPSR [Karp and Kung, 2000]. 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 AODV [Perkins and Royer, 1999]) 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 migra-

tory 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. The Communication Manager does not guarantee reordering of out of order messages or recovery of lost messages. But each exchanged message is identified by a sequence number which is accessible to the client (or service); hence, it is up to the client (or 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, re-binds the service to this node (i.e., 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 loss 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.

5.4.4 Service reliability

We believe that service reliability represents an essential factor in turning Urbanets into distributed service providers. Compared to traditional distributed computing systems where servers work correctly for long periods of time, Urbanets are more challenging in terms of reliability because communication, software, and hardware faults occur frequently and can thus render service provisioning unfeasible. Despite node mobility and limited resources, client applications demand stable interactions with services. In particular, the lack of reliability can seriously affect long-running stateful operations such as those targeted by Migratory Services. While numerous approaches [Cristian, 1991] have been proposed to provide reliability in distributed systems, similar mechanisms have not yet appeared in ad hoc networks. The only fault-tolerance issues stud-

ied so far in such networks are at the network layer [Chandra et al., 2001; Luo et al., 2004].

In the basic implementation of the Migratory Services framework, a migratory service provides implicit fault-tolerance to mobility and "predictable failures" of nodes. Each time a node becomes unsuitable for hosting a service (e.g., the node has moved away from a region of interest or it is running out of battery), the service can autonomously migrate to another node where it can compute semantically correct results. However, to provide reliable migratory services, in the presence of unpredictable service failures occurring when the hosting node crashes, is abruptly turned off, or loses connectivity with the network we have developed specialized reliability support. Our reliability model is based on the classical primary-backup approach [Budhiraja et al., 1993], in which one primary service interacts with the clients and one replica is stored on a secondary node. This technique is applied in a unique way to Migratory Services through two modifications: *(i)* the secondary node is dynamically selected at runtime, in a context-aware manner, and *(ii)* the backup frequency is constantly adapted according to the operating network conditions. This context-aware, adaptive solution allows the system to make dynamic decisions to constantly trade off reliability performance and induced overhead.

This reliability extension to the basic framework is optional due to the extra-load it induces in the network. The *Reliability Managers* at the nodes where the migratory service executes are the modules in charge of accomplishing reliability. The design and implementation of our service reliability support will be described in more detail in Section 5.5.

5.4.5 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 different 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 interacting with migratory services. This API shields the programmer from the underlying SM platform and the networking aspects.

Figure 5.5 shows code excerpts of a typical client application,

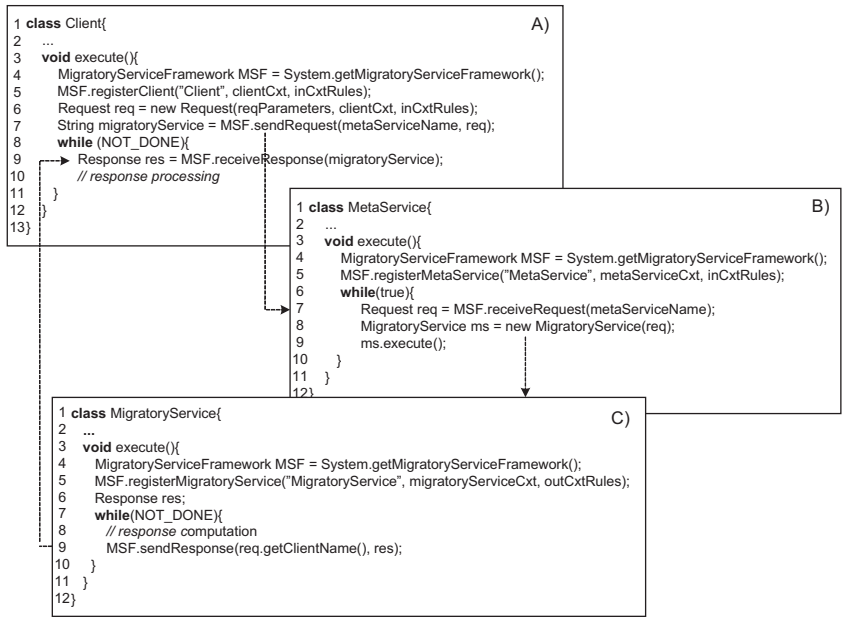

Figure 5.5: Pseudo-code of a typical client application (A), metaservice (B), and migratory service (C).

metaservice, and migratory service together with their interactions. To make the entire process of service migration transparent, each of these three entities has to register with the Migratory Services framework. At the registration invocation, the entities pass their class name, self-reference, name of context parameters to be monitored, and context rules to be evaluated at runtime. Class name and self-reference are needed by the framework to create underlying SMs associated with each entity. The class code will be the first code brick of the SM, while the self-reference (i.e., *this*) will be the first data brick. Besides these code and data bricks, the associated SM includes all the necessary framework-level code bricks and data bricks (e.g., multi-hop service migration and data delivery). 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 pro-

gramming model, the framework provides a similar API. All communication primitives are synchronous. Metaservices run in a loop and block waiting for client requests (Figure 5.5B, line 6 and 7). At the SM layer, this corresponds to an SM blocking on a tag. When a client application sends a request to a metaservice (Figure 5.5A, line 7), the SM tag is appended with the request parameters, and the metaservice is unblocked to receive the request. If the request is accepted, the metaservice instantiates a new migratory service that will be in charge of processing the client request (Figure 5.5B, line 8 and 9).

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 (Figure 5.5C, line 9) and the Communication Manager invokes the Reliability Manager to update the state on the secondary node. If the validation fails, the Communication Manager de-registers the service (removes its unique SM tag) and invokes an SM migration to transfer the service to a suitable node. Once 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 (Figure 5.5A, line 9); only responses that are positively validated by the InCxtRules are returned to the application.

5.5 Fault-tolerance support

This section presents the failure model, the backup process, and the service recovery protocols. We do not assume supervising entities in Urbanets, therefore node failures are detected by mutual monitoring of the nodes, and recovery is achieved collaboratively.

5.5.1 Failure model

In general, a service is correct if, in response to a request, it behaves in a manner consistent to its specification. We assume that

this specification prescribes that the service has to deliver responses that are semantically correct and produced within certain time intervals. A *service failure* occurs when the service does not behave according to its specification. Essentially, a service failure can occur due to transient or permanent failures of its components and communication links. Nodes in Urbanets can crash due to malfunctioning, resource exhaustion, and hardware errors or be deliberately switched off (e.g., to save battery or to reboot). Additionally, communication failures can occur due to mobility.

From a client's perspective, several types of service failures can occur [Cristian, 1991]. An *omission failure* occurs when the service omits to respond to a request. A *timing* or *performance failure* occurs when the service responds outside the time interval specified. A *response failure* occurs when the service response is semantically incorrect. In reaction to omission and timing failures, the client application re-sends its request. If after a certain number of attempts, the service omits to produce responses, the service is said to suffer a *crash failure*. As for response failures, the Migratory Services framework handles them in a transparent way to the client application. A migratory service constantly verifies the correctness of the produced result before sending it to the client. In the case the framework at the client node receives a wrong result, this is deleted and a message is automatically sent to the migratory service to update the request parameters.

In our work, we have focused on permanent *crash failures*. However, crash failures that are due to errors or packet losses typical of wireless communication channels are out of scope of our work. We assume that such failures are handled at the network layer.

5.5.2 Service backup

To attain protection against the failures mentioned above, and thus allow a client to receive an uninterrupted service, the Migratory Services framework adopts a primary-backup approach [Budhiraja et al., 1993]. The running migratory service, called *primary service*, relies on the existence of one backup service, called *secondary service*. The framework at the primary node creates such a backup service, moves it to a secondary node, and periodically *checkpoints* the state of the primary service on such a node. The secondary

service is inactive as long as the interaction between the primary service and the client functions correctly. If the primary service fails, a *failover* process occurs, and the secondary service takes over the service execution. This fault-tolerant extension of the service is completely transparent to the client application, which is still provided with the illusion of a single service node.

The classical primary-backup approach is applied to Migratory Services through two modifications:

1. context-aware selection of the secondary node and
2. adaptive checkpointing frequency.

In the rest of this section, we describe these techniques.

Context-aware selection of the secondary node

Fault-tolerance comes with the cost of the periodic state checkpointing process. This cost is mostly a communication cost and is proportional to *(i)* the physical distance between the primary and secondary node (i.e., the communication latency is implicitly proportional to this distance), *(ii)* the size of the service state, and *(iii)* the frequency of the state checkpointing process. As it is difficult to modify the size of the service state, the checkpointing overhead can be reduced by selecting a secondary node close to the primary node and by decreasing the checkpointing frequency. However, selecting a secondary node close to the primary node is not always beneficial in terms of the reliability that can be achieved, especially if the client node is far from primary and secondary.

We need to consider a trade-off between the desire to minimize the cost of the checkpointing process by selecting a close secondary and the reliability level that can be guaranteed. On the one hand, the secondary node should be close to the primary node in order to minimize the communication overhead in checkpointing. On the other hand, it should be close to the client node in order to improve the chances of a failover and to minimize the failover latency. In the extreme case, the secondary service can reside on the client node.

In general, a secondary node close to the client is preferred for highly critical applications that demand high reliability and fast recovery. However, when the service state is relatively large or

the checkpointing frequency is high, a secondary node close to the primary is preferred (e.g., in a tracking application, a service state consisting of large image files should be saved on the closest nodes).

Based on these considerations, the ideal distance d between the primary and the secondary is computed as follows:

$$d_f = \begin{cases} d_{PC} & \text{if } f \leq f_{min} \\ 0 & \text{if } f \geq f_{max} \\ d_{PC} \cdot \frac{f - f_{max}}{f_{min} - f_{max}} & \text{otherwise} \end{cases}$$

$$d_s = \begin{cases} d_{PC} & \text{if } s \leq s_{min} \\ 0 & \text{if } s \geq s_{max} \\ d_{PC} \cdot \frac{s - s_{max}}{s_{min} - s_{max}} & \text{otherwise} \end{cases}$$

$$d = \alpha \cdot d_f + (1 - \alpha) \cdot d_s \qquad \alpha = 1/2$$

The distance d is computed by a weighted combination of d_f and d_s. d_f is the ideal distance between the primary and the secondary based on the checkpointing frequency f. As f varies from f_{min} to f_{max}, d_f varies linearly from d_{PC} to 0, where d_{PC} is the distance between the primary and the client. Below the lower threshold f_{min}, d_f is equal to d_{PC}, meaning that the checkpointing frequency is so low that the client can act as secondary. Above the upper threshold f_{max}, d_{PC} is 0, meaning that the checkpointing frequency is so high that the secondary should be located on the primary node. Likewise, d_s is the ideal distance between the primary and the secondary based on the service state size s. It is computed in a similar way to d_f with s_{min} being the minimum threshold and s_{max} being the maximum threshold. Minimum and maximum thresholds of the checkpointing frequency and service state size are fixed by the framework.

However, selecting a secondary at the ideal distance from the primary does not necessarily mean to have found the best possible secondary node. Other context parameters have to be considered in making this decision.

The Migratory Services framework takes into account other *context* information such as available resources and mobility traces of the candidate secondary nodes. To accomplish this goal, the primary node could monitor the context of all nodes located in the

routing path to the client within the ideal distance d and thereby select the most appropriate secondary node among them. However, this monitoring process would turn out to be too expensive and not scalable. Instead, we adopt an on-demand solution in which the evaluation is distributed over the network.

First, the primary node computes the ideal position of the secondary node on the routing path between itself and the client using the previous formulas. Then, it broadcasts a `SEC_DISCOVERY` message in the network. This message contains current location and mobility traces of the primary, location of the client, and a list of N requirements that the secondary node should satisfy. These requirements are specified in a vector of conditions $< c_1, ..., c_N >$, where each condition expresses lower bounds on needed resources, such as minimum CPU, memory required, and communication capabilities. The `SEC_DISCOVERY` message is broadcasted in a geographical region of range r centered on the ideal position of the secondary. r cannot be lower than r_{min} and is computed in the following way:

$$r = max\{r_{min}, min\{2d, 2(d_{PC} - d)\}\} \qquad r_{min} = 500 \ m$$

Upon receiving a `SEC_DISCOVERY`, each node i locally evaluates if its characteristics match the requirements specified in the message and computes the following matching score:

$$score_i = Match(mob_i, mob_P) + \frac{1}{N} \cdot \sum_{j=1}^{N} c_j(p_{i,j})$$

This filtering aims to select the best secondary node among the available nodes. Since we consider mobile nodes, reliability and quality of the communication can highly improve if the primary and secondary nodes constantly move in the same direction and at similar speeds. The $Match$ function matches the mobility traces of the candidate node i (mob_i) and of the primary (mob_P) and returns highest scores for similar mobility profiles. Each trace contains the location coordinates (as $< latitude, longitude >$) of the node over the last N minutes, at every m seconds. For example, this kind of traces can be easily provided by GPS receivers. $Match$ computes the average distance between the positions of the two nodes over

the last N minutes; if the result is below a certain threshold, it returns 1, otherwise it returns 0.

The second part of the score computation considers the device profile (e.g., CPU, memory, and communication protocols). Reliable secondary nodes with higher capabilities are desirable because they are likely to be available for longer periods of time and be effective in promptly resuming the service execution when a fault occurs. Each condition c_j is evaluated against the corresponding profile parameter $p_{i,j}$ of the node i; if positively verified, 1 is returned, otherwise 0. Nodes that reach a *score* that is above a certain fixed threshold reply to the primary with their *score* value and profile information. The primary can then elect its secondary node.

Adaptive checkpointing frequency

Once a secondary node is selected, `STATE_UPDATE` messages between primary and secondary node are coordinated so that if the primary service fails the service state is available on the secondary node. The service state consists of service data and execution control state.

Instead of a fixed checkpointing frequency, the Migratory Services framework dynamically updates it. In the current design, the framework triggers a checkpoint of the service state every R responses sent to the client. This choice ensures that every checkpointed state contains information "relevant enough" to be saved. If R is high, the checkpointing overhead is reduced, but with the risk of having an inconsistent state on the secondary. If R is low the overhead increases, but the reliability improves.

Every `STATE_UPDATE` message must be acknowledged by the secondary node. Acknowledgments serve two purposes. First, they confirm the reception of the update message. Second, they carry control information such as distance in number of hops between primary and secondary and other status information of the secondary (e.g., remaining battery power, free memory, and status of network connectivity). Using this control information, the primary can decide to undertake a new secondary selection process. For example, if the secondary node has moved too far away or it does not have many resources left to act as reliable backup, a more suitable secondary node should be contacted.

5.5.3 Service recovery

Failures of the primary service are detected by using timeout mechanisms. In addition, timeouts are used also to detect disconnections between the primary and the secondary. To adapt to the changing network conditions, timeouts are constantly updated, but always in a range between a minimum and maximum value set by the framework at the beginning of the interaction. Ideally, only persistent disconnections or node crashes should trigger the replacement of the primary service with the backup service. Therefore, timeouts should be set in such a way to quickly detect permanent failures and tolerate transient failures.

When a failure of the primary service is detected, the backup service must take over the interaction. The recovery process involves activating the backup service, which should resume the interaction with the client in a short time. The recovery can be a push or pull process. In *pull recovery*, the client assumes that the primary has crashed when it stops receiving answers from it. After K failed attempts to reconnect to the primary, the client contacts the secondary. The client can retrieve the secondary identifier through the Migratory Services framework and use SM routing functions to communicate with it. The parameter K is application-specific and reflects the capacity of the application to tolerate long periods of disconnection from the service. In general, pull recovery particularly suits monitoring services that periodically compute and send results.

Push recovery is specific to situations where the client cannot tell if a fault has happened or not. For instance, this is the case of services that send responses only when certain events occur. In this case, the backup service needs to detect the failure, activate the recovery, and resume the interaction with the client. The backup service assumes a primary fault when it stops receiving its update messages.

Pull recovery

The first three examples in Figure 5.6 show three pull recovery scenarios. These examples model a typical "one request, multiple responses" interaction, in which responses are periodically sent to

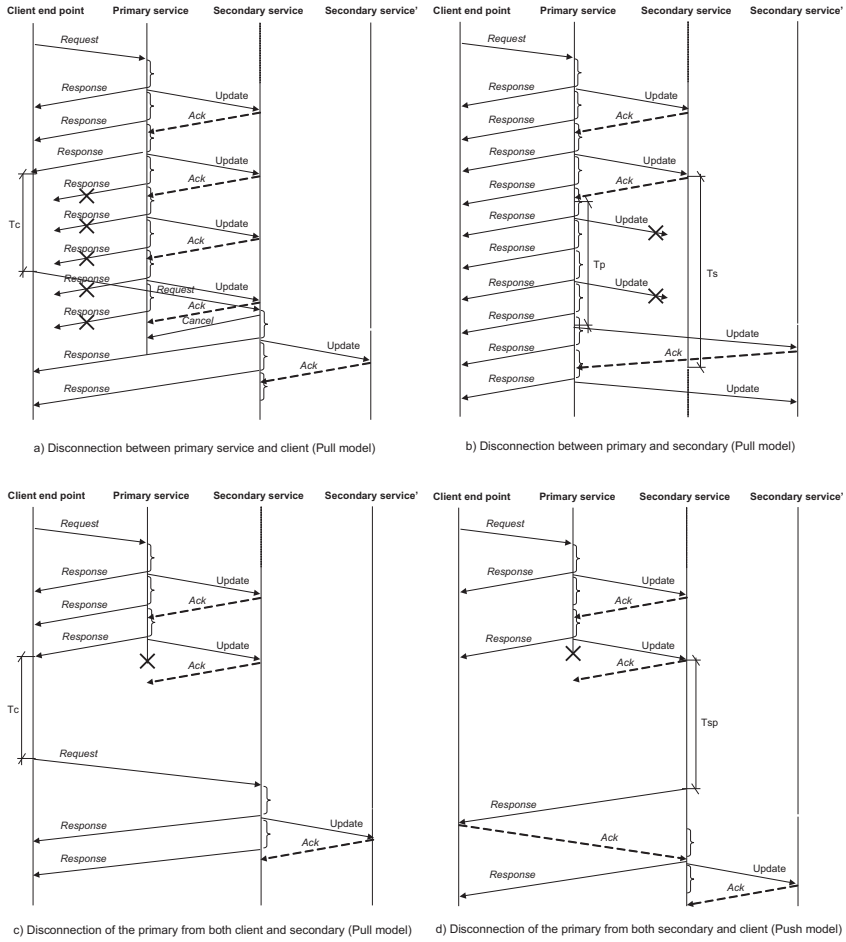

Figure 5.6: Sequence diagrams illustrating four Migratory Services failure scenarios.

the client.

In the first scenario, a disconnection between the client and the primary occurs, but the primary and the secondary continue communicating. For example, this could happen due to a network partitioning that isolates the primary node from the client node. The client does not receive responses and will then timeout. The timeout Tc is set to be N times the estimated RTT (round trip time) between client and primary. The RTT is constantly updated based on the jitter of the observed interresponse time. Since the timeout has expired, the framework at the client believes that the primary has crashed and thus re-sends the service request to the secondary node. The framework knows the identifier of the secondary node because this can always be uniquely determined by knowing the identifiers of the primary service and the client. The backup service resumes the computation and starts sending responses to the client. Furthermore, it sends a `CANCEL` message to the primary and selects a new secondary node for backup.

The second scenario shows the case in which a disconnection between the primary and the secondary occurs. At some point, the primary stops receiving update acknowledgments from the secondary. This could happen because the secondary node has crashed or has moved away. The primary times out and re-selects a new secondary node. Similarly to the previous case, the timeout Tp is computed and updated based on the observed primary-secondary RTT. The secondary will timeout as well and erase the stored data. In the case of a temporary disconnection of the secondary, it could happen that the second lost update that is shown in the figure manages to reach the secondary node and therefore triggers an acknowledgment. If this acknowledgment arrives after Tp has expired and a new secondary has already been selected, the primary will ignore it; the old secondary will not receive new update messages and therefore will timeout. However, the primary will increase its Tp of 1 RTT since it is likely that other transient failures will occur again. Tp will then be decreased again if no other transient failures occur in the next M periods.

In the third scenario, the primary service crashes and disconnects from both the client and secondary. The client times out and contacts the secondary. Consequently, the secondary assumes the role of the primary, resumes the interaction, and selects a new

secondary node. In this case, we need to guarantee that the client times out before the secondary, otherwise, the service state might get lost. Therefore, it is recommendable to always choose a Tc larger than Ts and set the minimum Tc larger than the minimum Ts. Another possibility, not shown in the figure, is that the primary service will reappear after some time by sending a new response to the client and a new update to the secondary. In this case, both the client and the secondary will reply to the primary with a `CANCEL` message. Additionally, the client increases its Tc of 1 RTT.

Push recovery

Push recovery is utilized in those cases where the client has no simple means of detecting a disconnection of the primary service. An example is that of an event-based interaction. The service is supposed to send answers only when a certain event occurs. Hence, after a disconnection of the primary, no timeout will expire at the client side. In this case, as shown in the last example of Figure 5.6, the disconnection must be detected by the secondary, after T_{sp} has expired. The primary periodically checkpoints its state on the secondary and therefore the secondary can realize when the primary has disconnected and subsequently contact the client. If the client believes that the primary is still active (i.e., the primary has probably disconnected only from the secondary), no reply is sent to the secondary that will timeout and remove the service state. Otherwise, the client will reply by asking the secondary to take over the execution. If the primary appears again after some time, both the client and secondary will reply with a `CANCEL` message and increase their timeouts by 1 RTT.

5.5.4 Reliability Manager implementation

Figure 5.7 shows the core software modules of the Reliability Manager and the interactions that take place between them during the backup and recovery process. The primary service generates an SM called `SecondaryDiscoverySM` to discover a qualified secondary node. This SM is broadcasted to all candidate secondary nodes located in the geographical region of interest, which was generated by the primary's framework as previously explained. Instances of

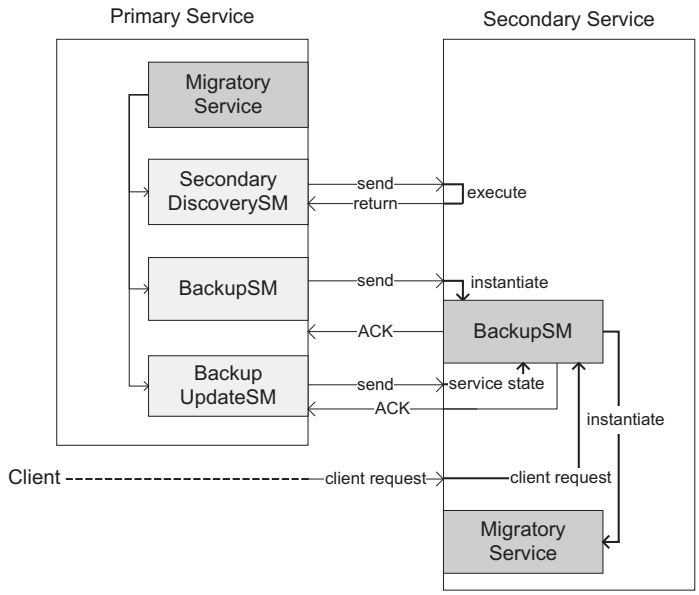

Figure 5.7: Software modules and interactions of the Migratory Services frameworks at the primary and secondary node.

this SM verify whether any node in the region meets the specified requirements, and migrate back to the primary when an appropriate secondary is found.

Based on the `SecondaryDiscoverySM`'s outcome, a secondary node is elected. Then, the primary service generates a `BackupSM` that migrates to the discovered secondary. The `BackupSM` carries the code bricks and data bricks necessary to instantiate and execute the migratory service on such a node. The code of the migratory service is cached at the secondary node. Similarly, the routing code is cached at intermediate nodes to improve the performance.

The `BackupSM` constantly blocks on `update` and `request` tags. The `update` tags are used to receive state updates from the primary. These updates are encapsulated in `BackupUpdateSMs`. These SMs migrate to the secondary node and unblock the `BackupSM` by writing an `update` tag containing the new service state. The `BackupSM` reads the tag, updates its state data, and acknowledges the reception of the message.

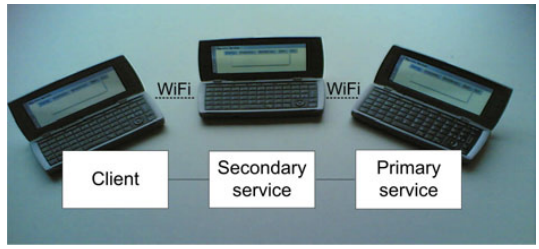

Figure 5.8: The Migratory Services experimental testbed.

The `request` tags are used by the client framework to trigger recovery upon failure of the primary. A `BackupRequestSM` (not shown in the figure) migrates to the secondary and writes a `request` tag containing the request parameters and the sequence number of the last response received by the primary. The `BackupSM` unblocks and instantiates a new `MigratoryService` that will take over the execution.

5.6 Experimental evaluation

Two versions of the Migratory Services framework exist. The first one used the original SM architecture implemented by modifying Sun Microsystem's KVM (See Section 2.4 for more details) and it was tested on HP iPAQs. In a second phase, the framework was extended to run on J2ME CDC devices by using Portable SM [Ravi et al., 2004] and it was tested on Nokia Series 80 phones. The Migratory Services's implementation on J2ME CDC is available under an open source license [Migratory Services, 2007].

To evaluate the Migratory Services framework, we ran experiments in a testbed consisting of three Nokia 9500 phones arranged in a line and communicating using IEEE 802.11b in ad hoc mode. Nokia 9500 phones run Symbian OS 7.0s, have an ARM processor at 150 MHz, and offer 76 MB of RAM. The experimental testbed is shown in Figure 5.8.

The experimental analysis is organized in three parts. First, we measured the latency of the context rules validation process, then, the cost of reliability, and, finally, the memory consumption of core

components of the framework. The cost of migrating services of different data sizes has been described in Section 4.6.1 and additional measurements assessing the cost of service migration and service resumption will be presented in the following, when evaluating the performance of the reliability support. To complete our analysis, in the next section, we will also present the experimental evaluation of a proof-of concept migratory service that we built using Migratory Services.

5.6.1 Context rules validation experiments

The latency cost of the entire context-aware migration process at the basis of Migratory Services includes the cost of validating the service's context rules, migrating and registering the service with the framework at the new node, and finally resuming its execution.

Table 5.1 presents the average latency for validating different sets of context rules. Context rules used in the experiments are comparison rules (see Section 4.4.3 for the definition of context rule) that verify, for example, if the level of memory and power consumption is acceptable or if the node is still located in the region of interest. Results show that the validation latency increases linearly with the number of rules. Based on the results, the cost of validating one context rule is approximately 530 μs.

Table 5.1: The average latency for validating an increasing number of context rules.

Number of rules	**Avg rule validation latency (μs) [90% Conf. Interval]**
10	5281.84 [1.75]
20	10642.01 [2.23]
30	15850.36 [2.99]
40	21573.43 [3.59]
50	26156.45 [3.56]

5.6.2 Reliability experiments

To understand the feasibility of the reliability support in Migratory Services, we assessed overhead and recovery latency of our prototype implementation.

We first measured the latency of the checkpointing process on the secondary node. This latency includes the time necessary for extracting the service state, saving it in the `BackupUpdateSM`, migrating the `BackupUpdateSM` to the secondary node, storing the new service state in the `BackupSM` running on the framework at the secondary node, and finally sending an acknowledgment carrying control information to the primary service. We measured the elapsed time from the beginning of the checkpointing process until the reception of the secondary's acknowledgment. We considered both the case in which the secondary is at a distance of one hop from the primary and the case in which it is at a distance of two hops from the primary. Given our testbed, the second case corresponds to having a secondary running on the client node.

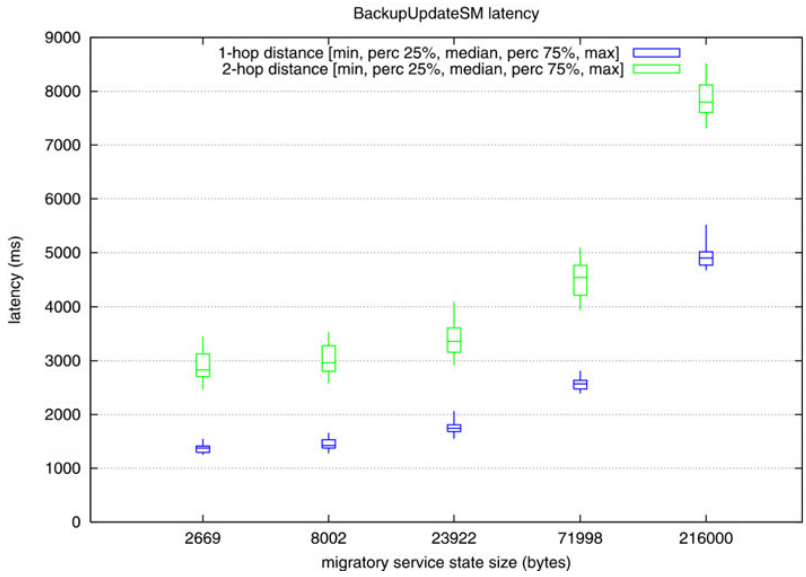

Figure 5.9: The latency of the checkpointing process at a distance of one and two hops.

Figure 5.9 summarizes the results of these experiments for different sizes of the service state and for distances of one hop and two hops. The x-axis is in logarithmic scale (base 10). We considered service states ranging from 2.7 kB, which is the service state size in our TJam prototype migratory service, up to 216 kB, which could be the service state size in a tracking system performing image processing. Notice that the databricks carried by the `BackupUpdateSM` actually contain the service state together with other control information, such as routing, that totally accounts for 691 bytes.

The backup latency time increases linearly with the size of the service state. The measured latencies present high variance due to the wireless communication media. Compared to case of backup at a distance of one hop, the latency of the backup process at a distance of two hops shows a faster raise as the service state size increases. These results confirm the assumption at the basis of our backup strategy that checkpointing large service states over multihop communication can largely compromise the effectiveness of the backup process and highly increase its overhead.

The break up analysis of the backup latencies shows that extracting the service state and generating the `BackupUpdateSM` accounts for approximately 280 ms, while storing the received service state and generating the acknowledgment message carrying control information accounts for approximately 415 ms. The rest is SM communication overhead, which includes connection establishment, transfer of data and code, and serialization. Notice that these experiments were executed without cached code, thereby also the codebricks of the `BackupUpdateSM` amounting to 7018 bytes needed to be transferred.

We then experimented with the four failure scenarios previously described and measured the recovery time. Phone failures were generated by turning off WiFi for a period long enough to trigger the timeout on the client node (pull case) or on the secondary node (push case). Table 5.2 presents test results for different sizes of the migratory service's state, called *MS state* in the table. The same state sizes of the previous experiments were used. The table also indicates the total size of the `BackupSM` that actually contains the MS state and other control information used by the routing, discovery, and migration functions. This additional overhead accounts

for 2364 bytes.

Table 5.2: The average latency of the entire recovery process consisting of failover, secondary discovery at a distance of one hop, BackupSM migration, BackupSM execution at the new secondary node, and reception of the BackupSM's acknowledgement. Tests for 5 different state sizes of the migratory service.

MS state/BackupSM (bytes)	**Failover (ms)**	**1-hop sec disc (ms)**	**BackupSM migr-ex-ack (ms)**
2669/ 5033	256	1850	7187
8002/ 10366	256	1850	7234
23922/ 26286	256	1850	7468
71998/ 74362	256	1850	8296
216000/218364	256	1850	10704

The latency of the recovery process consists of *(i)* resuming the service execution (*failover*); *(ii)* discovering a new, suitable secondary node based on the requirements of the primary service (*1-hop sec disc*); and *(iii)* creating and migrating the `BackupSM` that will start executing on the new node by first sending an acknowledgment back to the primary (*BackupSM migr-ex-ack*). In all experiments, the total size of the `BackupSM`'s codebricks amounts to 28757 bytes. This is because the `BackupSM` contains the migratory service's codebricks that must be transferred and stored at the secondary node. Notice that in the previous experiments, we measured the latency of `UpdateBackupSMs` that carry only the service state and do not contain the migratory service's codebricks because these are transferred only during the first update.

The cost of the recovery process is due mainly to the time necessary to migrate data and code of the backup service to the new secondary node. Even though this latency can be very high for services consisting of large databricks and codebricks, this does not directly impact on the performance observed by the end user. The delay of service responses sent to the client depends on the failover latency, which we found to be small, and on the timeouts detecting the primary failure.

5.6.3 Memory consumption experiments

These experiments have focused on quantifying the cost of Migratory Services in terms of memory consumption. These results may be useful in conjunction with the measurements of Section 4.6.3 to get an understanding of the amount of resources an Urbanet node needs to allocate to support mobile sensing applications.

Our memory measurements might not be highly accurate due to the difficulty of precisely measuring memory consumption in Java, but they definitely represent a lower bound. To run memory experiments we measure the heap size differences before and after our objects have been allocated. However, we need to consider that the JVM can decide to increase its current heap size at any time and in particular when running garbage collection. Therefore, we force garbage collection to happen before our measurements start and ensure that it does not interfere with our measurements[1].

Table 5.3 shows the memory that was allocated to support the execution of clients, metaservices, and migratory services. Primary and secondary services present the same cost. These numbers do not include the memory allocated to run the Java virtual machine and the Migratory Services framework itself; the corresponding heap memory size is about 172 kB.

Table 5.3: The average memory consumption of the `Client`, `MetaService`, and `MigratoryService` components for 5 different state sizes of the migratory service.

	Avg memory consumption (bytes)		
MS state/MS (bytes)	**Client**	**MetaService**	**MigratoryService**
2669/ 5253	68827	86689	111703
8002/ 10586	68827	92848	117852
23922/ 26506	68827	109127	134146
71998/ 74582	68827	157491	182580
216000/218584	68827	301825	326940

[1]We used a slightly modified version of the Sizeof class available at this URL: `http://www.javaworld.com/javaworld/javatips/jw-javatip130.html`

As done in the previous tests, we considered five cases for different sizes of the service state. All values are average values and we do not report the variances as the memory consumption deviations were of few bytes. Migratory services are the most expensive in terms of memory. Nokia 9500 phones have 76 MB of built-in memory. Technical specifications say that the device offers approximately 20 MB of RAM free to run programs. However, benchmark tests[2] show that the size of heap (RAM) is approximately 12.739 MB [9.094 ...16.384]. Based on these benchmarks, an upper bound for our experiments is that a migratory service consumes about 0.89% of the available RAM, which is more than acceptable for our use cases.

5.6.4 Experiments summary

These results demonstrated the feasibility of Migratory Services on modern smart phone platforms. The high latencies we measured are due mainly to the slow data transfer times and partly to the inefficient Java serialization. Furthermore, the final validation of our approach will come only upon extensive testing in large-scale Urbanets.

5.7 Application prototype

We have built TJam, a proof-of-concept migratory service, which 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. Figure 5.10 shows the TJam's user interface on a Nokia 9500 phone.

To evaluate the feasibility and effectiveness of TJam, and ultimately of the Migratory Services model, we ran experiments in a mobile ad hoc network testbed. Given the rapid changes of the nodes' configuration and location, it is crucial to evaluate how the

[2] `http://www.club-java.com/TastePhone/J2ME/MIDP_Benchmark.jsp`

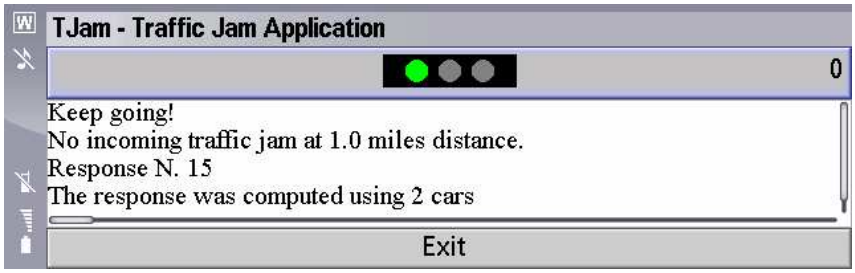

Figure 5.10: Screeshots of the TJam application.

service interaction can adapt to such changes and recover from disconnections. In the following, we first present the TJam application and then its experimental evaluation.

5.7.1 TJam application

We assume that cars on highways communicate using short-range wireless networking (e.g., 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 support the execution and development of such a service, in this case, a Migratory Services implementation is especially beneficial due to *(i)* the highly dynamic operating contexts of ad hoc networks of vehicles, *(ii)* the need to constantly update the coordinates of the monitored region according to the user's location and speed, and *(iii)* the need to transfer the execution state and maintain service history for estimating the traffic jam probability accurately.

In a typical example of interaction, the client first discovers the TJam metaservice and submits its request. A TJam service request includes 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 that will start running on the metaservice node itself. If the metaservice node is not located in the region of interest, the first action of the TJam migratory service

is to migrate 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 re-calculates 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 re-calibrate this estimation by updating the request parameters.

Traffic jams are locally congested phases, in which cars travel at slow or zero velocity. Based on this observation, the computation of the probability of traffic jam takes inspiration from the well-known algorithm Random Early Detection (RED) [Floyd and Jacobson, 1993] for congestion avoidance in packet-switched networks. In the same way RED predicts packet congestion by monitoring the average queue size at the gateway, we predict traffic jams along the road by monitoring two types of information that every car in the network has locally available:

1. the *number* of neighboring cars at a distance of one hop, and
2. the *speed* of neighboring cars at a distance of one hop.

Stated simply, the idea is that if the number of cars increases and their speed decreases it is likely that a traffic jam is forming.

The intermediate probability of traffic jam, called P'_{tjam}, is computed by a weighted combination of P_{number} and P_{speed}, where P_{number} is the probability of traffic jam given the average number of neighboring cars, called avg_{num}, and P_{speed} is the probability of traffic jam given the average speed of neighboring cars, called avg_{speed}.

$$P_{number} = \begin{cases} 1 & \text{if } avg_{num} \geq max_{num} \\ 0 & \text{if } avg_{num} \leq min_{num} \\ maxP_{number} \cdot \frac{avg_{num} - min_{num}}{max_{num} - min_{num}} & \text{otherwise} \end{cases}$$

$$P_{speed} = \begin{cases} 0 & \text{if } avg_{speed} \geq max_{speed} \\ 1 & \text{if } avg_{speed} \leq min_{speed} \\ maxP_{speed} \cdot \frac{avg_{speed} - max_{speed}}{min_{speed} - max_{speed}} & \text{otherwise} \end{cases}$$

$$P'_{tjam} = \alpha \cdot P_{number} + (1 - \alpha) \cdot P_{speed} \qquad \alpha = 1/2$$

where 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, meaning that the road is empty and therefore there is no traffic jam. Above the upper boundary max_{num}, P_{number} is 1, meaning that the road is crowded and therefore there is a traffic jam.

Conversely, as avg_{speed} varies from min_{speed} to max_{speed}, P_{speed} varies linearly from $maxP_{speed}$, which is the upper bound of P_{speed}, down to 0. Below the lower boundary min_{speed}, P_{speed} is 1, meaning that cars almost do not move or do not move at all and therefore there is a traffic jam. Above the upper boundary max_{speed}, P_{speed} is 0, meaning that cars travel faster than the road speed limit and therefore there is no traffic jam.

Finally, the probability of traffic jam P_{tjam} is the P'_{tjam} corrected by the ratio between the number of observations in which P_{tjam} was less than 0.5 (called N_{tjam}) and the total number of observations (called N_{total}).

$$P_{tjam} \quad = \quad P'_{tjam} \cdot \frac{N_{tjam}}{N_{total}}$$

To provide a more accurate computation of the traffic jam probability, it is necessary to tune the employed parameters according to traffic variations and history. For the sake of brevity, we simply say that minimum and maximum boundaries of number and speed of neighboring vehicles are adjusted based on the corresponding minimum and maximum values measured during the period of observation.

The probabilities $maxP_{number}$ and $maxP_{speed}$ are updated such that if it happens that avg_{num} is below the lower threshold, then $maxP_{number}$ is decreased, and if avg_{num} is greater than the upper

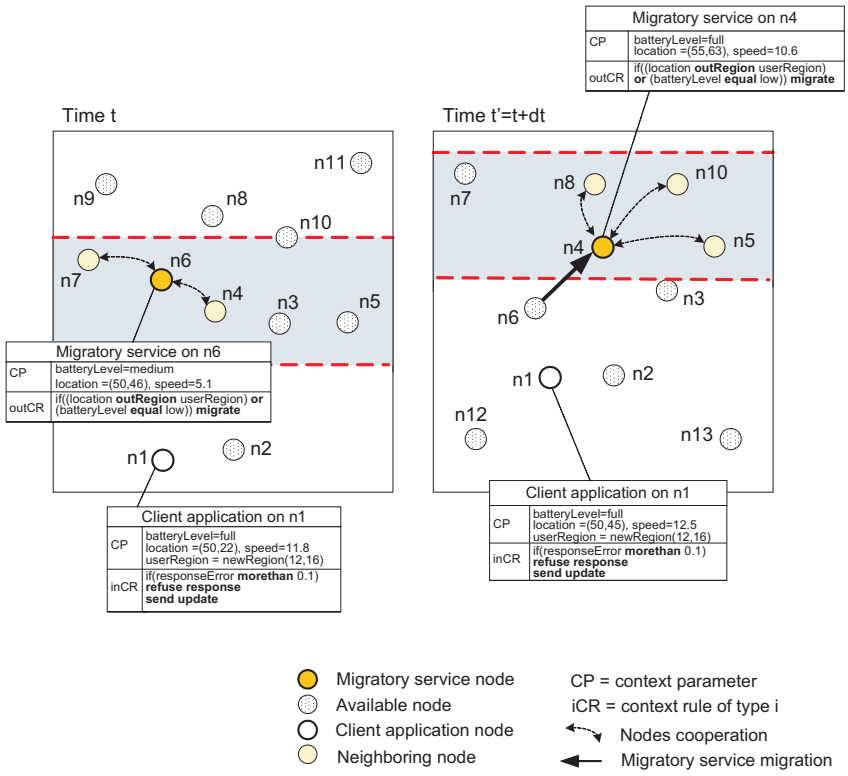

Figure 5.11: Example of execution of the TJam migratory service prototype.

threshold, then $maxP_{number}$ is increased. The opposite applies to $maxP_{speed}$.

$$maxP^{new}_{number} = \begin{cases} maxP^{old}_{number} \times \beta & \text{if } (avg_{num} < min_{num}) \\ maxP^{old}_{number} + \gamma & \text{if } (avg_{num} > max_{num}) \\ maxP^{old}_{number} & \text{otherwise} \end{cases}$$

$$\beta = 0.8 \ , \ \gamma = \frac{maxP^{old}_{number}}{10}$$

In order to illustrate an example of TJam migration along with some context parameters (CPs) and context rules (CRs), we refer

to Figure 5.11. The client application executes on node *n1*. At time *t*, the client interacts with the TJam migratory service which 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 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 Services 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 (i.e., min_{num}, max_{num}, min_{speed}, max_{speed}, $maxP_{number}$, $maxP_{speed}$, N_{tjam}, N_{total}) and history of user's locations and speeds for updating the user-defined region. In this way, the service can provide a continuous and more efficient interaction with the client.

5.7.2 TJam implementation

To exemplify the primitives offered by the Migratory Services framework, in the following, we illustrate some pseudo code of the TJam application.

In Figure 5.12, the client uses the Migratory Services framework (`MSF`) to issue a request for a `Tjam` service in a remote region `remoteR`. The request also contains information about client's current context and context rules. A context rule consists of condition and action pairs. In the proposed example (see lines 3-6), the rule states that when the error of the received response exceeds `MIN_ERROR`, the result is discarded.

A `TJam` metaservice available in the network receives the request and instantiates a migratory service `TJam-migra` to perform the computation. In order to execute on a certain node, the `TJam-migra` service registers with the hosting node and specifies its context rules. In the example of Figure 5.13, the context rule states that when the node moves out of the region specified by the client, a service migration must be triggered to resume the execution on a

```
 1. clientCxt.addElement(CXT_LOCATION);
 2. clientCxt.addElement(CXT_SPEED);
 3. ContextRule rule = new ContextRule();
 4. rule.setSourceID("c1");
 5. rule.setCondition(resError,morethan,MIN_ERROR);
 6. rule.setAction(ACTION_REFUSE);
 7. inCxtRules.addElement(rule);
 8. Region remoteR = new Region(distance,range);
 9. Request req = new Request("c1",remoteR,clientCxt,inCxtRules);
10. MSF.sendRequest("TJam",req);
```

Figure 5.12: Pseudo-code of the TJam client sending to the TJam migratory service a service request containing its context information and context rules.

node inside that region.

```
1. CtxRule rule = new CxtRule();
2. rule.setSourceID("TJam-migra-c1");
3. Region remoteR = new Region(distance,range);
4. rule.setCondition(OutOfRegion,remoteR);
5. rule.setAction(ACTION_MIGRATE);
6. MSF.registerCtxRule(rule);
```

Figure 5.13: Pseudo-code of the TJam migratory service registering a context rule with the Migratory Services framework (`MSF`).

During service execution, `MSF` constantly evaluates the registered context rules and acts upon them. In Figure 5.14, the service computes the probability of traffic jam and when the probability reaches the upper threshold, an alert is sent to the client. Note that migration is not part of the service code, as the framework triggers it when necessary and in a transparent way.

5.7.3 TJam evaluation

We first assessed the performance of TJam on Nokia Series 80 phones. We used an ad hoc network of 3 phones arranged on a line. The average interresponse time when the migratory service executes at a distance of two hops from the client is approximately 5.23 seconds. Note that the measured latencies include the sleep

```
1. int speed=0;
2. tjam_p=0;
3. Neighbors []n;
4. for(j=0;j<n.length;j++)
5.    speed + = n[j].getSpeed();
6. speed = speed/n.length;
7. tjam_p= computeTJamProbability(speed,n.length);
8. if(tjam_p>MAX_PROB)
9.    MSF.sendAlert("cl1",tjam_p);
```

Figure 5.14: Pseudo-code of the TJam migratory service computing the traffic jam probability.

time of 5 seconds between consecutive response computations. Similar latencies were observed both when reliability was enabled and disabled.

However, to investigate the performance of TJam in a slightly larger-scale mobile network, we used the network topology shown in Figure 5.15. This consists of 11 HP iPAQs running Linux and using Orinoco's 802.11b PC cards. Since it proved very difficult to run the experiments with the iPAQs moving at an adequate speed, we 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 (i.e., latitude and longitude) 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 experimental results, the service executes on a node that is 2-hop 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 (i.e., 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 and 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 coordinates are incorrect compared to the current context, the answer is discarded, and an update is sent to the service. Otherwise, the answer is deliv-

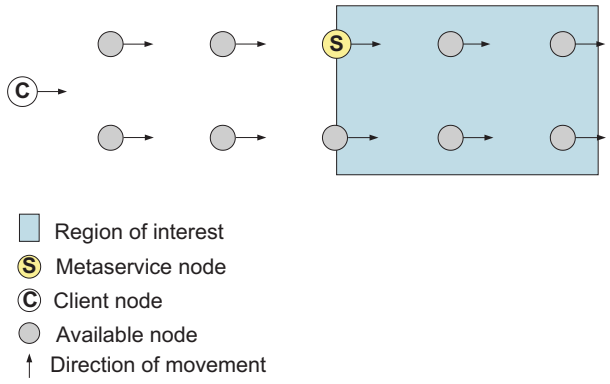

Figure 5.15: The initial network topology used in the TJam experiments.

ered to the client application and no update is sent. This approach is feasible for TJam since 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 timeout. In the experiments, the timeout is set to 7.5 seconds.

Four main metrics have been employed in the experimental evaluation.

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

Our main goal was to study how a migratory service allows the user to constantly receive correct observations in spite of disconnections and mobility. As Figure 5.16 shows, the service migration successfully follows the movement of the user. Specifically, the graph

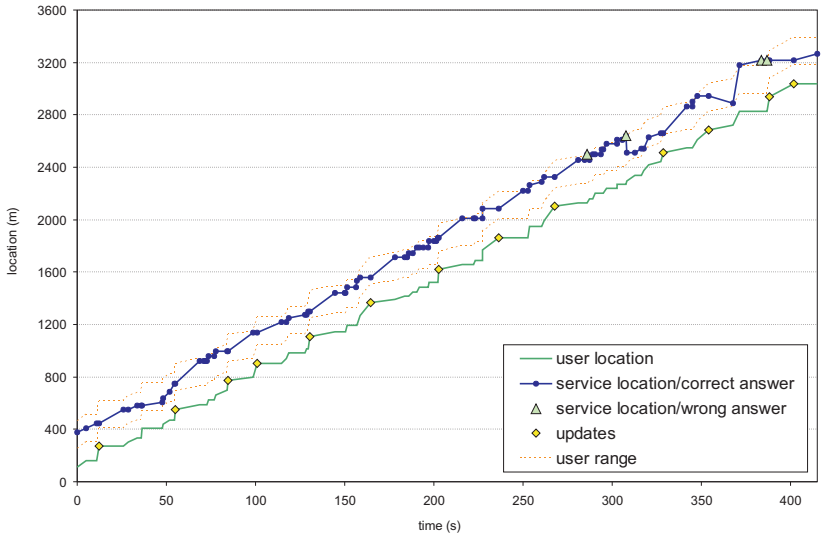

Figure 5.16: Location traces of the TJam migratory service following the user movement without any task interruption.

compares the location of the user to the location of the service. If the service 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 5.4 summarizes these experimental results. It reports the average value and the 90% confidence interval for every metric. In these experiments, the TJam's sleep time was only 500 ms. 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 re-calibrate the query parameters. However, these values are more than acceptable for this type of application.

In a more extensive study reported in [Riva et al., 2007], we simulated and evaluated two different versions of TJam, namely *TJam-Smart* and *TJam-Base* in order to investigate the scalability of our approach in larger-scale networks. TJam-Smart implements our model of context-aware migration. TJam-Base implements a baseline centralized approach. In TJam-Base, a few mobile nodes

Table 5.4: Results of the TJam migratory service experiments in an ad hoc network of 11 HP iPAQs.

Metric	**Avg [90% Conf. Interval]**
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]

host the service in the network. After receiving a client request, the TJam-Base service node computes the traffic jam probability by directly querying nodes in the region of interest, sends back the computed results to the client, automatically updates the query parameters, and finally initiates a new query cycle.

These services were simulated using the *ns-2* simulator [NS-2, 2007], enhanced with the CMU-wireless extensions [Monarch, 2007]. To simulate TJam-Smart and TJam-Base we used a microscopic simulation model 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 [Nadeem et al., 2004], where it was used to generate highway traffic and in [Zhou et al., 2005] where it was used to generate city traffics.

In the simulation study, we investigated the scalability of the Migratory Services approach by varying the number of clients, as well as the effects of the vehicular traffic variability by varying speed and density of the vehicles. Summarizing this analysis, the simulation results showed that TJam-Smart performs better than TJam-Base in terms of response time, network utilization, and packet overhead.

Based on the experimental results, we can conclude that besides increasing the density of the vehicles, also increasing the speed of the vehicles helps reduce the communication overhead and provide better performance. This occurs because higher mobility increases network connectivity [Grossglauser and Tse, 2002].

Finally, the 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, the Migratory Services approach guarantees better results than the statically centralized approach due to the smaller number of packets that need to be exchanged.

5.8 Concluding remarks

This chapter presented a context-aware service model that allows Urbanets to provide services that quickly adapt to changes of their execution context, while still guaranteeing service continuity to the client. A Migratory Services 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. This approach is particularly useful to support long-running and stateful end-to-end client-service interactions.

The Migratory Services framework provides communication primitives, migratory service control, context management, and reliability support. The framework was implemented in Java on top of the Smart Messages platform. We evaluated the framework by assessing its performance in terms of latency and memory consumption in a small-scale ad hoc network of smart phones. This framework was used to build 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. Its performance was evaluated using a slightly larger-scale ad hoc network of PDAs. The experimental results demonstrated the viability of our model. Furthermore, the programming interface offered by the framework has proved intuitive and flexible enough to be capable of supporting a large variety of applications.

CHAPTER 6

Conclusions

This chapter closes the dissertation by *(i)* summarizing our contributions, *(ii)* discussing some open issues, *(iii)* identifying several directions for future work, and *(iv)* presenting concluding remarks.

6.1 Contributions

The Internet has become such a great success over the years because of its appeal to regular people. This is not the case with sensor networks, which are nowadays perceived as "something" remote in the forest or on the battlefield. With mobile devices becoming ubiquitous, the time is ripe to bring sensor data out of their closed-loop networks into the center of daily urban life. This goal can be achieved through Urbanets, spontaneously created urban networks consisting of mobile multi-sensor platforms, such as smart phones and vehicular systems, individual sensors incorporated in buildings or roads, and sensor networks deployed by municipalities.

However, as the Urbanet applications domain diversifies, programming each application from scratch will be rather inefficient. A common distributed computing platform that can support such applications becomes then necessary. Urbanets are challenging programming environments due to their high volatility, software and hardware heterogeneity, resource constraints, and very large scale. These requirements hinder the direct adoption of traditional distributed computing models, which assume underlying networks with functionally homogeneous nodes, stable configurations, and known

delays. On the other hand, research on sensor networks and mobile ad hoc networks can be leveraged to provide novel architectures and models that address the Urbanet-specific requirements.

Our work has focused on designing, implementing, and evaluating a common distributed computing platform that can support the development and execution of Urbanet applications. The contributions of this work and the answers to the investigated research questions can be summarized as follows.

The design and development of two middleware platforms that can effectively support people-centric mobile sensing applications. Unlike existing solutions, these platforms *(i)* are distributed, *(ii)* are infrastructure-less, *(iii)* support real-time collection of sensed data, and *(iv)* provide on-demand collection at the frequency and with the accuracy specified by every single application. Their distributed nature and independence from available infrastructures permit achieving better resource utilization, improved scalability, and fault-tolerance. The implementation of these platforms on modern smart phones and their experimental evaluation in small-scale ad hoc networks have demonstrated the feasibility and the effectiveness of our design principles and models. Moreover, experiments in real testbeds and field trials organized during the sailing regatta gave our development work a realistic perspective. Additional contributions of our work come from the fact that the entire development was done on smart phones. Given their resource constraints, their limited programming environment, and poor debugging support, the implementation had to be lightweight in memory and processing.

A declarative and a client-server programming model that can support Urbanet applications. Contory offers a declarative programming paradigm that views Urbanets as a distributed sensor database and exposes a simple SQL-like interface to programmers. Context-aware Migratory Services provides a client-service model where services migrate to different nodes in the network to maintain a semantically correct and continuous interaction with clients. A number of prototype applications targeting real-life scenarios have been developed using such models. In programming these applications, compared to traditional socket-based approaches and middleware for static resource-rich networks, our middleware platforms

have been successful in hiding problems related to distribution, sensor failures, and resource and network volatility. High-level functions such as geographical-based execution, context queries, and resource-driven migration can definitely shorten the development time, as it has been in the case of the DYNAMOS sailing application implemented twice, before and after Contory was developed. However, besides providing transparency, these middleware platforms also allow programmers to control the execution dynamically through several context rules and to assess the quality of results through qualifying attributes. For example, a programmer using Contory can decide to specify precisely how and using which resources a query should be executed. Otherwise, he can let the middleware free to decide how the execution should be carried out and which amount and type of resources should be consumed.

A set of adaptation strategies and support for control policies to accomplish dynamic resource management, cope with network volatility, and guarantee fault-tolerance in networks of resource-constrained mobile devices. The proposed middleware architectures cope with network volatility in different ways. When Contory detects that a certain context provider has become unavailable, it dynamically selects an alternative provider currently available. In addition, Contory dynamically adapts its execution to the current device's status and available resources. The Context-aware Migratory Services framework responds to volatility by migrating the service execution to suitable nodes every time the execution context changes beyond the acceptable limits. In addition, it also maintains a backup service that takes over in case of service failure or network partitions. We implemented these strategies on smart phones and evaluated their effectiveness in testbed networks of phones and sensors. For example, Contory was able to tolerate temporary GPS disconnections. Migratory Services was able to recover from failures of the primary service or guarantee service continuity while executing in a network of mobile PDAs.

Finally, we believe that design principles and algorithms at the basis of our middleware architectures are generic enough to be applicable to solve similar problems of resource management, adaptability, network volatility, and reliability in a large number of application domains relative to mobile and ubiquitous computing. More-

over, our experiences with real system prototypes running on smart phones can be of interest also to developers of other research domains.

6.2 Open issues

There are still several problems that need to be solved before Urbanet applications can be widely and practically deployed. For example, our work does not directly address issues such as security, trust, privacy, and incentives for cooperation. On the other hand, we also believe that many of the proposed solutions for sensor networks and MANETs can be adapted to Urbanets.

Security, trust, and privacy are essential requirements to enable sharing of resources, information, and services in ad hoc networks. In particular, these aspects are crucial in our implementations that make use of mobile code. The migrating code needs to be protected from a malicious node and a hosting node needs to be protected from malicious migrating code. The underlying assumption of our prototype systems is that they can rely on a network of trusted entities or that the operating system together with the Smart Messages platform provide the protection needed. Compared to traditional mobile agents for relatively stable IP-based networks, the Smart Messages platform has also to cope with the lack of a central authority which hampers the direct adoption of well-know techniques based on key authentication and group management. As described in [Kang et al., 2004], the Smart Messages platform integrates a basic security architecture that provides protected access to the tag space, thus ensuring protection against malicious SMs.

More specifically, in Migratory Services, we can identify several security threats mainly caused by malicious nodes present in the network. First of all, control mechanisms are necessary to prevent a malicious migratory service to access private data or make an excessive use of resources of the hosting node. For example, a migratory service could exhaust the energy of the hosting device, corrupt users' personal data, launch an attack in the device, or infect other applications running on the device. These problems are partly solved by the Smart Messages platform by requiring SMs to express upper boundaries on resource usage and by performing ad-

mission control of incoming SMs. On the other hand, a malicious node in the network may alter the data (e.g., state) of a migratory service and compromise its correct execution. Or it may even alter the code of a migratory service and propagate viruses in the network. As proposed in Smart Messages, this can be solved by allowing migratory services to be migrated in an encrypted form. In this case, each node will need to carry a pair of public/private keys. To terminate or corrupt the execution of a migratory service, a malicious node may also attract a migratory service by pretending to possess false resources or to be in a certain location (i.e., "fake migratory services"). This can be partly solved with redundancy. For example, the secondary service can integrate mechanisms to detect possible corruptions of the primary service and thus promptly reestablish a correct execution. However, ultimately, a trusted third party is necessary to certify that nodes in the network claim to provide services and resources that they actually own. In the special case of fake locations, these can also be verified in a collaborative manner by querying neighboring nodes.

Trust issues are more critical than in traditional sensor networks because of the spontaneous and people-centric nature of Urbanets. Trust deals with the estimation of a node's future behaviour. In assessing the trust in a certain node, a node can rely on its direct experiences with such a node as well as on others' recommendations. A recommendation is "the perception that a node creates through past actions about its intentions and norms" [Mui et al., 2002]. To isolate malicious users, reputation mechanisms such as [Liu and Issarny, 2004; Michiardi and Molva, 2002; Zhu and Mutka, 2004] can be integrated in our systems.

Privacy issues directly influence the user acceptance of this type of systems. In environments full of sensors that keep track of any entity and any event, privacy concerns naturally arise. To address privacy issues there are two basic strategies that can be taken [Meyer and Rakotonirainy, 2003]. The first one ensures protection by instructing sensor devices to reduce the amount of private data being disseminated. The second strategy reduces the amount of information being acquired and stored to the minimum. Additionally, other important approaches to solve privacy issues are privacy preserving data mining techniques [Agrawal and Srikant, 2000; Verykios et al., 2004]. These modify the original data through different perturba-

tion algorithms so that the private data remain private without affecting the statistics to be collected. In our systems, the utilization and sharing of location information represent the most critical sources of privacy concerns. The user needs to control who gets access to her private location information and understand the usage that will be made of such an information. The IETF Geopriv Working Group has published a RFC [Cuellar et al., 2004] that describes the requirements for the Geopriv Location Object (LO) and for the protocols that use it. LO is a data structure used to securely exchange location data. It consists of location data and a set of user-specified privacy rules that specify to which entities, under which conditions, and at which granularity (e.g., street, city, country) location data can be released. Finally, we also believe that the actual deployment of fully working Urbanet services will make people less protective and thus will contribute to the social acceptance of approaches like ours.

A key assumption in our work is that nodes in Urbanets are willing to cooperate and they trust each others. Incentive models for cooperation are necessary so that entities can benefit from cooperating with others, even though some entity refuses to cooperate. A system of credits such as the one proposed in [Zhu and Mutka, 2004] could be integrated in our systems. A cost is associated to each query generation and a payment is received each time relevant sensor data are provided.

In addition, in the future, several types of Urbanet environments are expected to cooperate among themselves as well as with external services such as transport systems, surveillance and emergency services, or tourist information infrastructures. Therefore, each Urbanet middleware should offer a standardized interface to foster interoperability with other platforms and overcome hardware and software heterogeneity. Our solutions were designed to work despite Urbanets' heterogeneity with the assumption that the underlying Smart Messages platform provides a common execution environment across heterogeneous devices. Tiny sensors such as motes do not run our middleware; they can be accessed in two ways: either directly queried by mobile nodes in proximity (e.g., iMotes over Bluetooth) or accessed indirectly through a base station running our middleware.

Our experiences with these middleware platforms in small-scale

ad hoc networks of HP iPAQs and smart phones have been promising. Along with typical interference problems in places with high density of wireless devices, power remains the most constraining technical factor. Most of the algorithms and middleware services have also been simulated in order to investigate their performance in large-scale ad hoc networks. However, simulation experiments were conducted by co-authors and therefore their results were not included in this dissertation. The ultimate validation of our solutions will derive from extensive user evaluation in large-scale mobile networks of real-life scenarios.

6.3 Future work

In this section, we discuss several research topics for future work. We describe the main issues they involve and possible solutions to them.

Query and data aggregation in mobile ad hoc networks: Currently, Contory performs multi-query optimization only among queries submitted by the same device, but this can be extended to merge queries from different devices that have selection predicates with overlapping ranges of attributes. Moreover, in our future research, we will investigate mechanisms for in-network data aggregation that work in the presence of mobility. For instance, mobility can lead to situations where a certain context item is aggregated multiple times at different nodes, thus negatively afftecting the quality of results. Mechanisms for data and query aggregation have been so far proposed for static and homogenous sensor networks and understanding how to apply these to mobile and heterogenous environments is a challenging research problem.

Energy-awareness on mobile devices: Monitoring and reconfiguration mechanisms to control and adjust the utilization of resources on battery-powered devices will be more and more important in the future given the fact that there will always be an even higher demand for "mobile energy". Our experience was that policy-based approaches, despite their popularity, cannot always be flexible enough. Moreover, a priori specification of reconfiguration rules is almost impossible due to the required close coupling with the plat-

form characteristics. Instead, learning resource utilization seems to be more promising and potentially capable of finding out the right tricks for energy saving on each specific device. To fully address the energy problem, energy-aware mechanisms are needed at all system levels. Middleware needs to be energy-aware in performing network selection, aggregating data, and suspending and resuming applications. The operating system needs to be energy-aware in managing hardware devices and scheduling data transfers. In addition, the operating system must serve as a central monitoring unit for the entire device and propagate contextual information to the upper layers. Finally, it is necessary to coordinate energy-saving actions, resolve potential conflicts, and prioritize tasks whenever possible.

Dependability in spontaneous networking: Dependability of future pervasive middleware and applications and in particular of spontaneous networking are essential, albeit often ignored, requirements to be investigated. Ubiquitous connectivity has opened the way to a variety of interactions among mobile devices ranging from simple file transfers to more complex stateful interactions. Dependability has been a hot topic of distributed systems, but almost no mechanism has been applied to distributed networks of resource-constrained devices, such as ad hoc networks. The only fault-tolerance issues studied so far in ad hoc networks are at the network layer.

6.4 Concluding remarks

This dissertation has focused on providing middleware support and appropriate programming abstractions to support the development and execution of people-centric mobile sensing applications in Urbanets. In developing Urbanet applications, the choice of which middleware (or programming model) to employ depends on the applications' semantics.

With Contory, the intelligence is mainly in the middleware, while the applications are provided with a very simple programming interface to specify how to collect sensor data. Contory provides high transparency by adapting to changing operating conditions, but it also allows applications to assess the quality of results through qualifying attributes. Yet, Contory offers limited flexibility to program

complex distributed applications. Let us consider the case in which the sensed data need to be processed using complex algorithms and communicated to the application only if certain conditions are verified. If the application developer wants to specify and control the execution of such algorithms in a remote region, Contory does not offer an appropriate programming support. With Contory, observations are collected in the remote region and transferred to the application that can subsequently and locally process the data.

An alternative solution is that observations are collected and algorithms are executed by a node (i.e., service) located in the region of interest and only relevant results are then transmitted to the application. This kind of approach can turn out to be more efficient and effective depending on the type of algorithm to be executed, frequency of computation, distance of the region, and so forth. The Context-aware Migratory Services framework supports this second approach. Therefore, with this programming model, client applications are very simple, services can be very complex, and the middleware provides significant help by automatically and continuously adapting to the rapidly changing operating contexts.

To conclude, the convergence of sensor networks and mobile ad hoc networks into Urbanets offers a lot of opportunities to support mobile users in their daily urban lives. Without the need for any preexisting infrastructure, a variety of sensing applications can be flexibly and quickly established and executed to deliver customized services. The middleware platforms presented in this dissertation represent one of the first attempts to take advantage of these environments. Further steps will have to address more technical issues as well as social and legal concerns.

References

Abdelzaher, T., Anokwa, Y., Boda, P., Burke, J., Estrin, D., Guibas, L., Kansal, A., Madden, S., and Reich, J. 2007. Mobiscopes for Human Spaces. *IEEE Pervasive Magazine 6,* 2 (April-June), 20–29. IEEE Educational Activities Department.

Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., George, S., Gu, L., He, T., Krishnamurthy, S., Luo, L., Son, S., Stankovic, J., Stoleru, R., and Wood, A. 2004. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pp. 582–589. IEEE Computer Society.

Abowd, G. D. 1999. Classroom 2000: An Experiment with the Instrumentation of a Living Educational Environment. *IBM Systems Journal 38,* 4, 508–530.

Abowd, G. D., Iftode, L., and Mitchell, H. 2005. Guest Editors' Introduction: The Smart Phone - A First Platform for Pervasive Computing. *IEEE Pervasive Computing 4,* 2 (April-June), 18–19. IEEE Educational Activities Department.

Agrawal, R. and Srikant, R. 2000. Privacy-Preserving Data Mining. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*, pp. 439–450. ACM Press.

Akkaya, K. and Younis, M. 2005. A survey on routing protocols in wireless sensor networks. *Ad Hoc Networks 3,* 3 (May), 325–349. Elsevier B.V.

Akyildiz, I., Su, W., Sankarasubramaniam, Y., and Cayirci, E. 2002. A Survey on Sensor Networks. *IEEE Communications Magazine 40,* 8 (August), 102–114. IEEE Educational Activities Department.

Arbanowski, S., Ballon, P., David, K., Droegehorn, O., Eertink, H., Kellerer, W., van Kranenburg, H., Raatikainen, K., and Popescu-Zeletin, R. 2004. I-centric Communications: Personalization, Ambient Awareness, and Adaptability for Future Mobile Services. *IEEE Communications Magazine 42,* 9 (September), 63–69. IEEE Educational Activities Department.

Aura 2007. Aura Project. `www-2.cs.cmu.edu/~aura`.

Bagrodia, R., Chu, W., Kleinrock, L., and Popek, G. 1995. Vision, Issues, and Architecture for Nomadic Computing. *IEEE Personal Communications 2,* 6 (December), 14–27. IEEE Educational Activities Department.

Balan, R., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., and Yang, H.-I. 2002. The Case for Cyber Foraging. In *Proceedings of the 10th ACM SIGOPS European workshop: beyond the PC (EW10)*, pp. 87–92. ACM Press.

Baldus, H., Klabunde, K., and Müsch, G. 2004. Reliable Set-Up of Medical Body-Sensor Networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN'04)*, Volume 2920 of *Lecture Notes in Computer Science*, pp. 353–363. Springer-Verlag.

Banavar, G. and Bernstein, A. 2002. Software Infrastructure and Design Challenges for Ubiquitous Computing Applications. *Communications of the ACM 45,* 12 (December), 92–96. ACM Press.

Bardram, J. E. 2005. The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework

for Context-Aware Applications. In *Proceedings of the Third International Conference on Pervasive Computing (Pervasive'05)*, Volume 3468 of *Lecture Notes in Computer Science*, pp. 98–115. Springer-Verlag.

Barr, R., Bicket, J. C., Dantas, D. S., Du, B., Kim, T. W. D., Zhou, B., and Sirer, E. G. 2002. On the Need for System-Level Support for Ad Hoc and Sensor Networks. *ACM SIGOPS Operating Systems Review 36,* 2 (April), 1–5. ACM Press.

Basagni, S., Chlamtac, I., Syrotiuk, V. R., and Woodward, B. A. 1998. A Distance Routing Effect Algorithm for Mobility (DREAM). In *Proceedings of the Forth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98)*, pp. 76–84. ACM Press.

Beigl, M., Gellersen, H.-W., and Schmidt, A. 2001. MediaCups: Experiences with Design and Use of Computer-Augmented Everyday Objects. *Computer Networks 35,* 4 (March), 401–409. Elsevier B.V.

Bellavista, P., Corradi, A., Montanari, R., and Stefanelli, C. 2003. Context-Aware Middleware for Resource Management in the Wireless Internet. *IEEE Transactions on Software Engineering 29,* 12 (December), 1086–1099. IEEE Computer Society.

Bellavista, P., Corradi, A., Montanari, R., and Stefanelli, C. 2006. A Mobile Computing Middleware for Location- and Context-aware Internet Data Services. *ACM Transactions on Internet Technology (TOIT) 6,* 4 (November), 356–380. ACM Press.

Biegel, G. and Cahill, V. 2004. A Framework for Developing Mobile, Context-aware Applications. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, pp. 361–365. IEEE Computer Society.

Bisignano, M., Calvagna, A., Modica, G. D., and Tomarchio, O. 2003. Expeerience: A JXTA Middleware for Mobile

Ad-Hoc Networks. In *Proceedings of the Third International Conference on Peer-to-Peer Computing (P2P'03)*, pp. 214–215. IEEE Computer Society.

Blazevic, L., Boudec, J.-Y. L., and Giordano, S. 2005. A Location-Based Routing Method for Mobile Ad Hoc Networks. *IEEE Transactions on Mobile Computing 4,* 2 (March), 97–110. IEEE Educational Activities Department.

Bonnet, P., Gehrke, J., and Seshadri, P. 2001. Towards Sensor Database Systems. In *Proceedings of the Second International Conference on Mobile Data Management (MDM'01)*, Volume 1987 of *Lecture Notes in Computer Science*, pp. 3–14. Springer-Verlag.

Borcea, C., Intanagonwiwat, C., Kang, P., Kremer, U., and Iftode, L. 2004. Spatial Programming Using Smart Messages: Design and Implementation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pp. 690–699. IEEE Computer Society.

Borcea, C., Intanagonwiwat, C., Saxena, A., and Iftode, L. 2003. Self-Routing in Pervasive Computing Environments using Smart Messages. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom'03)*, pp. 87–96. IEEE Computer Society.

Borcea, C., Iyer, D., Kang, P., Saxena, A., and Iftode, L. 2002. Cooperative Computing for Distributed Embedded Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pp. 227–236. IEEE Computer Society.

Boulis, A., Han, C.-C., and Srivastava, M. B. 2003. Design and Implementation of a Framework for Programmable and Efficient Sensor Networks. In *Proceedings of the First International Conference on Mobile Systems, Applications and Services (MobiSys'03)*, pp. 187–200. ACM Press.

Braginsky, D. and Estrin, D. 2002. Rumor Routing Algorithm for Sensor Networks. *Proceedings of the First Workshop on Sensor Networks and Applications (WSNA'02)*, 22–31.

Broch, J., Maltz, D. A., Johnson, D. B., Hu, Y.-C., and Jetcheva, J. 1998. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proceedings of the Forth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98)*, pp. 85–97. ACM Press.

BTnodes 2007. BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. `http://www.btnode.ethz.ch`.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. 1993. The primary-backup approach. In S. Mullender (Ed.), *Distributed systems (2nd Ed.)*, ACM Press Frontier Series, pp. 199–216. ACM Press/Addison-Wesley Publishing Co.

Burrell, J., Brooke, T., and Beckwith, R. 2004. Vineyard Computing: Sensor Networks in Agricultural Production. *IEEE Pervasive Computing 3,* 1 (Jan-Mar), 38–45. IEEE Educational Activities Department.

Buttyán, L. and Hubaux, J.-P. 2003. Stimulating Cooperation in Self-Organizing Mobile Ad Hoc Networks. *Mobile Networks and Applications 8,* 5 (October), 579–592. Kluwer Academic Publishers.

Campbell, A. T., Eisenman, S. B., Lane, N. D., Miluzzo, E., and Peterson, R. 2006. People-Centric Urban Sensing. In *Proceedings of the Second ACM/IEEE Annual International Wireless Internet Conference (WICON'06).*

Campbell, R., Al-Muhtadi, J., Naldurg, P., Sampemane, G., and Mickunas, M. 2002. Towards Security and Privacy for Pervasive Computing. In *Proceedings of the Second Mext-NSF-JSPS International Symposium on Software Security - Theories and Systems*, pp. 1–15.

Capra, L., Emmerich, W., and Mascolo, C. 2003. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering 29,* 10 (October), 929–945. IEEE Computer Society.

Cardell-Oliver, R., Smettem, K., Kranz, M., and Mayer, K. 2005. A Reactive Soil Moisture Sensor Network: Design and Field Evaluation. *International Journal of Distributed Sensor Networks 1,* 2 (April-June), 149–162. Taylor & Francis.

Cartel 2007. CarTel Project. `http://cartel.csail.mit.edu`.

Cerpa, A., Elson, J., Estrin, D., Girod, L., Hamilton, M., and Zhao, J. 2001. Habitat Monitoring: Application Driver for Wireless Communications Technology. *2001 ACM SIGCOMM Workshop on Data communication in Latin America and the Caribbean*, 20–41. ACM Press.

Cerpa, A. and Estrin, D. 2004. ASCENT: Adaptive Self-Configuring Sensor Networks Topologies. *IEEE Transactions on Mobile Computing 3,* 3 (July), 272–285. IEEE Educational Activities Department.

Chakraborty, D. and Finin, T. 2006. Toward Distributed Service Discovery in Pervasive Computing Environments. *IEEE Transactions on Mobile Computing 5,* 2 (February), 97–112. IEEE Educational Activities Department.

Chandra, R., Ramasubramanian, V., and Birman, K. 2001. Anonymous Gossip: Improving Multicast Reliability in Mobile Ad-Hoc Networks. In *Proceedings of the 21th IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, pp. 275–283.

Chang, J.-H. and Tassiulas, L. 2000. Energy Conserving Routing in Wireless Ad-hoc Networks. In *Proceedings of the IEEE INFOCOM'00 Conference on Computer Communications*, pp. 22–31.

Chen, B., Jamieson, K., Balakrishnan, H., and Morris, R. 2001. Span: An Energy-efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MobiCom'01)*, pp. 85–96. ACM Press.

Chen, G., Li, M., and Kotz, D. 2004. Design and Implementation of a Large-Scale Context Fusion Network. In *Proceedings of*

the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04), pp. 246–255. IEEE Computer Society.

Chen, H., Finin, T., and Joshi, A. 2005. *The SOUPA Ontology for Pervasive Computing.* Whitestein Series in Software Agent Technologies. Springer-Verlag.

Cheng, S. Y. and Trivedi, M. M. 2006. Turn-Intent Analysis Using Body Pose for Intelligent Driver Assistance. *IEEE Pervasive Computing 5,* 4, 28–37. IEEE Educational Activities Department.

Cheong, E., Liebman, J., Liu, J., and Zhao, F. 2003. TinyGALS: A Programming Model for Event-Driven Embedded Systems. In *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC'03)*, pp. 698–704. ACM Press.

Chintalapudi, K., Fu, T., Paek, J., Kothari, N., Rangwala, S., Caffrey, J., Govindan, R., Johnson, E., and Masri, S. 2006. Monitoring Civil Structures with a Wireless Sensor Network. *IEEE Internet Computing 10,* 2, 26–34. IEEE Educational Activities Department.

Chlamtac, I., Conti, M., and Liu, J. J.-N. 2003. Mobile ad hoc networking: imperatives and challenges. *Ad Hoc Networks 1,* 1 (July), 13–64. Elsevier B.V.

Chong, C.-Y. and Kumar, S. 2003. Sensor Networks: Evolution, Opportunities, and Challenges. *Proceedings of the IEEE 91,* 8 (August), 1247–1256. IEEE Educational Activities Department.

Chu, M., Haussecker, H., and Zhao, F. 2002. Scalable Information-Driven Sensor Querying and Routing for ad hoc Heterogeneous Sensor Networks. *International Journal of High Performance Computing Applications 16,* 3 (August), 293–313. SAGE Publications.

Chung, E. S., Hong, J. I., Lin, J., Prabaker, M. K., Landay, J. A., and Liu, A. L. 2004. Development and Evaluation of Emerging Design Patterns for Ubiquitous Computing. In *Proceedings of the 2004 Conference on Designing Interactive Systems (DIS'04)*, pp. 233–242. ACM Press.

Contory 2007. Contory. `http://hoslab.cs.helsinki.fi/savane/projects/contory`.

Cooltwon 2007. Cooltown Project. `http://www.hpl.hp.com/archive/cooltown`.

Cooperstock, J. R., Fels, S. S., Buxton, W., and Smith, K. C. 1997. Reactive Environments. *Communications of the ACM 40,* 9 (September), 65–73. ACM Press.

Corson, S. and Macker, J. 1999. Mobile Ad Hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations. RFC 2501 (January), Internet Engineering Task Force. Available at `http://www.ietf.org/rfc/rfc2501.txt`.

Couloris, G., Dollimore, J., and Kindberg, T. 2001. *Distributed Systems Concepts and Design* (3rd ed.). Addison-Wesley.

Crespo, A., Buyukkokten, O., and Garcia-Molina, H. 2003. Query Merging: Improving Query Subscription Processing in a Multicast Environment. *IEEE Transactions on Knowledge and Data Engineering 15,* 1 (January), 174–191. IEEE Educational Activities Department.

Cristian, F. 1991. Understanding fault-tolerant distributed systems. *Communications of the ACM 34,* 2 (February), 56–78. ACM Press.

Cuellar, J., Morris, J., Mulligan, D., Peterson, J., and Polk, J. 2004. Geopriv Requirements. RFC 3693 (February), Internet Engineering Task Force. Available at `http://www.ietf.org/rfc/rfc3693.txt`.

Culler, D., Estrin, D., and Srivastava, M. 2004. Guest Editors' Introduction: Overview of Sensor Networks. *IEEE Computer 37,* 8 (August), 41–49. IEEE Educational Activities Department.

Das, S. R., Perkins, C. E., and Belding-Royer, E. M. 2000. Performance Comparison of Two On-demand Routing Protocols for Ad Hoc Networks. In *Proceedings of the IEEE INFOCOM'00 Conference on Computer Communications*, pp. 3–12.

Davies, N., Friday, A., Wade, S. P., and Blair, G. S. 1998. L2imbo: A Distributed Systems Platform for Mobile Computing. *ACM Mobile Networks and Applications (MONET), Special Issue on Protocols and Software Paradigms of Mobile Networks 3,* 2, 143–156. Kluwer Academic Publishers.

De Couto, D. S. J. and Morris, R. 2001. Location Proxies and Intermediate Node Forwarding for Practical Geographic Forwarding. Technical Report MIT-LCS-TR-824 (June), MIT Laboratory for Computer Science.

Dey, A. K. and Adowd, G. D. 1999. Toward a Better Understanding of Context and Context-Awareness. Technical Report GIT-GVU-99-22 (June), Georgia Institute of Technology, College of Computing.

Dey, A. K., Salber, D., and Abowd, G. 2001. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction 16,* 2-4, 97–166.

Dube, R., Rais, C., Wang, K., and Tripathi, S. 1997. Signal Stability based Adaptive Routing for Ad Hoc Mobile Networks. *IEEE Personal Communications 4,* 1 (February), 36–45. IEEE Educational Activities Department.

Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., and Burgelman, J.-C. 2001. Scenarios for Ambient Intelligence in 2010. *IST Advisory Group Final Report, Eur. Comm.*

DYNAMOS 2007. DYNAMOS Project. `http://virtual.vtt.fi/virtual/proj2/dynamos`.

EasyLiving 2007. EasyLiving Project. `http://research.microsoft.com/easyliving`.

Ember 2007. Ember. `http://www.ember.com`.

Endeavour 2007. Endeavour. `http://endeavour.cs.berkeley.edu`.

Estrin, D., Culler, D., Pister, K., and Sukhatme, G. 2002. Connecting the Physical World with Pervasive Networks. *IEEE*

Pervasive Computing 1, 1 (January), 59–69. IEEE Educational Activities Department.

Estrin, D., Girod, L., Pottie, G., and Srivastava, M. 2001. Instrumenting the World with Wireless Sensor Networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'01)*, Volume 4, pp. 2033–2036.

Fall, K. 2003. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '03)*, pp. 27–34. ACM Press.

Flinn, J., Park, S., and Satyanarayanan, M. 2002. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pp. 217–226. IEEE Computer Society.

Flinn, J. and Satyanarayanan, M. 2004. Managing Battery Lifetime with Energy-Aware Adaptation. *ACM Transactions on Computer Systems (TOCS) 22,* 2 (May), 137–179. ACM Press.

Floyd, S. and Jacobson, V. 1993. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking (TON) 1,* 4 (August), 397–413. IEEE Press.

Fok, C. L., Roman, G. C., and Lu, C. 2005. Mobile Agent Middleware for Sensor Networks: An Application Case Study. In *Proceedings of the Forth International Conference on Information Processing in Sensor Networks (IPSN'05)*, pp. 382–387. IEEE Press.

Ford, L. and Fulkerson, D. 1962. *Flows in Networks.* Princeton Univ. Press.

Foundation for Intelligent Physical Agents 2002a. *FIPA Device Ontology Specification.* Geneva, Switzerland. Specification number SI00091.

Foundation for Intelligent Physical Agents 2002b. *FIPA Quality of Service Ontology Specification.* Geneva, Switzerland. Specification number SC00094.

Fuego Core 2007. Fuego Core Project. `http://hoslab.cs.helsinki.fi/savane/projects/fuego-core`.

Fung, W. F., Sun, D., and Gehrke, J. 2002. COUGAR: The Network Is the Database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pp. 621–621. ACM Press.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Gehrke, J. and Madden, S. R. 2004. Query Processing In Sensor Networks. *IEEE Pervasive Computing 3,* 1, 46–55. IEEE Educational Activities Department.

Gerla, M. 2005. From battlefields to urban grids: New research challenges in ad hoc wireless networks. *Pervasive and Mobile Computing 1,* 1 (March), 77–93. Elsevier B. V.

Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., and Wetherall, D. 2004. System Support for Pervasive Applications. *ACM Transactions on Computer Systems (TOCS) 22,* 4 (November), 421–486. ACM Press.

Grimm, R. and et al 2001. Systems Directions for Pervasive Computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau/Oberbayern, Germany, pp. 147–151. IEEE Computer Society, Washington, DC.

Grossglauser, M. and Tse, D. N. C. 2002. Mobility increases the capacity of ad hoc wireless networks. *IEEE/ACM Transactions on Networking (TON) 10,* 4 (August), 477–486. IEEE Press.

Grudin, J. 2002. Group Dynamics and Ubiquitous Computing. *Commununications of the ACM 45,* 12 (December), 74–78. ACM Press.

Gummadi, R., Gnawali, O., and Govindan, R. 2005. Macroprogramming Wireless Sensor Networks Using Kairos. In *Proceedings of the First IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'05)*, Volume 3560 of *Lecture Notes in Computer Science*, pp. 126–140. Springer-Verlag.

Haas, Z. J. and Pearlman, M. R. 2001. The performance of query control schemes for the zone routing protocol. *IEEE/ACM Transactions on Networking (TON) 9,* 4 (August), 427–438. IEEE Press.

Hadim, S. and Mohamed, N. 2006. Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks. *IEEE Distributed Systems Online 7,* 3, 1. IEEE Educational Activities Department.

He, T., Krishnamurthy, S., Stankovic, J. A., Abdelzaher, T., Luo, L., Stoleru, R., Yan, T., Gu, L., Hui, J., and Krogh, B. 2004. Energy-Efficient Surveillance System using Wireless Sensor Networks. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (MobiSys'04)*, pp. 270–283. ACM Press.

Hedetniemi, S., Hedetniemi, S., and Liestman, A. 1988. A Survey of Gossiping and Broadcasting in Communication Networks. *Networks 18,* 4, 319–349. John Wiley & Sons.

Hedrick, C. L. 1988. RFC 1058: Routing Information Protocol. Available at: `http://www.ietf.org/rfc/rfc1058.txt`.

Heinzelman, W., Murphy, A., Carvalho, H., and Perillo, M. 2004. Middleware to Support Sensor Network Applications. *IEEE Network 18,* 1, 6–14. IEEE Educational Activities Department.

Heinzelman, W. R., Chandrakasan, A., and Balakrishnan, H. 2000. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS'00)*, Volume 8, pp. 8020. IEEE Computer Society.

Heinzelman, W. R., Kulik, J., and Balakrishnan, H. 1999. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)*, pp. 174–185. ACM Press.

Hightower, J. and Borriello, G. 2001. Location Systems for Ubiquitous Computing. *IEEE Computer 34,* 8 (August), 57–66. IEEE Computer Society Press.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System Architecture Directions for Networked Sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pp. 93–104. ACM Press.

Hill, J. L. 2003. System Architecture for Wireless Sensor Networks. Ph. D. thesis, University of California, Berkeley.

Hill, J. L. and Culler, D. E. 2002. Mica: A Wireless Platform for Deeply Embedded Networks. *IEEE Micro 22,* 6, 12–24. IEEE Computer Society Press.

Hohl, F., Mehrmann, L., and Hamdan, A. 2002. A Context System for a Mobile Service Platform. In *Proceedings of the International Conference on Architecture of Computing Systems (ARCS'02)*, pp. 21–33. Springer-Verlag.

Hong, I. and Landay, J. A. 2001. An Infrastrucutre Approach to Context-aware Computing. *Human-Computer Interaction 16,* 2-3, 287–303.

Hong, J. and Landay, J. 2004. An Architecture for Privacy-Sensitive Ubiquitous Computing. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (Mobisys'04)*, pp. 177–189.

Hong, X., Xu, K., and Gerla, M. 2002. Scalable routing protocols for mobile ad hoc networks. *IEEE Network Magazine 16,* 4, 11–21. IEEE Educational Activities Department.

Hou, T.-C. and Li, V. 1986. Transmission range control in multihop packet radio networks. *IEEE Transactions on Communications 34,* 1 (January), 38–44. IEEE Educational Activities Department.

Hull, B., Bychkovsky, V., Zhang, Y., Chen, K., Goraczko, M., Miu, A. K., Shih, E., Balakrishnan, H., and Madden, S. 2006. CarTel: A Distributed Mobile Sensor Computing System. In *Proceedings of the Forth ACM Conference on Embedded Networked Sensor Systems (SenSys'06)*, pp. 125–138. ACM Press.

Intanagonwiwat, C., Govindan, R., and Estrin, D. 2000. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the 6th Snnual International Conference on Mobile Computing and Networking (MobiCom'00)*, pp. 56–67. ACM Press.

Intel Mote 2007. Intel Mote. `http://www.intel.com/research/exploratory/motes.htm`.

Iwata, A., Chiang, C.-C., Pei, G., Gerla, M., and Chen, T.-W. 1999. Scalable Routing Strategies for Ad Hoc Wireless Networks. *IEEE Journal on Selected Areas in Communications, Special Issue on Ad-Hoc Networks 17,* 8 (August), 1369–1379. IEEE Educational Activities Department.

Johnson, D. B. and Maltz, D. A. 1996. Dynamic source routing in ad hoc wireless networks. *Mobile Computing 353*, 153–181. Kluwer Academic Publishers.

JSR 256 2007. JSR 256 ("Mobile Sensor API"). `http://jcp.org/en/jsr/detail?id=256`.

JSR 82 2007. JSR 82 ("Java^{TM} APIs for Bluetooth"). `http://jcp.org/en/jsr/detail?id=82`.

Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L. S., and Rubenstein, D. 2002. Energy-efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pp. 96–107. ACM Press.

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

Kangasharju, J., Tarkoma, S., and Lindholm, T. 2005. Xebu: A Binary Format with Schema-based Optimizations for XML Data. In *Proceedings of the 6th International Conference on Web Information Systems Engineering*, pp. 528–535.

Kansal, A., Somasundara, A. A., Jea, D. D., Srivastava, M. B., and Estrin, D. 2004. Intelligent Fluid Infrastructure for Embedded Networks. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (MobiSys'04)*, pp. 111–124. ACM Press.

Karp, B. and Kung, H. T. 2000. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of the 6th ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'00)*, pp. 243–254.

Katz, R. H. 1994. Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications 1,* 1, 6–17. IEEE Educational Activities Department.

Kehr, R., Zeidler, A., and Vogt, H. 1999. Towards a Generic Proxy Execution Service for Small Devices. *Workshop on Future Services for Networked Devices (FuSeNetD'99).*

Kiess, W. and Mauve, M. 2007. A Survey on Real-World Implementations of Mobile Ad-Hoc Networks. *Ad Hoc Networks 5,* 3 (April), 324–339. Elsevier B.V.

Kindberg, T. and Fox, A. 2002. System Software for Ubiquitous Computing. *IEEE Pervasive Computing 1,* 1, 70–81. IEEE Educational Activities Department.

Kiss, C. 2006. *Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 2.0.* W3C Working Draft.

Kleinrock, L. 2003. An Internet Vision: The Invisible Global Infrastructure. *Ad Hoc Networks 1,* 1 (July), 3–11. Elsevier B.V.

Kling, R. 2003. Intel Mote: An Enhanced Sensor Network Node. In *International Workshop on Advanced Sensors, Structural Health Monitoring, and Smart Structures.*

Ko, Y.-B. and Vaidya, N. H. 1998. Location-Aided Routing (LAR) in Mobile Ad Hoc Networks. In *Proceedings of the Forth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98)*, pp. 66–75.

Korpipää, P. 2005. Blackboard-based software framework and tool for mobile device context awareness. Ph. D. thesis, VTT Electronics, Espoo. Available at: `http://www.vtt.fi/inf/pdf/publications/2005/P579.pdf`.

Kranakis, E., Singh, H., and Urrutia, J. 1999. Compass Routing on Geometric Networks. In *Proceedings of the 11th Canadian Conference on Computational Geometry (CCCG'99)*, pp. 51–54.

Krishnamurthy, L., Adler, R., Buonadonna, P., Chhabra, J., Flanigan, M., Kushalnagar, N., Nachman, L., and Yarvis, M. 2005. Design and Deployment of Industrial Sensor Networks: Experiences from a Semiconductor Plant and the North Sea. In *Proceedings of the Third International Conference on Embedded Networked Sensor Systems (SenSys'05)*, pp. 64–75. ACM Press.

KVM 2007. K Virtual Machine. `http://java.sun.com/products/cldc`.

Landay, J. A. and Borriello, G. 2003. Design Patterns for Ubiquitous Computing. *IEEE Computer 36,* 8 (August), 93–95. IEEE Computer Society Press.

Levis, P. and Culler, D. 2002. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X'02)*, pp. 85–95. ACM Press.

Levis, P., Madden, S., Gay, D., Polastre, J., Szewczyk, R., Woo, A., Brewer, E., and Culler, D. 2004. The Emergence of Networking Abstractions and Techniques in TinyOS. *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'04)*, 1–14.

Li, S., Lin, Y., Son, S. H., Stankovic, J. A., and Wei, Y. 2004. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. *Journal of Telecommunication Systems, special issue on Wireless Sensor Networks 26,* 2–4 (June), 351–368. Kluwer Academic Publishers.

Liu, C. and Kaiser, J. 2005. A Survey of Mobile Ad Hoc network Routing Protocols. Technical report (October), MiNEMA Report. The Survey was published as University of Ulm Tech. Report Series, Nr.2003-08. Available at: `http://www.minema.di.fc.ul.pt/papers.html`.

Liu, J., Chu, M., Liu, J., Reich, J., and Zhao, F. 2003. State-Centric Programming for Sensor-Actuator Network Systems. *IEEE Pervasive Computing 2,* 4 (Oct.-Dec.), 50–62. IEEE Educational Activities Department.

Liu, J. and Issarny, V. 2004. Enhanced Reputation Mechanism for Mobile Ad Hoc Networks. In *Proceedings of the Second International Conference on Trust Management (iTrust2004)*, Volume 2995 of *Lecture Notes in Computer Science*, pp. 48–62. Springer-Verlag.

Liu, J. and Issarny, V. 2005. Signal Strength based Service Discovery (S3D) in Mobile Ad Hoc Networks. In *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05)*, Volume 2, pp. 811–815.

Liu, T. and Martonosi, M. 2003. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pp. 107–118. ACM Press.

Liu, T., Sadler, C. M., Zhang, P., and Martonosi, M. 2004. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (MobiSys'04)*, pp. 256–269. ACM.

Lorincz, K., Malan, D. J., Fulford-Jones, T. R. F., Nawoj, A., Clavel, A., Shnayder, V., Mainland, G., Welsh,

M., and Moulton, S. 2004. Sensor Networks for Emergency Response: Challenges and Opportunities. *IEEE Pervasive Computing 3,* 4, 16–23. IEEE Educational Activities Department.

Lundquist, J. D., Cayan, D. R., and Dettinger, M. D. 2003. Meteorology and Hydrology in Yosemite National Park: A Sensor Network Application. In *Information Processing in Sensor Networks (IPSN'03)*, pp. 518–528.

Luo, J., Eugster, P., and Hubaux, J. 2004. Pilot: Probabilistic lightweight group communication system for ad hoc networks. In *IEEE Transactions on Mobile Computing*, Volume 3, pp. 164–179.

Lyytinen, K. and Yoo, Y. 2002. Issues and Challenges in Ubiquitous Computing, Introduction. *Commununications of the ACM 45,* 12 (December), 62–65. ACM Press.

Macias, J. A. and y, J. G. 2006. WSN and MANET: Are they alike? *Book Chapter in Sensor Network and Configuration: Fundamentals, Techniques, Platforms, and Experiments*. Springer-Verlag.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2002. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *ACM SIGOPS Operating Systems Review 36,* SI (Winter 2002), 131–146. ACM Press.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2003. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pp. 491–502. ACM Press.

Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2005. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems (TODS) 30,* 1, 122–173. ACM Press.

Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., and Anderson, J. 2002. Wireless Sensor Networks for Habitat

Monitoring. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, pp. 88–97. ACM Press.

manet IETF WG 2007. Mobile Ad-hoc Networks (manet) IETF Working Group. `http://www.ietf.org/html.charters/manet-charter.html`.

Mascolo, C., Capra, L., and Emmerich, W. 2002. Mobile Computing Middleware. *Advanced Lectures on Networking*, 20–58. Springer-Verlag.

Mascolo, C., Capra, L., Zachariadis, S., and Emmerich, W. 2002. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Wireless Personal Communications 21,* 1, 77–103. Kluwer Academic Publishers.

Mauve, M., Widmer, A., and Hartenstein, H. 2001. A Survey on Position-Based Routing in Mobile Ad-Hoc Networks. *IEEE Network 15,* 6, 30–39. IEEE Educational Activities Department.

Mayer, K., Taylor, K., and Ellis, K. 2004. Cattle Health Monitoring using Wireless Sensor Networks. In *Proceedings of the Second IASTED International Conference on Communication and Computer Networks (CCN'04)*, pp. 64–75.

McQuillan, J., Richer, I., and Rosen, E. 1978. ARPANET Routing Algorithm Improvements - First Semiannual Technical Report. BBN Report 3803 (April).

Meier, R. and Cahill, V. 2002. STEAM: Event-Based Middleware for Wireless Ad Hoc Network. In *Proceedings of the First International Workshop on Distributed Event-Based Systems (DEBS'02)*, pp. 639–644. IEEE Computer Society.

MetroSense 2007. MetroSense Project. `http://metrosense.cs.dartmouth.edu`.

Meyer, S. and Rakotonirainy, A. 2003. A Survey of Research on Context-aware Homes. In *Proceedings of the Australasian Information Security Workshop Conference on ACSW Frontiers 2003 (CRPITS'03)*, Volume 21, pp. 159–168. Australian Computer Society, Inc.

Michahelles, F., Matter, P., Schmidt, A., and Schiele, B. 2003. Applying Wearable Sensors to Avalanche Rescue. *Computers and Graphics 27,* 6, 839–847. Elsevier B.V.

Michiardi, P. and Molva, R. 2002. Core: A Collaborative Reputation Mechanism to enforce node cooperation in Mobile Ad Hoc Networks. In *Proceedings of the IFIP TC6/TC11 Sixth Joint Working Conference on Communications and Multimedia Security*, pp. 107–121. Kluwer, B.V.

Migratory Services 2007. Context-aware Migratory Services. `http://hoslab.cs.helsinki.fi/savane/projects/msf/`.

MITes 2007. MIT Environmental sensors: A Portable Kit of Wireless Sensors for Naturalistic Data Collection. `http://web.media.mit.edu/~emunguia/miteswebsite`.

Monarch 2007. Monarch Project. `http://www.monarch.cs.rice.edu`.

Mui, L., Mohtashemi, M., and Halberstadt, A. 2002. A computational model of trust and reputation. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, pp. 2431–2439.

Murphy, A. L., Picco, G. P., and Roman, G.-C. 2001. LIME: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01)*, pp. 524. IEEE Computer Society.

Murthy, S. and Garcia-Luna-Aceves, J. J. 1996. An Efficient Routing Protocol for Wireless Networks. *ACM Mobile Networks and Applications 1,* 2 (October), 183–197. Kluwer Academic Publishers.

Musolesi, M., Mascolo, C., and Hailes, S. 2005. EMMA: Epidemic Messaging Middleware for Ad hoc networks. *Personal Ubiquitous Computing 10,* 1, 28–36. Springer-Verlag.

Nadeem, T., Dashtinezhad, S., Liao, C., and Iftode, L. 2004. TrafficView: Traffic Data Dissemination using Car-to-Car Communication. *ACM Mobile Computing and Communications Review (MC2R) 8,* 3 (July), 6–19. ACM Press.

Navas, J. C. and Imielinski, T. 1997. GeoCast - Geographic Addressing and Routing. In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'97)*, pp. 66–76. ACM Press.

Nelson, R. and Kleinrock, L. 1984. The Spatial Capacity of a Slotted ALOHA Multihop Packet Radio Network with Capture. *IEEE Transactions on Communications 32,* 6 (June), 684–694. IEEE Educational Activities Department.

Ni, Y. 2006. Programming ad hoc networks. Ph. D. thesis, Rutgers University, Department of Computer Science.

Ni, Y., Kremer, U., Stere, A., and Iftode, L. 2005. Programming Ad-Hoc Networks of Mobile and Resource-Constrained Devices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pp. 249–260. ACM Press.

Noble, B. D. and Satyanarayanan, M. 1999. Experience with adaptive mobile applications in Odyssey. *Mobile Networks and Applications 4,* 4 (December), 245–254. Kluwer Academic Publishers.

Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. 1997. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP'97)*, pp. 276–287. ACM Press.

NS-2 2007. Network Simulator ns-2. `http://www.isi.edu/nsnam/ns`.

Olsen, C. M. and Narayanaswami, C. 2006. PowerNap: An Efficient Power Management Scheme for Mobile Devices. *IEEE Transactions on Mobile Computing 5,* 7, 816–828. IEEE Educational Activities Department.

Oxygen 2007. Oxygen Project. `http://oxygen.csail.mit.edu`.

Ozdemir, O., Ray, P., Isik, C., C.K.Mohan, Varshney, P., Khalifa, H., and Zhang, J. 2005. Application of Wireless

Sensor Networks for AI-based Monitoring and Control of Built Environments. *Innovations and Commercial Applications of Distributed Sensor Networks (ICA DSN).*

Paradiso, J. A. and Starner, T. 2005. Energy Scavenging for Mobile and Wireless Electronics. *IEEE Pervasive Computing 4,* 1, 18–27. IEEE Educational Activities Department.

Park, V. D. and Corson, M. S. 1997. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *Proceedings of the IEEE INFOCOM'97 Conference on Computer Communications*, Volume 3, pp. 1405–1413.

Perkins, C. and Royer, E. 1999. Ad-Hoc On-Demand Distance Vector Routing. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, pp. 90–100.

Perkins, C. E. and Bhagwat, P. 1994. Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM'94)*, pp. 234–244. ACM Press.

Pon, R., Batalin, M. A., Gordon, J., Kansal, A., Liu, D., Rahimi, M., Shirachi, L., Yu, Y., Hansen, M., Kaiser, W. J., Srivastava, M., Sukhatme, G., and Estrin, D. 2005. Networked Infomechanical Systems: A Mobile Embedded Networked Sensor Platform. In *Proceedings of the Forth International Conference on Information Processing in Sensor Networks (IPSN'05)*, pp. 376–381. IEEE Press.

Portolano 2007. Portolano Project. `http://portolano.cs.washington.edu`.

Raatikainen, K. 2005. A New Look at Mobile Computing. In *Proceedings of the International Workshop on Convergent Technologies (IWCT'05).*

Raatikainen, K., Christensen, H., and Nakajima, T. 2002. Application Requirements for Middleware for Mobile and Pervasive Systems. *ACM Mobile Computing and Communications Review (MC2R) 6,* 4 (October), 16–24.

Raento, M., Oulasvirta, A., Petit, R., and Toivonen, H. 2005. ContextPhone: A Prototyping Platform for Context-Aware Mobile Applications. *IEEE Pervasive Computing 4,* 2 (Jan-March), 51–59. IEEE Educational Activities Department.

Ranganathan, A., Al-Muhtadi, J., Chetan, S., Campbell, R., and Mickunas, M. D. 2004. MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference (Middleware'04)*, Volume 3231 of *Lecture Notes in Computer Science*, pp. 397–416. Springer-Verlag.

Ranganathan, A. and Campbell, R. H. 2003. A Middleware for Context-Aware Agents in Ubiquitous Computing Environments. *In ACM/IFIP/USENIX International Middleware Conference (Middleware'03)*, 143–161.

Raverdy, P.-G., Riva, O., de La Chapelle, A., Chibout, R., and Issarny, V. 2006. Efficient Context-aware Service Discovery in Multi-Protocol Pervasive Environments. In *Proceedings of the 7th International Conference on Mobile Data Management (MDM'06)*, Volume 00, pp. 3–10. IEEE Computer Society.

Ravi, N., Borcea, C., Kang, P., and Iftode, L. 2004. Portable Smart Messages for Ubiquitous Java-Enabled Devices. In *The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)*, pp. 412–421. IEEE Computer Society.

Riekki, J., Huhtinen, J., Ala-Siuru, P., Alahuhta, P., Kaartinen, J., and Röning, J. 2003. Genie of the Net, an Agent Platform for Managing Services on Behalf of the User. *Computer Communications 26,* 11, 1188–1198.

Riekki, J., Isomursu, P., and Isomursu, M. 2004. Evaluating the Calmness of Ubiquitous Applications. In *5th International Conference on Product Focused Software Process Improvement (PROFES'04)*, Volume 3009 of *Lecture Notes in Computer Science*, pp. 105–119. Springer-Verlag.

Riva, O. 2006. Contory: A Middleware for the Provisioning of Context Information on Smart Phones. In *Proceedings of the 7th ACM International Middleware Conference (Middleware'06)*, Volume 4290 of *Lecture Notes in Computer Science*, pp. 219–239. Springer-Verlag.

Riva, O. 2007. Bibliography on Context-Awareness Support. Unpublished. Available at: `http://www.cs.helsinki.fi/u/riva/cas.pdf`.

Riva, O. and Borcea, C. 2007. The Urbanet Revolution: Sensor Power to the People! *IEEE Pervasive Magazine 6,* 2 (April-June), 41–49. IEEE Educational Activities Department.

Riva, O. and di Flora, C. 2006a. Contory: A Smart Phone Middleware Supporting Multiple Context Provisioning Strategies. In *Second IEEE International Workshop on Services and Infrastructure for the Ubiquitous and Mobile Internet (SIUMI'06)*. IEEE Computer Society.

Riva, O. and di Flora, C. 2006b. Unearthing Design Patterns to Support Context-Awareness. In *Third IEEE International Workshop on Middleware Support for Pervasive Computing (PerWare'06)*. Pisa, Italy: IEEE Computer Society.

Riva, O. and Kangasharju, J. 2007. Challenges and Lessons in Developing Middleware on Smart Phones. Under submission. Available at: `http://www.cs.helsinki.fi/u/riva/publications/riva_ieeecomputer07.pdf`.

Riva, O., Nadeem, T., Borcea, C., and Iftode, L. 2007. Context-aware Migratory Services in Ad Hoc Networks. *IEEE Transactions on Mobile Computing 6,* 12 (December), 1313–1328. IEEE Educational Activities Department.

Riva, O. and Toivonen, S. 2006. A Model of Hybrid Service Provisioning Implemented on Smart Phones. In *Proceedings of the Third IEEE International Conference on Pervasive Services (ICPS'06)*, pp. 47–56. IEEE Computer Society.

Riva, O. and Toivonen, S. 2007. The DYNAMOS Approach to Context-Aware Service Provisioning in Mobile Environments.

Journal of Systems and Software 80, 12 (December), 1956–1972. Elsevier B.V.

Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K. 2002. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing 1,* 4, 74–83. IEEE Educational Activities Department.

Römer, K. and Mattern, F. 2004. The Design Space of Wireless Sensor Networks. *IEEE Wireless Communications 11,* 6 (December), 54–61. IEEE Educational Activities Department.

Roussos, G., Marsh, A. J., and Maglavera, S. 2005. Enabling Pervasive Computing with Smart Phones. *IEEE Pervasive Computing 4,* 2, 20–27. IEEE Educational Activities Department.

Royer, E. M. and Toh, C.-K. 1999. A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks. *IEEE Personal Communications 6,* 2 (April), 46–55. IEEE Educational Activities Department.

Sadagopan, N., Krishnamachari, B., and Helmy, A. 2005. Active Query Forwarding in Sensor Networks. *Ad Hoc Networks 3,* 1, 91–113. Elsevier B.V.

Saha, D. and Mukherjee, A. 2003. Pervasive Computing: A Paradigm for the 21st Century. *IEEE Computer 36,* 3 (March), 25–31. IEEE Educational Activities Department.

Salem Hadim and Jameela Al-Jaroodi and Nader Mohamed 2006. Trends in Middleware for Mobile Ad Hoc Networks. *Journal of Communications (JCM) 1,* 4 (July), 11–21.

Satyanarayanan, M. 1996. Fundamental Challenges in Mobile Computing. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC'96)*, pp. 1–7. ACM Press.

Satyanarayanan, M. 2001. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications 8,* 4 (August), 10–17. IEEE Educational Activities Department.

Schilit, B. N., Adams, N. L., and Want, R. 1994. Context-Aware Computing Applications. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, pp. 85–90. IEEE Computer Society Press.

Schmidt, A. 2000. Implicit Human-Computer Interaction through Context. *Personal Technologies 4,* 2-3 (June), 191–199. Springer London.

Schmidt, A., Adoo, K. A., Takaluoma, A., Tuomela, U., Laerhoven, K. V., and de Velde, W. V. 1999. Advanced Interaction in Context. In *Proceedings of the First Symposium on Handheld and Ubiquitous Computing (HUC'99)*, Karlsruhe, Germany, pp. 89–101.

Schurgers, C., Tsiatsis, V., Ganeriwal, S., and Srivastava, M. 2002. Optimizing Sensor Networks in the Energy-Latency-Density Design Space. *IEEE Transactions on Mobile Computing 1,* 1, 70–80. IEEE Educational Activities Department.

SenseWeb 2007. SenseWeb Project. `http://research.microsoft.com/nec/senseweb`.

SensorPlanet 2007. Nokia SensorPlanet Project. `http://www.sensorplanet.org`.

Shankar, C., Al-Muhtadi, J., Campbell, R., and Mickunas, M. D. 2005. Mobile Gaia: A Middleware for Ad hoc Pervasive Computing. *IEEE Consumer Communications and Networking Conference (CCNC'05)*, 223–228.

Shen, C.-C., Srisathapornphat, C., and Jaikaeo, C. 2001. Sensor Information Networking Architecture and Applications. *IEEE Personal Communication Magazine 8,* 4 (August), 52–59. IEEE Educational Activities Department.

Singh, S. and Raghavendra, C. S. 1998. PAMAS: Power Aware Multi-Access Protocol with Signalling for Ad Hoc Networks. *ACM SIGCOMM Computer Communication Review 28,* 3 (July), 5–26. ACM Press.

Sloman, M. 1994. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management 2,* 4, 333–360. Springer Netherlands.

Smarts-Its 2007. Smart-Its Project. `http://www.smart-its.org`.

Soot 2007. Soot: A Java Optimization Framework. `http://www.sable.mcgill.ca/soot`.

Souto, E., aes, G. G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C., and Kelner, J. 2005. Mires: A Publish/Subscribe Middleware for Sensor Networks. *Personal Ubiquitous Computing 10,* 1, 37–44. Springer-Verlag.

Stojmenovic, I. and Lin, X. 2001. Power-Aware Localized Routing in Wireless Networks. *IEEE Transactions on Parallel and Distributes Systems 12,* 11 (November), 1122–1133.

Sun, J.-Z. 2001. Mobile Ad-hoc Networking: An Essential Technology for Pervasive Computing. In *Proceedings of International Conferences on Info-tech and Info-net*, Volume 3, pp. 316–321.

Szewczyk, R., Osterweil, E., Polastre, J., Hamilton, M., Mainwaring, A. M., and Estrin, D. 2004. Habitat Monitoring with Sensor Networks. *Commununications of the ACM 47,* 6 (June), 34–40. ACM Press.

Tafazolli, R. (Ed.) 2004. *Technologies for the Wireless Future: Wireless World Research Forum (WWRF).* John-Wiley & Sons.

Tafazolli, R. (Ed.) 2006. *Technologies for the Wireless Future: Wireless World Research Forum (WWRF), Volume 2.* John-Wiley & Sons.

Takagi, H. and Kleinrock, L. 1984. Optimal Transmission Ranges for Randomly Distributed Packet Radio Terminals. *IEEE Transactions on Communications 32,* 3 (March), 246–257. IEEE Educational Activities Department.

Tarkoma, S., Kangasharju, J., Lindholm, T., and Raatikainen, K. 2006. Fuego: Experiences with Mobile Data Communication and Synchronization. In *Proceedings of the 17th*

Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC'06), pp. 1–5.

TinyOS 2007. TinyOS. `http://www.tinyos.net`.

Tiwari, A., Ballal, P., and Lewis, F. L. 2007. Energy-Efficient Wireless Sensor Network Design and Implementation for Condition-based Maintenance. *ACM Transactions on Sensor Networks 3,* 1, 1–23. ACM Press.

Toh, C.-K. 1996. A Novel Distributed Routing Protocol To Support Ad hoc Mobile Computing. In *Proceedings of the IEEE 15th Annual International Phoenix Conference on Computers and Communications, (IPCCC'96)*, pp. 480–486. IEEE.

Tubaishat, M. and Madria, S. 2003. Sensor Networks: An Overview. *IEEE Potentials 22,* 2, 20–23. IEEE Educational Activities Department.

UrbanSensing 2007. Urban Sensing Project. `http://censweb.ats.ucla.edu/projects/2006/Systems/Urban_Sensing/`.

Vasilescu, I., Kotay, K., Rus, D., Dunbabin, M., and Corke, P. 2005. Data Collection, Storage, and Retrieval with an Underwater Sensor Network. In *Proceedings of the Third International Conference on Embedded Networked Sensor Systems (SenSys'05)*, pp. 154–165. ACM Press.

Verykios, V. S., Bertino, E., Fovino, I. N., Provenza, L. P., Saygin, Y., and Theodoridis, Y. 2004. State-of-the-art in Privacy Preserving Data Mining. *ACM SIGMOD Record 33,* 1, 50–57. ACM Press.

Waldo, J. 2005. Embedded Computing and Formula One Racing. *IEEE Pervasive Computing 4,* 3 (July-September), 18–21. IEEE Educational Activities Department.

Want, R., Borriello, G., Pering, T., and Farkas, K. I. 2002. Disappearing Hardware. *IEEE Pervasive Computing 1,* 1, 36–47. IEEE Educational Activities Department.

Want, R., Hopper, A., Falcao, V., and Gibbons, J. 1992. The Active Badge Location System. *ACM Transactions on Information Systems (TOIS) 10,* 1, 91–102. ACM Press.

WAP Forum 2001. *User Agent Profiling Specification. WAG UAProf. Version 20-Oct-2001.* WAP-248-UAProf-20011020-a.

Weiser, M. 1991. The Computer for the Twenty-First Century. *Scientific American 265,* 3 (September), 94–104.

Weiser, M. 1994. The World is not a Desktop. *ACM Interactions 1,* 1 (January), 7–8. ACM Press.

Weiser, M. and Brown, J. S. 1997. The Coming Age of Calm Technolgy. In P. J. Denning and R. M. Metcalfe (Eds.), *In Beyond Calculation: the Next Fifty Years*, New York, NY, pp. 75–85. Copernicus.

Welsh, M. and Mainland, G. 2004. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'04)*, pp. 29–42.

Werner-Allen, G., Lorincz, K., Welsh, M., Marcillo, O., Johnson, J., Ruiz, M., and Lees, J. 2006. Deploying a Wireless Sensor Network on an Active Volcano. *IEEE Internet Computing 10,* 2 (March), 18–25. IEEE Educational Activities Department.

Whitehouse, K., Liu, J., and Zhao, F. 2006. Semantic Streams: a Framework for Composable Inference over Sensor Data. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN'06)*, Volume 3868 of *Lecture Notes in Computer Science*. Springer-Verlag.

Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. 2004. Hood: A Neighborhood Abstraction for Sensor Networks. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (MobiSys'04)*, pp. 99–110. ACM.

Wies, R. 1994. Policies in Network and Systems Management - Formal Definition and Architecture. *Journal of Networks and Systems Management 2,* 1 (March), 63–83. Springer Netherlands.

Wireless World Research Forum 2001. *Book of Visions 2001.* `http://www.wireless-world-research.org/`.

Woo, K., Yu, C., Lee, D., Youn, H. Y., and Lee, B. 2001. Non-Blocking, Localized Routing Algorithm for Balanced Energy Consumption in Mobile Ad Hoc Networks. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01)*, pp. 117–124. IEEE Computer Society.

White, G., Kangasharju, J., Brutzman, D., and Williams, S. (Eds.) 2007. *Efficient XML Interchange Measurements Note*. World Wide Web Consortium. W3C Working Draft.

WRC 2007. Resource Description Framework (RDF). `http://www.w3.org/RDF`.

Xu, N., Rangwala, S., Chintalapudi, K. K., Ganesan, D., Broad, A., Govindan, R., and Estrin, D. 2004. A Wireless Sensor Network for Structural Monitoring. In *Proceedings of the Second International Conference on Embedded networked Sensor Systems (SenSys'04)*, pp. 13–24. ACM Press.

Xu, Y., Heidemann, J., and Estrin, D. 2001. Geography-informed Energy Conservation for Ad Hoc Routing. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MobiCom'01)*, pp. 70–84. ACM Press.

Yao, Y. and Gehrke, J. 2002. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM SIGMOD Record 31,* 3 (September), 9–18. ACM.

Yao, Y. and Gehrke, J. 2003. Query Processing in Sensor Networks. In *Proceedings of the First Biennal Conference on Innovative Data Systems Research (CIDR'03)*, pp. 233–244. ACM Press.

Yau, S. and Karim, F. February 2004. A context-sensitive middleware for dynamic integration of mobile devices with network infrastructures. *Journal Parallel Distributed Computing 64,* 2, 301–317. Academic Press, Inc.

Yau, S. S., Karim, F., Wang, Y., Wang, B., and Gupta, S. 2002. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. *IEEE Pervasive Computing 1,* 3 (July-Sept), 33–40. IEEE Educational Activities Department.

Ye, W., Heidemann, J., and Estrin, D. 2002. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the IEEE INFOCOM'02 Conference on Computer Communications*, Volume 3, pp. 1567–1576.

Yoneki, E. 2005. Evolution of Ubiquitous Computing with Sensor Networks in Urban Environments. *Workshop on Metapolis and Urban Life*.

Yu, C., Lee, B., and Youn, H. Y. 2003. Energy Efficient Routing Protocols for Mobile Ad Hoc Networks. *Wireless Communications and Mobile Computing Journal 3,* 8 (December), 959–973. John-Wiley & Sons.

Yu, Y., Govindan, R., and Estrin, D. 2001. Geographical and Energy Aware Routing: A Recursive Data Dissemination Protocol for Wireless Sensor Networks. Technical Report UCLA/CSD-TR-01-0023 (May), UCLA Computer Science Department Technical Report.

Zhang, W., Kantor, G., and Singh, S. 2004. Integrated Wireless Sensor/Actuator Networks in an Agricultural Application. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems (SenSys'04)*, pp. 317–317. ACM Press.

Zhong, S., Chen, J., and Yang, Y. R. 2003. Sprite: A Simple, Cheat-Proof, Credit-Based System for Mobile Ad-Hoc Networks. In *Proceedings of the IEEE INFOCOM'03 Conference on Computer Communications*, pp. 1987–1997.

Zhou, P., Nadeem, T., Kang, P., Borcea, C., and Iftode, L. 2005. EZCab: A Cab Booking Application Using Short-Range Wireless Communication. In *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications (PerCom'05)*, pp. 27–38. IEEE Computer Society.

Zhu, D. and Mutka, M. W. 2004. Promoting cooperation among strangers to access internet services from an ad hoc network. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, pp. 229–240. IEEE Computer Society.

TIETOJENKÄSITTELYTIETEEN LAITOS
PL 68 (Gustaf Hällströmin katu 2 b)
00014 Helsingin yliopisto

DEPARTMENT OF COMPUTER SCIENCE
P.O. Box 68 (Gustaf Hällströmin katu 2 b)
FIN-00014 University of Helsinki, FINLAND

JULKAISUSARJA **A**

SERIES OF PUBLICATIONS **A**

Reports may be ordered from: Kumpula Science Library, P.O. Box 64, FIN-00014 University of Helsinki, FINLAND.

A-2000-1 P. Moen: Attribute, event sequence, and event type similarity notions for data mining. 190+9 pp. (Ph.D. thesis).

A-2000-2 B. Heikkinen: Generalization of document structures and document assembly. 179 pp. (Ph.D. thesis).

A-2000-3 P. Kähkipuro: Performance modeling framework for CORBA based distributed systems. 151+15 pp. (Ph.D. thesis).

A-2000-4 K. Lemström: String matching techniques for music retrieval. 56+56 pp. (Ph.D.Thesis).

A-2000-5 T. Karvi: Partially defined Lotos specifications and their refinement relations. 157 pp. (Ph.D.Thesis).

A-2001-1 J. Rousu: Efficient range partitioning in classification learning. 68+74 pp. (Ph.D. thesis)

A-2001-2 M. Salmenkivi: Computational methods for intensity models. 145 pp. (Ph.D. thesis)

A-2001-3 K. Fredriksson: Rotation invariant template matching. 138 pp. (Ph.D. thesis)

A-2002-1 A.-P. Tuovinen: Object-oriented engineering of visual languages. 185 pp. (Ph.D. thesis)

A-2002-2 V. Ollikainen: Simulation techniques for disease gene localization in isolated populations. 149+5 pp. (Ph.D. thesis)

A-2002-3 J. Vilo: Discovery from biosequences. 149 pp. (Ph.D. thesis)

A-2003-1 J. Lindström: Optimistic concurrency control methods for real-time database systems. 111 pp. (Ph.D. thesis)

A-2003-2 H. Helin: Supporting nomadic agent-based applications in the FIPA agent architecture. 200+17 pp. (Ph.D. thesis)

A-2003-3 S. Campadello: Middleware infrastructure for distributed mobile applications. 164 pp. (Ph.D. thesis)

A-2003-4 J. Taina: Design and analysis of a distributed database architecture for IN/GSM data. 130 pp. (Ph.D. thesis)

A-2003-5 J. Kurhila: Considering individual differences in computer-supported special and elementary education. 135 pp. (Ph.D. thesis)

A-2003-6 V. Mäkinen: Parameterized approximate string matching and local-similarity-based point-pattern matching. 144 pp. (Ph.D. thesis)

A-2003-7 M. Luukkainen: A process algebraic reduction strategy for automata theoretic verification of untimed and timed concurrent systems. 141 pp. (Ph.D. thesis)

A-2003-8 J. Manner: Provision of quality of service in IP-based mobile access networks. 191 pp. (Ph.D. thesis)

A-2004-1 M. Koivisto: Sum-product algorithms for the analysis of genetic risks. 155 pp. (Ph.D. thesis)

A-2004-2 A. Gurtov: Efficient data transport in wireless overlay networks. [B 141 pp. (Ph.D. thesis)

A-2004-3 K. Vasko: Computational methods and models for paleoecology. 176 pp. (Ph.D. thesis)

A-2004-4 P. Sevon: Algorithms for Association-Based Gene Mapping. 101 pp. (Ph.D. thesis)

A-2004-5 J. Viljamaa: Applying Formal Concept Analysis to Extract Framework Reuse Interface Specifications from Source Code. 206 pp. (Ph.D. thesis)

A-2004-6 J. Ravantti: Computational Methods for Reconstructing Macromolecular Complexes from Cryo-Electron Microscopy Images. 100 pp. (Ph.D. thesis)

A-2004-7 M. Kääriäinen: Learning Small Trees and Graphs that Generalize. 45+49 pp. (Ph.D. thesis)

A-2004-8 T. Kivioja: Computational Tools for a Novel Transcriptional Profiling Method. 98 pp. (Ph.D. thesis)

A-2004-9 H. Tamm: On Minimality and Size Reduction of One-Tape and Multitape Finite Automata. 80 pp. (Ph.D. thesis)

A-2005-1 T. Mielikäinen: Summarization Techniques for Pattern Collections in Data Mining. 201 pp. (Ph.D. thesis)

A-2005-2 A. Doucet: Advanced Document Description, a Sequential Approach. 161 pp. (Ph.D. thesis)

A-2006-1 A. Viljamaa: Specifying Reuse Interfaces for Task-Oriented Framework Specialization. 285 pp. (Ph.D. thesis)

A-2006-2 S. Tarkoma: Efficient Content-based Routing, Mobility-aware Topologies, and Temporal Subspace Matching. 198 pp. (Ph.D. thesis)

A-2006-3 M. Lehtonen: Indexing Heterogeneous XML for Full-Text Search. 185+3 pp.(Ph.D. thesis).

A-2006-4 A. Rantanen: Algorithms for ^{13}C Metabolic Flux Analysis. 92+73 pp.(Ph.D. thesis).

A-2006-5 E. Terzi: Problems and Algorithms for Sequence Segmentations. 141 pp. (Ph.D. Thesis).

A-2007-1 P. Sarolahti: TCP Performance in Heterogeneous Wireless Networks.(Ph.D. Thesis).

A-2007-2 M. Raento: TCP Exploring privacy for ubiquitous computing: Tools, methods and experiments. (Ph.D. thesis).

A-2007-3 L. Aunimo: Methods for Answer Extraction in Textual Question Answering 127+18 pp. (Ph.D. Thesis).

A-2007-4 T. Roos: Statistical and Information-Theoretic Methods for Data Analysis. 82+75pp. (Ph.D. Thesis).

A-2007-5 S. Leggio: A Decentralized Session Management Framework for Heterogeneous Ad-Hoc and Fixed Networks 230 pp. (Ph.D. Thesis).