Report Number: WUCSE-2004-18

2004-04-22

Bringing Context-Awareness to Applications in Ad Hoc Mobile Networks

Christine Julien, Gruia-Catalin Roman, and Jamie Payton

Context-aware mobile applications require constant adapta-tion to their changing environments. Technological advancements have increased the pervasiveness of mobile computing devices such as laptops, handhelds, cellular phones, and embedded sensors. The sheer amount of context information necessary for adaptation places a heightened burden on application developers as they must manage and utilize vast amounts of data from diverse sources. Facilitating programming in this data-rich environment requires a middleware infrastructure for sensing, collect-ing, and providing context information to applications. In this paper, we demonstrate the feasibility of providing such a middleware that allows programmers to focus on high-level interactions among programs... **Read complete abstract on page 2.**

Bringing Context-Awareness to Applications in Ad Hoc Mobile Networks

Christine Julien, Gruia-Catalin Roman, and Jamie Payton

Complete Abstract:

Context-aware mobile applications require constant adapta-tion to their changing environments. Technological advancements have increased the pervasiveness of mobile computing devices such as laptops, handhelds, cellular phones, and embedded sensors. The sheer amount of context information necessary for adaptation places a heightened burden on application developers as they must manage and utilize vast amounts of data from diverse sources. Facilitating programming in this data-rich environment requires a middleware infrastructure for sensing, collect-ing, and providing context information to applications. In this paper, we demonstrate the feasibility of providing such a middleware that allows programmers to focus on high-level interactions among programs and to employ declarative abstract specifications of context in settings that exhibit high levels of mobility and transient interactions with opportunis-tically encountered components. We also discuss the novel context-aware abstractions the middleware provides and the programming knowledge necessary to write applications using our middleware. Finally, we provide examples demonstrating the flexibility of the infrastructure and its abil-ity to support differing tasks from a wide variety of application domains.

2004-18

Bringing Context-Awareness to Applications in Ad Hoc Mobile Networks

Authors: Julien, Christine; Roman, Gruia-Catalin; Payton, Jamie

April 22, 2004

Abstract: Context-aware mobile applications require constant adaptation to their changing environments. Technological advancements have increased the pervasiveness of mobile computing devices such as laptops, handhelds, cellular phones, and embedded sensors. The sheer amount of context information necessary for adaptation places a heightened burden on application developers as they must manage and utilize vast amounts of data from diverse sources. Facilitating programming in this data-rich environment requires a middleware infrastructure for sensing, collecting, and providing context information to applications. In this paper, we demonstrate the feasibility of providing such a middleware that allows programmers to focus on high-level interactions among programs and to employ declarative abstract specifications of context in settings that exhibit high levels of mobility and transient interactions with opportunistically encountered components. We also discuss the novel context-aware abstractions the middleware provides and the programming knowledge necessary to write applications using our middleware. Finally, we provide examples demonstrating the flexibility of the infrastructure and its ability to support differing tasks from a wide variety of application domains.

Bringing Context-Awareness to Applications in Ad Hoc Mobile Networks

Christine Julien, Gruia-Catalin Roman, and Jamie Payton

Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{julien, roman, payton}@wustl.edu

Abstract. Context-aware mobile applications require constant adaptation to their changing environments. Technological advancements have increased the pervasiveness of mobile computing devices such as laptops, handhelds, cellular phones, and embedded sensors. The sheer amount of context information necessary for adaptation places a heightened burden on application developers as they must manage and utilize vast amounts of data from diverse sources. Facilitating programming in this data-rich environment requires a middleware infrastructure for sensing, collecting, and providing context information to applications. In this paper, we demonstrate the feasibility of providing such a middleware that allows programmers to focus on high-level interactions among programs and to employ declarative abstract specifications of context in settings that exhibit high levels of mobility and transient interactions with opportunistically encountered components. We also discuss the novel context-aware abstractions the middleware provides and the programming knowledge necessary to write applications using our middleware. Finally, we provide examples demonstrating the flexibility of the infrastructure and its ability to support differing tasks from a wide variety of application domains.

1 Introduction

With the increasing popularity of mobile computing devices, software users find themselves living and interacting in environments characterized by the ability to communicate and coordinate with a wide variety of wirelessly networked resources. In the most extreme of cases, this network is completely disconnected from a wired infrastructure. As one example, imagine a network that forms among wirelessly enabled vehicles on a highway, in which the cars communicate directly with one another. This type of network, in which mobile components communicate directly with each other using wireless radio signals, is commonly referred to as an *ad hoc network*.

Ad hoc networks form opportunistically and change rapidly in response to the movement of the networked devices, or *mobile hosts*. These networks present an environment in which the network topology is both dynamic and unpredictable.

The lack of a static infrastructure requires the mobile hosts themselves to serve as routers for messages in the network. In addition, because communicating parties may be constantly moving, their interactions are inherently transient in nature. Two communicating parties may be only briefly connected and may never encounter each other again. Much work on supporting applications in this environment builds on the foundation of ad hoc routing protocols that create and maintain communication pathways between senders and receivers specified by IP address. As the topology of the ad hoc mobile network changes due to host mobility, the protocols adjust the paths to maintain end-to-end connectivity.

We attempt to bridge the gap between these current provisions for communication in ad hoc networks and the needs of applications in this highly dynamic environment. Specifically, we see two fundamental limitations application developers for ad hoc networks experience. First, the currently available communication primitives do not fully address the challenges of ad hoc networks previously outlined. While ad hoc routing protocols handle topology changes due to host mobility, applications in ad hoc networks are not likely to know in advance the IP addresses of the parties with which they want to communicate. These applications instead require communication constructs that support the transient connections they encounter in ad hoc networks and facilitate opportunistic interactions. Second, the existing communication primitives do not provide the best level of abstraction for enabling rapid development and dissemination of applications in ad hoc networks. The appropriate abstractions of the ad hoc network, on the other hand, will ease the development task of mobile applications.

In this paper, we apply lessons learned within the paradigm of context-aware computing to the unique challenges presented by ad hoc networks. With this approach, applications need not have explicit knowledge of the other mobile hosts in the network and the application's level of awareness rises to an environment with which it interacts. This abstraction of networked components as a *context* encompasses information that can be collected from hosts throughout the ad hoc network and facilitates the provision of natural programming constructs.

Given this expanded context, potential applications in many domains abound. The difficulty of programming these applications can be generalized to the need to manage large amounts of distributed and transiently available context data. This challenge motivated us to develop an infrastructure that facilitates program development by hiding the details associated with mobility, distribution, and transient connectivity. This middleware, EgoSpaces, provides mechanisms for applications to limit the portion of the context that they interact with, and each application can tailor its context to its individualized needs. An application may define different contexts that reflect diverse concurrent and changing needs and which encompass varying data from multiple sources. Each of these contexts may access a wide range of data, and EgoSpaces manages this information for the application. This relieves the programmer from having to open sockets between the communicating parties and manage the network disconnections common in mobile environments. The middleware's communication and management primitives facilitate rapid development of context-aware applications.

In the next section, we outline related work in context-aware computing and discuss how our application of context to the ad hoc network environment differs from previous approaches. Section 3 briefly overviews the EgoSpaces coordination model, highlighting how it manages sophisticated application contexts. Section 4 details the mechanics of using the infrastructure to program applications. Section 5 describes sample applications, shows how they rely on the middleware, and discusses lessons learned in their development. In Section 6, we present the middleware's design and implementation which focuses on modularity to allow reuse of components as necessary. Section 7 provides conclusions.

2 Context-Aware Computing

Context-aware computing first came to the forefront in the early 1990's with the introduction of small mobile devices. Active Badge [1] used infrared communication between badges worn by users and sensors placed in a building to monitor movement of the users and forward telephone calls to them. PARCTab [2] also used infrared communication between users' palm top devices and desktop computers to allow applications to adapt to the user's environment. These applications perform activities ranging from simply presenting information to the user about his current location to attaching a file directory to a room for use as a blackboard by users in the room. More recent work [3] in building such ubiquitous computing environments uses CORBA and operates over a wired network infrastructure that supports both localization and communication. These systems require extensive infrastructures which need constant maintenance. They also rely on wired communication and do not address the issues inherent to ad hoc networks, including the need to scale to large networks.

More recent context-aware applications, e.g., Cyberguide [4] and GUIDE [5], serve as tour guides by presenting information about the user's current environment. Fieldwork tools [6] automatically attach contextual information, e.g., time, to notes taken by researchers in the field. These applications each collect their own context information and focus on providing a specific type of context. While these applications have proven useful in their target environments, the applications commonly demanded by ad hoc networks share some characteristics that set them apart from previous context-aware applications. Most specifically, the applications benefit from opportunistic interactions. For example, an application for vehicles on a highway interacts with other cars locally (e.g., in the same area of a city) to collect traffic information. A particular driver has no advance knowledge about the cars providing the traffic information; instead the driver knows to collect traffic information from other nearby cars.

Generalized software built to support the development of context-aware computing in mobile environments has begun to be developed. Among the best known systems are the Context Toolkit [8] and the Context Fabric [9]. The Context Toolkit provides abstractions for representing context information through the use of context widgets. These widgets collect low-level sensor information and aggregate it into higher-level information more easily handled by application developers. These widgets form a library that developers can use when constructing context-aware applications. The Context Fabric attacks a similar problem but uses an infrastructure approach. While the Context Toolkit and Context Fabric offer developers much needed building blocks for constructing context-aware applications, even those for collecting information from distributed sets of sensors, these systems do not explicitly address the needs of applications in ad hoc networks to dynamically discover and operate over a constantly changing context.

Ubiquitous computing environments build on traditional approaches to context-aware computing. The GAIA project [10] introduces the notion of *Active Spaces* as immersive computing environments for context-aware applications. Users move from one Active Space to another, seamlessly extracting from one space and integrating into a new one. This work addresses the needs of context-aware applications in small networked environments where the available resources in the space can be centrally managed by a kernel. This type of approach does not map well to large-scale context-aware applications in completely wireless environments.

The CORTEX project [11] proposes an infrastructure for context-awareness in nomadic mobile environments that combine mobile entities with a wired infrastructure. This project focuses on quality of service guarantees that can be provided within a region of the network and uses gateways to connect these various regions. Similarly, Solar [12] provides an infrastructure to support context acquisition and operation for nomadic wireless networks. The goals of these system are in line with our goals—to support large-scale mobile computing—but the target environment differs in that the concerns apparent in ad hoc networks require specialized solutions that are not applicable in nomadic networks.

From this review, it becomes apparent that context-aware computing provides abstractions similar to those that would be useful for supporting applications in mobile ad hoc networks. The distinguishing characteristics of ad hoc networks and the applications likely to be desired in these dynamic environments necessitate a redefinition of what it means to be context-aware. The key components of this new definition of context-awareness are:

1. Context should be generalized so that applications interact with different types of context (e.g., location, bandwidth, etc.) in a similar manner.
2. Different applications require contexts tailored to their individual and changing needs.
3. In an ad hoc network, an application's context includes information collected from a distributed network of devices surrounding the application's host.
4. Due to the large-scale nature of the environment, applications require a decentralized solution for interacting with their contexts.
5. Abstractions of this distributed context ease the programming burden.

In the remainder of this paper, we describe a middleware that provides exactly this perception of context driven by the specific needs of applications in ad hoc networks. This middleware builds on the asymmetric model of coordination first introduced in [13]. Throughout the presentation, we relate the middleware's abstractions to application scenarios common in these dynamic environments and show how the middleware facilitates the development of these applications.

3 EgoSpaces Model Overview

In our computing model, hosts can move in physical space, and applications are structured as a community of logically mobile agents that can move among these hosts. Agents can communicate among themselves and move among hosts when the hosts involved can physically communicate. These software agents control pieces of data they share with other agents to foster coordination; this data can include application information or data generated by environmental sensors.

The EgoSpaces model uses asymmetric coordination to gather and utilize context information in an ad hoc mobile environment, i.e., each agent filters the world around it through its own unique and changing perspectives. The amount of context information ultimately available to an application may span a large network; this generates an overwhelming amount of data for the application to manage. EgoSpaces allows an individual application agent to precisely specify the context necessary for completing its tasks, and the infrastructure provides this context for the application. As the environment changes, the set of data satisfying the application's specification also changes, and the infrastructure adapts the context accordingly. Throughout this paper, we use the term *reference agent* to refer to the particular agent whose context we are discussing.

3.1 The View Concept

To provide scalable coordination in an ad hoc network, EgoSpaces relies on an abstraction called a *view*. This concept is agent-centric because a reference agent defines views with respect to its individual needs. In practice, the reference agent's behavior generally relies on information available in a region surrounding its location. In EgoSpaces, an agent sees the world through a set of personalized views that it may alter at will, and each view presents a projection of all data available to the reference agent. The unique properties of ad hoc mobile networks force context restriction based on attributes of the network hosts and links. EgoSpaces combines these network and host constraints with restrictions on the data and agents that own the data. An agent describes its personal needs through a declarative view specification. For example:

> Traffic information (reference to data) collected by traffic monitoring agents (reference to agents) on cars (reference to hosts) within 100 meters in front of my current position (network restriction).

Network Constraints. To restrict the scope of the network, the application specifies an abstract metric over network properties. This metric calculates a logical distance from the reference host to other network hosts. The application also provides a bound over allowable distances which restricts which hosts belong to the neighborhood. In EgoSpaces, we support this abstraction using a communication protocol like the one detailed in [14].

Host and Agent Constraints. Host and agent constraints allow an application to restrict a view's data based on properties of the hosts and agents that hold the data. In EgoSpaces, every host and agent creates a profile describing its properties. Host properties might include a unique host id, the host's owner, or services the host provides. Agent properties might include the agent's host or the agent's application task. To restrict which hosts and agents contribute to its view, a reference agent provides constraints over profile properties. Constraints are evaluated over profiles in a content based manner, detailed later.

Data Constraints. The data constraints allow a reference agent to restrict the individual data items in the view. Applications can associate "meta-data" with each data object that describes the data or its intended use. The data constraints can then operate over this meta-data to restrict view membership.

Access Controls. In EgoSpaces, each agent specifies an individualized function that limits the ability of other agents to access its local data. From the opposite direction, when an agent specifies a view, it attaches to the view a set of credentials that verify it to other agents. The specifying agent also declares the operations it intends to perform on the view. When determining whether a particular data item belongs to a view, EgoSpaces evaluates the contributing agent's access control function over the view's credentials and operations. The access control function is evaluated for each individual data item, which provides a fine level of granularity. In addition, the access control function can account for properties of the environment, making it context-sensitive in its own right.

Given these components, a view specification consists of three patterns (one over data items, one over agent profiles, and one over host profiles), the network constraints (consisting of a metric for network path costs and a bound on the metric), and an operation list and credentials that allow provision of access controls. With this information, our middleware constructs the application's desired view, at a logical level, the set of data items in the network that satisfy all four levels of constraints.

3.2 Tuple Space Based Coordination

In Linda [15], distributed processes coordinate through a shared tuple space. A tuple is an ordered list of typed fields. In Linda, coordinating processes interact directly with a single, centralized tuple space. Adaptations of Linda divide this tuple space to accommodate mobility. MARS [16] associates a tuple space with each host in the network and allows coordination among co-located application agents. Lime [17] associates a tuple space with each agent, and the tuple space moves with the agent. In this model, the tuple space an application operates on is defined as the union of all tuple spaces within communication range. EgoSpaces also associates tuple spaces with individual application agents because it flexibly supports both physical host mobility and logical agent mobility.

Processes place tuples in the tuple space using **out**(t) operations. Data access occurs in a content based manner by matching a tuple against a pattern, or template, constraining the values of the fields in the tuple.

Tuple Definition and Pattern Matching. EgoSpaces extends the tuple definition to provide more flexible coordination. Specifically, a tuple is an unordered set of fields, each consisting of a name, type, and value. A tuple can have only one field of a given name. The use of this name field allows us to relax restrictions on tuple pattern matching. In Linda, patterns must be the same length as the tuple, and the fields of the tuple and pattern are matched in order. We extend pattern matching to operate over unordered tuples. A pattern is similar to a tuple, but each field's value is replaced with a constraint that restricts the field's value. A tuple matches a pattern if, for every constraint in the pattern, there exists a field in the tuple with the same name and type. The value of the field must also satisfy the corresponding constraint function. While the matching mechanism does require that every constraint in the pattern is satisfied, it does not require that every field in the tuple is constrained, i.e., a tuple must contain exactly the fields in the pattern, but the tuple can contain additional fields.

3.3 View Operations

A view is the set of tuples that satisfy the reference agent's restrictions. Agents use operations similar to Linda tuple space operations to interact with the view's tuples. EgoSpaces preserves Linda's atomic blocking operations, **rd**(p) and **in**(p), which provide a pattern a matching tuple must satisfy. The operations do not return until a tuple in the view matches the pattern. When a match exists, both operations return the matching tuple, and an **in** operation also deletes the tuple from the network. The atomicity of these operations guarantees that, if a matching tuple exists in the view, it will be found and returned.

Common extensions to Linda provide atomic probing operations, **rdp**(p) and **inp**(p) that carry the same atomicity guarantees as the original operations but return immediately instead of blocking. If no tuple in the view immediately matches, an empty value is returned. Other Linda extensions utilize aggregate operations that return all matching tuples. EgoSpaces provides these operations in both blocking (**rdg**(p) and **ing**(p)) and probing (**rdgp**(p) and **ingp**(p)) forms.

Finally, in the dynamic ad hoc environment, atomic operations are often costly to provide. While some applications (e.g., those involving money transfer) require strong guarantees, other applications can take advantage of or even benefit from operations with weaker guarantees. In EgoSpaces, scattered probing operations provide this style of context interaction by providing best-effort semantics. EgoSpaces provides both single (**rdsp**(p) and **insp**(p)) and group (**rdgsp**(p) and **ingsp**)p)) scattered probing operations.

Formal semantic definitions for the view operations can be found in [13]. Additional programming constructs such as reactive programming and behavioral extensions [18] are also available; they are not detailed in this paper.

4 Rapid Development Potential

EgoSpaces reduces programming context-aware mobile applications to simple operations tailored to the capabilities of novice programmers. The middleware provides all network communication programming and presents the programmer with a high-level agent coordination interface. In this section, we show how

EgoSpaces's abstractions ease development by simplifying the programming interfaces while retaining the necessary power of coordination.

EgoSpaces uses the software agent as the unit of modularity and mobility. To use EgoSpaces' abstractions, applications extend the `Agent` class, which allows access to the view specification mechanics and communication capabilities.

Agent Extension. Figure 1 shows the interface for the abstract `Agent` class. An application's agent inherits three key fields: the unique `AgentID`, the `AgentProfile`, and the `AccessControlFunction`. The `AgentID` is not modifiable by the extending class, and its initialization guarantees its uniqueness.

```
public abstract class Agent {
   protected final AgentID aID;
   protected AgentProfile profile;
   protected AccessControlFunction acf;
   public Agent();
   public AgentProfile getProfile();
   protected final void register();
   protected final void out(ETuple tuple);
}
```

Fig. 1. The API for the `Agent` class

An agent's profile fosters powerful coordination by allowing other agents to include or exclude the agent from coordination based on the agent's properties. Initially, the `AgentProfile` contains two fields named "Agent ID" and "Host ID" that contain the `AgentID` and the id of the agent's host. EgoSpaces represents profiles as tuples, so a field in a profile consists of a name, type, and value. The field types can be determined at runtime, therefore an agent need only specify the field's name and value. An agent can use the three methods shown in Figure 2 to modify its profile.

```
public class AgentProfile {
   public void addProperty(String name, Serializable value);
   public void removeProperty(String name);
   public void modifyProperty(String name, Serializable newValue);
}
```

Fig. 2. The API for the `AgentProfile` class

An application agent also inherits the `Agent`'s `AccessControlFunction`. The default function grants all access requests. Agents can personalize this function to exercise access control over their data by extending the `AccessControlFunction` and overriding the `evaluate` method. This function evaluates incoming access requests based on the credentials provided by the reference agent, the view the request comes from, and the particular tuple being accessed.

In extending the `Agent` base class, the application agent receives two methods. The first method registers the `Agent` with the `EgoManager`, a component described in more detail in Section 6. By registering with the `EgoManager`, an application agent delegates responsibility for data management and communication. This also facilitates agent migration among hosts, which we discuss later.

The final Agent method allows agents to create tuples. When the agent is registered with the EgoManager, these data items are available for coordination. Agents generate tuples without respect to their views or their current location. If an agent moves to a new host, all its data moves with it.

View Definition and Use. The view abstraction allows application agents to coordinate over an ad hoc network. Once registered with the EgoManager, an agent can define and use views. Figure 3 shows the public API of the View class.

```
public class View {
  public View(Metric m, Cost bound,
              HostConstraints hc, AgentConstraints ac,
              DataConstraints dc, Credentials cred);
  public ETuple rd(ETemplate template);
  public ETuple rdp(ETemplate template);
  public ETuple rdsp(ETemplate template);
  public ETuple[] rdg(ETemplate template);
  public ETuple[] rdgp(ETemplate template);
  public ETuple[] rdgsp(ETemplate template);
  public ETuple in(ETemplate template);
  public ETuple inp(ETemplate template);
  public ETuple insp(ETemplate template);
  public ETuple[] ing(ETemplate template);
  public ETuple[] ingp(ETemplate template);
  public ETuple[] ingsp(ETemplate template);
}
```

Fig. 3. The API for the View class

We first examine the View constructor. The first two components represent the network constraints and are part of the NetworkAbstractions interface. The Metric defines the costs of paths in the network based on properties of hosts and links. Based on this Metric and the Cost that defines a bound on the lengths of paths, the NetworkAbstractions protocol can build a subnet containing exactly the hosts that satisfy the view's network constraints. EgoSpaces provides commonly used Metric definitions, for example, a metric based on hop count and another based on physical distance. More sophisticated application developers can build their own Metric and Cost definitions by following the procedure outlined in [19]. The HostConstraints and AgentConstraints provide restrictions that hosts and agents must satisfy to contribute data to the view. Because EgoSpaces represents profiles as tuples, both types of constraints can be provided as patterns over tuples. The DataConstraints in a View specification are a pattern over data items that appear in the view.

The View's Credentials identify the reference agent to remote agents. Remote agents' AccessControlFunctions use the Credentials when determining whether to allow the reference agent access to tuples. The Credentials are a subset of the AgentProfile and contain, at a minimum, the reference agent's AgentID. If an application represents agents' Credentials as tuples, AccessControlFunctions can be given via patterns.

Once a `View` is defined, the reference agent sees it as the set of data items that satisfy the associated restrictions. The reference agent uses the operations shown in Figure 3 to access data. Each operation takes a pattern, which allows an application to provide a final restriction that any returned tuple must satisfy.

5 Sample Applications

The best demonstration of the middleware's ability to ease context-aware application development is by example. In this section, we present three applications that show different uses of the view concept in varying application domains.

5.1 Emergency Vehicle Warning System

Our first application warns cars of emergency vehicles along their projected path or appearing from other directions. When a driver needs to clear the road for the emergency vehicle, a light on the dashboard appears.

View Definition. Key to this application is being able to notify the car in time for it to give way for the emergency vehicle. The car's view constraints are:

- *Network constraint.* The network is restricted based on physical distance between hosts.
- *Host constraint.* Only emergency vehicles' hosts contribute to the view.
- *Data constraint.* The view contains only emergency warning tuples.

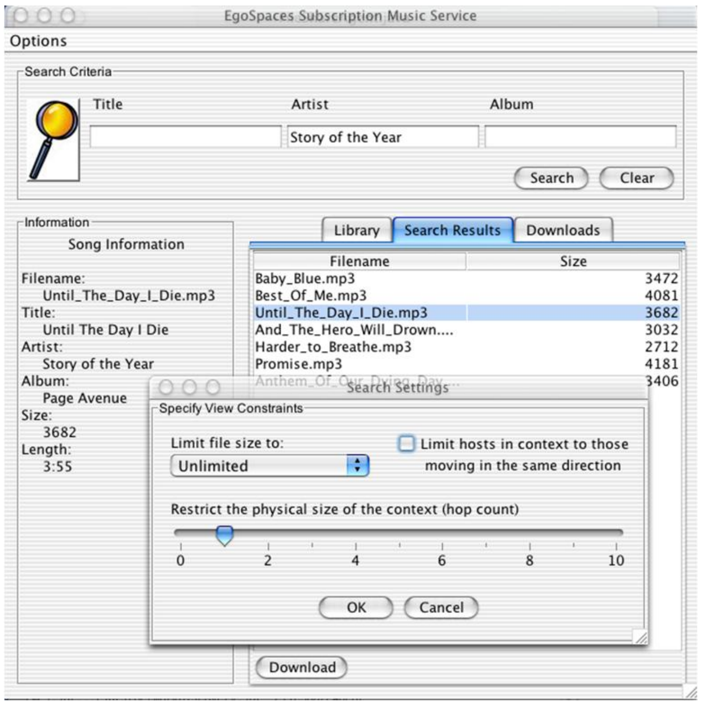
Agent Interaction. Only the emergency vehicle generates tuples. An emergency vehicle creates a tuple when it turns its siren on and removes the tuple when it turns its siren off. The access controls for the emergency vehicle prevent any other agent from removing the warning tuple from the tuple space (i.e., no **in** operations are allowed except by the emergency vehicle's application agent).

Given the view defined above, a car issues a **rd** operation on the view. This operation will match any warning tuple and blocks until a warning tuple appears in the view, indicating an emergency vehicle's presence. At this time, the light on the dashboard warns the driver. The application can probe the view (with periodic **rdp** operations, e.g., at one second intervals) to wait for the disappearance of the warning tuple. After the emergency vehicle has passed, the application can reissue the **rd** operation and the driver can continue. If multiple emergency vehicles appear, this implementation ensures that the driver remains pulled over until all emergency vehicles have passed.

Lessons Learned. The key to successful implementation of this application lies in the definition of the view. Because both the car and the emergency vehicles speeds are variable, the scope of the view depends on their velocities. Given a well-defined view, the application agent's minimal interaction with EgoSpaces involves only simple view operations. The car is guaranteed to be notified as soon as possible of the approach of an emergency vehicle. Notification that the emergency vehicle has departed is not guaranteed to be as timely. This latter behavior could be provided using reactive extensions [18] to the middleware.

5.2 Subscription Music Service

The second application enables music sharing on a network of cars and requires more sophisticated agent coordination. Users subscribe to a music file sharing service which allows them to manage their music and share music with other subscribers they meet on the highway. The application allows a user to manage his music files, search a region of the highway for music, and download these files. If a download only partially succeeds, the application remembers the user's desire for the song, and, when the file is encountered again, the download completes. Figure 4 shows the user interface.



Search Criteria

Title:

Artist: Story of the Year

Album:

Information

Song Information

Filename:
 Until_The_Day_I_Die.mp3
Title:
 Until The Day I Die
Artist:
 Story of the Year
Album:
 Page Avenue
Size:
 3682
Length:
 3:55

Filename	Size
Baby_Blue.mp3	3472
Best_Of_Me.mp3	4081
Until_The_Day_I_Die.mp3	3682
And_The_Hero_Will_Drown....	3032
Harder_to_Breathe.mp3	2712
Promise.mp3	4181
Anthem_Of_Our_Dying_Day...	3406

Specify View Constraints

Limit file size to: Unlimited

☐ Limit hosts in context to those moving in the same direction

Restrict the physical size of the context (hop count)

Fig. 4. The subscription music service

View Definition. The dialog box in Figure 4 allows the user to change his view's constraints. The constraints the user can manipulate are:

– *Network constraint.* The span of the view is defined by network hops.
– *Host constraint.* Restricting the hosts in the view to those traveling in the same direction provides more stability in the contents of the view, making successful downloads more likely.
– *Data constraint.* The user can limit potential downloads based on file size.

As one example, Figure 5 shows the code to build the data constraint based on the file size, where `LTConstraint` requires data items to have values in the size field less than `maxSize`.

```
LTConstraint lt = new LTConstraint(new Integer(maxSize));
EConstraint ec = new EConstraint(‘‘Size’’, Integer.class, lt)
dc.addConstraint(ec);
```

Fig. 5. Building a data constraint

Agent Interaction. The application represents each song in multiple tuples. One tuple holds information about the song, and multiple additional tuples hold the song data. The data is divided into multiple tuples to facilitate the ability of the application to continue interrupted downloads. Figure 6 shows the application code used to generate an information tuple. This code is part of the `FileShareAgent`, which extends the `Agent` base class.

```
ETuple songTuple = new ETuple();
songTuple.addField(new EField("Filename", file));
songTuple.addField(new EField("Title", title));
songTuple.addField(new EField("Artist", artist));
songTuple.addField(new EField("Album", album));
songTuple.addField(new EField("Size", size));
songTuple.addField(new EField("Length", length));
out(songTuple);
```

Fig. 6. Generating information tuples

When the user performs a search, the “Search Results” tab displays the results. The user can choose to download a file, and the progress appears in the “Downloads” tab. The “Library” tab allows the user to manage his music files.

To perform searches, the user enters restrictions in the search panel, which the application constructs into a template. The user can select a file based on its title, artist, or album. Because a music subscription service does not require atomicity guarantees, we use scattered probing operations. Figure 7 shows the code for querying the view.

```
ETemplate template = new ETemplate();
template.addConstraint(titleConstraint);
template.addConstraint(artistConstraint);
template.addConstraint(albumConstraint);
ETuple[] results = searchView.rdgp(template);
```

Fig. 7. Accessing the view

Lessons Learned. The subscription music service takes full advantage of the simplified programming interface in EgoSpaces. Using the view abstraction and coordination constructs, EgoSpaces allows the programmer to focus on how the music subscription application uses the information collected instead of having to explicitly discover and communicate with other agents in the network.

5.3 Collaborative Puzzle Game

The final application demonstrates how the EgoSpaces coordination model can be applied to cooperative work applications. In this example, several users collaborate to complete a puzzle whose pieces are distributed throughout the ad hoc network. Figure 8 shows the screens of two puzzle participants.

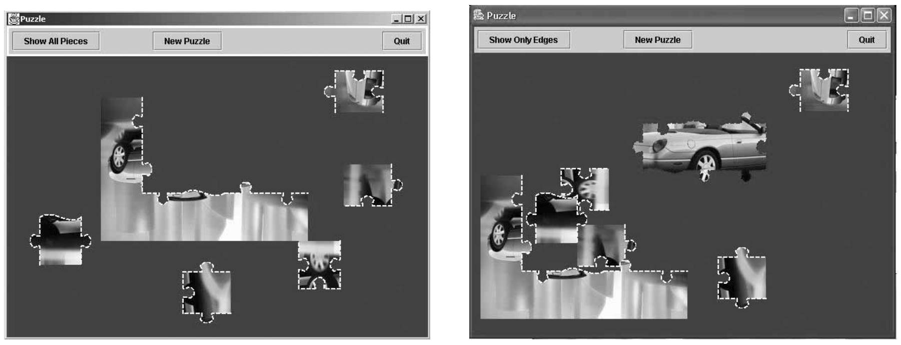


Fig. 8. Two views of a puzzle game

View Definition. This application uses the view constraints to limit the amount of data displayed based on properties of the puzzle to be solved. This view is more logical in nature and can be as simple as to contain only data constraints. The specific constraints used depend on a particular user's goals; as one example, the view might be defined to contain only edge pieces, or only pieces of a certain color. An example of the data constraint required to define the former is shown in Figure 9. It makes use of the `EqualConstraint` function included in EgoSpaces that requires the field's value to equal a designated value.

```
EqualConstraint e = new EqualConstraint(new Boolean(true));
EConstraint ec = new EConstraint(``edgePiece'', Boolean.class, e);
dc.addConstraint(ec);
```

Fig. 9. Seeing only edge pieces

Puzzle players may find many different view definitions useful. If player agents have an idle status, a player might define a view that contains only pieces owned by idle players. If a player is facing a hole of a certain shape, he might specify his view to contain only the partially assembled piece he is working on and any pieces that are the correct shape for the hole. In the puzzle application, choices for defining these views are provided through a series of menus and dialog boxes.

Agent Interaction. One player in the game initializes the puzzle by loading an image. The pieces of the puzzle are represented by tuples in the data space of the agent initializing the puzzle. Each agent (representing a single player in the puzzle game), can define views that determine which puzzle pieces are displayed at a given time. Each agent initially starts with the maximal view, i.e., the view contains all pieces owned by any connected agents. As new agents connect, they too define this view and can see the puzzle pieces available in the system. A user can select a piece by clicking on it. When the user does so, the tuple corresponding to the puzzle piece is removed from its owner and placed in the user's local data space. To all users, this change appears as a change in the color of the border of the displayed puzzle piece. Players can assemble their pieces, and these changes are also reflected in the displays of connected agents.

When a user defines a different view of the puzzle pieces, the display changes appropriately. For example, if the user defines a view to contain only edge pieces, the player will see only these pieces, and all of the interior pieces are hidden. This is the view seen by the agent on the left-hand side of Figure 8. The player on the right has the default view and sees all the pieces. Changes made by the player on the left are displayed to the player on the right, but the reverse is not necessarily true. This is because the player on the right may make changes that affect only interior pieces not included in the other player's view.

Lessons Learned. In the previous two application scenarios, the view definitions were based on obvious notions of distance and relative location. With the puzzle game example, on the other hand, we see that the same abstractions can be used to define more logical views in perhaps smaller scale networks where a user wants to interact with a subset of all of the available data. While the particular subset was determined partially by the data's location in the previous examples, in the puzzle game only properties of the data or agents matter. Other applications that involve cooperative work by distributed parties can be implemented in a similar way. If the collaborative project does span a large-scale network, the application can be extended to account for the relative locations of the data items, in much the same way as in the music sharing example.

6 Infrastructure Design and Implementation

The programming abstractions presented in Section 3 facilitate rapid program development of applications in ad hoc networks. Figure 10 shows the high-level system architecture of EgoSpaces. The gray boxes represent components we assume to exist (message passing and the ad hoc physical network) or components the programmer provides (the application). The white boxes represent components we provide.

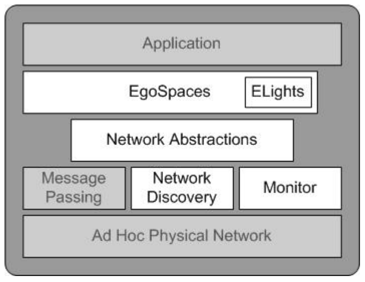


Fig. 10. The system architecture

6.1 Supporting Packages

To build EgoSpaces, we implemented three support packages (a discovery package, a monitor package, and a network abstractions package) that provide lightweight implementations of services necessary for building the view abstraction. The eLights package provides the tuple matching mechanism described in Section 4.

Discovering Network Neighbors. In ad hoc networks, no wired infrastructure with dedicated routing nodes exists. Instead, all hosts serve as routers. To distribute messages, a host must maintain knowledge of its current set of neighbors, and, as movement causes this set to change, the host must be notified. Our system utilizes a discovery service that uses a periodic beaconing mechanism parameterized with policies for neighbor addition and removal.

Monitoring Environmental Conditions. To adapt to context information, applications must sense environmental changes. Our purposes require a lightweight mechanism in which both local and neighboring sensors are accessed in a context-sensitive manner. The sensed information is used to calculate the network restriction discussed next. Our monitor service provides context information by maintaining a registry of monitors available on the local host and neighboring hosts. An application tailors the monitor package to its needed capabilities, e.g., to add a location monitor, the application provides code that interacts with a particular GPS monitor. New monitors must adhere to a standard monitor interface.

Defining Metrics on the Network. EgoSpaces uses the network abstractions protocol [19] to construct a subnet of the ad hoc network. The protocol uses sensor information from monitors and the view's metric and bound to build a tree over the subnet of the ad hoc network that contains exactly the hosts in the network that satisfy the network constraints. The protocol can also maintain the tree as the hosts in the network move and the path costs change. The network abstractions protocol provides EgoSpaces the ability to send messages to exactly the hosts in the context. EgoSpaces can also use the network abstraction interface to register persistent operations on the context hosts. As new hosts move into the context, they receive notification of any registered operations, and as hosts move out of the context, registered operations are removed.

6.2 EgoSpaces Implementation

Figure 11 depicts the middleware's details. The previous sections explained how the application agent interacts with the upper portions in this figure. In this section, we detail how the underlying components support the view abstraction while being attentive to the need for a lightweight and efficient system. Each host supports a single `EgoManager` object that facilitates agents' interactions.

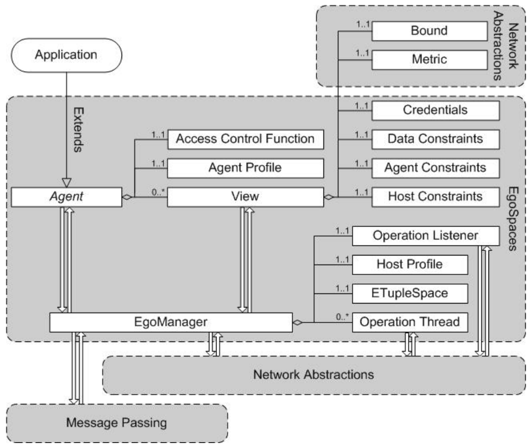


Fig. 11. Internal class diagram of EgoSpaces

Agent Registration and Migration. When an agent is created, a data structure within the agent holds the agent's tuples. EgoSpaces hides this data structure from the extending class. However, if the agent generates tuples via **out** operations before it registers with the `EgoManager`, the tuples are placed in this local storage. These tuples are not available for access by other agents; essentially the agent owning the tuples does not exist in the system. When the agent calls the `register` method, the agent is registered with the `EgoManager`.

Upon registration, the contents of the agent's local tuple storage are placed in a host-level tuple space. During the transfer from the agent's local storage to the host-level tuple space, each tuple is annotated with the owning agent's id. We use a single host-level tuple space instead of maintaining the agent level tuple spaces to reduce the overhead of remote operations. This justification will become more apparent in the discussion of operation processing.

With the registration mechanism described above, facilitating agent migration is reduced to a few simple steps. Upon migrating, an agent is first deregistered from the current `EgoManager`. This moves the agent's tuples from the host-level tuple space to the agent's local storage. This extraction is simplified by the fact that every tuple is labeled with the owning agent's id. After deregistration, the application agent's code and state are moved to the destination host, where the agent is registered with the local `EgoManager`.

View Creation and Maintenance. Any registered agent can define views. For each view, the `EgoManager` uses `NetworkAbstractions` to construct the subnet of hosts that define the network over which the view operates. This construction is performed on-demand; `NetworkAbstractions` only builds and maintains views for the `EgoManager` when operations are issued to avoid unnecessary communication overhead. This is important to ensuring as efficient an implementation as possible.

View Operation and Agent Interaction. When the reference agent issues an operation on a `View`, the operation and view constraint information are passed to the `EgoManager`. The `EgoManager` creates a dedicated operation thread for the request.

Atomic Blocking Operations. Figure 12 shows a sequence diagram describing an **in** operation. The calling thread blocks until the operation thread finds a tuple matching the operation's template. The operation thread uses `NetworkAbstractions` to distribute a persistent query to every host in the context, and the query remains registered on those hosts until the operation thread deregisters it. If new hosts move into the context while the query remains active, they receive the query. Similarly, as hosts move out of the context, the query is removed from them.

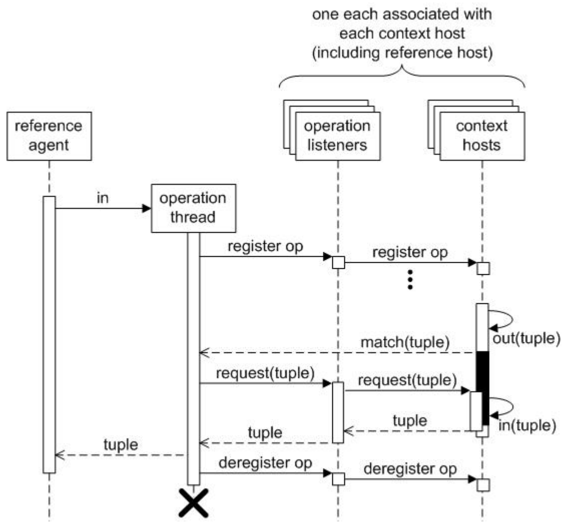


Fig. 12. Sequence diagram of an **in**

Two things can happen when an operation is registered. First, a tuple may immediately match. If so, the context host notifies the operation thread. If not, the context host stores the registration and checks every tuple generated to see if it matches. When a tuple matches the request, the context host reserves the matching tuple for the requesting agent until either the operation thread requests it be removed and returned or the operation's query is deregistered (indicated as the blackened active period in Figure 12). A match may also be triggered by a new host with a matching tuple moving into the view. The registration of the operation on this arriving host (as well as deregistration from any departing host) is handled implicitly by the `NetworkAbstractions` protocol.

When the operation thread receives notification of a matching tuple, it sends a message to the owning host to remove the tuple. It is possible that the operation thread will receive multiple matches for an **in** operation from multiple context hosts; it chooses one non-deterministically. Once the operation is ready to return, the persistent operation query is deregistered from the context hosts.

The other blocking operations have a similar form. When a context host finds a match to a **rd** operation, it simply returns the match and waits for the operation thread to deregister the query. Aggregate operations perform the same steps as their counterparts, but to ensure they return all matching tuples, when the operation finds a match, the operation thread issues an aggregate atomic probing operation, described next.

Atomic Probing Operations. The sequence diagram in Figure 13 shows a **rdp** operation. Again, when the reference agent issues its operation, the `EgoManager` spawns a dedicated operation thread; the reference agent remains active, waiting for a response. If, after checking each host in turn, the operation thread finds no matching tuple, it will return a null value. The operation thread first collects the ids of hosts within the view by sending a query to the hosts defined by the view's network constraints. Every host within the context responds with its host's id and the host ids of its children in the tree. The `EgoManager` on the reference agent's host

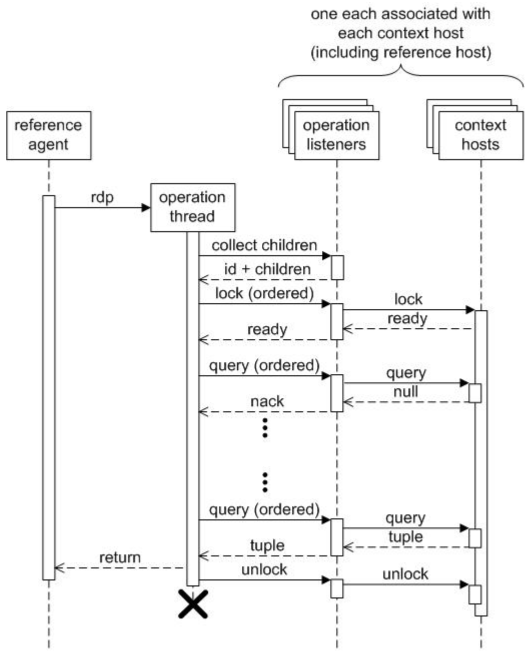


Fig. 13. Sequence diagram of a **rdp**

uses this information to ensure that it hears from every member of the context before continuing. At this point, the set of hosts on which the operation will be performed is fixed. If new hosts move within the constraints of the view, their addition to the context is delayed until this operation completes.

When the operation thread has gathered the ids of all context hosts, it locks them in order of increasing id. The ordered locking prevents deadlock because every operation thread locks hosts in the same order. Locking a tuple space prevents other threads from modifying the tuple space's contents. When a context host receives a locking request, it waits until its tuple space is not locked by another thread, then returns positively. The operation thread waits to hear from each context host before locking the next host.

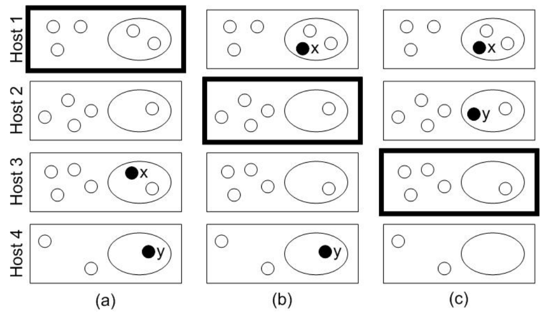


Fig. 14. Locking example

The need for locking is not immediately obvious. Consider, however, the case shown in Figure 14, which shows four host tuple spaces that contain tuples in the reference agent's view. The ellipse inside each host tuple space contains the tuples that satisfy the view constraint. The black tuples also satisfy the operation's template. In this figure, the operation queries the host tuple spaces for matching tuples in order; the outlined rectangle indicates the host tuple space being queried. In Figure 14(a), the operation thread first queries Host 1. Being unsuccessful, in part (b), the operation thread then queries Host 2. At the same time, a different operation thread moves tuple x from Host 3's tuple space to Host 1's tuple space. This is allowed because the tuple spaces are not locked. In part (c), because the operation thread did not find a matching tuple, it queries Host 3, while the tuple y is moved to Host 2. The operation thread finds no match at Hosts 3 or 4. This violates the semantics of the atomic probing operation because a matching tuple existed in the view the entire time the operation was processed.

After locking every host in the context, the operation thread requests a matching tuple from every host in order. For the **rdp** operation, as soon as the operation thread finds a single match, it returns the tuple. For an **inp** operation, the operation thread also returns the first match, but the matching tuple is removed from the owning agent's host tuple space. For aggregate operations, the actions performed are the same, except that the operation thread must query every host tuple space instead of halting once it finds a match.

Scattered Probing Operations. These operations provide weaker semantics than the previous two in that the operations are allowed to miss matching tuples in the view. That is, the case shown in Figure 14 is acceptable. The weakened semantics of these operations allow more efficient implementations that do not require locking. The sequence of events in executing a scattered probing operation follows those of an atomic probing operation, without the need to lock the context hosts. Thus, context hosts are active only while responding directly to the operation thread.

7 Conclusions

Context-aware abstractions of the operating environment prove essential to the rapid development of applications designed for ad hoc mobile networks. EgoSpaces utilizes a novel approach that combines the use of context-awareness and asymmetric coordination that proves successful in easing the development burden of applications in this environment. These constructs are especially useful in dealing with the large amounts of data encountered in large-scale ad hoc networks. As the ubiquity of mobile computing devices continues to grow, the need for middleware providing this style of coordination becomes increasingly apparent. This paper highlighted the software engineering gains associated with providing application programmers abstractions of the unpredictable operating contexts the resulting applications may encounter over their lifetimes. Specifically, EgoSpaces facilitates opportunistic interactions through the view concept which dynamically changes to represent a changing environment. The programming constructs are provided as a simplified interface to the application's world, yet they retain the flexibility and expressiveness required from the variety of applications needed in mobile ad hoc networks.

ACKNOWLEDGMENTS

This research was supported in part by the Office of Naval Research under ONR MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsoring agencies. The authors would also like to thank Rohan Sen and Tom Elgin for the help in the middleware implementaiton.

References

1. Want, R., Hopper, A., Falco, V., Gibbons, J.: The Active Badge location system. ACM Transactions on Information Systems **10** (1992) 91–102
2. Want, R., et al.: An overview of the PARCTab ubiquitous computing environment. IEEE Personal Communications **2** (1995) 28–33
3. Harter, A., Hopper, A., Steggles, P., Ward, A., Webster, P.: The anatomy of a context-aware application. Mobile Networks **8** (2002) 187–197
4. Abowd, G., Atkeson, C., Hong, J., Long, S., Kooper, R., Pinkerton, M.: Cyberguide: A mobile context-aware tour guide. ACM Wireless Networks **3** (1997) 421–433

5. Cheverst, K., Davies, N., Mitchell, K., Friday, A., Efstratiou, C.: Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In: Proceedings of MobiCom, ACM Press (2000) 20–31
6. Pascoe, J.: Adding generic contextual capabilities to wearable computers. In: Proceedings of the 2nd International Symposium on Wearable Computers. (1998) 92–99
7. Rhodes, B.: The wearable remembrance agent: A system for augmented memory. In: Proceedings of the 1st International Symposium on Wearable Computers. (1997) 123–128
8. Salber, D., Dey, A., Abowd, G.: The Context Toolkit: Aiding the development of context-enabled applications. In: Proceedings of CHI'99. (1999) 434–441
9. Hong, J., Landay, J.: An infrastructure approach to context-aware computing. Human Computer Interaction **16** (2001)
10. Roman, M., Hess, C., Cerqueira, R., Ranganat, A., Campbell, R., Nahrstedt, K.: Gaia: A middleware infrastructure to enable active spaces. IEEE Pervasive Computing **1** (2002) 74–83
11. Verissimo, P., Cahill, V., Casimiro, A., Friday, K.C.A., Kaiser, J.: CORTEX: Towards supporting autonomous and cooperating sentient entities. In: Proceedings of European Wireless. (2002)
12. Chen, G., Kotz, D.: Solar: An open platform for context-aware mobile applications. In: Proceedings of the 1^{st} International Conference on Pervasive Computing. (2002) 41–47
13. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. In: Proceedings of the 10^{th} International Symposium on the Foundations of Software Engineering. (2002)
14. Roman, G.C., Julien, C., Huang, Q.: Network abstractions for context-aware mobile computing. In: Proceedings of the 24^{th} International Conference on Software Engineering. (2002) 363–373
15. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems **7** (1985) 80–112
16. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. Internet Computing **4** (2000) 26–35
17. Murphy, A.L., Picco, G.P., Roman, G.C.: Lime: A middleware for physical and logical mobility. In: Proceedings of the 21^{st} International Conference on Distributed Computing Systems. (2001) 524–533
18. Julien, C., Roman, G.C.: Active coordination in ad hoc networks. In: Proceedings of the 6^{th} International Conference on Coordination Models and Languages. (2004) 199–215
19. Julien, C., Roman, G.C.: A protocol supporting context provision in wireless mobile ad hoc networks. Technical Report WUCSE-03-57, Washington University (2003)