Active Coordination in Ad Hoc Networks

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

ABSTRACT

The increasing ubiquity of mobile devices has led to an explosion in the development of applications tailored to the particular needs of individual users. As the research community gains experience in the development of these applications, the need for middleware to simplify such software development is rapidly expanding. Vastly different needs of these various applications, however, have led to the emergence of many different middleware models, each of which approaches the dissemination of contextual information in a distinct way. The EgoSpaces model consists of logically mobile agents that operate over physically mobile hosts. EgoSpaces addresses the specific needs of individual agents, allowing them to define what data is to be included in their operating context by means of declarative specifications constraining properties of the data items, the agents that own the data, the hosts on which those agents are running, and attributes of the ad hoc network. In the resulting coordination model, agents interact with a dynamically changing environment through a set of views, custom defined projections of the set of data objects present in the surrounding ad hoc network. This paper builds on EgoSpaces by allowing agents to assign automatic behaviors to the agent-defined views. Behaviors consist of actions which are automatically performed in response to specified changes in the view. Behaviors discussed in this paper encompass reactive programming, transparent data migration, automatic data duplication, and event capture. Formal semantic definitions and programming examples are given for each behavior.

1. INTRODUCTION

The mobile ad hoc environment is an extreme among networks where the lack of an infrastructure necessitates a reinvestigation of network protocols and communication paradigms. These opportunistically formed networks change rapidly in response to nodes entering and leaving communication range. Roving robots on an uninhabited planet may explore the terrain, collect information, and coordinate to assimilate this information and perform tasks. Automobiles on a highway communicate to gather traffic information about the road ahead or to distribute digital coupons for nearby restaurants. Rescue workers in a disaster recovery scenario must communicate to perform their tasks quickly and safely, but the communication infrastructure is often crippled or even destroyed. Units in a military situation must remain aware of the current status of other units, but the communication infrastructure may belong to the enemy. These domains share the potential for a wealth of applications requiring coordination among mobile components.

Much research in the area has focused on developing a variety of protocols tailored to the specialized needs of these constrained networks. Ad hoc routing protocols [2, 9, 11, 15] have made great strides to bridge the communication gaps that exist among groups of connected hosts. These protocols have transformed the ad hoc network domain and have brought the possibility of large scale ad hoc networks ever closer to reality. In such environments, however, the massive amounts of available information quickly overwhelms applications. This information serves as the context for an application operating in the network, and the application may need to adapt to changes in this context over time.

An application's desire for adaptability manifests itself in the diversity of context-aware applications for traditional networks [6, 18]. FieldNote [16] allows researchers to implicitly attach context information to field notes, while tour guides [1, 4] display information based on the user's location. The radically different properties of ad hoc networks, however, require new context-awareness models tailored to the environment's specific complexities. The Context Toolkit [17] and Context Fabric [7] take steps to generalize context in various environments, but they do not address the need for a distributed coordination model.

Applications running in this information-rich environment benefit from coordination mechanisms that assist in managing, operating over, and reacting to contextual information. As the demand for new context-aware applications tailored to the ad hoc environment grows, producing these applications places an increasingly heavy burden on programmers. Research in mobile computing middleware has shown that providing coordination constructs in middleware can simplify programming. While early middleware solutions focused on localizing reactions to individual hosts [3] or limiting interactions to symmetric communication [10], the EgoSpaces model [8] and middleware introduced the novel notion of asymmetric coordination, giving each application direct control over the size and scope of its personalized context. This approach is essential to accommodating programming for large, dense ad hoc networks.

Given the large amounts of context data and the coordination constructs that have proven historically useful, however, the basic operations provided by EgoSpaces fall short of the abstractions programmers require for rapid application development. This paper extends EgoSpaces to provide a variety of these high-level coordination constructs, including reactive programming, data migration, data duplication, and event capture. Of particular interest is our ability to reduce the specialized behaviors to a single construct, the reaction, giving promise for an efficient implementation that maximizes application responsiveness and minimizes communication overhead without sacrificing simplicity.

The next section reviews EgoSpaces. Section 3 presents the extensions. Section 4 addresses performance considerations in presenting the constructs' implementations. Conclusions appear in Section 5.

2. THE EGOSPACES COORDINATION MODEL

EgoSpaces introduced an agent-centered notion of context whose scope extends beyond the local host to contain data and resources associated with hosts and agents surrounding the agent of interest. This asymmetric relation among participants is new to coordination research and is motivated by our desire to accommodate high-density and wide-coverage ad hoc networks. This section reviews EgoSpaces and highlights its key components.

2.1 Computational Model

EgoSpaces considers systems entailing both logically mobile agents (units of modularity and execution) executing over physically mobile hosts (simple containers for agents). Communication among agents and agent migration can occur whenever the hosts involved are connected. A closed set of these connected hosts defines an ad hoc network.

EgoSpaces bases its coordination on the notion of a Global Virtual Data Structure (GVDS) [12]. In this model, all data available in the entire network appears, to the programmer, to be stored in a single, common data structure, even though the data items themselves are distributed among the participating agents. The programmer interacts with the data structure via the standard operations for that structure. At any given time, the available data depends on connectivity.

Each agent manages its own data items, stored in a local data repository. When agents move within communication range of each other, their data structures logically merge to form a single "global" structure. The structures over which agents ultimately operate are projections of this global data structure called *views*.

2.2 View Concept

In principle, an agent's context includes all data available in the entire ad hoc network. As discussed previously, providing access to this vast amount of information proves costly. For this and other reasons, EgoSpaces structures data access in terms of *views*, projections of the GVDS. Since one's context is relative, we use the term *reference agent* to denote the agent whose context we are considering. Each agent defines individualized views by providing declarative specifications constraining properties of the network, hosts, agents, and data. As an example, imagine a building with a fixed infrastructure of sensors and information appliances providing contextual information. Sensors provide information regarding the building's structural integrity, the frequency of sounds, the movement of occupants, etc. Engineers and inspectors carry PDAs or laptops that provide additional context and assimilate context information. Different people have specific tasks and will therefore use information from different sensors. As an engineer moves through the building, he wishes to see structural information not for the whole building, but for the floors adjacent to his current floor. An agent running on his PDA might declare the following view:

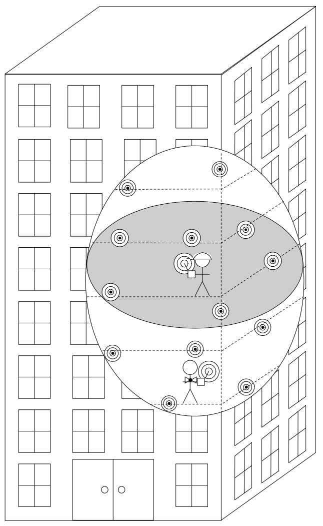
Figure 1: Example view definition.

> *Data from the past hour (reference to data) gathered by structural agents (reference to agents) on sensors (reference to hosts) within one floor of my current location (property of reference host).*

Figure 1 depicts this example, where the shaded area represents the view of the engineer (in the hard hat). As shown, this view contains sensors embedded in the building and the PDA of an inspector on the adjacent floor.

EgoSpaces transparently maintains views. As hosts and agents move, the view's contents automatically reflect these changes without the reference agent's explicit action. As the engineer changes floors, his view is automatically updated to include data from a different set of sensors and devices.

EgoSpaces employs agent-specified *access control functions* to limit the ability of other agents to access an agent's local data. When a reference agent defines a view, it attaches a list of operations it intends to perform over the view and a set of credentials verifying itself to other agents. When determining the contents of a view, EgoSpaces evaluates, for each tuple meeting the view specification, the contributing agent's access control function with respect to the provided operations and credentials. The view contains only tuples that have qualified via the access control function. More details on view specifications, transparent maintenance, and access control can be found in [8] and [14].

2.3 Basic Data Access Operations

For the remainder of this paper, we focus on using a tuple space based data structure underlying our views. Each agent, therefore, carries its own local tuple space. The operations provided over views are variations of the standard Linda [5] tuple space operations for tuple creation (**out**), tuple reading (**rd**), and tuple removal (**in**). In EgoSpaces, however, the scope of the former is restricted to the local tuple space, while the scope of the latter two operations is constrained to a single view.

The EgoSpaces model introduces a new tuple structure in which a tuple is a set of unordered triples of the form:

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

where the *name*s of the fields must be unique within the tuple. The Linda retrieval operations (**rd** and **in**) operate by matching a pattern against data in the tuple space. EgoSpaces patterns contain constraints over the fields in a tuple. To match a tuple, every constraint in the pattern must be satisfied by a corresponding field in the tuple.

Agents create tuples using **out** operations. A new tuple is placed in the creating agent's local tuple space and is available in any view whose constraints it satisfies. To read and remove tuples, agents use variations of **rd** and **in** operations restricted to individual views. Because **in** operations remove tuples from the tuple space, they may affect other views if the tuple removed is contained in multiple views. The **rd** and **in** operations block until a matching tuple exists and then return the match. If more than one tuple matches, the one returned is chosen non-deterministically.

Variations of these operations include aggregate operations (**rdg** and **ing**) that block until a match exists and then return all matches, and probing versions of both single (**rdp** and **inp**) and aggregate operations (**rdgp** and **ingp**) which return ϵ if no match immediately exists. All operations listed thus far act over the view atomically, requiring a transaction over all view participants. Because this can become costly, EgoSpaces offers scattered probes for both single (**rdsp** and **insp**) and aggregate (**rdgsp** and **ingsp**) operations. They provide a weaker consistency because they check the tuple spaces one at a time without locking the entire view, thus they may miss a matching tuple. All operations and their semantics are provided in [8] with a formal description of tuples, patterns, and the associated matching function.

Programming Example. To operate in EgoSpaces, a programmer defines views and issues operations over them. The building engineer might retrieve structural information about a single floor, perform some local processing to assimilate that data, and then output a tuple indicating whether the structural integrity of the current floor is sound. In this case, the engineer's agent does not want to consume the data items, because they might be useful to the operation of other application agents. The following code accomplishes this in EgoSpaces:

```
ν = [data from structural agents on the current floor]
p = ⟨(strain, number, any),
      (acoustic emission, number, any),
      ...,
      (timestamp, time, [within 10 minutes])⟩
data[] = ν.rdgp(p)
[local processing using data]
result = [tuple containing result]
out(result)
```

The first line creates a view that contains only data from structural agents on this floor. The details of view specification are outside the scope of this paper; for more information see [8]. In the definition of the pattern p, the constraint any indicates that the tuple must contain a field with the indicated name and type but the value is unrestricted.

3. EXTENDING EGOSPACES

For some applications, these basic operations suffice, but in many cases, the application requires more sophisticated coordination mechanisms. For example, a built-in reactive construct has been shown to be extremely useful in programming for ad hoc networks. This section presents several such constructs, including a powerful reactive mechanism, data migration, data duplication, and event capture. We show how using these sophisticated constructs decreases the amount of code the programmer must write, increases the encapsulation of this code, and increases code reusability.

3.1 Advanced Constructs

All the constructs previously described involve explicit data access. If a mobile component needs to wait for a piece of data before performing additional actions, it must poll. This costly and inefficient mechanism prevents the component from performing other work in the meantime. Furthermore, the EgoSpaces primitives provide no mechanism for grouping operations in a transactional fashion. We introduce reactions to address the former concern and transactions to address the latter. We then combine the two constructs to build an even more powerful reactive construct.

3.1.1 Reactions

EgoSpaces provides reactive programming constructs that allow agents to adapt their behavior in response to the presence of particular tuples. Similar abstractions have proven useful in other mobile systems [10, 3]. An EgoSpaces reaction associates a trigger (i.e., a pattern) with a set of possible simple actions to perform when a tuple in the view matches the pattern. A reaction is registered on all agents contributing to the view and fires once for every tuple in the view matching its pattern. Disabling and re-enabling a reaction causes it to fire again for all matching tuples. Similarly, disconnection followed by reconnection causes reactions to re-fire. The burden of handling these cases falls on the application programmer. A reaction can remove its trigger from the tuple space and/or output the trigger modified in some way. This modification is achieved through a *tuple_modifiers* subroutine the user defines that can add, remove, or change fields before outputting the trigger tuple. For example, if an agent with unique id ID1 has retrieved the tuple:

$$\langle(ID, TupleID, 5)(destination, AgentID, \texttt{ID1}),\\ (timestamp, time, 8{:}41), temperature, celsius, 28)\rangle$$

and wants to change the timestamp field, remove the destination field, and add an owner field, it defines the following *tuple_modifiers*:

```
tuple_modifiers(t) =
      {t.changeField(timestamp, currentTime),
       t.removeField(destination),
       t.addField(owner, AgentID, ID1)
       t.newID()}
```

where currentTime is a local variable containing the current time. The newID method allows the tuple's new owner to give it a new, unique ID. If the new tuple ID generated was 12 and the new time 9:36, the resulting tuple would be:

$$\langle(ID, TupleID, 12)(timestamp, time, 9{:}36),\\ (temperature, celsius, 28), (owner, AgentID, \texttt{ID1})\rangle$$

If the *tuple_modifiers* attempt to add a field that already exists in the tuple, the current value of the field is replaced with the value being added. The key to using the *tuple_modifiers* is that the tuple output to the tuple space will have the same ID (unless it is changed by the *tuple_modifiers*), and therefore the reactive construct will not fire repeatedly on the same tuple.

A reaction has one of two scheduling modalities, eager or lazy, indicating when it should fire. Reactions with eager modalities occur immediately following the insertion of a matching tuple into the view. Only other eager reactions can preempt them. Therefore, an eager reaction is guaranteed to fire for every matching tuple present in the view while the reaction is enabled, except under one condition discussed below. A lazy modality brings a much weaker guarantee—eventual triggering of the reaction is guaranteed if the tuple remains in the view long enough. Other operations (including both reactions and basic EgoSpaces operations) may occur in the meantime, possibly removing the tuple before the lazy reaction fires. Finally, reactions have a priority that arranges a hierarchy of firing within each scheduling modality. Priorities are integers; within each modality, reactions with higher priorities fire before reactions with lower priorities (the highest priority being 1). When more than one reaction with the same modality and same priority can be triggered by the same tuple, the reaction fired first is chosen non-deterministically. If the first reaction removes the trigger, then the second reaction will not be fired. Reactions take the form:

$$\rho = \textbf{react to}\, p\, [\textbf{remove}]\ \ [\textbf{and out}(\textit{tuple\_modifiers}(\tau))]$$

where the local name τ is bound to the trigger tuple; p is the reactive pattern; the keyword **remove** causes tuple removal; and the optional **out**(*tuple_modifiers*(τ)) places the trigger tuple with the *tuple_modifiers* applied into the reference agent's tuple space. A reference agent enables and disables a reaction using:

$$\begin{array}{l}\textbf{enable}\ \rho\ \ \textbf{with}\ \textit{sched\_modality}, \textit{priority}\ \textbf{over}\ \nu \\ \textbf{disable}\ \rho\ \ \textbf{over}\ \nu\end{array}$$

where *sched_modality* is either eager or lazy, and *priority* is an integer. As with other operations, reactions affect the contributing agents' access controls. When specifying a view, the reference agent must indicate if it intends to register reactions on it.

Triggering the reaction and executing the associated actions occur as a single atomic step. If used, the **out** places a tuple in the reference agent's local tuple space at the completion of the reaction's execution. This tuple can trigger other reactions registered on the same view or different ones.

The next section discusses EgoSpaces transactions, which we will use later to add flexibility to this reactive construct. First, however, we give an example of how programming with this reactive construct compares to programming using only the EgoSpaces primitives.

Programming Example. To achieve this behavior using only the primitives, a programmer must write the entire reactive behavior by hand. Consider the application scenario in which the original temperature sensors placed in the building generate Fahrenheit temperatures. Most of these sensors have been replaced by sensors that generate Celsius temperatures. To provide a standard system, the Celsius sensors contain an agent that reacts to the presence of Fahrenheit readings, converts the values to Celsius, and replaces the readings in the tuple space. If using only the basic EgoSpaces operations, the programmer must write the following code:

```
ν = [temperature data on this floor and adjacent ones]
p = ⟨(tempType, string, = "Fahrenheit")⟩
seenTuples = new Vector()
while(true)
   data[] = ν.rdgp(p)
   if data ≠ null
      for i=1 to data.length
         if !seenTuples.contains(data[i])
            ν.inp(data[i])
            data[i].changeField(tempType, "Celsius")
            data[i].changeField(tempValue,
                                convert(oldT))
            out(data[i])
            seenTuples.add(data[i])
   else
      sleep(time)
```

This code is slightly simplified because it refers to the Fahrenheit temperature as "`oldT`", but this value must really be retrieved from the tuple (`data[i]`). The programmer must manage this code in an extra thread independent of the agent's other operations. The agent creates and executes the thread whenever it wants to "enable" the reaction, and stops it when it wants to "disable" the reaction. In this programming by hand example, the thread awakens periodically to check the reactive condition. To check this, the thread first must read all tuples matching p from the tuple space and check if they have been processed before. If not, the actions execute for the tuple, and the tuple is added to the list of processed tuples.

With the built-in reactive construct, the code becomes:

```
ν = [temperature data on this floor and adjacent ones]
p = ⟨(tempType, string, ="Fahrenheit")⟩
t_m(t) = {t.changeField(tempType, "Celsius"),
          t.changeField(tempValue, convert(oldT))}
ρ = react to p remove and out t_m(τ)
enable ρ with eager, 1 over ν
```

The definition of the view and the pattern is the same, but, from the programmer's perspective, this is a simpler, more straightforward way to code the reactive behavior. In this example, the programmer is enabling a high priority, eager reaction. Not only does this reactive construct simplify the programmer's task, it adds subtle, useful semantics. Instead of achieving a polling behavior as in the first example, if the reaction is eager, the application is guaranteed that it will fire immediately following the insertion of a tuple matching p into the tuple space unless another eager reaction fires and removes the tuple. In the first example, however, this is not the case. Because the "reactive" thread is periodically sleeping, it is possible that tuples will be inserted and removed before the thread awakens to check for matches. Because this behavior is built into the EgoSpaces system, its actions can be optimized. For example, instead of having to gather all of the possible matches at the reference agent each time before determining if the tuples have previously been processed, EgoSpaces can perform this check at each remote host, before returning the tuples as triggers to the reference host. Finally, the application programmer has encapsulated the reaction and can reuse it on other views if desired.

3.1.2 Transactions

From an agent's perspective, performing several operations sequentially is not an atomic action because other agents' operations can interleave with its operations. For example, if an agent performs a successful **rdp** operation and then immediately attempts to **in** the same tuple, the

in operation might be unsuccessful if another agent has, in the meantime, removed the tuple from the tuple space. At times, an application agent may want to guarantee that a sequence of operations is atomic with respect to all other operations on the involved views. For example, if an application wants to replace a piece of data with an update, but does not want it to ever appear to the world that the data is unavailable, it needs to group the removal and the replacement operation into a single atomic step. To accomplish this, we introduce the notion of *transactions* to EgoSpaces.

A transaction is a named sequence of simple actions that can include plain code, atomic or scattered probing operations, and tuple creation. Transaction code can access local state variables to save information retrieved from the tuple space inside the transaction. Because transactions must complete, they cannot include blocking operations that could halt the transaction indefinitely. Transactions are individual atomic actions; their intermediate results are not visible from the outside.

When creating a transaction, the reference agent provides a view restriction listing the involved views and serving as a contract between the reference agent and EgoSpaces. Any attempt inside the transaction to perform operations outside the view restriction generates an exception. The view restriction makes a deadlock-free implementation of the transaction mechanism possible (see Section 5).

A transaction takes the form:

$$T = \textbf{transaction over } v_1, v_2, \ldots \\ \textbf{begin } op_1, op_2, \ldots \textbf{ end}$$

where T is the transaction's name; $v_1, v_2 \ldots$ is the view restriction; and $op_1, op_2, \ldots$ is the sequence of operations. An agent executes a transaction using:

$$\textbf{execute } T$$

3.1.3 Augmenting Reactions

The reactive construct introduced previously was limited because the only actions that an agent could perform in response to a trigger tuple were the removal of the tuple and the output of an augmented version of the trigger tuple. More powerful reactive constructs that allow application agents to perform arbitrary actions in response to the trigger provide a more useful and flexible programming construct. Next, we introduce an extension of a reaction to a local trigger using a transaction. After that, we show how the semantics change if the trigger is a non-local tuple.

Transactions can extend reactions to allow them to perform a varied set of operations over any of the reference agents' views. The reaction's triggering, optional trigger removal, optional **out**, and transaction are performed as a single atomic action. For the first case we consider, the trigger must be located in the reference agent's tuple space so that the agents involved in the transaction can be locked in order (thus preventing deadlock in the system). An extended reaction has the form:

$$\rho = \textbf{react to } p \text{ [}\textbf{remove}\text{] [}\textbf{and out}(\textit{tuple\_modifiers}(\tau))\text{]} \\ \textbf{extended by } T(\tau)$$

where T is the transaction that executes in response to the trigger, and τ is a local variable bound to the trigger tuple. An agent enables an extended transaction using:

$$\textbf{enable } \rho \textbf{ with } \textit{sched\_modality}, \textit{priority} \textbf{ over } \nu_l$$

Upon enabling, EgoSpaces verifies that ν_l is a local view, restricted in scope to only the reference agent.

An agent may desire the same style of interaction in response to remote agents' tuples. In this case, however, trigger, removal, and notification are a single atomic action, while the execution of the associated transaction is a separate atomic action. The most important ramification of this subtle difference is that the trigger might not be available to the transaction when it executes because other operations can interleave with the reaction's triggering and the transaction. The transaction does, however, receive a copy (τ) of the trigger tuple. This more generalized reaction has the form:

$$\rho = \textbf{react to } p \text{ [}\textbf{remove}\text{] [}\textbf{and out}(\textit{tuple\_modifiers}(\tau))\text{]} \\ \textbf{followed by } T(\tau)$$

The use of the word **followed** in place of **extended** indicates the separation of the triggering of the reaction and the execution of the transaction. The enabling mechanism for generalized reactions is identical to basic reactions.

Programming Example. Imagine an agent that averages temperatures generated by sensors on the current floor over the past hour and replaces all of the old temperature readings with an average reading. Assume that temperature readings are generated once a minute. To implement this behavior without reactions, a programmer writes something like:

```
$\nu$ = [Celsius temperature data on current floor]
p = $\langle(\textit{timestamp}, \textit{time},$ minutes = :00$)\rangle$
seenTuples = new Vector()
while(true)
   data = $\nu$.rdp(p)
   if data $\neq$ null
      if !seenTuples.contains(data)
         $p_1$ = $\langle(\textit{tempValue}$, any, any$)$,
               $(\textit{timestamp}, \textit{time},$ [within past hour]$)\rangle$
         temps[] = $\nu$.inpg($p_1$)
         avg = average(temps[])
         average = [tuple with average information]
         out(average)
         seenTuples.add(data)
   else
      sleep(time)
```

With the built-in construct, however, the code consists of defining and enabling the reaction:

```
$\nu$ = [Celsius temperature data on current floor]
p = $\langle(\textit{timestamp}, \textit{time},$ minutes = :00$)\rangle$
$T(\tau)$ = transaction over $\nu$
         begin
            $p_1$ = $\langle(\textit{tempValue}$, any, any$)$,
                  $(\textit{timestamp}, \textit{time},$ [within past hour]$)\rangle$
            temps[] = $\nu$.inpg($p_1$)
            avg = average(temps[])
            average = [tuple with average information]
            out(average)
         end
$\rho$ = react to p followed by $T(\tau)$
enable $\rho$ with eager, 1 over $\nu$
```

In this code, the programmer explicitly declares the views over which its operations will act. If the operations do not hold to this contract, the system generates an exception. This contract, however, allows the system to provide atomicity guarantees associated with the execution of the operations; in the latter case, all n operations of the transaction are executed as a single atomic step, while in the hand-coded case, each operation may be interleaved with other operations issued by this agent or other agents in the system.

3.2 Behavioral Extensions

The reactive constructs make programming within the EgoSpaces system a more flexible task. They also provide more powerful semantics and guarantees. Most importantly, they allow agents to define general-purpose responses to trigger tuples in a view, because any code can appear inside the reaction's transaction component. In some cases, the types of actions an application performs in response to a trigger tuple will be common with other applications. For example, an application may want to duplicate certain data it encounters. This section classifies three such behaviors and shows how the semantics of these new constructs can be expressed in terms of the previously defined reactive construct.

Building these types of behaviors into the system reduces the programming burden in common cases. In this section, we describe data migration, data duplication, and event capture. We also leave the system open to extension as additional useful coordination mechanisms arise in the future.

A reference agent attaches behaviors to views, and, as long as the behavior is enabled, encountering certain conditions triggers an automatic action. In general, behaviors share several key components. First, a behavior responds to a trigger, identified via a pattern. Once enabled, EgoSpaces compares the behavior's pattern to tuples in the view and triggers the behavior whenever the pattern is matched. Like reactions, behaviors respond once to each matching tuple. Again, if tuples leave the view and return or the behavior is disabled and re-enabled, the behavior executes again.

Like reactions, behaviors can be either eager or lazy indicating when the behaviors occur. Eager behaviors execute as soon as the trigger is matched, and only other eager constructs can preempt them. Lazy behaviors will eventually execute if the behavior remains enabled and the trigger stays present. Behaviors can also include tuple modifiers, which allow the reference agent to insert, change, or remove fields in resulting local tuples. How this is used will become apparent as we present the different behaviors. Finally, behaviors have an optional transaction executed at the behavior's completion. In general, behaviors take the form:

$$\beta = \texttt{act}(p)\ \ [\mathit{tuple\_modifiers}(\tau)]\ \ [\textbf{followed by}\ \ T(\tau)]$$

where `act` is the name of the behavior (e.g., "migrate" or "duplicate"). Names must be agreed upon to allow access control implementation. A reference agent must identify which behaviors it might attach to a view. Contributing agents consider this set when evaluating access control functions. Reference agents enable and disable behaviors using:

$$\begin{array}{l}\textbf{enable}\ \beta\ \ \textbf{with}\ \mathit{sched\_modality}\ \textbf{over}\ \nu\\ \textbf{disable}\ \beta\ \textbf{over}\ \nu\end{array}$$

We discuss each behavior individually, providing a brief description and syntax. We then show the behaviors' semantics by reducing them to reactions and transactions. For each behavior, we also include a programming example to show how the behavior is used.

3.2.1 Data Migration

Mobile agents encounter a lot of data, but both data and agents are constantly moving. A particular agent may want to collect certain data without having to explicitly read each piece immediately. When the consistency of data is important agents cannot make duplicates of data items and operate on them because other agents might operate on the originals. A common solution is replica management, where multiple copies of data are kept consistent, but this solution is undesirable in ad hoc environments because agents carrying originals and duplicates meet sporadically and may never be in contact again. Transparent data migration, which avoids replicating the data, offers a solution. In this scheme, only one copy of the data item exists, and the migration behavior allows an agent to collect data matching a particular pattern. For example, building engineers might respond to work orders generated by distributed components sensing particular needs. A single engineer should take responsibility for each work order because if multiple engineers pick up the same job, work will be wasted. When an engineer encounters a work order he should perform, the work order should move from the component generating it to the engineer.

When a migration is enabled, all tuples in the view matching the pattern automatically move from their current location to the reference agent's local tuple space. Because EgoSpaces evaluates contributing agents' access control functions before determining which tuples belong to the view, contributing agents implicitly allow tuple transfer. Once migrated, the tuples become subject to the reference agent's access controls. This may affect the contents of other views defined by the reference agent or other agents. If desired, a migration uses tuple modifiers to change migrated tuples. For example, an engineer's agent that collects work orders might want to mark the migrated tuples as "assigned" to the engineer so that when others are encountered in the system, the work orders are not migrated again.

Semantics. A migration reduces to a reaction that removes the trigger and generates a new tuple in the reference agent's tuple space:

$$\begin{array}{l}\mathcal{M} = \texttt{migrate}\,p\ \ [\mathit{tuple\_modifiers}(\tau)]\\ \qquad\triangleq \rho_m = \textbf{react to}\,p\ \ \textbf{remove and out}(\mathit{tuple\_modifiers}(\tau)))\end{array}$$

If the programmer supplies the optional tuple modifiers, the tuple placed in the local tuple space is the trigger tuple with the tuple modifiers applied. Otherwise, the tuple is exactly the trigger tuple. Even though the migrated tuple is the same tuple (unless the tuple modifiers change the ID), tuple migration may trigger reactions in the new location that have already fired for the tuple in the previous location. As described in Section 4, this is because bookkeeping for reactions is done on a per-agent basis.

Enabling a migration with a particular scheduling modality reduces to enabling the above reaction using the same scheduling modality and a low priority (e.g., 10):

$$\begin{array}{l}\textbf{enable}\,\mathcal{M}\ \textbf{with}\ \mathit{sched\_modality}\ \textbf{over}\ \ \nu\\ \qquad\triangleq \textbf{enable}\,\rho_m\ \textbf{with}\ \mathit{sched\_modality}, 10\ \ \textbf{over}\ \nu_r\end{array}$$

where ν_r is the same view as ν with an added agent constraint that eliminates the reference agent from the view. An example of this view definition is seen below in the programming example. This prevents the EgoSpaces system from "migrating" tuples that are already local. In providing behaviors, EgoSpaces uses a scheduling scheme that maximizes the number of behaviors that execute, e.g., the system ensures that duplicates are made before tuples migrate. A migration's low priority allows other reactions and behaviors of the same modality to trigger first. If any of these actions remove the tuple, however, the migration will not occur.

Programming Example. The following code shows how a programmer would accomplish migration using only the basic EgoSpaces constructs. This code implements the work order collection application described above.

```
$\nu$ = [work orders on this floor and adjacent ones]
$\nu_r$ = [data in $\nu$ not owned by this agent]
p = $\langle(taken, boolean, =\text{false})\rangle$
while(true)
    data[] = $\nu_r$.rdgp(p)
    if data $\neq$ null
        for i=1 to data.length
            $\nu$.inp(data[i])
            data[i].changeField(taken, true);
            out(data[i])
    else
        sleep(time)
```

The tuple output to the tuple space has the same id as the one read in, but the "taken" field has been set to true. This code shows just one possible implementation; the actual code depends on the application's real-world needs and desired behavior. As with the reactive constructs, it is possible that this implementation will miss matching tuples if they happen to appear and disappear while the thread is sleeping. To ensure that local tuples are not infinitely migrated, the programmer must explicitly define ν_r, or the remote portion of a view ν. In this example, the definition of ν_r prevents tuples in the local tuple space (e.g., work orders created by this engineer that other engineers should perform) from being "migrated" to their current host.

To accomplish a similar thing using the migration behavior, a programmer must create and enable the behavior over the view. In this process, the declaration of ν_r is hidden from the programmer.

```
$\nu$ = [work orders on this floor and adjacent ones]
p = $\langle(taken, boolean, =\text{false})\rangle$
$t\_m(t) = \{t.\texttt{changeField(taken, true)}\}$
$\mathcal{M}$ = **migrate** $p$ $t\_m(t)$
**enable** $\mathcal{M}$ **with** eager **over** $\nu$
```

As before, the semantics of this operation differ from that of the hand-coded migration. Because this behavior is integrated with the system, we can guarantee, for eager migrations, that tuples are migrated if they appear in the reference agent's view. Again, this guarantee is conditional on no other reactive constructs removing the tuple first.

3.2.2 *Data Duplication*

Under different circumstances than the above, data availability is more important than data consistency, and an application would rather duplicate the data items to make them available upon disconnection, with the knowledge that duplicates will not remain consistent with the originals. A duplication behavior copies tuples matching some pattern, and places the copies in the reference agent's local tuple space, leaving the originals unaffected. In our example application, the building engineer may collect sensor data for processing off-site. The engineer does not, however, want to remove the data because others may need it.

Duplicated tuples may match the original view specification and be infinitely duplicated. An agent can prevent this using tuple modifiers, e.g., by tagging all duplicates with a new field. Also, duplicated tuples may satisfy view specifications of other agents. As was the case with the migration behavior, applications can deal with these concerns individually using the tuple modifiers. Copies become the responsibility of the owning agent. Before these tuples appear in any views, EgoSpaces evaluates the owning agent's access control function. Again, because replica management proves too costly, duplicates do not remain consistent with originals, even if both persist in the view.

In some application instances, an agent may want to respond to the appearance of a particular tuple and generate an entirely new tuple in response, a process known as data generation. The data duplication behavior can accomplish this behavior by using the tuple modifiers to remove all of the fields and add all new fields.

Semantics. Duplication reduces to a reaction that does not remove the trigger and generates a new tuple in the reference agent's tuple space. This new tuple must have a new, unique ID, which is generated by the reference agent.

$$\begin{aligned}\mathcal{D} = \texttt{duplicate}\; p\;\; & \mathit{tuple\_modifiers}(\tau)\\ \triangleq\; & \mathit{tuple\_modifiers}'(\tau) = \{\tau.\texttt{newID}()\}\\ & \rho_d = \textbf{react to}\; p\; \textbf{and}\\ & \qquad \textbf{out}(\mathit{tuple\_modifiers}(\tau) \cup \mathit{tuple\_modifiers}'(\tau))\end{aligned}$$

A duplication which specifies no tuple modifiers creates an exact copy (with a new tuple id), while one that adds a field "copied" marks all duplicates.

Enabling a duplication reduces to enabling the above reaction with the provided scheduling modality and a high priority (e.g., 1):

$$\begin{aligned}&\textbf{enable}\; \mathcal{D}\; \textbf{with}\; \mathit{sched\_modality}\; \textbf{over}\; \nu\\ &\qquad \triangleq \textbf{enable}\; \rho_d\; \textbf{with}\; \mathit{sched\_modality}, 1\; \textbf{over}\; \nu\end{aligned}$$

We use a high priority to ensure that duplication, which does not affect the original tuple, occurs before other actions, e.g., migration.

Programming Example. Programming the duplication behavior using only the basic EgoSpaces operations is similar to programming the migration behavior except that the tuple is not removed from its remote location, and the new tuple created in the reference agent's tuple space should have a new unique ID. Imagine an engineer that wants to duplicate the structural integrity data it encounters on the current floor and the adjacent floors and then return to his office to process it. His agent would have code similar to:

```
$\nu$ = [structural agent data on this floor and adjacent ones]
p = $\langle(strain, number, \text{any}), (acoustic\ emission, number, \text{any}), \ldots\rangle$
seenTuples = new Vector()
while(true)
    data[] = $\nu$.rdgp(p)
    if data $\neq$ null
        for i=1 to data.length
            data[i].newID()
            out(data[i])
            seenTuples.add(data[i])
    else
        sleep(time)
```

Again, accomplishing a similar functionality using the built-in EgoSpaces duplication behavior requires much less code and reduces to defining a view, creating a duplication behavior, and enabling it on the view.

```
$\nu$ = [structural agent data on this floor and adjacent ones]
p = $\langle(strain, number, \text{any}), (acoustic\ emission, number, \text{any}), \ldots\rangle$
$\mathcal{D}$ = **duplicate** $p$
**enable** $\mathcal{D}$ **with** eager **over** $\nu$
```

As in the previous examples, this behavior is enabled as an eager one over the view. In this case, it is guaranteed that the behavior will duplicate all matching tuples that appear in the view without missing any, while the hand-coded example may miss some matching tuples. If this duplication behavior is enabled as a lazy behavior, the semantics become identical to those of the hand-coded example.

3.2.3 Event Capture

The EgoSpaces primitives, reactions, and behaviors all operate over the state of the system by interacting with the available data. Many applications also benefit from the ability to react to events raised in the system. For example, an agent might want to be notified when another agent accesses a particular piece of its data. In our system, examples of events include the arrival of a new view contributer and another agent's data access operations. As an example, the engineer might adapt his behavior in response to the arrival of a building inspector.

EgoSpaces events are special tuples. An agent registers its interest in an event via a pattern over such tuples, and these interests are propagated to view participants. Once registered, event notifications for events matching the pattern propagate from the location of the event to the reference agent. To prevent superfluous event generation, EgoSpaces raises event tuples only for specific registrations, and the event's callback execution consumes the event tuple created for it. This allows multiple registrations for the same event, even by multiple agents such that when a matching event occurs, all parties registered for it receive notification. A reference agent uses a transaction to specify the event's callback, which executes after the reference agent receives the notification. Section 4 covers the details of event tuples and the mechanism for raising them.

Semantics. The event behavior reduces to a pair of reactions. The first generates a copy of the event tuple augmented with the id of this event registration and places it in the reference agent's local tuple space. The second reacts to the generated tuple and executes the callback:

$$\begin{aligned}
&\mathcal{E} = \texttt{event}(p)\ \textbf{followed by}\ T_e(\tau)\\
&\quad \triangleq \mathit{eid} = \textbf{new}\,\mathit{event\ id}\\
&\qquad \rho_{e1} = \textbf{react to}\,p\ \textbf{and out}(\tau \oplus \{(\texttt{eID}, \mathit{event\ id}, \mathit{eid})\})\\
&\qquad \rho_{e2} = \textbf{react to}\,(p \oplus \{(\texttt{eID}, \mathit{event\ id}, = \mathit{eid})\}\textbf{remove}\\
&\qquad\qquad \textbf{extended by}\ T_e(\tau)
\end{aligned}$$

The $\oplus$ indicates that the provided field, in this case the new event id, is added to the tuple or template. The generation of the event copy and the callback execution are not an atomic action. However, as long as the reference agent prevents other agents from stealing its event tuples (by using its access control function), this should not pose a problem.

Enabling an event behavior reduces to enabling the two reactions defined above:

$$\begin{aligned}
&\textbf{enable}\,\mathcal{E}\ \textbf{with}\ \mathit{sched\_modality}\ \textbf{over}\ \nu\\
&\quad \triangleq \textbf{enable}\,\rho_{e1}\ \textbf{with}\ \mathsf{eager}, 1\ \textbf{over}\ \nu\\
&\qquad \textbf{enable}\,\rho_{e2}\ \textbf{with}\ \mathit{sched\_modality}, 1\ \textbf{over}\ \nu_l
\end{aligned}$$

The first reaction (generating a personal copy of the event) is enabled with eager modality and high priority, guaranteeing the reference agent receives notification of the event regardless of the behavior's modality. The second reaction's scheduling modality corresponds to the behavior's modality and also executes at high priority. This reaction is enabled on a local view (ν_l) defined specifically for this behavior whose constraints cause it to contain only event tuples local to the reference agent matching p.

Ths behavior's semantics differ slightly from the others. Every event behavior, eager or lazy, is guaranteed to be triggered because a trigger event tuple is created specifically for each registration. In the lazy case, however, by the time the agent's callback executes, the agent or other entity that caused the event may be no longer connected.

This reduction assumes mechanisms exist to generate events and clean up event tuples. The former is discussed in Section 4, and the latter is accomplished by a reaction that removes event tuples:

$$\rho_{gc} = \textbf{react to}\,p\ \ \textbf{remove}$$

where p matches any event tuple, for example, $p = \langle(\mathit{eventType}, \mathit{string}, \mathsf{any})\rangle$. This reaction is enabled with eager modality and a priority of at least 2, guaranteeing all event copies have been generated (at priority 1):

$$\textbf{enable}\,\rho_{gc}\ \textbf{with}\ \mathit{sched\_modality}, 2\ \ \textbf{over}\ \nu_e$$

Every agent automatically has this reaction defined and enabled on its event view, so it is not necessary for an agent to redefine this reaction each time it enables an event behavior.

Programming Example. Because the event capture behavior requires the addition of an event generation mechanism, there is no way to accomplish this same behavior using the initial EgoSpaces operations. In this section, therefore, we show only an example use of the event capture behavior. Assume that a tuple indicating the arrival of a new host is represented with an event tuple similar to the following:

$$\langle(\mathit{eventType}, \mathit{string}, \mathit{hostArrival}), (\mathit{ID}, \mathit{HostID}, \mathit{newHost}), \ldots\rangle$$

Then if the building engineer wants to receive a notification of the arrival of an inspector on adjacent floors, his application agent would have to implement something like the following code:

```
$\nu$ = [this floor and adjacent ones]
p = $\langle(\mathit{eventType}, \mathit{string},\ =\text{hostArrival})\rangle$
$T_e(\tau)$ = **transaction over** null
        **begin**
            [display message to user]
        **end**
$\mathcal{E}$ = event(p) **followed by** $T_e(\tau)$
**enable** $\mathcal{E}$ **with** eager **over** $\nu$
```

The null view restriction indicates that the operations in the transaction do not affect any views.

4. DESIGN STRATEGIES

The design and implementation of the EgoSpaces extensions discussed previously build on the current EgoSpaces prototype. In some cases (e.g., event generation, reaction registration), the new features are integrated into the core system, while others build on top of this integrated system.

4.1 View Construction and Maintenance

View construction and maintenance protocols directly influence the operations' implementations. Inefficient view building limits performance. Our initial efforts led to the development of a network abstractions protocol that, given neighborhood restrictions requested by the reference agent, provides a list of qualifying agents, represented as a tree. For details of this protocol, see [14]. In short, the protocol builds the tree and maintains it in the face of mobility. Other view maintenance protocols under development may provide some uncertainty regarding the view participants, allowing similar implementations with slightly weaker semantics.

4.2 Blocking Operations

An efficient implementation of blocking operations takes advantage of reactions to prevent expensive polling. For example, an **ing** operation entails a (low priority) eager reaction that does not remove its trigger. When this reaction fires, a transaction follows and attempts an **inpg**. If this operation returns anything other than ϵ, the operation returns and disables its associated reaction. If the operation is unsuccessful, another operation removed the tuple first. Because these operations are serializable with respect to the view, this is within the operation's semantics.

4.3 Atomic Probe Operations

Atomic probes equate to transactions performed over a single view. They require locking all view participants, performing the operation, and unlocking the participants. This locking mechanism is discussed below in the description of the transaction implementation. Once the agents are locked, the query is sent to all participants, who return copies of all matches. For single operations, a tuple is chosen nondeterministically and returned; **in** operations also remove the tuple from the tuple space. Group operations return all matches found. Agents benefit from intelligent view definition, as this type of operation becomes costly on views involving large numbers of agents.

4.4 Scattered Probe Operations

A variety of possible implementations for scattered probes exist because they perform a best effort search for a match. The simplest implementation polls the view's participants in order (by id). When a match is found, it is removed if the operation is an **in** and returned. If all participants have been queried and no match found, the operation returns ϵ. Group operations query all participants and return all matches. More sophisticated implementations of the single operations can take advantage of the environment; one might query the physically closest agents or the agents with the highest bandwidth connections first.

4.5 Transactions

A transaction must operate over several views with explicit guarantees that its internal state is not visible from outside. As such, transactions are inherently costly. EgoSpaces reduces this cost by requiring a reference agent to explicitly declare what other agents need to be blocked for the duration of the transaction by providing a list of views over which to execute the transaction. Because of the underlying view maintenance, the agents contributing to each view are known, and EgoSpaces can create an ordered list of them (ordered by id). EgoSpaces then locks the transaction's participants (including the reference agent) in order. If any other agent also performs a transaction, it will lock agents in the same order, avoiding deadlock. If a contributing agent moves out of the view while a transaction is locking agents, it must be unlocked before departing. If the transaction's operations are already executing, the agent's departure must be delayed until the transaction's completion. We assume enough time to complete the transaction before the agent disappears from communication range. Such a guarantee can be provided using *safe distance* [13]. If a new agent moves into the view while the reference agent is performing a transaction, its arrival is delayed until the transaction completes.

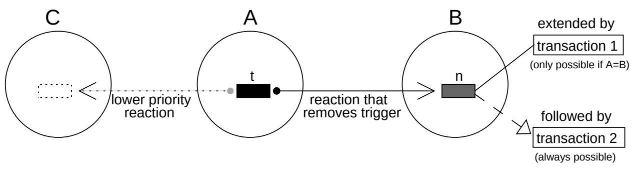

Figure 2: The Reaction Mechanism

4.6 Reactions

Because the reactive mechanism lies at the core of the EgoSpaces extensions, an efficient reaction implementation is essential to a successful system. This includes the propagation of registrations and the return of notifications.

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 insert in the tuple space when the reaction fires (if any), and the transaction that extends or follows this reaction (if any). A reaction list entry contains the reaction's id, 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 reaction registration, the message propagates to all view participants (discussed below) and is inserted in each participant's reaction list. Upon registration, all tuples in the view are checked against the pattern. For all tuples that match, the reaction fires. This firing sends a notification (containing a copy of the trigger) to the registering agent. If specified, the tuple is removed from the tuple space. While the reaction remains enabled, new tuples that satisfy the view specification are checked against the pattern. For each match, the registering agent receives a notification and locates the reaction in the reaction registry. If necessary, it performs the appropriate **out** operation and either executes or schedules any associated transaction.

Figure 2 shows the reaction mechanism. Agents B and C register reactions on agent A, which both match t. The reaction with the highest priority (B's reaction) fires first, generating notification n for B. Because this reaction removes the trigger, 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 a local tuple (i.e., A=B).

Reactions are treated as persistent operations by the view maintenance protocol. During the view's construction, new agents receive the reaction registration and add it to their reaction list. As new agents move into the view's scope, they receive any registered reactions. As agents move out of the view, they remove information regarding the reference agent's registered reactions. If these agents return, they receive the registrations and fire the associated reactions again if matching tuples exist.

Because reactions should only fire once per registration per matching tuple, we introduce a bookkeeping mechanism to maintain the ids tuples that have already been processed. Each reaction contains one data structure per agent to monitor this. When the reaction fires for a trigger tuple, the id of that tuple is stored. Before firing for a trigger, the system checks this data structure. If the reference agent deregisters the reaction, this data structure is deleted and recreated if the reaction is reregistered. In this case, the reaction will refire for duplicate tuples. The behaviors also affect reaction triggering; e.g., if a tuple triggers a registered reaction and then migrates to another agent where the reaction is also registered, the reaction will fire again.

4.7 Behaviors

Because the semantics of behaviors are written as reactions, their implementation relies on the reaction's implementation. Again, the key reason for building these behaviors into the system is to provide common actions as simple operations and to allow for code encapsulation and reuse.

4.8 Event Generation

To successfully implement the event capture behavior, we must add an event raising mechanism to EgoSpaces. In this scheme, event tuples define system level events. Recall the example in which the building engineer responded to the event generated when an inspector arrived. Some example event types include host arrival and departure, agent arrival and departure, and data access operations. Each type of operation has a defined type string (e.g., *hostArrival*) and some secondary information (e.g., the *HostID* for a host arrival or departure event).

The event generation mechanism we add to EgoSpaces raises events only if some reference agent has registered for the event. Upon generation, special event tuples are created for each registered agent, and these tuples are transmitted to the agent. The event's callback then executes according to the registration's modality (eager or lazy).

5. CONCLUSION

The success of a coordination middleware for ad hoc mobile environments lies in its ability to address the key issues of this constrained environment. First, the vast amount of information available necessitates mechanisms to easily and abstractly limit one's operating context. Second, the variety of applications forces the middleware to provide programming abstractions tailored to specific application domains while remaining general enough to maintain a small footprint on devices with constrained memory requirements. Finally, the communication restrictions and responsiveness requirements inherent in wireless applications directs design. The original EgoSpaces model began to address the first of these three concerns. The additional constructs and behavioral extensions introduced in this paper complete this task and provide the needed high-level coordination mechanisms. The reduction of the behaviors into a unifying construct, the reaction, decreases the required middleware support. Finally, our approach to protocol development and implementation focuses on limiting communication overhead and increasing the operations' responsiveness. With such a direct attack on complexities specific to ad hoc mobile networks, EgoSpaces and its extensions promise to transform application development in this environment. Additionally, this paper shows how these behavioral extensions can serve as a powerful abstraction for practical systems. An equally interesting domain treats these behavioral extensions as a computational model for investigating well known problems like routing and self-stabilization in novel ways.

ACKNOWLEDGEMENTS

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 National Science Foundation or the Office of Naval Research.

6. REFERENCES

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

[2] J. Broch, D. B. Johnson, and D. A. Maltz. The dynamic source routing protocol for mobile ad hoc networks. Internet Draft, March 1998. IETF Mobile Ad Hoc Networking Working Group.

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

[4] 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 *Proceedings of MobiCom*, pages 20–31. ACM Press, 2000.

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

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

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

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

[9] Y. Ko and N. Vaidya. Location-aided routing (LAR) in mobile ad hoc networks. In *Proc. of MobiCom*, pages 66–75, 1998.

[10] 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 Dist. Comp. Systems*, pages 524–533, 2001.

[11] V. Park. and M. S. Corson. Temporally-ordered routing algorithm (TORA) version 1: functional specification. Internet Draft, August 1998. IETF Mobile Ad Hoc Networking Working Group.

[12] G. P. Picco, A. L. Murphy, and G.-C. Roman. On global virtual data structures. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, pages 11–29, 2002.

[13] G.-C. Roman, Q. Huang, and A. Hazemi. Consistent group membership in ad hoc networks. In *Proc. of the 23rd Int'l. Conf. on Software Engineering*, 2001.

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

[15] E. Royer and C.-K. Toh. A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications*, pages 46–55, 1999.

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

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

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