Egocentric Context-Aware Programming in Ad Hoc Mobile Environments

Christine Julien and Gruia-Catalin Roman
Department of Computer Science and Engineering
Washington University
Saint Louis, MO 63130
{julien, roman}@cs.wustl.edu

ABSTRACT

Some of the most dynamic systems being built today consist of physically mobile hosts and logically mobile agents. Such systems exhibit frequent configuration changes and a great deal of resource variability. Applications executing under these circumstances need to react continuously and rapidly to changes in operating conditions and must adapt their behavior accordingly. The development of such applications demands a reexamination of the notion of context and the mechanisms used to manage the application's response to contextual changes. This paper introduces EgoSpaces, a coordination model and middleware for ad hoc mobile environments. EgoSpaces focuses on the needs of application development in ad hoc environments by proposing an agent-centered notion of context, called a view, whose scope extends beyond the local host to data and resources associated with hosts and agents within a subnet surrounding the agent of interest. An agent may operate over multiple views whose definitions may change over time. An agent uses declarative specifications to constrain the contents of each view by employing a rich set of constraints that take into consideration properties of the individual data items, the agents that own them, the hosts on which the agents reside, and the physical and logical topology of the ad hoc network. This paper formalizes the concept of view, explores the notion of programming against views, discusses possible implementation strategies for transparent context maintenance, and describes our current prototype of the system. We include examples to illustrate the expressive power of the view abstraction and to relate it to other research on coordination models and middleware.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.
Copyright 2002 ACM 1-58113-514-9/02/0011 ...$5.00.

General Terms

Design, Reliability, Human Factors, Algorithms

Keywords

Mobility, Context-Aware, Ad Hoc Network, Coordination, Scalability

1. INTRODUCTION

In the recent past, a number of researchers advocated the ability for applications to adapt their behavior in response to changes in their environments. Applications in which context sensors capture changes in the environment and pass them to the application are called context-aware. As user mobility has increased, context-aware models of computation have gained popularity because they offer the ability to include information about the current environment as part of the computation, thereby enhancing the services available to the users. Context-aware applications and development frameworks existing today generally gain access to context information through context sensors. Cyberguide [1] and GUIDE [7], two tour guide applications, use events generated by location sensors to update the user's screen according to physical location. The Stick-e Document [4] framework allows users to create notes that are triggered when the user encounters the associated context. Events indicate context changes, and they trigger the display of a stored note. Both Active Badge [11] and PARCTab [20] provide a framework on top of which developers build context-aware applications. In both cases, small devices moving about the research complex beacon location events to the architecture via infrared communication links. In PARCTab, each device, called a tab, has an associated agent responsible for mediating the context events. Each tab has a single current application to which the tab's agent forwards location events generated by the tab. The Active Badge framework allows applications to query the device's current context. Additionally, applications can register for notifications of specific context changes by specifying a filter indicating the contextual information that interests it and a callback to invoke when the filter is satisfied. The FieldNotes [16] application extends the types of context used to include time, weather, and user information by allowing researchers in the field to attach varied contextual information to their notes. With the increase in the variety and complexity of context information, frameworks and systems for generalizing its treatment are being developed. The Context Toolkit [17]

generalizes interactions among components through context widgets, while the Context Fabric [12] provides a service infrastructure. By and large these systems limit the context to what a component can immediately sense, ignoring what other networked components sense. While discussions of context-aware software [18] have hinted at the need for an extended view of context, none of the better known systems allows such access without requiring application developers to code it explicitly.

In this paper, we consider systems that entail both physical and logical mobility. For presentation purposes, we assume a system existing of mobile agents (representing units of modularity and logical mobility) that execute over potentially mobile hosts (units of physical mobility). The mobility of hosts and software agents adds a new degree of complexity to this changing environment. As hosts and agents become more mobile and travel to completely new environments, the resources they can access change and the manner of this access evolves. The presence of other agents, the availability of resources associated with them, the specific host providing the agent's execution environment, and the connectivity of other hosts and their particular location or movement behavior all have the potential to affect the behavior of a single agent. The initial impetus for this work came from our attempt to apply other models for ad hoc mobile coordination to vehicle to vehicle communication applications on the highway. Such applications can vary from time and safety critical ones like negotiating the safe passage of automobiles through intersections without traffic lights to entertainment applications including file sharing between cars or distributed video games. In each case, the application constructs its view based on different criteria and requires different guarantees for its operation over the view. For example, in a safe intersection example, a single car will collect information from other cars in close proximity and from sensors in the intersection. As a second example, imagine a building or construction site with a fixed infrastructure of sensors and information appliances that provide contextual information for applications running inside the building. Sensors can provide a multitude of information, including data regarding the structural integrity of the building, the frequency of sounds in the building, or the movement of building occupants. Additionally, engineers or inspectors carry PDAs or laptops that provide additional context and assimilate context information to accomplish specific tasks. Different people have different tasks and will therefore use information from different sets of sensors.

Our research goal is to develop a formal abstract treatment of context-awareness and offer middleware to the programmer to manage an extended notion of context. The middleware should allow the programmer not only to define a specific context but to influence that context's definition. It should further provide a flexible and general usage pattern for the context. Our approach centers on the concept of *view* as an abstraction of a particular agent's operating context with respect to a specific agent interest. An agent can specify multiple views, each designed to meet the agent's needs for specific contextual data. The agent has full control over the specification of each view and may change it at runtime. Formally, a view could encompass all the data that the agent can reach in the network. However, the key is to allow the agent to control the scope of the individual views in a manner that facilitates easy program development (in the mobile setting) without excessive performance penalties. In this manner, the agent defines exactly which pieces of data belong in the set of data the view encompasses. Our middleware will provide an agent with specification mechanisms (a language) through which it can define views based on properties of the network topology, properties of the hosts in the network, properties of other agents in the network, and properties of individual data items owned by other agents. As an agent moves through space and time, the content of each view changes to reflect the currently available data. In this paper, we start with the premise that the operating context is all-encompassing, provide a precise definition of a view as an agent-defined restriction of the operating context, and develop a formal approach for view specification. We move on to explain the dynamics of view maintenance and usage in the presence of mobility and to demonstrate its implementability and expressive power.

The remainder of this paper is organized as follows. In Section 2, we explain our extended definition of context in a mobile setting. Section 3 explains the view abstraction in detail and discusses what it encompasses, how it is accessed, and how it is maintained. Section 4 discusses how an agent programs to a view. Section 5 reviews the current status of our implementation and presents an example application. Section 6 discusses the view's expressive power. Finally, conclusions appear in Section 7.

2. EXTENDED CONTEXT

Our extended notion of context potentially includes everything available in the network. The target applications for our middleware, however, may operate in environments where the network could grow unmanageably large. This necessitates mechanisms for restricting this large context to some smaller operating context, or view. This section precisely defines our computational model and gives an overview of how and why applications provide view specifications.

Computational Model. We assume a computing model in which hosts can move in physical space, and the applications they support are structured as a community of mobile software agents that can migrate from one host to another. Thus, in our computing model, an agent is the unit of modularity, execution, and mobility, while a host is a container for agents characterized, among other things, by its location in physical space. We use the term agent to refer to any stand-alone piece of software code capable of moving between connected hosts. Communication among agents and agent migration can take place whenever the hosts involved can physically communicate with each other, i.e., they are connected. A closed set of these connected hosts defines what we will refer to as an ad hoc network.

Since the notion of context is always relative, we use the term *reference agent* to denote the agent whose context we are considering, and we will refer to the host on which the agent is located as the *reference host*. In principle, the context associated with a given agent in an ad hoc network consists of all the information available in that network. Of course, such broad access to information is generally costly to implement. In addition, various parts of the same application may need different resources at different times during the execution of the program. For this reason, we believe that it is important to structure the context in terms of fine-grained units which we call views. A *view* is a projection of everything available to the reference agent together with an

interpretation that defines the rules of engagement between the agent and the particular view. An agent can be associated with one or more views (which can be redefined over time) and can operate on each view in a manner compatible with the view definition. The actual *contents* of the view may be visible to the agent directly or indirectly, depending on the abstract interpretation associated with that view. For the purposes of this paper, we assume that a view contains discrete data items that the reference agent accesses using various data access operations.

Declarative View Specification. The view concept is egocentric in the sense that every view is defined relative to a reference agent and with respect to its needs for resources from and knowledge about its environment. Although we will focus on the specification of and operation over a single view, an agent sees the world through a set of views that may be altered at will by defining, redefining, and discarding views as processing requirements demand. An agent describes its contextual needs to the underlying context maintenance system by providing a declarative specification of a projection of the maximal context. Through this specification, the programmer controls the scope of the view (a larger or smaller neighborhood of the network) and the size of the view (the range of entities included). The former is accomplished by providing constraints over the properties of the network, hosts, and agents, while the latter is achieved through the use of constraints over the data itself. For example, an agent mediating an automobile's safe passage through an intersection might declare the following view:

> *All location data (reference to data) entered by collision warning agents (reference to agents) on cars within 100 meters (reference to hosts) of my current location (property of the reference host).*

Figure 1 shows an evaluation of the declarative view specification. The figure shows cars on a highway; the arrows indicate their approximate movement patterns. In the figure, the rectangle labeled "X" represents the reference agent's car. To simplify this picture, we assume only a single agent per car. In the top picture, the reference agent provides a restriction of the cars that participate in the view. The center picture shows how hosts and data items (circles in the picture) map to cars. Because the reference agent is interested only in location data (represented by blackened circles in the bottom picture) the actual view contains only these data items. As discussed in later sections, these restrictions on the network, hosts, agents, and data and the evaluation of these restrictions by the underlying context maintenance system account not only for the needs of the reference agent but also for security and network considerations.

Transparent Context Maintenance. The use of declarative context specifications allows applications to shift the burden of sensing and maintaining context information to the underlying middleware. Upon specifying a view, an application relies on the middleware to perform transparent updates of that view when other hosts or agents in the network move or contextual information changes. Imagine the previously introduced example of a building inspector examining a building while carrying a PDA. As he moves through the building, the inspector wishes to see information not for the whole building, but for his quadrant on the floors adjacent to the one he is currently inspecting. As he changes floors, his view is automatically updated to reflect the changing context. The transparent context maintenance provided by the underlying middleware gives the application programmer explicit control over the cost associated with context maintenance. As discussed later, operations of varying consistency allow a programmer to govern the relative cost of executing a particular operation on a view.

3. DATA CONTEXT

An application agent's success resides in its ability to easily but specifically define a restriction of the operating context. Our extension of an agent's context to include all data belonging to agents on reachable hosts enlarges the agent's operating context to include more than it may ever need to access. The previous section touched upon our use of declarative view specifications for narrowing the set of data an agent "sees" to exactly the data needed for operation. Next, we turn our attention to the details of view specifications and the data over which they are defined.

3.1 Data Representation

The manner in which each agent perceives and accesses data has ramifications for the ease of programming and the efficiency of operations over data in the view. Different application needs as well as a programmer's expertise all contribute to the choice of a mechanism for data representation. Therefore, we separate the contents of the view from the presentation of the view to the application agents. That is, we assume a single data representation as a basis for coordination. Other forms of interaction can be easily added to the middleware through the use of thin veneers over our chosen tuple space representation.

Underlying Database of Tuples. Tuple space representations based on the Linda tuple space model [10] enjoy a great deal of popularity due to the content-based manner in which data is accessed. In mobile computing specifically, several systems have found success using shared tuple spaces. MARS [5] focuses on logical mobility, or the movement of agents over physically stationary hosts. This system uses a tuple space to allow coordination among co-located

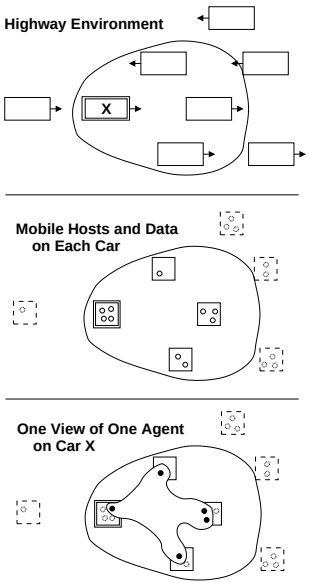

Figure 1: View used by a collision warning agent on car X

mobile agents, while LIME [14] combines support for logical mobility with support for physical mobility and relies on transient sharing of tuple spaces among agents and hosts within communication range. We reuse this notion of transient sharing of tuple spaces, combine it with a more flexible tuple representation, and allow an agent to use a declarative view specifications to indicate with which other components it wants to share data.

A tuple is an unordered set of triples of the form:

$$\langle (name, type, value), (name, type, value), \ldots \rangle.$$

For each field, *name* is the name given to the field, and *type* is the data type of *value*. In a given tuple, the *names* of each field must be unique. The *name* field allows us to relax the ordering restrictions seen in traditional uses of tuples, allowing EgoSpaces more flexibility and openness.[1] Fundamentally, users access tuple spaces by matching patterns against contents of tuples. While adhering to the content-based nature of Linda pattern matching, we extend the traditional semantics to allow the provision of more flexible constraints over fields. A pattern takes the form:

$$\langle (name, type, constraint), (name, type, constraint), \ldots \rangle.$$

In patterns, *name* and *type* are identical to their counterparts in tuples. The *constraint*s are functions that provide requirements that the *value* in a field must match for the field in the tuple to match the field in the pattern. More specifically, the matching function $\mathcal{M}$ is defined over a tuple θ and a pattern p as:

$$\begin{aligned}\mathcal{M}(\theta, p) \equiv \langle \forall c : c \in p :: \\ \langle \exists f : f \in \theta \wedge f.name = c.name \\ \wedge f.type \texttt{ instanceof } c.type \\ :: c.constraint(f.value) \rangle \rangle.\end{aligned}$$ [2]

The matching function, $\mathcal{M}$, requires that, for every constraint c in the pattern, there must be a field f in the tuple with the same name, the same type or a derived type, and a value that satisfies the constraint. While the function does require that each constraint is satisfied, it does not require that every field in the tuple is constrained, i.e., a tuple must contain exactly the fields contained in the pattern, but the tuple can contain additional fields. Because a field's constraint is a function evaluated over the field's value, it allows both positive and negative constraints. We will discuss the use of this pattern matching in more detail in Section 4.2.

Presentation. Many applications benefit from direct access to the tuple space. Other applications, however, operate more naturally over a different structure, for example a tree or a self-organizing list. Because different applications benefit from different presentations of the underlying data, individualized veneers can abstract the data representation. Such veneers provide wrappers for the tuple space calls. The application operates as if it sees, for example, a tree, and the veneer translates operations on the tree into operations on the underlying tuple space. Because these veneers and their target applications lie outside the scope of this paper, we discuss only operations directly on the tuple space.

[1] LIME 2.0 adopts this type of tuple and tuple matching.

[2] The three-part notation $\langle \mathbf{op}\, quantified\_variable : range :: expression \rangle$ used throughout the text is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression*, producing a multiset of values to which **op** is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for **op**, e.g., *true* when **op** is $\forall$ or zero if **op** is "+" .

3.2 View Specification Mechanics

In our computational model, properties of hosts, agents, and data all contribute to the definition of the operational context. In providing a view specification, an agent indicates the specific data that should comprise its view. This view, therefore, consists of a subset of all the data available on reachable hosts in the network. The data, the agents owning the data, the hosts where the agents are located, and the paths to those hosts must all satisfy the view specification.

Network Constraints. As indicated previously, we extend the availability of context information beyond a host's immediate scope, i.e., a host should be able to gather information from a subnet of the entire ad hoc network. Doing so requires an abstraction of the network topology and its properties. After specifying some constraints including the application's specific definition of distance, an application on the reference host would like a qualifying list of acquaintances. That is:

> *Given a host, α, in an ad hoc network, and a positive bound value, D, find the set of all hosts, Q_α, such that the cost of the shortest path from α to each host in Q_α is less than D.*

Abstractly, one can view this list as a subnet around the reference host.

To build this list, we first define a way to determine the cost of a path. Costs derive from quantifiable aspects of the reference host's context. In any network, both hosts and the links between them have attributes that affect the communication in the network. We abstract these properties by combining the quantified properties of two connected nodes with the quantified properties of the link between them to achieve a single weight for each link in the network. An application has the freedom to specify which properties define the weights of links. As a simple example, each link can have a weight of one. This allows us to count the number of hops between two nodes in the network.

Once a weight has been defined and calculated for each link, a cost function specified by the application can be evaluated over these weights to determine the cost of a particular path in the network. Continuing the network hop count example, the cost function specified by the application would be the sum of the weights of the links along a path. Because the weight of each link is one, the number of hops from the source of the path to that node determines the cost at that node. The only restriction placed on the cost function is that the cost of a given path must strictly increase as the number of hops from the reference host grows. We will see below how this allows us to apply a bound to the computation of the context. In a real network, however, multiple paths may exist between two given nodes. Therefore, we build a tree rooted at the reference host that includes only the lowest cost path to each node in the network.

Because we aim to restrict the scope of an application's context, calculating the lowest cost to every node in the network is not reasonable. To limit the context specification, we require the application to specify a bound for its cost function. Nodes to which the cost is less than the bound are included in the context. Because cost functions must strictly

increase, once the computation reaches a node that lies outside the bound, all nodes farther on the same path must also lie outside the bound. By combining the previous components of the network abstraction, we see that in providing this piece of the context specification, the application agent must include three things: the mechanism for calculating the weight of a link, the cost function used to determine the cost of the path, and a bound on that cost function. For the hop count example, an entire context specification might be written as: all nodes which can be reached in fewer than five hops. The evaluation of this context specification results in a tree rooted at the reference node and spanning a subnet of the entire ad hoc network. The path to every node in this tree satisfies the restrictions imposed by the context specification's cost function and bound, and this tree is maintained by the underlying system as long as necessary for supporting the application. That is, as hosts move in the network, the properties defining this tree change, thus changing both the contents and the topology of the tree.

Our previous work [15] on this abstraction has shown that it is amenable to elegant formal treatment. We have also developed a protocol that builds and maintains a network abstraction individualized for the specifying reference host.

Host Constraints. While the network constraints deal with physical properties of the host, the host constraints handle logical properties. Examples of such properties include the host's id, the identity of the owner of the computer, or services provided by the computer. A host stores these properties in a *host profile*, which can be viewed as a special private tuple where the fields are host attributes:

$$\langle (att\_name, type, value), (att\_name, type, value), ... \rangle$$

Host constraints can then be provided as a pattern over this profile with the matching function and semantics outlined previously. For example, a host wanting to print a document could restrict contributing hosts to color printers of a certain quality. In such a case, printers might have attributes representing the service they provide, the type of printer, whether or not the printer is color, and the dpi of the printer. An example profile for a printer might be:

$$\langle (service,\ enumeration,\ printer),\\ (printer\ type,\ enumeration,\ laser),\\ (color,\ boolean,\ true),\\ (quality,\ integer,\ 1200) \rangle.$$

A constraint that would match this profile and satisfy the previous example host's requirements would be:

$$\langle (service,\ enumeration,\ printer),\\ (color,\ boolean,\ true),\\ (quality,\ integer,\ > 800) \rangle.$$

The example constraint does not restrict the type of the printer because the printer type does not interest the specifying host. Because the host demands that the printer be of at least a certain quality, the last constraint provides a function over the printer's dpi that must be satisfied.

Agent Constraints. Every EgoSpaces agent defines a profile similar to a host profile, containing agent properties instead of host properties. Providing constraints over agent profiles allows application agents to restrict the set of agents who contribute data to the view. Because agents are mobile pieces of code, an obvious agent property is the host on which the agent is located. Other, more application specific properties are also useful. For example, in the building inspection domain discussed in the introduction, some application agents may sense air quality throughout the building, while other agents on the same devices monitor physical vibrations. Restricting operations to one type of agent or the other increases the efficiency with which coordination can occur by decreasing the number of parties involved.

Data Constraints. In the same way that agent constraints allow an application agent to restrict the agents contributing to the view, the data constraints allow the same application agent to restrict the individual data items in the view. To accomplish such a restriction, the application agent simply supplies a data pattern that all data in the view must satisfy. The use of this constraint can be extended if an application attaches "meta-data" to a piece of data by inserting extra fields in the application's tuples that can be used in matching data constraints.

3.3 Formal View Definition.

Given these four types of constraints, a view specification consists of three patterns (one over data items, one over agent profiles, and one over host profiles) and the network constraints (consisting of a metric for link weights, a network cost function, and a bound on that function). With this information, our middleware constructs a view for the application. The view is defined by the set of tuples belonging to it. Given a reference host r, we first define η, the subnet of the ad hoc network that satisfies the provided network constraints (n) to be a subset of the closure of r's network. η must be a tree, r must belong to η, and η must satisfy n.

Given the network constraints (n), the host constraints (h), the agent constraints (a), and the data constraints (d), a view specified by a reference agent r contains the tuples defined by:

$$\begin{aligned} &\mathsf{view}_{\mathsf{r}}(n,h,a,d) \triangleq \\ &\quad \langle \mathsf{set}\ \eta,\gamma,\alpha,\theta : \eta \subseteq Closure(\mathsf{r}) \wedge tree(\eta) \\ &\qquad\qquad \wedge\, \mathsf{r} \in \eta \wedge \eta\ \mathtt{sat}\ n \\ &\qquad\qquad \wedge\, \gamma \in \eta \wedge \mathcal{M}(\gamma.\mathsf{profile}, h) \\ &\qquad\qquad \wedge\, \alpha.\mathsf{loc} = \gamma \wedge \mathcal{M}(\alpha.\mathsf{profile}, a) \\ &\qquad\qquad \wedge\, \theta \in \alpha.T \wedge \mathcal{M}(\theta, d) \\ &\qquad\qquad :: \theta \rangle, \end{aligned}$$

where γ is a host, α is an agent, and θ is a tuple. $\alpha.T$ refers to the agent α's local tuple space. This function assumes that the host on which an agent is currently located is accessible through a variable at the agent, loc. Throughout our discussion, we will refer to a view as ν.

As hosts and agents move and the available data changes, the middleware updates the tuples available in the view. From the application's perspective, all of these changes are transparent and manifest only in changes in the set of available data items. Therefore, the application agent can operate over a view without regard for the changes occurring in that view. The application also has the freedom to change the constraints associated with its view dynamically, and, when it does, the middleware recalculates the view to reflect the application's new needs.

3.4 Access Controls

As discussed previously, both agents and hosts constantly move, and EgoSpaces provides flexible and efficient communication in the face of such changes. However, in any shared data space, access control becomes a real problem. EgoSpaces addresses the access control issue by adding an

access control function. 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. Additionally, the specifying agent declares the operations it intends to perform over the view. When determining the contents of a view, EgoSpaces evaluates each contributing agent's access control function over the view's credentials and potential operations. The fact that the access control function is evaluated on an individual basis for each tuple adhering to the view constraints provides a very fine level of granularity. The definition of the view becomes dependent on the evaluation of these access control functions. The following definition shows the previous formal view definition augmented to account for the credentials (κ) of the reference agent, the operations that will be performed on the view (ops), and the access control function for an agent α ($\alpha.acf$):

$$\begin{aligned}&\mathsf{view_r}(n, h, a, d, \kappa, ops) \triangleq \\ &\quad \langle \mathsf{set}\ \eta, \gamma, \alpha, \theta : \eta \subseteq \mathit{Closure}(\mathsf{r}) \wedge \mathit{tree}(\eta) \\ &\qquad \wedge \mathsf{r} \in \eta \wedge \eta\ \mathsf{sat}\ n \\ &\qquad \wedge \gamma \in \eta \wedge \mathcal{M}(\gamma.\mathsf{profile}, h) \\ &\qquad \wedge \alpha.\mathsf{loc} = \gamma \wedge \mathcal{M}(\alpha.\mathsf{profile}, a) \\ &\qquad \wedge \theta \in \alpha.T \wedge \mathcal{M}(\theta, d) \\ &\qquad \wedge \alpha.acf(\kappa, ops, \theta) \\ &\quad :: \theta \rangle.\end{aligned}$$

The provision of *ops* is a contract between the specifying agent and the EgoSpaces system. Any attempt by the specifying agent to perform operations not declared for a view will result in an error. A tuple belongs to a view only if it satisfies the view constraints and the reference agent meets the requirements of the access control function of the agent owning the tuple. The next section covers in more detail how agents perform operations over views they specify.

4. VIEW PROGRAMMING

An agent interacts with the world by specifying views that define projections of the set of all available tuples. EgoSpaces then allows an agent access to its views both through traditional tuple space access operations and through active views, which attach additional functionality to a specified view.

4.1 Multiple Views

An agent operates over the ad hoc network through a set of views, with their own view specifications. These views can overlap, i.e., two views can contain the same pieces of data, but an agent operates over only a single view at a time. Because the view concept is agent-centric, each agent defines the views needed for its successful operation. Given the specifications, EgoSpaces constructs and manages all these views transparently from the application's perspective.

4.2 Basic Operations

Basic tuple space operations can be divided into two groups: tuple generation operations that place new tuples in the agent's local tuple space and on-demand tuple access operations that allow a reference agent to read and remove tuples in one of its views.

Tuple Generation. An agent creates a tuple by performing an **out** operation on its local tuple space (T):

$$\boxed{\mathbf{out}(T, t)}$$
$$T := T \cup \{(\mathtt{ID}, \mathit{tuple\ id}, \mathit{newId}()] \oplus t\}.$$

We use this notation throughout the discussion to denote the operational semantics of tuple space and view operations. The operation appears in the box, and its operational semantics follow as an abstract program. The **out** operation augments the data tuple t provided by the application with a unique ID field and places the tuple in the local tuple space. Tuple insertion into the local tuple space is atomic with respect to all other operations on that local tuple space.

On-Demand Tuple Access. Application agents gain access to tuples through pattern matching over the tuples' contents. The scope of such access operations is restricted to a single view. To operate on the tuple space, an agent provides a pattern for the desired tuple. The detection of a matching tuple in the view uses the previously defined matching function, $\mathcal{M}$, and can be formalized as:

$$\mathsf{matchExists}(\nu, p) \triangleq \langle \exists\, \theta : \theta \in \nu :: \mathcal{M}(\theta, p) \rangle.$$

In this definition, p is a pattern used for matching tuples, and ν refers to a specific view defined by the reference agent. The content of ν changes to reflect the current data available in the context. Additionally, ν reflects the evaluation of the access control functions; tuples must pass the access control restrictions of their owning agents before becoming available in the view. We will reuse these variable names with the same meaning throughout the view programming discussion. The requested operation on the view is ultimately performed only on the set of tuples belonging to the view that match the pattern:

$$\mathsf{matchingSet}(\nu, p) \triangleq \langle \mathsf{set}\ \theta : \theta \in \nu \wedge \mathcal{M}(\theta, p) :: \theta \rangle.$$

The two basic types of operations allowed mirror the access operations in Linda. As in Linda, these operations are blocking, meaning that they return immediately upon finding a matching tuple in the view; if a matching tuple does not exist upon issuance of the operation, the operation blocks until one does exist. The first type, a **rd** operation, returns a copy of a tuple in the specified view that matches the provided pattern. A **rd** copies a tuple by selecting one nondeterministically from the matchingSet and returning a duplicate of it. The nondeterministic selection of a tuple from the matchingSet uses the *nondeterministic assignment statement* [3]. A statement $x := x'.Q$, assigns to x a value x' nondeterministically selected from among the values satisfying the predicate Q. If such an assignment is not possible, the statement aborts:

$$\boxed{t := \mathbf{rd}(\nu, p)}$$
$$\langle \mathbf{await}\ \mathsf{matchExists}(\nu, p) \rightarrow t := t'.(t' \in \mathsf{matchingSet}(\nu, p)) \rangle^{3}.$$

The use of the matchExists guard guarantees that at least one tuple exists in the matchingSet, and, therefore, the nondeterministic assignment will succeed. If the guard evaluates to false, the test is attempted later until the synchronization condition evaluates to true. A discussion of a non-blocking **rd** operation and how it differs follows below. Tuples returned by a **rd** operation remain in the tuple space.

[3]The $\langle \mathbf{await}\, B \rightarrow S \rangle$ construct [2] allows a program to delay execution until the condition B holds. When B is true, the statements in S are executed in order. The angle brackets enclosing the construct indicate that the statement is executed atomically, i.e., when S executes, B is guaranteed to still be true, and no state internal to S is visible outside the execution of S.

The second type of operation, **in**, returns a tuple in the specified view that matches the provided pattern. Unlike **rd**, however, **in** removes the returned tuple from the tuple space. Removal of a tuple is accomplished by nondeterministically selecting a tuple from the pattern's matchingSet, removing the tuple from the tuple space, and returning it:

$$\boxed{t := \textbf{in}(\nu, p)}$$
$$\langle \textbf{await } \mathsf{matchExists}(\nu, p) \rightarrow t := t'.(t' \in \mathsf{matchingSet}(\nu, p)) \; || \langle || \alpha : t \in \alpha.T :: \alpha.T := \alpha.T - \{t\} \rangle \rangle^{4},$$

where α is the agent owning the tuple t.

Several common extensions of the Linda primitives [14, 13, 9, 19] include probing operations. As alluded to above, these operations differ from the blocking operations by returning immediately, even if a matching tuple does not exist in the view. As an example, **rdp** returns a copy of a matching tuple if one exists; otherwise it returns ϵ:

$$\boxed{t := \textbf{rdp}(\nu, p)}$$
$$\langle \textbf{if } \mathsf{matchExists}(\nu, p) \textbf{ then } t := t'.(t' \in \mathsf{matchingSet}(\nu, p)) \textbf{ else } t := \epsilon \textbf{ fi} \rangle.$$

The definition of **inp** is the same but removes the tuple.

Access operations can return single tuples or groups of tuples. We refer to operations returning only one tuple as single operations and to those returning multiple tuples as aggregate operations. All the operations we have discussed thus far fall in the category of single operations. Because a single operation returns only a single tuple, if the operation finds more than one matching tuple, it nondeterministically chooses which to return. Aggregate operations, on the other hand, return the entire set of matching tuples. Aggregate operations can be either blocking or probing. Blocking aggregate operations (**rdg** and **ing**) block until at least one tuple in the view matches the pattern. A **rdg** returns a copy of all matching tuples. The **ing** operation builds on this by additionally removing all of the matching tuples from their respective tuple spaces:

$$\boxed{tset := \textbf{ing}(\nu, p)}$$
$$\langle \textbf{await } \mathsf{matchExists}(\nu, p) \rightarrow tset := \mathsf{matchingSet}(\nu, p) \; || \langle || \theta, \alpha : \theta \in \mathsf{matchingSet}(\nu, p) \wedge \theta \in \alpha.T :: \alpha.T := \alpha.T - \{tset\} \rangle \rangle.$$

The probing versions of aggregate operations closely resemble the other probing operations—they return immediately and do not wait for a matching tuple to appear. Instead, they return all of the tuples available that match, and, if none do, the operations return an empty set. A **rdgp** simply returns the matchingSet, while an **ingp** returns the matchingSet and removes all of the tuples in the set from their respective tuple spaces. Their formal definitions are identical for the definitions of **rdg** and **ing**, except for the wrapping of the matchExists guards in the if/else clause.

4.3 Consistency Concerns

All operations discussed thus far act over the view atomically. This requires a transaction over all participants in the view. As the number of participants increases, this can become costly. From a different perspective, the previously discussed operations come with strict guarantees—if a matching tuple (or tuples) exists in the view, it (or they) will be returned. To more efficiently accommodate applications that do not require these strong guarantees, we introduce scattered probes that provide a weaker consistency because they are allowed to miss a matching tuple in the view. Scattered probes provide a best-effort solution and return ϵ (or an empty set) if they do not find a matching tuple. Several different implementations of scattered probes might apply in different application situations. The general intuition behind the operations, however, is a simple polling of the agents contributing to the view one at a time. EgoSpaces keeps track of which agents have been polled, and if it has covered all contributing agents without finding a matching tuple, the operation returns ϵ (or an empty set). To define these operations more formally, we first define a helper macro that builds the set of agents contributing to the view:

$$\mathsf{contrib}(\nu) \triangleq \langle \mathsf{set}\ \alpha : \langle \exists\, \theta : \theta \in \nu \wedge \theta \in \alpha.T \rangle :: \alpha \rangle.$$

We must also provide matchExists and matchingSet functions constrained to specific agents. For brevity, we omit the formal definitions of these functions; they closely resemble their counterparts that operate over the entire view.

We refer to the single scattered probe operations as **rdsp** and **insp**. The following function shows the definition of **rdsp**, in which the operation checks each contributing agent for a match, and, if all agents have been checked without finding a match, the operation returns ϵ:

$$\boxed{t := \textbf{rdsp}(\nu, p)}$$
$$A := \emptyset;\ t := \epsilon;\ \textbf{while } \mathsf{contrib}(\nu) - A \neq \emptyset \textbf{ do } \alpha := \alpha'.(\alpha' \in (\mathsf{contrib}(\nu) - A)); \langle \textbf{if } \mathsf{matchExists}(\alpha, \nu, p) \textbf{ then } t := t'.(t' \in \mathsf{matchingSet}(\alpha, \nu, p)); \textbf{break fi} \rangle; A := A \cup \{\alpha\} \textbf{ od}.$$

The definition of **insp** is identical to that of **rdsp**, but removes the tuple that it returns.

The aggregate scattered probe operations, **rdgsp** and **ingsp**, build on the matchingSet function. They also poll the contributing agents one at a time, copying or removing tuples as they go, and building a set of tuples to return:

$$\boxed{tset := \textbf{rdgsp}(\nu, p)}$$
$$A := \emptyset;\ tset := \emptyset;\ \textbf{while } \mathsf{contrib}(\nu) - A \neq \emptyset \textbf{ do } \alpha := \alpha'.(\alpha' \in \mathsf{contrib}(\nu)); \langle \textbf{if } \mathsf{matchExists}(\alpha, \nu, p) \textbf{ then } tset := tset \cup \mathsf{matchingSet}(\alpha, \nu, p) \textbf{ fi} \rangle; A := A \cup \{\alpha\} \textbf{ od}.$$

The definition of **ingsp** is identical to that of **rdgsp**, but removes the set of tuples it returns.

4.4 Reactive Programming

EgoSpaces provides reactive programming abstractions by allowing agents to adapt their behavior in response to the presence of tuples matching a provided pattern. Similar

[4]The || notation indicates that the quantified statements execute simultaneously. That is, all of the statements satisfying the conditions are executed in a single atomic step.

notions of reactive programming for tuple space based middleware for mobility have proven useful in other systems including LIME [14] and MARS [5]. In EgoSpaces, a reaction associates a tuple in a view with a sequence of program statements which can include both plain code and non-blocking tuple space operations:

$$\rho = \textbf{reaction}(p) \textbf{ in mode } \textit{sched\_modality} \\ \textbf{begin } op_1, op_2, \ldots \textbf{end}.$$

where $op_1, op_2, \ldots$ is the sequence of tuple space operations and code fragments to execute when the reaction fires.

Agents can enable and disable reactions over views using:

$$\textbf{enable } \rho \textbf{ over } \nu \\ \textbf{disable } \rho \textbf{ over } \nu.$$

As indicated by the *sched_modality*, behaviors can have a high priority (a *sched_modality* of "eager") or a lower priority (a *sched_modality* of "lazy"). An eager modality guarantees that the execution of the reaction's callback occurs immediately upon the insertion (or removal) of a tuple matching pattern p in the view. Only other eager operations can preempt an eager reaction. A lazy modality brings a much weaker guarantee—eventual firing of the reaction is guaranteed if the tuple remains in the view long enough. Other operations are allowed in the meantime, and it is therefore possible that the tuple is removed by one of these operations.

The execution of a reaction always removes the tuple that triggered it. Therefore, if a single tuple can trigger multiple reactions, only one will fire; the reaction that fires is chosen nondeterministically. When a reaction fires, the reference agent is notified, and the callback is scheduled for execution.

The execution of the statements in the reaction's callback occurs as a single atomic step. If the statements **out** tuples, all the tuples are placed in the tuple space at the same time at the completion of the callback execution. These tuples can then trigger other reactions registered on the same view or different views. However the execution of these reactions, even in the eager modality, does not force a distributed transaction, since they are independent atomic actions. Hosts locked for the execution of one reaction callback do not remain locked after its completion. Instead, subsequent callbacks lock only hosts necessary for their execution, and other hosts are free to continue processing.

As with all operations on views, registering reactions on a view affects the evaluation of the contributing agents' access controls. When specifying a view, the reference agent must indicate if it intends to register reactions on the view.

4.5 Active Views

In addition to regular tuple space operations and reactive programming, our evaluation of current models led us to develop active views. EgoSpaces allows applications to attach a variety of behaviors to particular views, including the ability to migrate tuples matching a certain pattern from a remote tuple space to the agent's local tuple space, the ability to duplicate tuples in the view, and the ability to react to events generated by tuple space access operations. The notion of active views is extensible in that EgoSpaces allows application programmers to define individualized types of behaviors in addition to those provided by the system.

In general, behaviors take the following form:

$$\beta = \textbf{behavior } \texttt{act}(p) \textbf{ in mode } \textit{sched\_modality} \\ \textbf{begin } \textit{tuple modifiers } \textbf{end},$$

where `act` can be a migrate, duplicate, capture, or a user-defined behavior; p is the pattern that triggers the behavior; and *sched_modality* can be eager or lazy. The *tuple modifiers* allow the reference agent to change local tuples resulting from the behavior. These modifications can insert or remove fields in the tuple. The necessity for the ability to perform such modifications will become especially apparent in the discussion of automatic data duplication.

An agent can enable and disable behaviors over views in a manner identical to that for reactions. Again, behaviors associated with a view affect the access controls of the view. EgoSpaces treats behaviors in the same way as any other operation—when specifying a view, the reference agent must indicate what types of behaviors it might enable over the view. The access control functions of the contributing agents account for this information when determining which tuples to contribute to the view.

4.5.1 Transparent Data Migration

Data migration allows a mobile agent to gather data on which it wishes to operate. This type of behavior can prove very useful in the ad hoc mobile environment because agents come and go. By collecting certain tuples while it is connected to other agents, an agent allows itself to operate on that data after the previous owner moves outside the scope of the view. Migration ensures that only a single instance of the tuple exists in the tuple space.

EgoSpaces provides this data migration as a pull operation. More specifically, the agent to which the tuple moves indicates (via a pattern) which tuples it wants migrated to it. As new tuples appear in the view, whether by a tuple insertion operation or because a new agent contributes to the view, EgoSpaces evaluates each of the new tuples with the migration pattern and moves those that match. The tuples are removed from their current tuple space and placed in the reference agent's local tuple space without changing the tuple id. After migration, the migrated tuples still appear in the same view. Because the current owner's access control function was evaluated before the tuple was allowed to appear in the view, the current owner has implicitly granted permission for the migration to occur.

4.5.2 Automatic Duplication

Automatic duplication allows a reference agent to cache tuples from a view without affecting the originals in the tuple space. The reference agent can then use these copies after the original tuple disappears from the view, whether because the owner agent moved outside the view's reach or because another operation removed the original. The key difference between tuple duplication and tuple migration is that, in duplication, the local copies of the tuple are only copies, and removing the copy does not affect the original.

Tuple duplication occurs when a matching tuple is inserted in the view or when an agent with a matching tuple moves into the view. A contributing agent moving out of the view and then returning may cause reduplication of tuples. Managing these reduplicates is the responsibility of the application performing the duplication.

When a tuple matching the duplication pattern appears in the view, EgoSpaces automatically creates a copy of it. The duplicate tuple will likely match the specification of the view to which the duplicate behavior is attached. The programmer must prevent an infinite reduplication of these tuples. The ability to associate tuple modifiers to a duplication behavior provides an easy way to accomplish this.

4.5.3 Event Capture

Capturing events allows an agent to adapt its behavior in response to operations performed by other agents on the view. EgoSpaces allows agents to receive events generated when agents insert tuples into the tuple space and remove tuples from the tuple space. When an event occurs in the view on a tuple matching the event behavior's pattern, EgoSpaces creates a special event tuple and places it in the agent's local tuple space.

5. IMPLEMENTATION

An initial prototype of the EgoSpaces middleware is complete, and we have begun development of the applications that spurred these investigations. The prototype is built on top of another newly developed middleware for coordination among mobile components, LimeLite. Because this system also utilizes tuple space based coordination, the data access operations described in the previous section take advantage of the LimeLite primitives for their implementation. LimeLite handles all network communication necessary for data sharing. The prototype includes host, agent, and data constraints, but due to the model of the underlying system, implementation of network constraints is not possible in the current prototype. However, protocols for supporting network constraints have been explored in [15], and an implementation of the work described is complete. Future work on the development of the EgoSpaces middleware will use these protocols to fill in the provision of true asymmetric behavior and allow for an empirical evaluation of the performance of applications. Additionally, further development of reactive and active views will add functionality to the prototype.

On top of this prototype implementation, we have built our first EgoSpaces application. This application is an adaptation of the `RoamingJigsaw` built for the Lime middleware [14]. In this application, a group of players cooperate to reconstruct a puzzle from its pieces. One player initializes the puzzle by loading an image. In the game's implementation, puzzle pieces are represented by tuples in the tuple space. Initially, all of the puzzle piece tuples are located in the local tuple space of the agent initializing the puzzle. The agent can define views over the tuple space that determine which puzzle pieces are displayed at a given time. In the game's implementation, the agent initially starts with the maximal view, i.e., the view that 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. An agent can select a piece by clicking on it. When the agent does so, the tuple corresponding to the puzzle piece is removed from the tuple space and placed in the selecting agent's local tuple 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 puzzle piece tuples move outside of a given agent's view, they are no longer displayed on the screen. At any given time, therefore, an agent can work only with the pieces currently available in its view.

The puzzle application also allows users to define new views. For example, a common way to assemble a puzzle is to start with the edge pieces. Therefore, the puzzle player can define a view that contains only edge pieces. As long as this is the view currently displayed, the player will see only available edge pieces, and all interior pieces are hidden. Figure 2 shows such an example. The player on top has defined a view containing only edge pieces. The player on bottom is working in the default view and sees all the pieces. Changes made by the player on top are displayed to the player on bottom, but the reverse is not true. This is because the changes the player on bottom has made affect only interior pieces, which are not included in the view defined by the player on top.

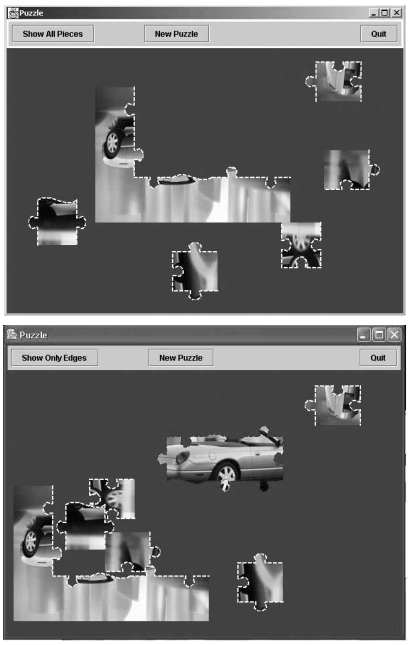

Figure 2: Two puzzle participants with different views

Puzzle players may find many different view definitions useful. For example, if player agents have an idle status, a player might define a view that contains only puzzle pieces owned by idle players. If these players are not currently working on assembling the puzzle, it will not interrupt them for another player to take control of their pieces and assemble them. As another example, if a player is facing a hole of a certain shape, he might want to 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.

While serving as an interesting demonstration, the puzzle application is a good example of an application that benefits from the transparent sharing available in EgoSpaces. Other applications that involve collaborative work by distributed parties can be implemented in similar ways.

6. DISCUSSIONS

The development of this model was driven by our efforts to build applications for vehicle to vehicle coordination on the highway. Because a network of transitively connected automobiles can grow very large, even to include an entire interstate, the desirability for an application to restrict its operating context to some reasonable subnet surrounding itself became quickly apparent. The asymmetric abstractions of the EgoSpaces view concept allow exactly this.

The original Linda model is a special case in EgoSpaces where both hosts and agents are stationary and their transiently shared tuple spaces form a single globally persistent

tuple space. Every agent defines a single view encompassing every tuple, and operates on this view using only the Linda primitives that carry over into EgoSpaces. The reduction of EgoSpaces to Linda is simplified by the fact that, with the exception of the Linda `eval` primitive, all of the primitives available in Linda are available in EgoSpaces.

In LIME agents define multiple named tuple spaces which are transiently shared based on their names. To accomplish this in EgoSpaces, an agent creates a view corresponding to a named tuple space. When an agent places a tuple in the tuple space, it tags the tuple with the view (tuple space) name. The view specifications are formed such that a view contains every tuple tagged with the corresponding name.

EgoSpaces can also express event based models like JEDI [8] and SIENA [6]. These models take advantage of asynchronous communication through the exchange of event notifications. Components receive event notifications by subscribing for particular events. EgoSpaces can achieve this behavior by attaching the event behavior to a view. Event notifications can be wrapped in tuples and placed in the tuple space. Event notification tuples matching the pattern of a registration will fire the corresponding callback.

7. CONCLUSIONS

EgoSpaces introduces a novel coordination model that focuses on the context of a particular component in a mobile ad hoc network. The system allows the mobile agent to structure its context at a very fine level of granularity. In doing this, however, EgoSpaces achieves an unprecedented level of flexibility, both in terms of the agent's specification of its operating context and in the provision of mechanisms for operating on this context. EgoSpaces promises to simplify programming by providing a flexible infrastructure on which developers build applications for ad hoc mobile environments. The model results directly from our reexamination the applicability of standard models in the high density highway environment. EgoSpaces introduces an asymmetric style of coordination that gives each individual agent direct control over the size and scope of the data it accesses, an approach that is essential to accommodating programming for very large and dense ad hoc networks.

ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation under Grant No. CCR-9970939. 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 National Science Foundation. The authors wish to thank Amy Murphy and Gian Pietro Picco for the lively intellectual discussions that led to the development of this model and Tom Elgin for the prototype implementation.

8. REFERENCES

[1] G. Abowd, C. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: A mobile context-aware tour guide. *ACM Wireless Networks*, 3:421–433, 1997.

[2] G. Andrews. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Company, 1991.

[3] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13(2–3):133–180, 1990.

[4] P. J. Brown. The stick-e document: A framework for creating context-aware applications. In *Proc. of EP'96*, pages 259–272, 1996.

[5] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *Internet Computing*, 4(4):26–35, 2000.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area even notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, 2001.

[7] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou. Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In *Proc. of MobiCom*, pages 20–31. ACM Press, 2000.

[8] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. on Software Engineering*, 27(9):827–850, 2001.

[9] N. Davies, A. Friday, S. Wade, and G. Blair. L^{2}imbo: A distributed systems platform for mobile computing. *ACM Mobile Networks and Applications (MONET), Special Issue on Protocols and Software Paradigms of Mobile Networks*, 3(2):143–156, 1998.

[10] D. Gelernter. Generative communication in Linda. *ACM Trans. on Programming Languages and Systems*, 7(1):80–112, 1985.

[11] A. Harter and A. Hopper. A distributed location system for the active office. *IEEE Networks*, 8(1):62–70, 1994.

[12] J. Hong and J. Landay. An infrastructure approach to context-aware computing. *Human Computer Interaction*, 16, 2001.

[13] IBM. T Spaces. http://www.almaden.ibm.com/cs/TSpaces/, 2001.

[14] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proc. of the 21st Int'l. Conf. on Distributed Computing Systems*, pages 524–533, 2001.

[15] G.-C. Roman, C. Julien, and Q. Huang. Network abstractions for context-aware mobile computing. In *Proc. of the 24th Int'l. Conf. on Software Engineering*, pages 363–373, May 2002.

[16] N. Ryan, J. Pascoe, and D. Morse. Fieldnote: A handhelod information system for the field. In *1st Int'l. Workshop on TeloGeoProcessing*, 1999.

[17] D. Salber, A. Dey, and G. Abowd. The Context Toolkit: Aiding the development of context-enabled applications. In *Proc. of CHI'99*, pages 434–441, 1999.

[18] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, 1994.

[19] Sun. Javaspaces. http://www.sun.com/jini/specs/jini1.1html/js-title.html, 2001.

[20] R. Want et al. An overview of the PARCTab ubiquitous computing environment. *IEEE Personal Communications*, 2(6):28–33, 1995.