The Urbanet Revolution: Sensor Power to the People!

With mobile devices becoming ubiquitous, the time is ripe to bring sensor data out of close-loop networks into the center of daily urban life.

Oriana Riva
Helsinki Institute for Information Technology

Cristian Borcea
New Jersey Institute of Technology

The Internet has become a great success because its applications appeal to regular people. This isn't the case with sensor networks, which are generally perceived as "something" remote in the forest or on the battlefield. With few exceptions, first-generation sensor networks address application-specific, static-sensor deployments to accurately monitor the sensed environment in real time. Commonly, these networks involve a centralized data-collection point and no sharing of data outside the organization that owns it. Although this operational model can accommodate many application scenarios, it significantly deviates from the pervasive computing vision of ambient intelligence, where people seamlessly access anytime, anywhere data produced by sensors embedded in the surroundings.

Urban environments offer the elements to build large-scale, people-centric sensing platforms. As mobile devices become ubiquitous, they can begin to serve purposes beyond email and Web access:

- acting as collaborating, multisensor 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 applications.

In particular, smart phones and vehicular systems are becoming attractive, convenient mobile sensor platforms. Compared to the 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. In the near future, they will include other sensors as well. However, energy remains a main constraint for them. Vehicular systems, on the other hand, don't have energy restrictions. Additionally, they offer powerful processors, significant memory, plenty of storage capacity, and many types of sensors.

Spontaneous urban networks

We use the term *Urbanet* to define a new type of spontaneous urban network composed of heterogeneous and potentially mobile sensors. In Urbanets, sensor networks and mobile ad hoc networks (MANETs) meet to create rich, open sensing environments where people, municipalities, and community organizations can share their resources to give mobile users real-time access to sensed data. Much of this data will be incorporated in novel applications running on our personal mobile devices. Urbanets differ from first-generation sensor networks not only in their goal to support concurrent people-centric sensing applications across cities but also in their hardware and software heterogeneity, high volatility, and very large scale. Although Urbanets will extend sensing coverage and let developers incorporate

1536-1268/07/$25.00 © 2007 IEEE ■ *Published by the IEEE Computer Society*

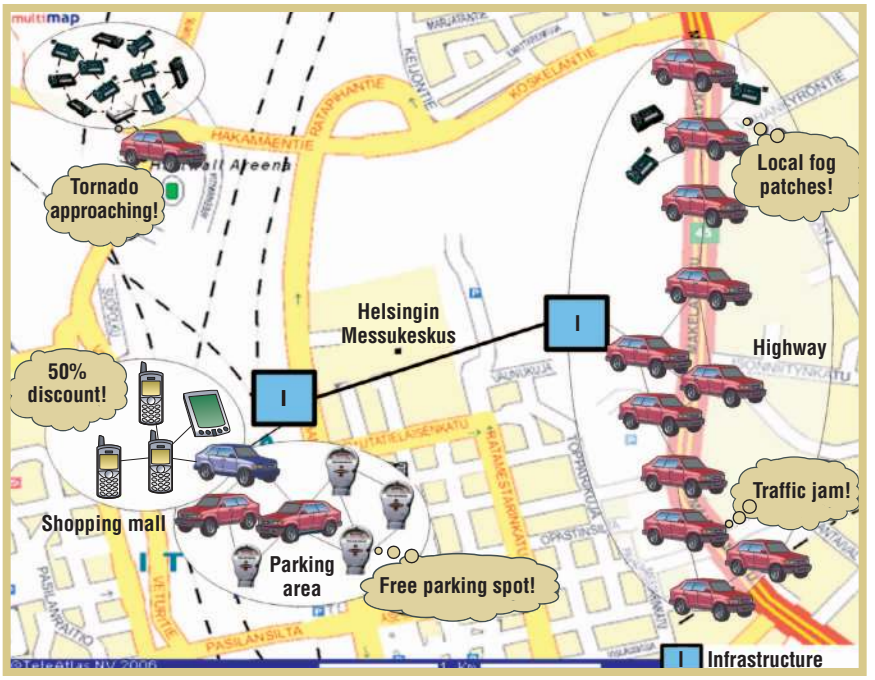

Figure 1. Example Urbanet scenarios. A network of cars and environmental sensors cooperate to warn drivers about upcoming traffic jams, recommend alternative routes, and detect traffic hazards such as fog patches. Smart phones or vehicular systems can query parking meters on streets adjacent to the shopping mall and offer directions to free spots. In the mall, smart phones read product RFIDs, exchange product information, and give users personalized recommendations. Farther away, a municipal weather-monitoring sensor network alerts cars to an approaching tornado.

sensed data in a large spectrum of mobile applications, they aren't expected to achieve the same sensing-fidelity level of static sensor networks whose nodes are primarily dedicated to sensing. Urbanets also differ fundamentally from MANETs in terms of enabled applications. So far, most MANET applications have focused on traditional file transfers. Instead, Urbanet applications acquire, process, and distribute real-time sensing information from devices in proximity of regions, entities, or activities of interest. Figure 1 depicts several Urbanet scenarios.

Several recent projects confirm a growing interest in large-scale people-centric sensor networks. For example, the "Related Work" sidebar presents projects under way at Microsoft, the Massachusetts Institute of Technology, Dartmouth University, and the University of California, Los Angeles. Although we share broad goals with these projects, our focus is on cooperative, infrastructureless solutions for programming mobile sensing applications in Urbanets. We present middleware platforms that capitalize on spontaneously created sensing coverage provided by Urbanets to enable applications running on mobile devices. These platforms let applications collect real-time sensed data each time they want to, even when Internet connectivity isn't available or is too expensive.

Distributed programming in Urbanets

The research community has been quite successful in designing device platforms, protocols, and even network architectures that can apply to Urbanets. However, programming people-centric, mobile sensing applications has received just marginal attention. As the Urbanet applications domain diversifies, programming each application from scratch will be almost impossible. So, we expect an increasing demand for a common distributed-computing platform to support the development, deployment, and execution of such applications. Several questions drive our research: How do we define a user application in Urbanets? What are the right programming abstractions for Urbanets? What type of middleware or runtime systems can support such programming abstractions? What are the trade-offs between ease of programming and efficiency? Or, in other words, how much of the underlying network complexity can a platform hide from programmers while still giving them enough flexibility in application development?

Urbanets can't 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; they evolve unpredictably over time and space, making it hard to know the exact number or location of their resources. Such volatility requires more flexible programming models.

Furthermore, traditional models assume fixed bindings between names and node addresses. This naming is too rigid for Urbanets, where contextual properties—such as available sensors, location, computational resources, or energy—determine the nodes of interest. Urbanet applications require data-centric or property-based naming that sensor networks use.

To be useful, Urbanet applications can't fail each time something goes wrong in the network. Also, the applications must work even when confronted with highly variable sensor data fidelity. This makes "best effort" semantics desirable—an approach that can tolerate the network dynamics while providing applications a certain quality of results. An Urbanet middleware platform must let applications trade the quality of results for network resources.

Unlike sensor network nodes, Urbanet nodes aren't always

Related Work

Urbanets share the goal of building people-centric urban sensor networks with several recent projects:

- SenseWeb (http://research.microsoft.com/nec/senseweb) aims to provide a Web-based platform and tools that let people easily publish and query sensor data. SenseWeb can use sensor metadata to dispatch and answer queries in real time.
- CarTel (http://cartel.csail.mit.edu) focuses on building a delay-tolerant mobile sensing architecture based on opportunistic communication. A continuous query processor running on a central portal answers user queries.
- MetroSense (http://metrosense.cs.dartmouth.edu) proposes a three-tier architecture for scalable support of concurrent people-centric applications.
- Urban Sensing (http://research.cens.ucla.edu/projects/2006/Systems/Urban_Sensing) seeks to build short-term, community-oriented urban sensor networks.

These projects assume central collection points across the Internet that perform data and task management and act as mediators between users and the network. Our proposed solutions present a complementary, decentralized view for programming distributed sensing applications; they don't require servers or Internet connectivity.

Researchers have recently proposed several distributed-programming abstractions and middleware for ubicomp environments and sensor networks. Many existing ubicomp solutions target more stable networks in schools or homes, sometimes assume powerful servers, and lack the flexibility to work in highly volatile, data-centric environments. Successful abstractions for sensor networks are more appropriate for leveraging in Urbanet programming models. We can categorize these abstractions according to whether they focus on localized behavior—exposing some network details to programmers—or on global behavior—hiding the network details from programmers. State-centric programming,[1] Hood,[2] and Abstract Regions[3] belong to the first category; TinyDB[4] and Kairos[5] belong to the second.

All our models provide global abstractions. Contory sees the network as a database, Spatial Programming provides a virtual naming across the network, and Migratory Services can execute on any node satisfying the specified context rules. However, Spatial Programming and Migratory Services applications often exhibit localized behavior.

REFERENCES

1. J. Liu et al., "State-Centric Programming for Sensor-Actuator Network Systems," *IEEE Pervasive Computing*, vol. 2, no. 4, 2003, pp. 50–62.
2. K. Whitehouse et al., "Hood: A Neighborhood Abstraction for Sensor Networks," *Proc. 2nd Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 04), ACM Press, 2004, pp. 99–110.
3. M. Welsh and G. Mainland, "Programming Sensor Networks Using Abstract Regions," *Proc. 1st Usenix/ACM Symp. Networked Systems Design and Implementation* (NSDI 04), Usenix Assoc., 2004, pp. 29–42.
4. S.R. Madden et al., "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Systems*, vol. 30, no. 1, 2005, pp. 122–173.
5. R. Gummadi, O. Gnawali, and R. Govindan, "Macroprogramming Wireless Sensor Networks Using Kairos," *Proc. 1st IEEE Int'l Conf. Distributed Computing in Sensor Systems* (DCOSS 05), LNCS 3560, Springer, 2005, pp. 126–140.

fully dedicated to sensing applications. For example, phones are used primarily to make and receive phone calls and only secondarily to support Urbanet applications. So, an Urbanet middleware must dynamically optimize resource utilization to the sensing activity, network conditions, and local resources.

Finally, Urbanets are distinguished from regular sensor networks in that they must support concurrent user applications. Managing simultaneous user applications can generate large data traffic in often resource-impoverished environments. An Urbanet middleware must balance resource utilization across multiple applications and limit the geographic scope of the control updates. The middleware should also aggregate data for multiple applications when possible.

To introduce our middleware solutions to these Urbanet-specific challenges, we'll consider the design of a mobile application that helps drivers detect traffic jams in a city. One possibility is to instruct Urbanets to monitor vehicle speeds in the region of interest; the middleware would periodically transfer these observations to the application, which subsequently computes the traffic jam probability. A second possibility is to specify the region of interest and dispatch a distributed task to execute on that region's nodes; the task then informs the user when it detects a traffic jam. A third possibility is to register an interest with a traffic jam service that's executing in the region of interest; the service collects local traffic observations, computes the traffic jam probability, and alerts the client application when necessary.

According to these potential solutions, we present three middleware platforms that support three different programming models:

- *Contory*[1] (*Cont*extfact*ory*) supports a declarative programming model that views Urbanets as a distributed-sensor

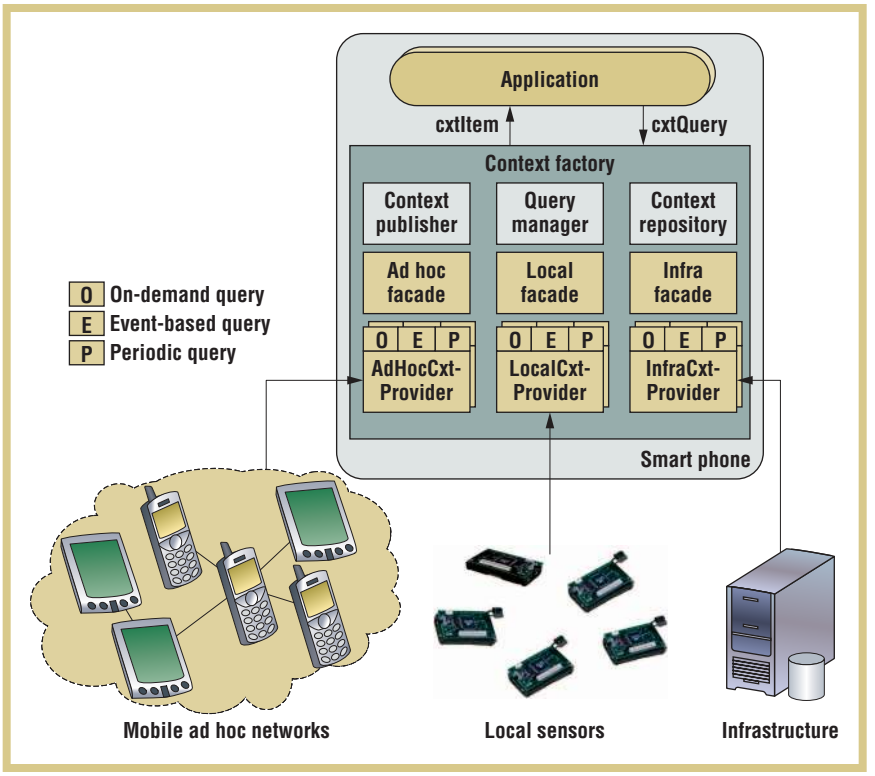

Figure 2. Contory architecture.

database and exposes a simple SQL-like interface to programmers.
- *Spatial Programming*[2] supports an imperative programming model that views an Urbanet as a single virtual name space that applications use to access individual resources at nodes.
- *Migratory Services*[3] supports a client-service model where services migrate to different network nodes to maintain a semantically correct and continuous interaction with clients.

We've implemented all these middleware platforms on top of our Smart Messages distributed-computing platform,[4] which provides naming, routing, and execution migration. In the following, we use the traffic jam application as a reference example to describe each middleware platform.

Contory

The Contory platform supports mobile applications that must be aware of both their local context and the context of remote entities or physical environments. A certain context is expressed as a set of context items, consisting of type, sensed data, sensor type, and other qualifying properties. Example context items include spatial information, environmental conditions, and network resource availability.

While other projects have demonstrated how well declarative programming suits sensor networks,[5] the question remained open as to whether this model was adaptable to highly mobile, heterogeneous networks. To cope with Urbanet dynamism and, in particular, sensor failures and resource constraints, Contory integrates three alternative mechanisms for context provisioning:

- local sensors integrated in or connected to mobile devices,
- context infrastructures possibly available in the environment, and
- MANETs of sensing devices.

Accordingly, Contory's software architecture (see figure 2) integrates three types of ContextProvider modules: LocalCxtProvider, AdHocCxtProvider, and InfraCxtProvider. ContextProviders aren't bound to a single network technology and can therefore provide more adaptability to changing network conditions (similar to Haggle's approach[6]). In particular, AdHocCxtProviders support one-hop context provisioning over Bluetooth and multihop context provisioning over Wi-Fi. Additional architectural components are Facade modules to coordinate the access to context providers, ContextRepository to support context data storage, ContextPublisher to make local context data available to external entities, and QueryManager to manage query assignment and optimization.

The query language is specialized to address the concurrent requirements of different applications. Each application can specify which data to collect and how to combine and summarize it. Contory uses standardized attributes to qualify sensor data of interest and filter a subset of it. Example attributes are data freshness, accuracy, and correctness. Furthermore, the query must specify space and time requirements. For instance, a region-bound query collects sensor data in a specified geographic region without a priori knowledge of the mobile sensors that dynamically join and leave the region. Finally, Contory supports long-running queries, such as event-based and periodic queries.

The following example shows a query sent to collect accurate `speed` values of all nodes found in `remote_region`:

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

Contory returns results when the average speed drops below `min_speed`. To compute the probability of a traffic jam, the application combines these data with the number of collected sam-

Figure 3. WeatherWatcher using Contory: (a) the user inputs a location of interest. Then WeatherWatcher collects meteorologic observations around this location and (b) displays the infrared weather conditions.

ples (that is, density of cars), past observations, and possibly knowledge of road topology.

The FROM clause offers three ways to specify the context source types, one for each provisioning mechanism: sensors, infrastructures, and ad hoc networks. Programmers can also leave this clause unspecified, and Contory will decide which context sources to employ. The provisioning mechanism is selected on the basis of present operating conditions, estimated resource consumption for query processing, and quality of the expected results. The initial query assignment can change at runtime multiple times. For example, if Contory detects a malfunction in a sensor connected to the phone, it discovers a new sensor source in the network.

Currently, Contory performs multiquery optimization only among queries submitted by the same device, but we plan to extend it to merge queries from different devices that have selection predicates with overlapping attribute ranges. We also plan to investigate mechanisms for in-network data aggregation that work in the presence of mobility. For instance, mobility can lead to situations where Contory aggregates a certain context item multiple times at different nodes, thus negatively impacting the result quality.

We've implemented Contory using Java 2 Platform Micro Edition (J2ME). Currently, two implementations exist: one for Connected Limited Device Configuration 1.0 and Mobile Information Device Profile 2.0 APIs, and one for Connected Device Configuration (CDC) 1.0. We performed all software development using Nokia Series 60 and Nokia Series 80 phones.

We've used Contory to implement a WeatherWatcher application (see figure 3), which retrieves weather-related sensed data from user-specified regions. For example, the WeatherWatcher running on a car driver's personal phone can query sensors integrated in neighboring cars or available along the highway, analyze the data, and possibly alert the driver about upcoming fog patches.

Spatial Programming

The declarative programming model proposed by Contory has the advantage of simplicity for application programmers. However, as several programming abstractions proposed for sensor networks have shown,[7–9] it's not a panacea for every task. Imperative programming can be more appropriate for complex tasks that go beyond data collection, especially tasks with algorithmic details that can't be left to a common middleware. Additionally, powerful nodes such as smart phones, vehicular systems, or intelligent video cameras can be programmed more effectively when applications have fine-grained control over individual node resources.

Spatial Programming is a runtime system that offers a location-aware programming model. SP enables Urbanet nodes to perform collaborative tasks by abstracting an Urbanet as a single virtual name space (similar to Kairos' global name space[10]). An application written under the SP 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, SP shields application programmers from the distributed, volatile nature of Urbanets. This high-level network view is similar to the way shared-virtual-memory systems shield programmers from message-passing communication while offering a shared virtual address space for distributed applications. A major difference, however, is that shared virtual memory works over a stable, robust network with an acceptable upper bound for memory access time, whereas SP must tolerate dynamic network configurations with unknown time bounds for accessing resources at nodes. Figure 4 illustrates this analogy and the simple abstractions SP defines to support programming in Urbanets—namely, space regions and spatial references.

SP defines a spatial reference as a {space:tag[index], timeout} tuple, which it maps to an Urbanet node. *Space* is a name associated with a region that represents the node's geographical scope. *Tag* is the name of a property the node provides. *Index* differentiates among nodes with the same space-tag pair referenced in the same application. SP requires application programmers to reason about the possibility of not reaching a node by imposing a time-out on each spatial reference. If the application can't reach a node in the specified time interval, the SP runtime system throws a time-out exception; the application catches the exception and decides further actions. SP guarantees reference consistency: each time an application uses a certain spatial reference, it accesses

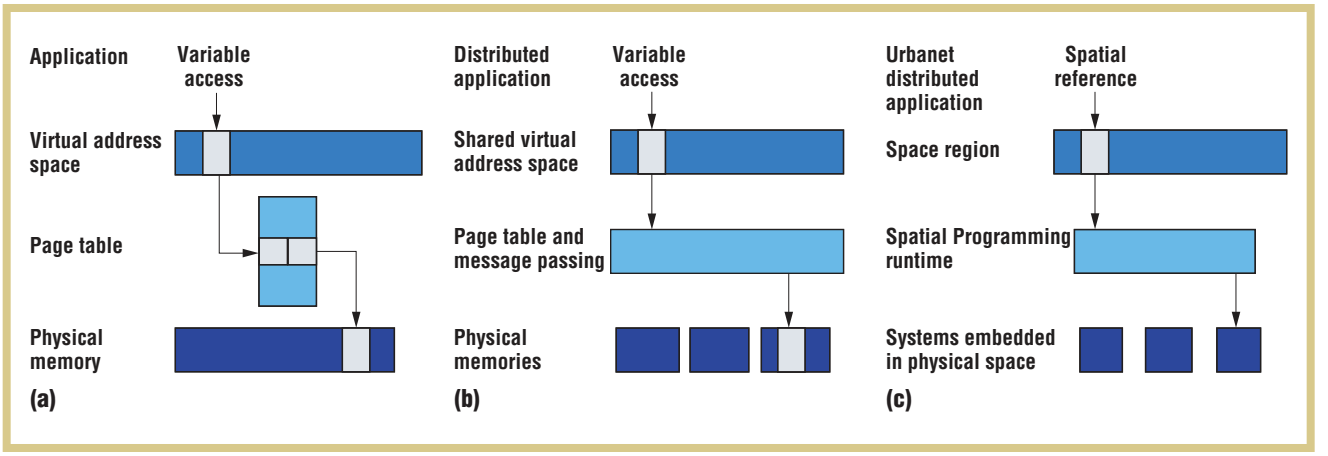

Figure 4. An analogy between two traditional programming models—(a) conventional computer system and (b) shared virtual memory—and (c) Spatial Programming.

the same node as long as the node remains in its original space.

The following code excerpt shows a simplified implementation of the traffic jam application (for instance, we didn't include the timeout in spatial references):

```
int i=0, tjam_p=0;
  while(NOT_DONE){
  try{
     Neighbors []n = {remote_region:car[i]}.neighbors;
     int speed = 0;
     for(int j=0; j<n.length; j++)
        speed += n[j].speed;
     speed = speed/n.length;
     tjam_p = computeTJamProbability(speed, n.length);
     if (tjam_p>MAX_PROB)
        {driver_region:driver_name[0]}.tjam=true;
  }catch(SpaceViolationException e){
     i++;
  }
}
```

The application uses the spatial reference {remote_region:car[i]} to access the list of one-hop neighbors on that node. We assume that each car provides this list together with its neighbors' speeds. When the probability of a traffic jam exceeds a certain threshold, the application informs the driver identified by {driver_region:driver name[0]}. The application continues to use the same spatial reference for a car in the desired region as long as this reference is semantically acceptable—that is, the car remains in the region of interest. If the car moves out of the region, SP triggers an exception and looks for a new car in that region.

The example demonstrates how programmers can use spatial references to access individual Urbanet resources in the same way they use variables to access memory locations in a conventional programming model. Similar to a conventional computer's mappings from virtual to physical memory, SP maintains mappings between spatial references and nodes in the physical space. These mappings are maintained in a per-application mapping table and are persistent during the SP program execution. SP creates a mapping when an application first accesses a resource. The SP runtime system takes care of name resolution, routing, and access to resources.

We implemented this runtime system on top of the Smart Messages platform. Under this implementation, SP applications are Java programs, and each access to a network resource is translated into a Smart Messages migration. We used this implementation to prototype a simple intrusion-detection application using a Wi-Fi-based ad hoc network of Hewlett-Packard iPAQs, their associated light sensors, and an attached video camera.[2]

Migratory Services

The client-service model is another well-understood, simple programming model. To support it in Urbanets, we need to consider two issues. First, rapidly changing operating conditions can often lead to situations where a node providing a certain service suddenly interrupts the interaction. For example, the target of an object-tracking service can move out of its video camera's sensing range. Second, many scenarios must deal with context changes at the client side: users operate in highly dynamic environments, so request parameters are subject to frequent context-induced changes. For instance, if a driver wishes to be continuously informed about traffic conditions in a region 10 miles ahead of her location, the service must continuously adapt to the user's movement. To summarize, a node might become unsuitable to host a service when context changes occur at either the service or client side.

A simple solution is to pass the problem to the client and require it to discover a similar service on another node. However, this approach would lose any state associated with the old interaction, thus seriously affecting long-term stateful client-ser-

vice interactions. Additionally, another node providing that service might not exist in the Urbanet.

To address these issues, we propose Migratory Services, a new client-service model for Urbanets. Under this model, services can migrate to different network nodes to effectively accomplish their task. They execute on a certain node as long as they can provide semantically correct results. When this becomes impossible, they migrate until they find a new node where they can resume execution. As figure 5 shows, the Migratory Services model provides transparent service migration and maintains a continuous client-service interaction. Although a migratory service is physically located on different nodes over time, it constantly presents a single virtual end-point to the client. This concept is similar to virtual mobile nodes.[11]

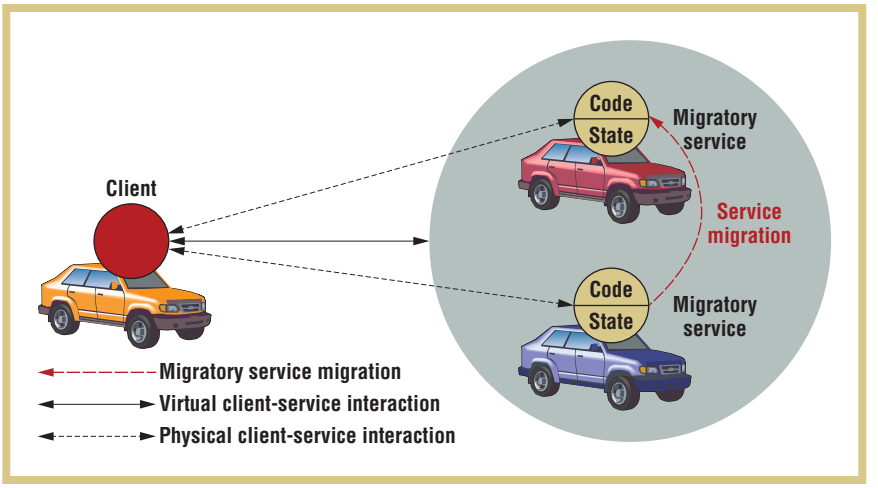

Figure 5.The Migratory Services model.

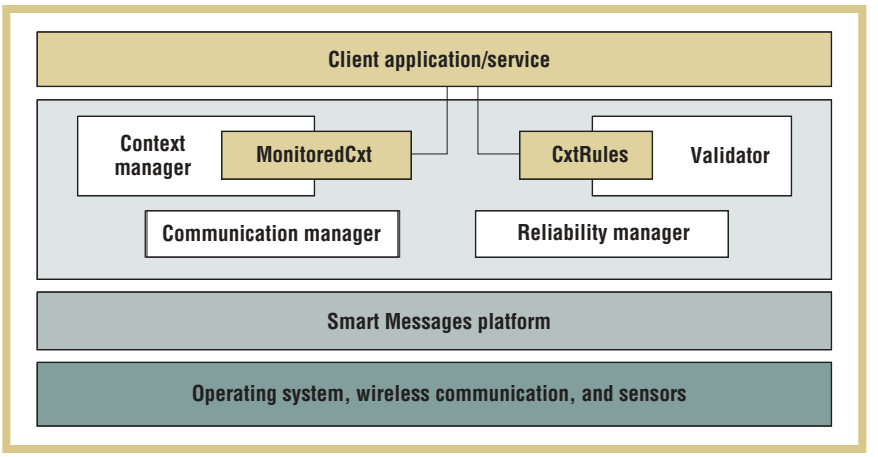

Figure 6. The Migratory Services Framework. The Smart Messages platform provides execution migration, naming, routing, and admission control. On top of the Smart Messages layer, the framework provides context provisioning, context rules validation, client-service communication, and service reliability.

The Migratory Services model involves three main mechanisms supported by the development and execution Migratory Services Framework (MSF), shown in figure 6. The first mechanism monitors interacting entities' dynamism by assessing context parameters that characterize their environment and available resources. The second uses rules to specify how the service execution should adapt to context parameter variations. The third makes the service capable of migrating from node to node and of resuming its execution once migrated. We call this process *context-aware service migration* because it's triggered by context changes.

To exemplify these concepts, we consider again the application for detecting traffic jams. The client uses MSF to issue a request for a traffic jam (Tjam) service in a remote region:

```
Request req = new Request(client_id, remote_region);
MSF.sendRequest(Tjam, req);
```

A Tjam service available in the network receives the request and starts the computation. This service must register with the hosting node and specify its context rules. A context rule consists of (condition, action) pairs. For example, the following rule states that when the node moves out of the region specified by the client, MSF must trigger a service migration to resume execution on a node in that region:

```
CtxRule rule = new CxtRule(service_id);
rule.setCondition(OutOfRegion, remote_region);
rule.setAction(MIGRATE);
MSF.registerCtxRule(rule);
```

During service execution, MSF constantly verifies the registered context rules and acts on them. The service computes the probability of a traffic jam, and when the probability exceeds a certain threshold, it sends an alert to the client:

```
int speed=0, tjam_p=0; Neighbors []n;
for(j=0; j<n.length; j++)
  speed + = n[j].getSpeed();
```

```
speed = speed/n.length;
tjam_p= computeTJamProbability(speed, n.length);
if(tjam_p>MAX_PROB)
  MSF.sendAlert(client_id, tjam_p);
```

Migration isn't part of the service code—the framework triggers it when necessary.

We implemented this framework in Java and tested the traffic jam application over a WiFi-based network of Hewlett-Packard iPAQs. Recently, we extended the implementation to J2ME CDC and tested it on Nokia 9500 phones.

Discussion

Routing, localization, cooperation incentives, security, privacy, and trust are all significant Urbanet challenges. We believe that many proposed solutions for sensor networks and MANETs can be adapted for Urbanets. Trust issues are more critical than in traditional sensor networks because of the Urbanets' spontaneous, people-centric nature. We're investigating how to create trusted ad hoc networks using a trusted execution monitor built as a kernel module on top of a Trusted Platform Module. In terms of localization, GPS or systems based on signals from existing Wi-Fi access points or cellular base stations can provide cheap, reasonably accurate solutions.

Our experiences with these middleware platforms in small-scale ad hoc networks of HP iPAQs and smart phones have been promising. Besides typical interference problems in places where wireless devices are dense, energy remains the most constraining technical factor. Because most applications must be aware of their location, we quantified the lifetime of a phone running a modified fingerprinting version of Intel's Place lab. The phone computed its location and sent it as a SOAP message to another node over Wi-Fi at intervals from 10 seconds to 1 minute. The phone lifetime varied from 4 to 6 hours.

The Smart Messages platform provides a common execution environment for our Urbanet middleware solutions, and its design for resource-constrained devices offers flexible, property-based naming and routing support.

We designed our solutions to work despite Urbanet heterogeneity. The underlying Smart Messages platform provides a common execution environment across heterogeneous devices, and its design for resource-constrained devices such as smart phones offers flexible property-based naming and routing support. However, tiny sensors such as motes won't run our middleware. Running the middleware on more powerful devices allows access to tiny sensors either directly through queries (for example, iMotes over Bluetooth) or indirectly through a base station.

Each middleware copes with network volatility in different ways. When Contory detects that a certain context provider becomes unavailable, it dynamically selects an alternative available provider. In SP, programmers associate time-outs with each resource access, and the runtime raises exceptions when the access isn't successful during the specified time-out. This mechanism lets applications dynamically adjust their requirements to the network conditions (for example, they can accept lower-quality results that are still semantically correct). Migratory Services respond to volatility by migrating the execution to suitable nodes every time the execution context changes beyond specified limits. In addition, they also maintain a backup service that takes over in case of service failure or network partitions. To ensure that the service can resume, the backup service can also reside at the client node.

The choice of which middleware to employ depends on the application's semantics. With Contory, intelligence is mostly in the middleware; applications receive a very simple programming interface to collect sensor data. Contory provides high transparency by adapting to changing operating conditions while also letting applications assess the results quality through qualifying attributes. However, it offers limited flexibility to program complex distributed applications. For these applications, Spatial Programming provides the necessary fine-grained access to network resources while maintaining a relatively simple programming interface. In Spatial Programming, the middleware is thin, and most of the complex logic resides in the application. The Migratory Services middleware presents an intermediate solution: client applications are very simple, services are more complex, and the middleware provides significant help by automatically adapting to the operating context. It's particularly useful for long-running and stateful end-to-end interactions in highly volatile conditions.

Of course, the ultimate validation of our platforms will come from experiences in larger scale networks with real mobile users. P

ACKNOWLEDGMENTS

We thank Liviu Iftode for his contribution to the design of Spatial Programming and Migratory Services. We also thank the anonymous reviewers for their useful comments. This work has been supported in part by US National Science Foundation grants CNS-0520033, CNS-0454081, and IIS-0534520, and in part by the Finland's National Technology Agency (Tekes) Dynamos project and the Nokia Foundation.

REFERENCES

1. O. Riva, "Contory, A Middleware for the Provisioning of Context Information on Smart Phones," *Proc. 7th ACM Int'l Middleware Conf.* (Middleware 06), LNCS 4290, Springer, 2006, pp. 219–239.
2. C. Borcea et al., "Spatial Programming Using Smart Messages: Design and Implementation," *Proc. 24th Int'l Conf. Distributed Computing Systems* (Icdcs 04), IEEE CS Press, 2004, pp. 690–699.
3. O. Riva et al., "Context-Aware Migratory Services in Ad Hoc Networks," to be published in *IEEE Trans. on Mobile Computing*, 2007.
4. P. Kang et al., "Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems," *Computer J.*, vol. 47, no. 4, 2004, pp. 475–494.
5. S.R. Madden et al., "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Systems*, vol. 30, no. 1, 2005, pp. 122–173.
6. J. Scott et al., "Haggle: A Networking Architecture Designed around Mobile Users," *Proc. 3rd Ann. Conf. Wireless On-Demand Network Systems and Services* (WONS 06), Inria, 2006, pp. 78–86.
7. J. Liu et al., "State-Centric Programming for Sensor-Actuator Network Systems," *IEEE Pervasive Computing*, vol. 2, no. 4, 2003, pp. 50–62.
8. M. Welsh and G. Mainland, "Programming Sensor Networks Using Abstract Regions," *Proc. 1st Usenix/ACM Symp. Networked Systems Design and Implementation* (NSDI 04), Usenix Assoc., 2004, pp. 29–42.
9. K. Whitehouse et al., "Hood: A Neighborhood Abstraction for Sensor Networks," *Proc. 2nd Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 04), ACM Press, 2004, pp. 99–110.
10. R. Gummadi, O. Gnawali, and R. Govindan, "Macroprogramming Wireless Sensor Networks Using Kairos," *Proc. 1st IEEE Int'l Conf. Distributed Computing in Sensor Systems* (Dcoss 05), LNCS 3560, Springer, 2005, pp. 126–140.
11. S. Dolev et al., "Virtual Mobile Nodes for Mobile Ad Hoc Networks," *Proc. 23rd Ann. ACM Symp. Principles of Distributed Computing* (PODC 04), ACM Press, 2004, p. 385.

the AUTHORS

Oriana Riva is a researcher at the Helsinki Institute for Information Technology and a doctoral student in computer science at the University of Helsinki. Her research interests include middleware for pervasive systems, ad hoc networking, and context-aware computing. She received her MSc in telecommunication engineering from Politecnico di Milano. Contact her at the Dept. of Computer Science, PO Box 68, FIN-00014, Univ. of Helsinki, Finland; oriana.riva@cs.helsinki.fi.

Cristian Borcea is an assistant professor in the New Jersey Institute of Technology's Department of Computer Science. His research interests include mobile computing, middleware for ubiquitous networked systems, vehicular networks, and sensor networks. He received his PhD in computer science from Rutgers University. He's a member of the IEEE, ACM, and Usenix. Contact him at the Dept. of Computer Science, New Jersey Inst. of Technology, University Heights, Newark, NJ 07102; borcea@cs.njit.edu.

IEEE pervasive COMPUTING
MOBILE AND UBIQUITOUS SYSTEMS

How to Reach Us

Writers
For detailed information on submitting articles, write for our Editorial Guidelines (pervasive@computer.org) or access www.computer.org/pervasive/author.htm.

Letters to the Editor
Send letters to

Shani Murray, Lead Editor
IEEE Pervasive Computing
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
pervasive@computer.org

Please provide an email address or daytime phone number with your letter.

On the Web
Access www.computer.org/pervasive or http://dsonline.computer.org for information about *IEEE Pervasive Computing*.

Subscription Change of Address
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Pervasive Computing*.

Membership Change of Address
Send change-of-address requests for the membership directory to directory.updates@computer.org.

Missing or Damaged Copies
If you are missing an issue or you received a damaged copy, contact membership@computer.org.

Reprints of Articles
For price information or to order reprints, send email to pervasive@computer.org or fax +1 714 821 4010.

Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at copyrights@ieee.org.