EgoSpaces: Facilitating Rapid Development of Context-Aware Mobile Applications

Christine Julien
Gruia-Catalin Roman

TR-UTEDGE-2005-004

EgoSpaces: Facilitating Rapid Development of Context-Aware Mobile Applications

Christine Julien and Gruia-Catalin Roman

Abstract— **Today's mobile applications require constant adaptation to their changing environments, or contexts. Technological advancements have increased the pervasiveness of mobile computing devices such as laptops, handhelds, and embedded sensors. The sheer amount of context information available 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 that provides context information to applications in an abstract form. In this paper, we demonstrate the feasibility of such a middleware that allows programmers to focus on high-level interactions among programs and to employ declarative abstract context specifications in settings that exhibit 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 it. Finally, we provide examples demonstrating the infrastructure's ability to support differing tasks from a wide variety of application domains.**

Index Terms— **context-awareness, middleware, mobile ad hoc networks, programming abstraction**

I. Introduction

With the increasing popularity of mobile computing devices, users find themselves living and interacting in environments characterized by the ability to coordinate with a variety of wirelessly networked resources. Imagine a network that forms on a highway among vehicles communicating directly with one another. Such *ad hoc networks* form opportunistically and change rapidly in response to the movement of the devices, or *mobile hosts*, resulting in a network topology that is both dynamic and unpredictable. Because communicating parties may be constantly moving, their interactions are inherently transient. Routing protocols have been devised to create and maintain communication pathways among mobile hosts even as the network topology changes but do not sufficiently abstract communication to the level of applications' operations.

Consider an automobile network. An individual driver might first want to keep track of all cars likely to collide with him. If another car comes too close, a light warns the driver, and he can attempt to avoid the collision. The driver might also monitor traffic conditions for his specified route. As a second example, imagine a building with a fixed infrastructure of sensors that provide information about the building's structural integrity, occupants' movements, etc. Engineers and inspectors carry PDAs that interact with the sensors. As an engineer moves, he wishes to see structural information determined by his task or location. He may also want to respond to events, *e.g.*, the arrival of an inspector. As support for pervasive computing devices builds, the possibility for such applications in various domains abounds.

Christine Julien is with the Department of Electrical and Computer Engineering at the University of Texas at Austin, 1 University Station, C5000, Austin, TX 78712, Email: c.julien@mail.utexas.edu

Gruia-Catalin Roman is with the Department of Computer Science and Engineering at Washington University in Saint Louis, Campus Box 1045, One Brookings Drive, St. Louis, MO 63130, Email: roman@wustl.edu

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

Mobile application programming difficulties can be generalized to the need to manage large amounts of distributed and transiently available context data. This challenge motivated us to hide the details of mobility, distribution, and transient connectivity. The resulting middleware, EgoSpaces, allows an individual application to limit the portion of the context it interacts with. An application may define different contexts that reflect diverse concurrent and changing needs and which encompass data from multiple sources. EgoSpaces manages this information for the application, relieving the developer from having to handle network connections and disconnections common in mobile environments.

This work represents a significant step in creating a context-aware computing infrastructure for simplifying adaptive mobile application development. Specifically, our contributions include: 1) a redefinition of context-awareness in mobile environments, 2) the elucidation of a conceptual model amenable to dynamic applications and tailored to a novice programmer's capabilities, and 3) the implementation of a middleware that enables simplified context-aware application development.

In this paper, we first examine the state of the art in context-aware computing. Section III uses this foundation to develop a novel model of emerging context-aware mobile applications. The following section demonstrates how applications operate within that model to exchange and interact with dynamic data. In Section V, we use this mode to develop a middleware specifically designed to reduce the software engineering burden in mobile ad hoc networks. Section VI provides examples from varying application domains, while Section VII experimentally evaluates the middleware's performance through simulation. Related work and conclusions appear in Sections VIII and IX, respectively.

II. Context-Aware Computing

In context-aware computing, the applications' behavior is determined by the circumstances in which they find themselves. The environment has a powerful impact on an application component either because the latter needs to adapt to changing external conditions or because it relies on resources whose availability continuously changes.

A. Collecting and Adapting to Context

Context-aware computing became prevalent with the emergence of mobile devices. Active Badge [46] uses infrared communication between users' badges and sensors placed in a building to forward telephone calls. PARCTab [47] also enables adaptive applications which, for example, can attach a file directory to a room for use as a blackboard. More recent work [20] in ubiquitous computing uses CORBA and operates over a wired network that supports localization and communication. These systems require constant maintenance and do not address issues inherent in ad hoc networks, including the need to scale to large and unpredictable networks.

Context-aware tour guides [1], [11] present information about the user's current environment. Fieldwork tools [35] automatically attach context information (*e.g.*, time) to researchers' field notes. Memory aids [38] record notes about the current context that might later be useful to the user. These applications collect their own context information and focus on a specific context types, while mobile applications share characteristics that set them apart. Specifically, mobile hosts do not have *a priori* knowledge of the parties with which they interact. These new applications instead rely on opportunistic interactions. For example, an application for vehicles on a highway interacts with other cars locally to collect traffic information. A particular driver has no advance knowledge about which cars will provide the traffic information.

Generalized software built to support context-aware computing in mobile environments has also become a focus of much research [18], [21], [43]. Among the best known systems is the Context Toolkit [43], which provides abstractions for representing context through widgets that collect low-level sensor information and aggregate it to be more easily handled by application developers. While these approaches offer much needed building blocks for constructing applications, they do not address an application's need to dynamically discover and operate over a constantly changing context.

B. A Novel Notion of Context-Awareness

While the above approaches demonstrate that context-aware computing provides abstractions useful in supporting mobile applications, they do not directly address the distinguishing characteristics of ad hoc networks and their desired applications. Specifically, we build on the above context definitions but take an application-level approach to adaptation, focusing on how applications specify and use context elements:

- Context should be generalized so that applications interact with different context types (*e.g.*, location, bandwidth, etc.) in a similar manner.
- Different applications require contexts tailored to their individual and changing needs.
- An application's context includes information collected from a distributed network, which must be specified without significant *a priori* knowledge.
- Due to the large-scale environment, applications require decentralized context interaction.
- High-level abstractions ease the programming burden.

In this paper, we use this new definition to design and develop a middleware infrastructure that supports rapid context-aware application development in mobile ad hoc networks.

III. A Conceptual Model of Context-Aware Applications

Armed with this new perspective on context-awareness, we developed a conceptual model to describe mobile application behavior and to provide support for their rapid development.

A. Computational Model

We assume a computing model in which hosts move in physical space, and applications are structured as a community of mobile software agents that can migrate among hosts. An agent is the unit of modularity and mobility, while a host is a container that is characterized by, among other things, its location in physical space. Communication among agents and agent migration can take place whenever the hosts involved can communicate. A closed set of connected hosts forms an ad hoc network.

Since context is relative to a particular application on a particular host, we use the term *reference agent* to denote the agent whose context we are considering, and we will refer to the host on which this agent is located as the *reference host*. In principle, an agent's context consists of all the information available in the network. Such broad access to information is costly to implement and undesirable in large networks. Consider the application in which a driver collects traffic information. Automobiles may be transitively connected for hundreds of miles, but only local traffic information is of interest to the driver. For these reasons, we structure the context in terms of fine-grained units called *views*.

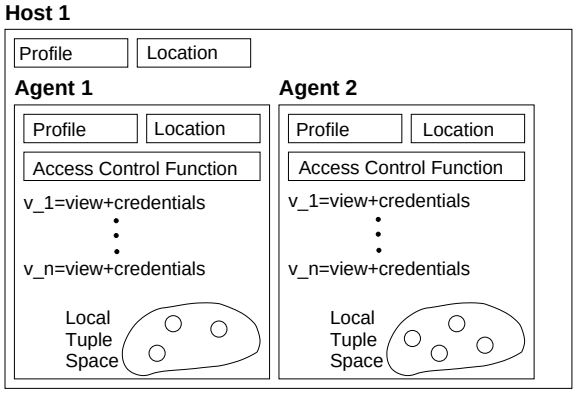


Fig. 1. The computational model

Fig. 1 shows our computational model whose components are discussed in detail throughout this section. A host has a physical location and a profile describing its properties. An agent has a profile and a logical location, the host on which it is running. Each agent can define multiple views by providing a specification, whose construction is described in more detail in Section III-C. Finally, every agent stores its data in a local tuple space. While all these aspects are essential to our computational model, successful asymmetric coordination through the view concept is our middleware's cornerstone. We next explore this concept in more detail before continuing with the access operations in Section IV.

B. Data Representation

Our model intentionally does not separate the notions of *data* and *context*. That is, given our redefinition of context from Section II, we envision that any data available in the network has the potential to impact an agent's behavior and is therefore context. The manner in which an agent perceives data has ramifications on the ease of programming and the efficiency of operations. Therefore we explicitly separate the specific data items an application can access from the manner in which they are presented to the application. That is, we assume a single data representation as a basis for coordination. Other interaction forms can be swapped in for our choice; the investigation of such *context-sensitive data structures* [36], [37] is outside the scope of this paper.

In our model, applications perceive the network as an underlying database of tuples. Tuple space representations based on Linda [16] enjoy a great deal of popularity due to the use of content-based data access. Several mobile computing systems have found success using shared tuple spaces [7], [33]. We support transient tuple space sharing, combine it with a flexible tuple representation, and allow an agent to use a declarative view specification to indicate with which other components it wants to share data.

To support tuple spaces, we developed ELIGHTS, in which 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 each *value*. In any tuple, the field names must be unique. The name allows us to relax the ordering restrictions seen in traditional tuples. Fundamentally, users access tuple spaces by matching a pattern against a tuple's contents. An ELIGHTS pattern has the form:

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

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

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

In the three-part notation $\langle \textbf{op}\ quantified\_vars : range :: exp \rangle$, the variables from *quantified_vars* take on all values permitted by *range*. If *range* is missing, the domain of the variables is restricted by context. Each instantiation of the variables is substituted in *exp*, producing a multiset of values to which **op** is applied, yielding the value of the three-part expression. If no instantiation satisfies *range*, the value of the expression is the identity element for **op**, *e.g.*, *true* when **op** is $\forall$ or zero if **op** is "+." For each constraint in the pattern, the tuple must have a field with the same name, the same type or a derived type, and a value that satisfies the constraint. The function requires that each *constraint* is satisfied, but it does not require every *field* to be constrained. In enabling coordination among distributed application components, it is assumed that some meta-knowledge about the type and representation of the context is shared among the components. This assumption enables the model to be more flexible in allowing any data types to be stored as context but makes application development slightly more complex by leaving the definition of a naming scheme to the developer.

C. The View Concept

A *view* is a projection of all data available to the reference agent. An agent can define multiple views (which can be redefined over time as needs change). In the remainder of this section, we first define the view and the components of its specification and behavior informally. We end with a formalization of the view and its contents.

Declarative View Specifications. The view concept is egocentric in that every view is defined with respect to a reference agent and its needs for resources from its environment. An agent requests a view by providing a *declarative specification* which controls the scope of the view (a larger or smaller network neighborhood) 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 on the data. For example, an automobile's collision avoidance agent might declare the following view:

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

Fig. 2 shows an evaluation of the declarative view specification. The figure shows cars on a highway; the arrows indicate their approximate movement patterns. The "X" represents the reference agent. To simplify the picture, we assume only a single agent per car. In the picture on the left, the reference agent provides a restriction of the cars that participate in the view. The center picture shows how data items (circles in the picture) map to cars. Because the reference agent is interested only in location data (black circles in the picture on the right), the actual view contains only these data items.

Network Constraints. We extend the availability of context information beyond a host's immediate scope, *i.e.*, a host can gather context from a subset of the entire ad hoc network.

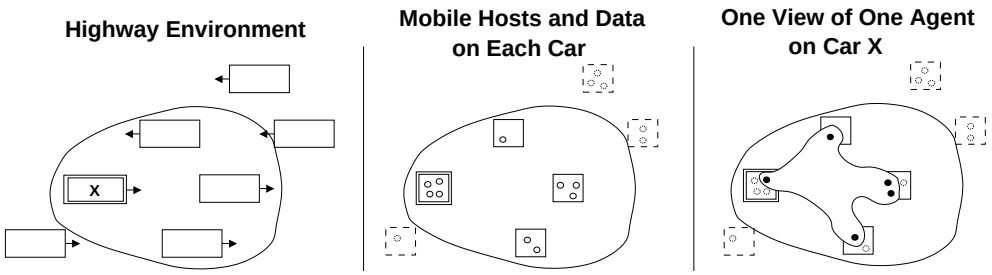


Fig. 2. View used by a collision warning agent on car X

Doing so requires an abstraction of the network topology and its properties. After specifying some constraints, including an individualized definition of distance, an application desires a qualifying list of acquaintances. That is,

> *Given a host α and a positive bound 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.*

To provide such an abstraction of the ad hoc network, we use the Source Initiated Context Construction (SICC) protocol and its network abstraction [23], [27], [39]. SICC provides an abstraction of the network as a tree that contains only those hosts that are within a specified *distance* from the reference host, where the distance can be calculated via an application-specified metric. Using this abstraction, applications delegate responsibility for low-level communication and focus instead on application-level adaptation and coordination. To use SICC, 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.

The computation results in a tree rooted at the reference node and spanning a subnet of the network. The path to every node satisfies the restrictions imposed by the cost function and bound, and the tree is maintained as long as needed. As hosts move, the properties defining the tree change, thus changing both the contents and the topology of the tree. Building on the automobile example from above, the network constraints portion of the view specification would restrict context hosts to only those within 100 meters.

Host Constraints. While the network constraints deal with physical properties, the host constraints handle logical properties. Examples include the host's id, the identity of the device's owner, or services the device provides. A host stores the properties in a *host profile*, which is a special private tuple where the fields of the host are attributes:

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

An example profile for a car might be:

$$\begin{aligned}\langle &(vehicle\_type,\ enumeration,\ car),\\ &(direction,\ string,\ NORTH),\\ &(speed,\ integer,\ 65)\rangle.\end{aligned}$$

Host constraints are a pattern over this profile. For example, the following constraints restrict the view to only vehicles moving in the same direction at nearly the same speed:

$$\begin{aligned}\langle &(direction,\ enumeration,\ =\ \mathsf{mydirection}),\\ &(speed,\ integer,\ <\ \mathsf{myspeed}+2),\\ &(speed,\ integer,\ >\ \mathsf{myspeed}-2)\rangle.\end{aligned}$$

The example constraint does not restrict the type of vehicle because that property does not interest the specifying host. This constraint also refers to three local variables (that start with "my," which refer to values stored in the specifying host's profile. In Section VI-B, we show a complete example in which such a specification is useful because it provides a relatively consistently connected set of hosts.

Agent Constraints. Every agent defines another profile containing agent properties. Providing constraints over agent profiles allows application agents to restrict the set of agents that contribute data to the view. Restricting operations to one type of agent or another increases the efficiency of coordination by decreasing the number of parties involved. Revisiting the car example from above, the agent profiles would be defined similarly to the car's profile above, but with agent properties (*e.g.*, the profile may contain a service of type traffic or weather that the particular agent offers).

Data Constraints. In the same way that agent constraints restrict the agents contributing to the view, the data constraints restrict individual data items. 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" by inserting extra fields in the application's tuples that can be used in matching data constraints. With respect to the automobile example, the data constraints could select only location data no more than 30 seconds old.

Transparent View Maintenance. Applications use the above process to define views in a relatively static manner (although applications are allowed to redefine views and their constraints at any time). As hosts and agents move and the available data changes, the view is automatically updated. From the application's perspective, these changes are transparent and manifest themselves only in changes in the set of available data. Therefore, an agent can operate over a view without explicit regard for context dynamics. This update occurs only at a perceptual level; views and the data belonging to them are not actually calculated until or unless an application uses the view. The overhead of constructing and maintaining a view is incurred only when the application is

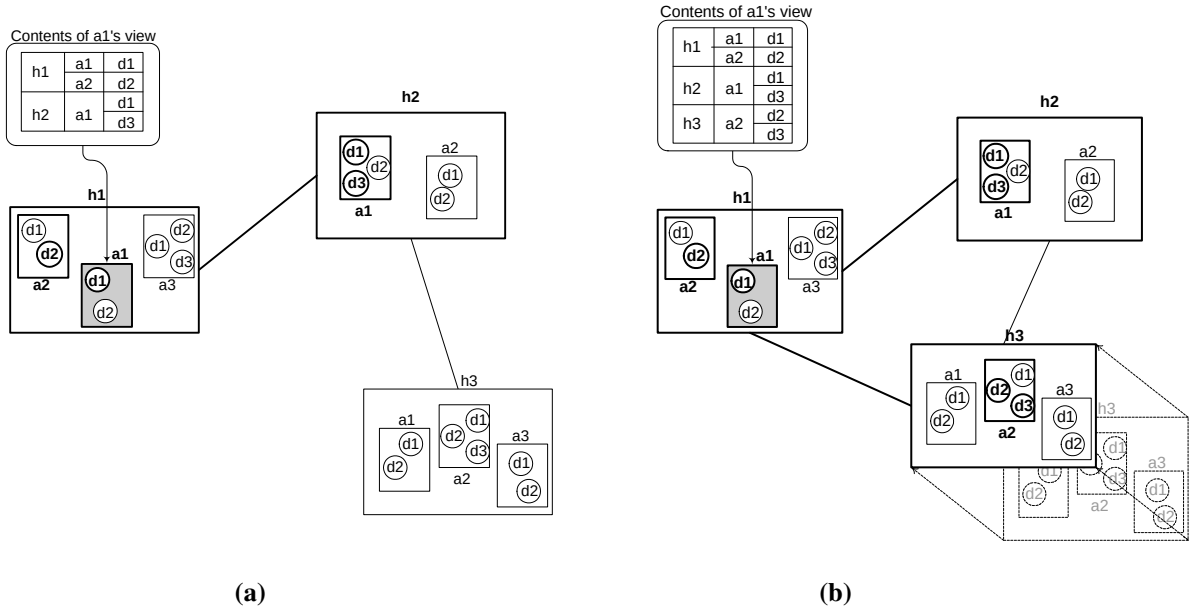


Fig. 3. View dynamics.

actively using the view, but the application benefits from the perception of a persistent data structure that reflects the view's current contents.

The dynamic nature of the view is illustrated in Fig. 3, where the depicted view of agent a1 changes as the distance between hosts h1 and h3 decreases. Hosts, agents, and data that contribute to the view have darkened borders. In (a), due to a1's specification, only h1 and h2 qualify to contribute agents to the view. Because of the restrictions on agent and data properties, only certain data items on certain agents on these hosts appear in the view. The balloon pointing to a1 shows a table of the view's contributors. In (b), when h3 moves closer to h1, it satisfies the view's constraints, and its qualifying agents can contribute qualifying data.

Formal View Definition. Given the four types of constraints, a view specification consists of three patterns (over data, agent profiles, and host profiles) and the network constraints (consisting of a link weight metric, a cost function, and the function's bound). Given these constraints, the view is informally defined as the set of all tuples that satisfy the data constraints, are owned by agents that satisfy the agent constraints, and are located on hosts that satisfy the host constraints. Finally, these hosts must lie within the boundaries defined by the network constraints.

Given a reference host r, we define η, the subnet of the network that satisfies the 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:

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

γ is a host, α is an agent, and θ is a tuple. $\alpha.T$ refers to the agent α's local tuple space. loc refers to an agent's host. This definition has been extended to allow agents to control access to the tuples they own. For brevity, details have been omitted from this paper, but the interested reader is pointed to [24] for further details. Throughout the remainder of the paper, we will refer to a view as ν.

IV. Interacting with Views

An agent interacts with the world by specifying views that are presented to the application as tuple spaces. This section overviews the operations allowed within the view concept.

A. Basic Operations

Basic tuple space operations can be divided into two categories: tuple generation that places new tuples in the agent's local tuple space and on-demand access operations that allow a reference agent to read and remove tuples in its views. These operations and descriptions of their behavior are shown in Table I; complete operational semantics can be found in [25].

B. Consistency Concerns

The above operations act over a view atomically, which requires a *transaction* over all view participants. In some applications (*e.g.*, those involving money), this transactional

Operation	Description
$\mathsf{out}(T, t)$	Places the tuple t in the agent's local tuple space (designated, for completeness, as T).
$t := \mathsf{rd}(\nu, p)$	Returns in t a copy of a tuple that satisfies the view ν's specification *and* the pattern p. If no such tuple exists in ν, the agent blocks until one does.
$t := \mathsf{in}(\nu, p)$	Returns in t a tuple that satisfies the view ν's specification *and* the pattern p, and also deletes the tuple returned. If no such tuple exists in ν, the agent blocks until one does.
$t := \mathsf{rdp}(\nu, p)$	Returns in t a copy of a tuple that satisfies the view ν's specification *and* the pattern p. If no such tuple exists in ν, ϵ (a null value) is returned.
$t := \mathsf{inp}(\nu, p)$	Returns in t a tuple that satisfies the view ν's specification *and* the pattern p, and also deletes the tuple returned. If no such tuple exists in ν, ϵ (a null value) is returned.
$tset := \mathsf{rdg}(\nu, p)$	Returns in $tset$ the set of copies of all tuples that satisfy the view ν's specification *and* the pattern p. If no such tuple exists in ν, the agent blocks until one does.
$tset := \mathsf{ing}(\nu, p)$	Returns in $tset$ the set of all tuples that satisfy the view ν's specification *and* the pattern p, and deletes the tuples returned. If no such tuple exists in ν, the agent blocks until one does.
$tset := \mathsf{rdgp}(\nu, p)$	Returns in $tset$ the set of copies of all tuples that satisfy the view ν's specification *and* the pattern p. If no such tuple exists in ν, ϵ (a null value) is returned.
$tset := \mathsf{ingp}(\nu, p)$	Returns in $tset$ the set of all tuples that satisfy the view ν's specification *and* the pattern p, and also deletes the tuples returned. If no such tuple exists in ν, ϵ (a null value) is returned.

TABLE I
BASIC OPERATIONS ON VIEWS

Operation	Description
$t := \mathsf{rdsp}(\nu, p)$	Returns in t a copy of a tuple that satisfies the view ν's specification *and* the pattern p. If no such tuple can easily be found in ν, ϵ (a null value) is returned.
$t := \mathsf{insp}(\nu, p)$	Returns in t a tuple that satisfies the view ν's specification *and* the pattern p, and also deletes the tuple returned. If no such tuple can easily be found in ν, ϵ is returned.
$tset := \mathsf{rdgsp}(\nu, p)$	Returns in $tset$ the set of copies of all tuples that satisfy the view ν's specification *and* the pattern p. If no such tuple can easily be found in ν, ϵ (a null value) is returned.
$tset := \mathsf{ingsp}(\nu, p)$	Returns in $tset$ all tuples that satisfy the view ν's specification *and* the pattern p, and also deletes the tuples returned. If no such tuple can easily be found in ν, ϵ is returned.

TABLE II
SCATTERED PROBING OPERATIONS ON VIEWS

behavior is required. 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. In certain conditions, we can provide such transactional guarantees, even in the face of mobility. Section VII provides some performance characterizations that demonstrate how well this assumption holds. As described in more detail in Section V, transactional semantics can be provided by relying on a second protocol that defines legal links for "safe" communication. However, as the number of participants increases, this can become costly and difficult. To more efficiently accommodate applications that do not require these strong guarantees, we introduce *scattered probes* that provide a best-effort alternative.

Different implementations of scattered probes apply in different scenarios. The general intuition is a simple one-at-a-time polling of agents contributing to a view. The operation 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). Table II shows these operations, their operational semantics can be found in [25].

C. Active Views

Using the previous constructs, to wait for a piece of data, an agent must either block or poll, which prevents it from performing other work. To provide expressive, application-centered constructs, we augment the view model to integrate *transactions*, *reactions*, and generic *behaviors* with views. These new operations are summarized in Table III.

Transactions. Performing several operations sequentially is not atomic because other operations can interleave. For example, if an agent performs a successful rdp operation and immediately attempts to in the same tuple, it may be unsuccessful if another agent has, in the meantime, removed the tuple. An application may want a sequence of operations to be atomic with respect to other operations on the involved views. To support this, we introduce *transactions* that must explicitly specify the views over which they will operate, and they are restricted to acting only on those views. In the collision detection example, it may be imperative that if a car is present, its current location is reflected in the tuple space. In this situation, the car could use a transaction to remove an old location and replace it with a new location as a single atomic action (so that it never appears to any other car that the location reading was not available).

Reactions. Coordination systems [7], [33] and publish-subscribe systems [8], [14] have found the ability to react to data essential for adaptation. In the highway example, an application may react to the presence of a location tuple that is "too close" as defined by application level properties.

Operation	Description
$T =$ **transaction** **over** $v_1, v_2, \ldots$ **begin** $op_1, op_2, \ldots$ **end**	Performs the specified operations ($op_1, op_2, \ldots$) as a transaction over the specified views ($v_1, v_2, \ldots$). Any attempt by the operations inside the transaction to use views not included in the list results in an exception.
$\rho =$ **react to** p [**remove**] [**and** out(tuple_modifiers(τ))]	Basic reaction triggered by the presence of a tuple matching the pattern p within the view on which the reaction is registered (not shown). A basic reaction can remove (delete) the basic trigger tuple, and/or output a permutation of the trigger using tuple_modifiers.
$\rho =$ **react to** p [**remove**] [**and** out(tuple_modifiers(τ))] **extended by** $T(\tau)$	Extended reaction triggered by the presence of a tuple matching the pattern p within the view on which the reaction is registered (not shown). In addition to optionally removing the trigger and outputting a modified tuple, an extended reaction can include a transaction that is executed in the same atomic step as the triggering. To ensure the behavior's atomicity, the trigger tuple must be local.
$\rho =$ **react to** p [**remove**] [**and** out(tuple_modifiers(τ))] **followed by** $T(\tau)$	Followed reaction triggered by the presence of a tuple matching the pattern p within the view on which the reaction is registered (not shown). In addition to optionally removing the trigger and outputting a modified tuple, a followed reaction can include a transaction that executes after the triggering (but not in the same atomic step).
$\mathcal{M} =$ migrate p [tuple_modifiers(τ)]	Migration moves any tuple matching the pattern p that appears in the view on which the behavior is registered. The trigger is moved to the agent that created the behavior, altered according to the tuple_modifiers.
$\mathcal{D} =$ duplicate p [tuple_modifiers(τ)]	Duplication creates a copy of any tuple matching the pattern p that appears in the view on which the behavior is registered. The copy of the trigger is altered according to the tuple_modifiers and placed in the specifying agent's local space.
$\mathcal{E} =$ event(p) **followed by** $T_e(\tau)$	Event registrations are triggered when a matching event occurs in the view on which the event behavior was registered. A single event tuple is generated for each registration. The triggering of an event registration is followed by the specified transaction.

TABLE III
ACTIVE VIEW CONSTRUCTS

In the view model, a *Basic Reaction* associates a pattern with actions to perform when a tuple in the view matches the pattern. We further augment reactions to allow them to execute a transaction in response to a trigger. An *Extended Reaction* couples the triggering and response as a single atomic action but, to ensure atomicity, requires that the trigger tuple is local. A *Followed Reaction* treats the triggering and the response as separate atomic actions, the implication being that the triggering tuple may not be available to the responding transaction. In all cases, each agent maintains a list of the reactions registered on it on behalf of other agents. If the agent leaves a view, the associated reactions are deregistered. If the agent returns, the reactions are reregistered *as new reactions*, which may cause them to fire again for data items that have already been reacted to. Details of the three types of reactions can be found in [26].

Behaviors. In our use of the basic model, we discovered that many applications create generic behaviors using the basic constructs. Capturing these behaviors as built-in programming constructs reduces the programming burden in common cases and provides powerful high-level abstractions that promote reuse and reduce programming errors. We have classified three such behaviors: automatic data duplication, data migration, and event generation. The use of these behaviors enables novice programmers to create applications that involve sophisticated, repetitive coordination activities with minimal added overhead.

A mobile agent may want to collect data without explicitly having to read each piece. When data consistency is important, a common solution is data replication and associated replica management, where copies of the data are kept consistent. This solution is impractical in ad hoc environments where agents carrying originals and duplicates meet unpredictably. Instead of attempting to resolve this issue, we avoid the excessive overhead of replica management by providing two alternatives: *data migration*, in which only one copy of a tuple persists, and *data duplication*, in which independent copies of data items are made. The application is left with the responsibility of managing consistency in these situations. Duplicated tuples may match the view specification and be infinitely duplicated. They may also appear in other agents' views. Applications deal with these concerns individually, *e.g.*, by tagging all duplicates and preventing duplication of tagged tuples.

All of the previously described constructs act over state. Many applications also benefit from reacting to events. Events include an agent's arrival, another agent's data access operations, etc. We introduce an event generation mechanism to our model (see Section V), represent events as special tuples, and register an agent's interest in an event via patterns. To allow multiple registrations for the same event yet prevent superfluous event generation, we raise events only when a matching registration exists. A unique event tuple is created for each specific registration, and each callback consumes the event tuple created for it.

V. EGOSPACES MIDDLEWARE

The programming constructs described above enable novice programmers to build complex applications. In this section, we describe EgoSpaces, the middleware that provides the programming abstractions. Fig. 4 shows the middleware's high-level architecture. Gray boxes represent components we assume to exist (message passing and the ad hoc physical network) or components the programmer provides (the ap-

plication). White boxes represent pieces of our architecture.

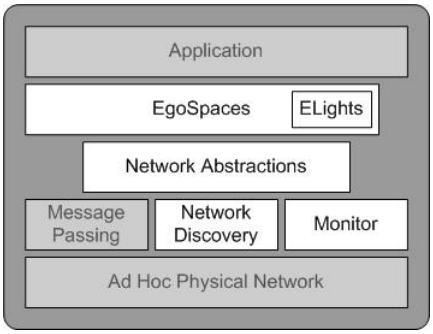


Fig. 4. The EgoSpaces system architecture

A. Supporting Packages

To build EgoSpaces, we implemented three support packages that provide lightweight implementations of necessary services.

1) Discovering Network Neighbors: In ad hoc networks, all hosts serve as routers. To distribute messages, a host must maintain up-to-date knowledge of its current neighbors. We utilize a discovery service with periodic beaconing parameterized with policies for neighbor addition and removal. The error associated with neighbor knowledge is directly dependent on the beaconing period. The impact of this error is explored in detail in Section VII.

2) Monitoring Environmental Conditions: We developed CONSUL [18], a general-purpose monitoring framework which maintains a registry of sensors available locally and on neighboring hosts (within one hop). An application tailors CONSUL to its needed capabilities. As an example, to add a location monitor, the application provides code that interacts with, for instance, a GPS device. In general, a monitor contains its current value (*e.g.*, the value of a GPS monitor might be represented by a variable of type `Location`) and allows an application to access the value or react to changes. The information EgoSpaces gathers from CONSUL is essential in enabling communication to adapt to the changing context.

3) Defining Network Metrics: To provide network constraints, we use the SICC protocol [27], [39] to construct a subnet of the ad hoc network based on network properties. As it processes queries in a distributed fashion, SICC uses local sensor information from CONSUL and the view's metric and bound to build a tree over the subnet of the network that contains exactly the hosts that satisfy the view's network constraints. When the application accesses the view, the system routes over this tree to service queries. The protocol also maintains the tree as hosts move and paths change. The protocol allows EgoSpaces to send messages to exactly the hosts in the context, *i.e.*, those hosts that contribute to the view.

B. Application Interaction with EgoSpaces

EgoSpaces reduces programming context-aware mobile applications to simple operations tailored to novice programmers' capabilities. An application developer extends the `Agent` base class, which allows access to view specification mechanics and communication capabilities.

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

Fig. 5. The API for the `Agent` class

1) Agent Extension: Fig. 5 shows the API of the abstract `Agent` class. An application's agent inherits two fields: the unique `AgentID` and the `AgentProfile`. An `AgentID` consists of the unique id of the host on which it was created coupled with a counter incremented by that host. Even if the agent moves within the network, it retains an id associated with the host where it was first created. An agent's profile fosters coordination by allowing other agents to include or exclude the agent from coordination based on its properties (via agent constraints). Initially, the profile contains two fields named "Agent ID" and "Host ID" that contain the `AgentID` and the id of the agent's host. An agent can add, remove, and modify properties in its profile (except for the `AgentID` and the `HostID`, which are controlled by the system).

In extending the `Agent` base class, an application agent receives two methods. The first registers the `Agent` with the local `EgoManager` which delegates responsibility for data management and communication. The second method, `out`, allows an agent to create tuples. When the agent is registered, these data items are available for coordination.

2) View Definition and Use: Once registered with the `EgoManager`, an agent can define views. The `View` API includes a constructor, data access operations, and the ability to enable behaviors. The constructor requires the agent to provide the four constraints that define a view. The network constraints are provided via a metric and bound as required by SICC. Because EgoSpaces represents profiles as tuples, the remaining constraints can be provided as patterns over tuples. Once a `View` is defined, the reference agent sees it as the set of data items that satisfy the restrictions and uses the constructs discussed in Section III to access data.

C. EgoSpaces Implementation

Agent Registration and Migration. When an agent is created, a data structure is initialized to hold any tuples the agent creates. 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 yet available for access by other agents. When the agent calls the `register` method, the EgoSpaces system registers the agent with the `EgoManager`, and the contents of the agent's local storage are placed in a host-level tuple space. During the transfer to the host-level tuple space, each tuple is annotated with the owning agent's id.

We use a single host-level tuple space to reduce the overhead of remote operations.

The registration mechanism described above reduces agent migration to a few simple steps. Upon migrating, an agent is deregistered from the current `EgoManager`. This moves the agent's tuples from the host-level tuple space to the agent's local storage. The agent's code and state are then 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 SICC to construct the subnet of hosts over which the view's operations are issued. The `EgoManager` only builds and maintains views when operations are issued to avoid unnecessary communication.

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`, which creates a dedicated operation thread for the request. From this point, the steps necessary to implement each operation depend on the operation's semantics.

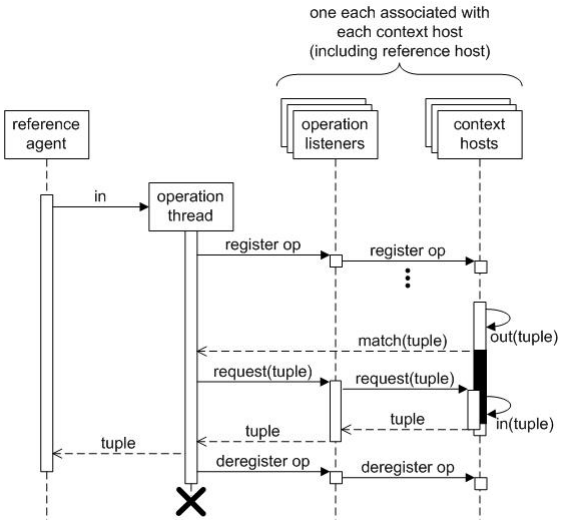


Fig. 6. Sequence diagram of an `in`

Atomic Blocking Operations. Fig. 6 shows a sequence diagram of an `in` operation. The calling thread blocks until the operation thread finds a match. The operation thread uses SICC to distribute a 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.

Two things can happen when the operation is registered. First, a tuple in the host's tuple space may immediately match (not shown). 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 query is deregistered (indicated as the blackened period in Fig. 6). A match may also be triggered by a new host with a matching tuple moving into the view. When the operation thread receives notification of a match, 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`; it chooses one nondeterministically. Once the operation is ready to return, the query is deregistered from all of the context hosts.

When a context host finds a match to a `rd`, it simply returns it 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 the first match, it issues an aggregate atomic probe to complete the operation.

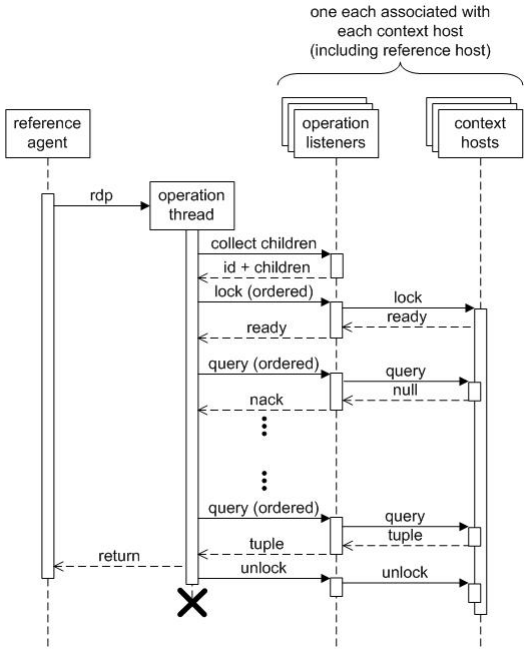


Fig. 7. Sequence diagram of a **rdp**

Atomic Probing Operations. The sequence diagram in Fig. 7 shows a `rdp`. When the reference agent issues its operation, the `EgoManager` spawns an operation thread; the reference agent waits for a response. The operation thread first collects the ids of the view's hosts using a SICC query. Every host in the context responds with its id and the ids of its children in the tree. The `EgoManager` on the reference agent's host uses this information to ensure that it hears from every contributor before continuing. At this point, the set of hosts for the operation is fixed. If new hosts move into the view, their addition is delayed until this operation completes. Once the operation thread has gathered the ids of all context hosts, it locks them in order of increasing id. Locking a tuple space prevents other threads from modifying the tuple space's contents; ordered locking prevents deadlock. When a host receives a locking request, it waits until its tuple space is not locked by another thread, then returns positively.

The need for locking is not immediately obvious. Consider the case shown in Fig. 8, in which four tuple spaces contain tuples in the reference agent's view. The ellipse inside each

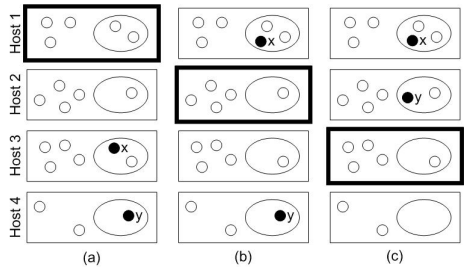


Fig. 8. Locking example

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 tuple spaces for matching tuples in order without locking them; the outlined rectangle indicates the tuple space being queried. In part (a), the operation queries Host 1. Being unsuccessful, the operation thread then queries Host 2 (part (b)). At the same time, a different operation moves tuple x from Host 3's tuple space to Host 1's tuple space. 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 operation's semantics because a match existed for the duration of the operation.

After locking every host in the context, the operation thread requests a matching tuple from each host in order. For the `rdp`, as soon as the operation thread finds a single match, it returns the tuple. For an `inp`, the operation thread returns the first match, but also removes the matching tuple. For aggregate operations, 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 Fig. 8 is acceptable. The weakened semantics of these operations allow more efficient implementations. 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.

Transactions. A transaction operates over several views. As such, transactions are inherently costly. EgoSpaces reduces this cost by requiring a reference agent to explicitly declare which other agents need to be locked for the transaction by providing a list of views. Because the agents contributing to each view are known, EgoSpaces can lock the transaction's participants (including the reference agent) in order (by id). If a new agent moves into the view while a transaction is in progress, its arrival is ignored until the transaction completes. If a contributing agent moves out of the view while a transaction is locking agents, it is unlocked before departing. If the transaction's operations are already executing, the agent's departure must be delayed until the transaction completes. We guarantee enough time to complete the transaction before the agent disappears from communication range using *safe distance* [22]. The latter is defined as a function over the speed and direction of the nodes involved in the communication and the maximum time necessary to complete a requested transaction. If transactions can have longer durations, the safe distance that defines allowable network links becomes shorter.

Reactions. Because reactions are the core of the EgoSpaces behaviors, an efficient implementation is essential. This implementation is similar to blocking operations with added bookkeeping for maintaining the registrations. Each agent keeps a reaction registry (containing all reactions it has registered) and a reaction list (containing all reactions this agent should fire on behalf of other agents, including itself). A reaction registry entry contains a reaction's id, the tuple to output when the reaction fires (if any), and the transaction that extends or follows this reaction (if any). A reaction list entry contains the reaction issuer's id, the reaction's pattern, the view's data pattern, and a boolean indicating whether or not to remove the trigger. Upon registration, the reaction is inserted in each view participant's reaction list. For all matching tuples, the reaction fires, sending a notification (containing a copy of the trigger) to the registering agent. If specified, the tuple is deleted. As long as the reaction remains enabled, new tuples are checked against the pattern. For each match, the registering agent receives a notification and locates the reaction in its reaction registry. If necessary, it performs the appropriate `out` operation and schedules any associated transaction. In

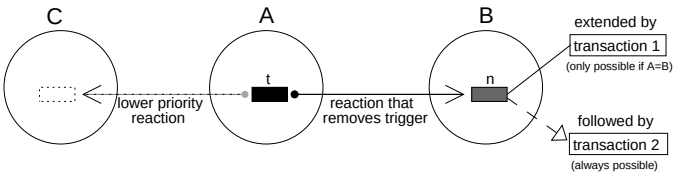


Fig. 9. The Reaction Mechanism

Fig. 9, agents B and C register reactions, which both match t. The reaction with the highest priority (B's reaction) fires first, generating notification n. Because this reaction removes t, C's lower priority reaction will not fire. B's reaction can be extended or followed by a transaction. The former is only allowed when the trigger is local (*i.e.*, A=B). As agents move out of the view, they remove information regarding registered reactions. If these agents return, they receive the registrations and fire the reactions again for matching tuples.

Behaviors. Because the semantics of behaviors are written as reactions (see [26]), their implementations rely on the reaction's implementation. We build these behaviors into the system to provide common actions as simple operations and to allow for code encapsulation and reuse.

Event Generation. To implement event capture, we add an event raising mechanism. Each type of event has a defined string (*e.g.*, *hostArrival*) and some secondary information (*e.g.*, the *HostID* for a host arrival event). Upon generation, special event tuples are created for each registered agent, and these tuples are transmitted to the agent and trigger the event's callback.

VI. Simplifying Application Development

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

A. Emergency Vehicle Warning System

Our first application warns cars of nearby emergency vehicles. When a driver needs to clear the road for the emergency vehicle, a light on the dashboard turns on.

View Definition. Key to this application is the ability 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. 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 (*i.e.*, no `in` operations are allowed except by the emergency vehicle's agent).

A car issues a `rd` operation on its 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 which time, the light on the dashboard warns the driver. The application can probe the view (with periodic `rdp` operations) to wait for the disappearance of the warning tuple. After the emergency vehicle has passed, the application can reissue the `rd`, 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 cars' 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 may not be as timely. This latter behavior could be further accomplished using the reactive constructs.

B. Subscription Music Service

The second application enables music sharing on a network of cars. Users subscribe to a music service which allows them to 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 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 picks up where it left off and completes. Fig. 10 shows the user interface.

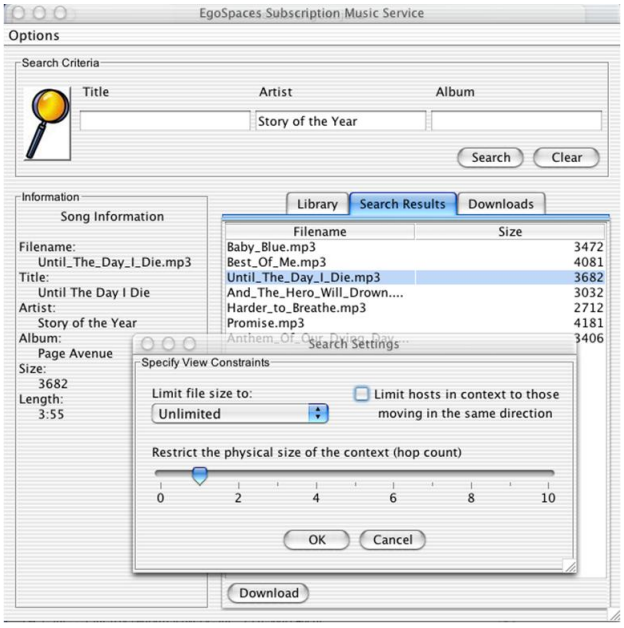


Fig. 10. The subscription music service

View Definition The constraints the user can manipulate include:

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

The following code builds 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);
```

`EConstraint` builds a pattern over tuples; `dc` represents the set of patterns that define the agent's data constraints.

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. The following code generates an information tuple:

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

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. To query the view, the agent uses the following code:

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

Lessons Learned. 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.

C. Collaborative Puzzle Game

The final application demonstrates how EgoSpaces can be useful to cooperative work applications. In this example, several users collaborate to complete a distributed puzzle. Fig. 11 shows the screens of two puzzle participants.

View Definition. This application uses the view constraints to limit the amount of data displayed based on properties of the puzzle. This view is logical 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. An example of such a data constraint is:

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

The `EqualConstraint` function included in EgoSpaces requires the field's value to equal the designated value.

Puzzle players may find many different view definitions useful. If players 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 shape of the hole.

Agent Interaction. The pieces of the puzzle are represented by tuples in the data space of the agent initializing the puzzle. Each agent (a player in the puzzle game), can define views that determine which puzzle pieces are displayed at a given time. A user can select a piece by clicking on it. When the user does so, the tuple corresponding to the puzzle piece is moved to 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 reflected in the displays of connected players.

When a user defines a different view, his display changes. For example, if the user defines a view to contain only edge pieces, all of the interior pieces are hidden (the view at the left of Fig. 11). 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. In this example, we see that the same abstractions can be used to define logical views in smaller scale networks. 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.

VII. Performance Evaluation

EgoSpaces's goal is to simplify the development of context-aware mobile applications. While the programming interface and its use described in the previous sections are important to this goal, the performance of the middleware must also be a concern to ensure that the overhead associated with using the middleware is not detrimental to applications' operations. In this section, we quantify the performance characteristics of the operations described in Section IV under varying environmental and application conditions. The goal of this evaluation is to provide application developers that use EgoSpaces information about the performance they can expect from the middleware and the overhead of employing, for example, operations with transactional semantics.

A. Simulation Settings

For the purposes of this evaluation, we used the open source OMNeT++ discrete event simulator [44] and its mobility framework extension [29]. All of the results we report are for 50 node networks in which the nodes are dispersed in a rectangular area of size 3000x600m^2. The nodes move according to the random waypoint mobility model [6], in which each node is initially placed randomly in the space, chooses a random destination within that space and moves in the direction of the destination at a given speed. Once the node reaches the destination, it pauses for a specified interval (the

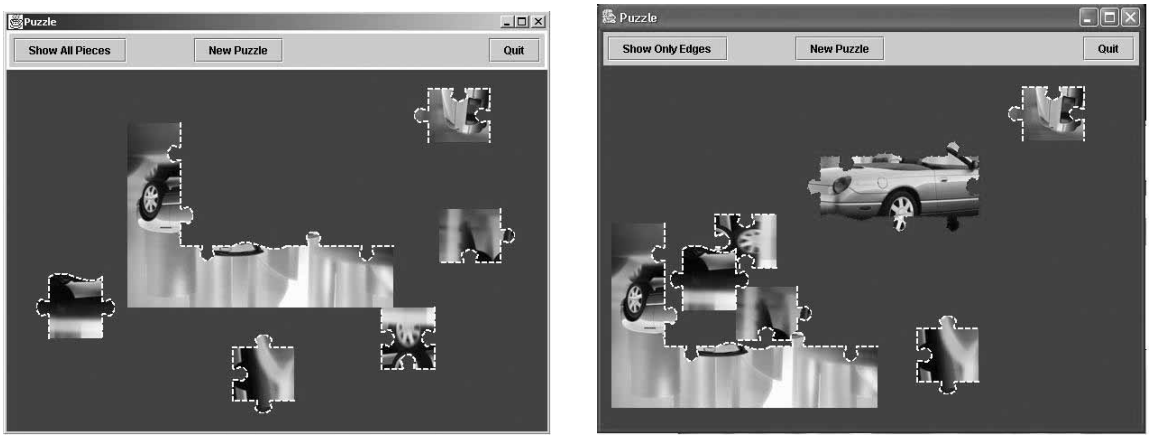


Fig. 11. Two views of a puzzle game

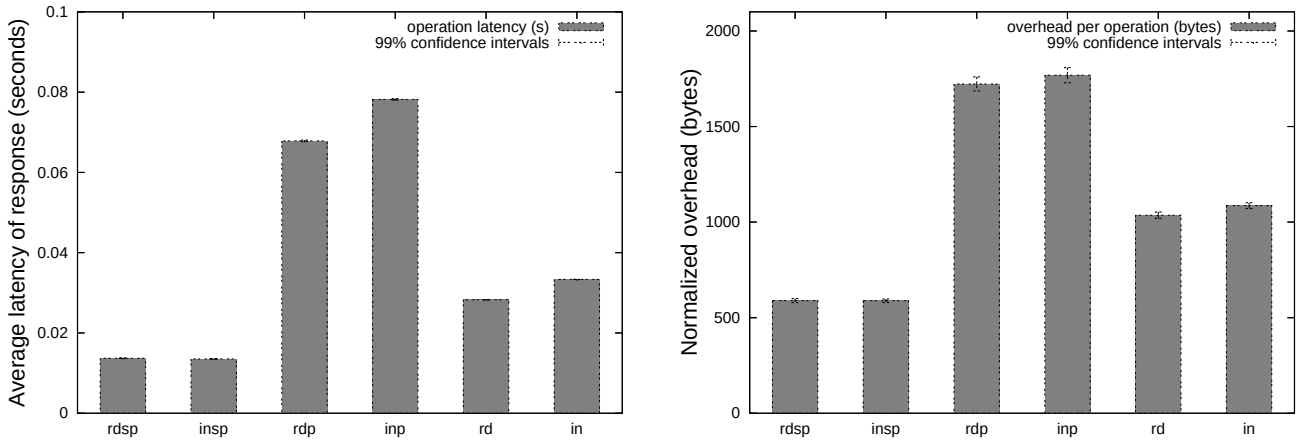
pause time) then repeats the process. In all of our simulations, we use a pause time of 0 seconds to provide relatively dynamic networks. We used the 802.11 MAC protocol with a bandwidth of 1Mbps. The results we present here are for simple views whose network constraints are based on a hop count metric, but, because the packets carry all information for calculating views with them, simulating views based on other properties is straightforward. We chose the simple metric for presentation purposes because it makes the views and their scopes easier to visualize. In addition, because more of the overhead of view construction is dependent on the communication costs than on computation costs, a hop count-based view provides more generalizable results. Data availability was modeled randomly as each node having a 10% probability of possessing a requested data item. Queries were assumed to consume no more than 64 bytes (including the constraints and the operation's pattern), and the data carried in the reply was assumed to fit in 1024 bytes. All results below are reported with 99% confidence intervals.

B. Comparing Operations with Differing Semantics

Our first set of results compares basic performance metrics for the six standard operations: rd, in, rdp, inp, rdsp, and insp. The evaluation of these basic operations can be generalized to express the performance of the more sophisticated operations as well. In the worst (*i.e.*, most expensive) case, a transaction is a sequence of atomic probing operations, and the performance for a transaction is the aggregation of the performance for the operations that it comprises. Our implementation of a reaction is based on the implementation of the atomic blocking operations (in fact, a reaction is implemented as an atomic blocking operation augmented with additional bookkeeping and persistence). Finally, as described in Section V, behaviors (*i.e.*, migration, duplication, and event capture) have been implemented as reactions.

For this initial set of simulations, we set the node speed to be 20m/s (a fairly dynamic scenario; equivalent to automobiles on a city street). The host constraints and agent constraints were unrestrictive and, as indicated above, each agent had a 10% chance of having the requested data item. The network constraints defined a cost function based on hop count with a bound of two hops (*i.e.*, any host within two hops of the reference agent was included in the view). For each data point, 50 runs lasting 900 seconds were performed. In each run, five hosts were randomly selected to be requesters, and they issued a new operation every half second. This is a high-level of traffic when considering a user's interaction with an application, but corresponds to an application that may periodically monitor a condition in its environment (*e.g.*, the positions of nearby automobiles). In the case of the blocking operations, the operation was registered for the full half second, when it was deregistered and a new operation was issued. While the operation remains registered, the view is maintained, though no new data items are introduced during the registration, so a blocking operation is never unblocked.

Fig. 12(a) shows the operation latency for the six operations. In the case of the probing operations (both scattered probes and atomic probes), the time reported is the time to return a result, whether it be the actual data or a null value. In the case of the blocking operation, the average reflects only instances when the data was actually available, ignoring cases when the operation did not return a result before the application canceled it. The in($*$) operations take slightly longer than the rd($*$) operations because they require an extra round of communication between the requester and the data provider to ensure that the data item is removed. As expected, the atomic operations take significantly more time to complete than the scattered probing operations, but, perhaps surprisingly, the atomic probing operations take more than twice as long as the blocking operations. This is due to the fact that, to be able to reliably return a null value assuring the application that the data item did not exist, an atomic probe must perform a two-phase commit protocol. These transactional semantics are expensive, but, as discussed previously, necessary for some applications. A final thing to notice about Fig. 12(a) relates to the expense of an EgoSpaces transaction. Recall that a transaction can be made up of any sequence of non-blocking operations. Given the results displayed in the figure, a transaction consisting of five *dependent* operations (meaning each operation must wait for the previous one to finish before



(a) (b)

Fig. 12. (a) Operation latency for the six single operations in EgoSpaces (in seconds). (b) Overhead per operation issued for each operation type (in bytes).

being issued) takes less than half of a second. For applications that demand this added consistency, this latency represents a reasonable tradeoff for the strengthened semantics.

Fig. 12(b) shows the amount of overhead (in bytes) generated for each request that is sent. In counting the overhead, we counted *everything* except the one-way transmission of the data item. Also, each propagation of the same packet is counted as another packet of overhead. What is important, then, in this characterization, is not necessarily the number assigned to the overhead but the relationship between the overhead for different operations. First, as in the latency, the overhead for the in($*$) operations is slightly larger than for the rd($*$) operation because the former require extra control communication to confirm the removal of a data item. The overhead for the atomic operations is greater than for the non-atomic because they require a beaconing mechanism that enables each node to keep track of its neighbors. The effect of the beacon interval on overhead is explored in more detail below. For the probing operations, the beacon is used to ensure the transactional semantics (*i.e.*, to ensure that every participant in the view has been queried before returning). In the blocking operations, the beacon ensures that nodes moving into the view are notified of an operation while nodes moving out of the view can remove the operation. As above, the overhead for a transaction is dependent on the number of operations it comprises, and the overhead of a reaction is similar to the blocking operations.

C. Impact of Environmental and Network Factors

To be able to perform transactions on changing views and to maintain views as they change over time, it is necessary for hosts to have an up-to-date knowledge of its neighboring (one-hop) nodes. This is accomplished through a beaconing mechanism in which each node periodically broadcasts a "hello" message. Nodes that hear other nodes' beacons add them to their neighbor lists. After not hearing a node's beacon for three beacon intervals, a node removes the departed node from its neighbor list.

Fig. 13 shows two measurements on the same graph. The simulation settings are the same as above, and a rdp operation was used to generate these results. First, the dashed line indicates how the measured overhead changes with changing beacon interval. Not surprisingly, as the beacon interval increases, nodes send fewer beacons, so less overhead traffic is generated. This decrease in overhead comes at a cost, however, with respect to the consistency guarantees that can be provided. The solid line measures the degree with which EgoSpaces could guarantee the consistency of an atomic operation. The distance of this line from 1 indicates the percentage of times that a rdp operation had to abort, *i.e.*, it could not return a data value, but it could also not guarantee that one did not exist. For example, with a beacon interval of one second, 91.5% of rdp operations completed successfully (either with a matching data item or with a guaranteed null value). The value compounds in a transaction consisting of multiple atomic probes; a transaction of five dependent rdp operations would complete successfully only 64.1% of the time. This value decreases with increasing beacon intervals due to the fact that the neighbor lists are increasingly inconsistent. Two other important points should also be noted. First, in inp operations, we did not encounter any instances in which a discovered data item could not subsequently be removed. Second, every time the consistency assumption was not met (even in inp operations), it was possible to notify the application and rollback the operation. However, the same would not necessarily be true in a transaction consisting of multiple operations. These simulations did not incorporate the safe distance algorithm [22] described earlier. Doing so would further restrict the size of the view but provide greater reliability to the application.

We next evaluate the scalability of the view with respect to increasing the size of the view (*i.e.*, the number of participants in the view). Fig. 14 compares a view's size (in number of hops) along the x-axis to both the latency of operations issued on the view (in this case, rd operations) and the overhead incurred in both issuing the request, and, in the case of the rd operation shown, maintaining the view until the request is

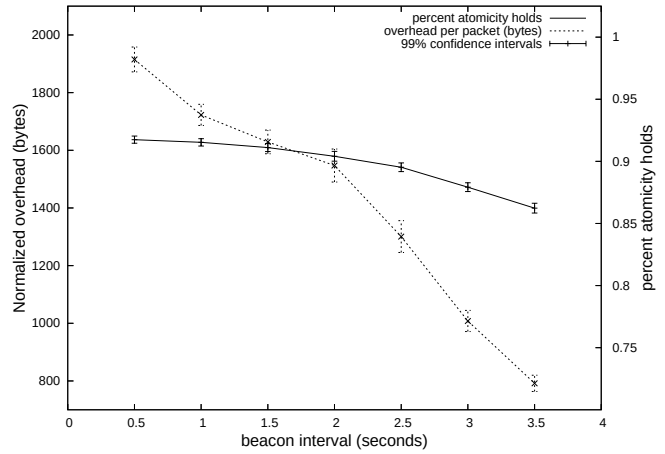


Fig. 13. The tradeoff between guaranteed consistency and overhead for changing beacon intervals (for rdp operations).

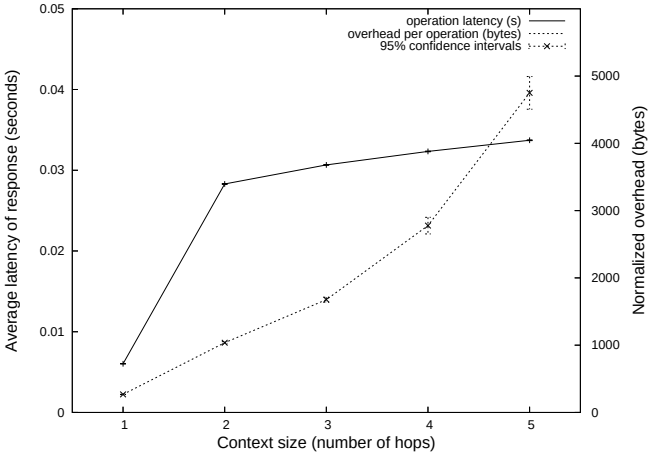


Fig. 14. The relationship between the size of the view and the latency of the operations (solid line) and the overhead incurred (dashed line).

satisfied or canceled. The latency of the operation increases significantly from the one-hop case to the multi-hop case, but only marginally thereafter. This is due to the fact that the one-hop case requires only a single broadcast, while the multi-hop cases require the request to be rebroadcast, resulting in interference and the need for the lower protocol layers to backoff to ensure message delivery. The overhead for distributing view requests increases a bit faster than linearly, but the increase is proportional to the increasing number of view participants.

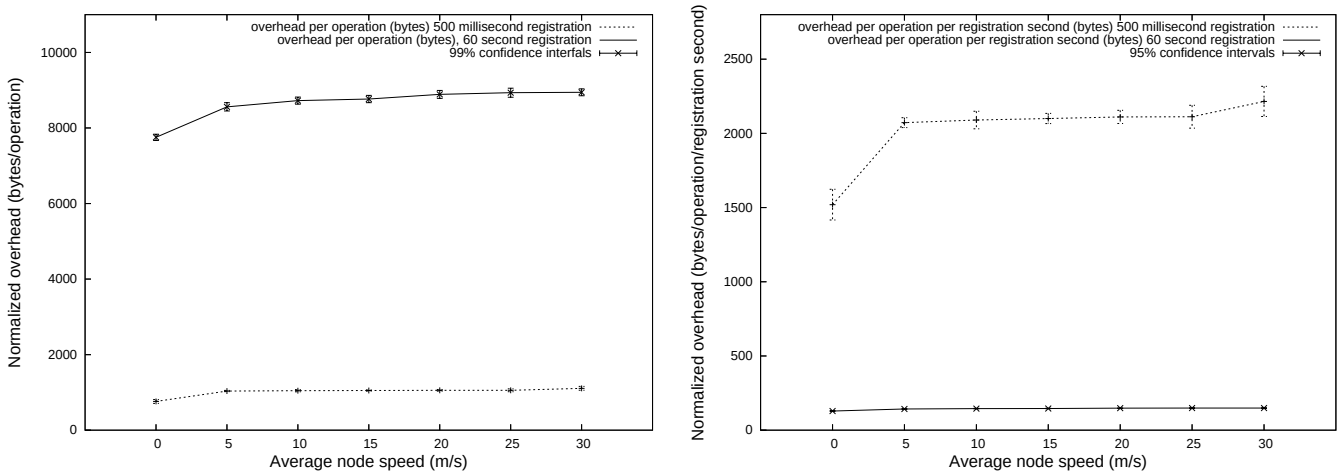
Our final measurements, shown in Fig. 15, compare the impact of changing the average node speed on the overhead of issuing operations. We use a blocking operation (rd) for this measurement since it requires maintaining the view until a matching data item is found or the operation times out (whichever is first). In this example, the view's size was still two hops, but we varied the duration of the registration between .5 seconds (the value used above) and 60 seconds (indicating a significantly longer registration, and therefore a significantly longer portion of time over which the operation remained registered). Fig. 15(a) compares the overhead for the two registration durations. As expected, a longer duration registration incurs significantly more overhead. In both cases, the overhead associated with maintaining the view while the operation is registered increases as the speed of the nodes increases (*i.e.*, as the network becomes more dynamic), but the increase is very gradual. Fig. 15(b) shows the same information, but amortizes the overhead incurred over the duration of the registration. In this case, the overhead for a longer duration registration is much lower than for the short registration, indicating that most of the overhead stems from the initial multicast, and the subsequent overhead of the distributed algorithm that maintains the view is comparatively small.

The view concept's use of asymmetric coordination represents such a significant deviation from existing coordination mechanisms that it is difficult to compare its performance to alternatives. Through this evaluation, we have shown that our implementation of the view concept is manageable within reasonable traffic and mobility assumptions. In addition, Fig. 15 shows that the maintenance aspect of the view is inexpensive in comparison to distribution of multicast messages over a dynamic network, which is the approach that underlies most other coordination approaches. We have also shown that maintaining the consistency assumption required for providing atomic operations is feasible to a certain extent, with two benefits worth repeating. First, when the atomicity assumption *does* fail for a particular operation, the application can be notified, and, for single operations, data was never left in an inconsistent state (*i.e.*, the removal portion of an in operation never failed). Second, combination of the view's construction mechanism with our consistent group membership can guarantee consistency for transactions of a longer duration, but may require further restricting the view's participants according to the calculated *safe distance*.

VIII. Related Work

Our experiences and the above applications have shown that EgoSpaces's programming abstractions dramatically simplify the development of mobile applications. These abstractions are founded on the observation that representing the dynamic environment through an egocentric data structure allows natural interactions for any novice programmer and reduces the need for complex and error-prone network programming.

EgoSpaces is not the first programming environment to use such abstractions for mobile computing. LIME [33] aims to simplify the software development process and has been shown to facilitate context interactions [32]. LIME enables mobile coordination by abstracting communication into a global virtual data structure, the tuple space. At any instant, a device's perception of the world is through this tuple space which contains the data available on all connected devices. LIME requires strong assumptions about the operating environment that fail to hold as the number of devices, connections, and the degree of mobility grows. Limone [15] centers the coordination tasks around *acquaintances*, and knowledge of specific coordinating partners is essential to Limone's functionality. EgoSpaces, on the other hand, takes a device agnostic view, favoring complete abstraction of the network and its devices in to the available context or data items.



(a) (b)

Fig. 15. (a) Overhead per rd operation for varying node speed (in bytes) (b) Normalized overhead per rd operation per second of registration (in bytes).

Reactive tuple space approaches like MARS [7], TuCSoN [34], and TOTA [30] augment tuple spaces with reactive capabilities. Mars and TuCSoN focus on coordination among co-located mobile agents but do not enable coordination across the networks, requiring agents to move to hosts where resources are located to perform their computation. TOTA provides an alternative to EgoSpaces, but instead of EgoSpaces's egocentric pull-based interactions, TOTA propagates tuples away from a reference node based on context properties, in a manner similar to content based multicast [50].

Recent middleware have been developed to enable the rapid development of pervasive or ubiquitous computing applications. GAIA [41] introduces *Active Spaces* as immersive computing environments for context-aware applications. Users move from one Active Space to another, seamlessly integrating into new spaces. GAIA functions in small networked environments where the available resources in the space can be centrally managed by a kernel. This approach does not map well to large-scale applications in mobile ad hoc networks that necessitate an entirely decentralized solution. CORTEX [45] proposes an infrastructure for context-awareness in nomadic mobile environments and focuses on quality of service guarantees within a region of the network. Similarly, Solar [10] provides an infrastructure to support context acquisition and operation for nomadic wireless networks. The goals of these systems are in line with our goals—to support large-scale mobile computing—but the target environment differs in that the solutions apply only to nomadic networks.

While EgoSpaces abstracts all of the context as data items stored in a distributed tuple space, other context-aware middleware approaches use the service abstraction to represent available resources. Context-aware resource bindings update the connections between clients and services as processing or environment dictates [4], [28]. Context-sensitive bindings [19], [40] use a *follow-me session* to transfer a service connection from one provider to another. This approach builds on the EgoSpaces notion of an asymmetric definition of context. Service-oriented network sockets [42] provide a similar abstraction but use existing service discovery mechanisms to gather *all* matching services locally before deciding which services to connect to. This can incur significant amounts of overhead in environments that are highly dynamic. iMash [2] presents a dynamic application session handoff scheme that relies on a knowledgeable intermediary to handle service switches on behalf of applications. Similarly, Atlas [12] uses a central server to mediate the transfer of a service binding from one provider to another.

A complementary approach to coordination relies on event based interactions. Context-aware publish subscribe systems [5], [8], [13], [14], [31], [48] generate events and propagate them through the network towards matching subscriptions. EgoSpaces's event generation mechanism is analogous to these approaches, yet it allows an application to express its interest in an event based on the view concept, further restricting the network over which a subscription must be propagated.

Other middleware approaches adapt services within the infrastructure to changing context properties, allowing applications to become relatively ignorant of the restrictions their environment might impose. MobiPADS [9] employs adaptive fidelity techniques to tailor services based to a particular device's capabilities or network properties like available bandwidth. CARMEN [3] uses mobile agents that track client devices, providing customized services based on user profiles and environmental conditions. ReMMoC [17] similarly attempts to simplify the development of applications that rely on distributed services by unifying the discovery and interaction mechanisms through a single web-services based interface. Satin [49] uses encapsulation and component mobility to dynamically reconfigure services to adapt to applications' changing needs on-demand. These approaches have a significantly different goal than our work. EgoSpaces focuses on enabling applications to adapt to changes while these systems place the adaptation in the middleware, making the application and user experiences the same regardless of the environment.

In addition to the differences highlighted above, EgoSpaces focuses specifically on enabling context-aware coordination through data sharing. Our approach chooses to abstract avail-

able context information into a data structure that we allow applications to access and respond to. We have explicitly favored application-awareness over transparency in an effort to enable applications to dynamically respond to their environments in manners that are tailored to the applications' instantaneous needs.

IX. Conclusions

This paper describes a simplified application development process for programmers in mobile ad hoc networks. The investigation began with a careful study of emerging applications and the classification of these needs into a redefinition of context-awareness. Given the lessons learned from this exploration, we built a conceptual model of mobile applications. The use of context-awareness within mobile computing and for the purpose of simplifying development for novice programmers has shown significant promise. It is coupled with the introduction of asymmetric coordination (via the *view* construct). The need for asymmetry is based on the observation that mobile applications tend to be egocentric in that they define their needs from the environment independent of the needs of other applications. The usefulness of the EgoSpaces middleware has been demonstrated through the successful and simple construction of dynamic applications from varying domains and its performance characterized through simulation.

Acknowledgments

The EgoSpaces prototype implementation and further documentation are available at `http://www.ece.utexas.edu/~julien/egospaces.html`. This research was supported in part by the National Science Foundation under Grant No. CCR-9970939 and by the Office of Naval Research MURI Research Contract No. 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.

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(5):421–433, October 1997.

[2] R. Bagrodia, S. Bhattacharyya, F. Cheng, S. Gerding, R. Guy, Z. Ji, J. Lin, T. Phan, E. Skow, M. Varshney, and G. Zorpas. iMASH: Interactive mobile application session handoff. In *Proc. of the 1^{st} Int'l. Conf. on Mobile Syst., App., and Services*, pages 259–272, May 2003.

[3] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Context-aware middleware for resource management in the wireless internet. *IEEE Trans. on Software Eng.*, 29(12):1086–1099, December 2003.

[4] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Dynamic binding in mobile applications. *IEEE Internet Comput.*, 7(3):34–42, 2003.

[5] R. Boyer and W. Griswold. Fulcrum: An open-implementation approach to internet-scale context-aware publish/subscribe. In *Proc. of the 38^{th} Hawaii Int'l. Conf. on System Sciences*, 2005.

[6] J. Broch, D. Maltz, D. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proc. of the ACM/IEEE MobiCom*, pages 85–97, October 1998.

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

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

[9] A. Chan and S.-N. Chuang. MobiPADS: A reflective middleware for context-aware mobile computing. *IEEE Trans. on Software Eng.*, 29(12):1072–1085, December 2003.

[10] G. Chen and D. Kotz. Solar: An open platform for context-aware mobile applications. In *Proc. of the 1^{st} Int'l. Conf. on Pervasive Comput.*, pages 41–47, March 2002.

[11] 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 the 6^{th} Int'l Conf. on Mobile Comput. and Networking*, pages 20–31, August 2000.

[12] A. Cole, S. Duri, J. Munson, J. Murdock, and D. Wood. Adaptive service binding middleware to support mobility. In *Proc. of the ICDCS Wkshps.*, pages 396–374, May 2003.

[13] P. Costa and G. Picco. Semi-probabilistic content-based publish-subscribe. In *Proc. of the 25^{th} Int'l. Conf. on Dist. Comput. Syst.*, pages 575–585, June 2005.

[14] 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 Eng.*, 27(9):827–850, September 2001.

[15] C.-L. Fok, G.-C. Roman, and G. Hackmann. A lightweight coordination middleware for mobile computing. In *Proc. of the 6^{th} Int'l. Conf. on Coordination Models and Languages*, February 2004.

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

[17] P. Grace, G. Blair, and S. Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *ACM SIGMOBILE Mobile Comput. and Commun. Review*, 9(1):2–14, January 2005.

[18] G. Hackmann, C. Julien, J. Payton, and G.-C. Roman. Supporting generalized context interactions. In T. Gschwind and C. Mascolo, editors, *Software Eng. and Middleware: 4^{th} Int'l. Wkshp., Revised Selected Papers*, volume 3437 of *LNCS*, pages 91–106. March 2005.

[19] R. Handorean, R. Sen, G. Hackmann, and G.-C. Roman. Context-aware session management for services in ad hoc networks. In *Proc. of the Int'l. Conf. on Services Comput.*, pages 113–120, July 2005.

[20] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The anatomy of a context-aware application. *Wireless Networks*, 8(2/3):187–197, March–May 2002.

[21] J. Hong and J. Landay. An infrastructure approach to context-aware computing. *Human Computer Interaction*, 16(2–4), 2001.

[22] Q. Huang, C. Julien, and G.-C. Roman. Relying on safe distance to achieve strong partitionable group membership in ad hoc networks. *IEEE Trans. on Mobile Comput.*, 3(2):192–205, April–June 2004.

[23] C. Julien. *Supporting Context-Aware Application Development in Ad Hoc Mobile Networks*. PhD thesis, Washington University in Saint Louis, 2004.

[24] C. Julien, J. Payton, and G.-C. Roman. Adaptive access control in coordination-based mobile agent systems. In R. C. et al, editor, *Software Eng. for Large-Scale Multi-Agent Syst. III*, volume 3390 of *LNCS*, pages 254–271, February 2005.

[25] C. Julien and G.-C. Roman. Egocentric context-aware programming in ad hoc mobile environments. In *Proc. of the 10^{th} Int'l. Symp. on the Foundations of Software Eng.*, pages 21–30, November 2002.

[26] C. Julien and G.-C. Roman. Active coordination in ad hoc networks. In *Proc. of the 6^{th} Int'l. Conf. on Coordination Models and Languages*, volume 2949 of *LNCS*, pages 199–215, February 2004.

[27] C. Julien and G.-C. Roman. Supporting context-aware interaction in dynamic multi-agent systems (invited paper). In *Environments for Multiagent Syst.*, volume 3374 of *LNCS*, February 2005.

[28] M. Klein and B. Konig-Ries. Combining query and preference: An approach to fully automize dynamic service binding. In *Proc. of the Int'l. Conf. on Web Services*, pages 788–791, July 2004.

[29] M. Loebbers, D. Willkomm, and A. Koepke. The Mobility Framework for OMNeT++ Web Page. `http://mobility-fw.sourceforge.net`.

[30] M. Mamei, F. Zambonelli, and L. Leonardi. Tuples on the air: A middleware for context-aware computing in dynamic networks. In *Proc. of the ICDCS Wkshps.*, pages 342–348, 2003.

[31] R. Meier and V. Cahill. STEAM: Event-based middleware for wireless ad hoc networks. In *Proc. of the 22^{nd} Int'l. Conf. on Distib. Comput. Wkshps.*, pages 639–644, July 2002.

[32] A. L. Murphy and G. P. Picco. Using coordination middleware for location-aware computing: A LIME case study. In *Proc. of the 6^{th} Int'l. Conf. on Coordination Models and Languages*, volume 2949 of *LNCS*, pages 263–278, February 2004.

[33] 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 Distib. Comput. Syst.*, pages 524–533, April 2001.

[34] A. Omicini and F. Zambonelli. TuCSoN: A coordination model for mobile information agents. In *Proc. of the 1st Int'l. Wkshp. on Innovative Internet Information Syst.*, pages 177–187, June 1998.

[35] J. Pascoe. Adding generic contextual capabilities to wearable computers. In *Proc. of the 2nd Int'l. Symp. on Wearable Computers*, pages 92–99, October 1998.

[36] J. Payton, C. Julien, and G.-C. Roman. Context-sensitive data structures supporting software development in ad hoc networks. In *Proc. of the 3rd Int'l. Wkshp. on Software Eng. for Large Scale Multi-Agent Syst.*, pages 42–48, 2004.

[37] J. Payton, C. Simon, and G.-C. Roman. A query-centered perspective on context-awareness in mobile ad hoc networks. Technical Report WUCSE-05-8, Washington University in Saint Louis, Department of Computer Science and Eng., 2005.

[38] B. Rhodes. The wearable remembrance agent: A system for augmented memory. In *Proc. of the 1st Int'l. Symp. on Wearable Computers*, pages 123–128, October 1997.

[39] 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 Eng.*, pages 363–373, May 2002.

[40] G.-C. Roman, C. Julien, and A. L. Murphy. A declarative approach to agent-centered context-aware computing in ad hoc wireless environments. In *Software Egineering for Large-Scale Multi-Agent Syst.*, volume 2603 of *LNCS*, pages 94–109, 2003.

[41] M. Roman, C. Hess, R. Cerqueira, A. Ranganat, R. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Comput.*, 1(4):74–83, October–December 2002.

[42] U. Saif and J. Paluska. Service-oriented network sockets. In *Proc. of the 1st Int'l. Conf. on Mobile Syst., Apps., and Services*, pages 159–172, May 2003.

[43] D. Salber, A. Dey, and G. Abowd. The Context Toolkit: Aiding the development of context-enabled applications. In *Proc. of the Conf. on Human Factors in Comput. Syst.*, pages 434–441, May 1999.

[44] A. Vargas. OMNeT++ Web Page. `http://www.omnetpp.org`.

[45] P. Verissimo, V. Cahill, A. Casimiro, K. C. A. Friday, and J. Kaiser. CORTEX: Towards supporting autonomous and cooperating sentient entities. In *Proc. of European Wireless*, February 2002.

[46] R. Want, A. Hopper, V. Falco, and J. Gibbons. The Active Badge location system. *ACM Trans. on Information Syst.*, 10(1):91–102, January 1992.

[47] R. Want, B. Schilit, N. Adams, R. Gold, K. Petersen, D. Goldberg, J. Ellis, and M. Weiser. An overview of the PARCTab ubiquitous computing environment. *IEEE Personal Commun.*, 2(6):28–33, December 1995.

[48] E. Yoneki and J. Bacon. An adaptive approach to content-based subscription in mobile ad hoc networks. In *Proc. of the 1st Int'l. Wkshp. on Mobile and P2P Comput.*, pages 92–97, 2004.

[49] S. Zachariadis, C. Mascolo, and W. Emmerich. SATIN: A component model for mobile self-organisation. In *Proc. of the Int'l. Symp. on Distib. Objects and Apps.*, October 2004.

[50] H. Zhou and S. Singh. Content based multicast (CBM) in ad hoc networks. In *Proc. of the 1st ACM Int'l. Symp. on Mobile Ad Hoc Networking and Comput.*, pages 51–60, 2000.