Report Number: WUCSE-2002-34

2002-09-19

Using EgoSpaces for Scalable, Proactive Coordination in Ad Hoc Networks **PLEASE SEE WUCSE-03-11**

Gruia-Catalin Roman and Christine Julien

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... **Read complete abstract on page 2.**

Using EgoSpaces for Scalable, Proactive Coordination in Ad Hoc Networks **PLEASE SEE WUCSE-03-11**

Gruia-Catalin Roman and Christine Julien

Complete 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. The resulting model is one in which 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 are given for each behavior. Since performance is a real concern in the ad hoc environment, this paper also presents protocol implementations tailored to each behavior type.

2002-34

Using EgoSpaces for Scalable, Proactive Coordination in Ad Hoc Networks **PLEASE SEE WUCSE-03-11**

Authors: Roman, G.-C.; Julien, C.

September 19, 2002

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. The resulting model is one in which 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 are given for each behavior. Since performance is a real concern in the ad hoc environment, this paper also presents protocol implementations tailored to each

Using EgoSpaces for Scalable, Proactive Coordination in Ad Hoc Networks

Gruia-Catalin Roman and Christine Julien
Department of Computer Science and Engineering
Washington University
Saint Louis, MO 63130
{roman, julien}@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. The resulting model is one in which 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 are given for each behavior. Since performance is a real concern in the ad hoc environment, this paper also presents protocol implementations tailored to each behavior type.

Keywords

Ad Hoc Mobility, Context-Aware, Coordination, Middleware, Software Engineering

1 Introduction

The unique combination of ad hoc networks, context-aware computing, and the need for personalization in such environments drives the work presented in this paper. Mobile ad hoc networks form opportunistically and change rapidly in response to host movement. This constantly changing network necessitates the ability for applications running in it to adapt their behavior to their environment, a style of computing referred to as context-aware computing. Ad hoc routing protocols [1, 8, 11, 14], however, have expanded connectivity outside the immediately accessible wireless broadcast region, providing applications access to unmanageable amounts of data. Ad hoc context-aware applications, therefore, could benefit from the ability to define personalized contexts that include only the pieces of this maximal context that interest them.

Notions of context-awareness have been explored for both static and nomadic mobile networks [5, 16, 6, 15]. The radically different properties of ad hoc networks, however, require new context-awareness models tailored to the environment's specific complexities. While applications in static networks may optionally use context information, applications in ad hoc networks often require such knowledge. Because the nature of context-aware computing causes various applications to need access to different context information relative to each other and over time, such applications desire personalized operating contexts. The EgoSpaces model and middleware [7] introduces the novel notion of asymmetric coordination, giving each application direct control over the size and scope of its personalized context, an approach essential to accommodating programming for large, dense ad hoc networks.

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. There is a need for middleware providing high-level coordination abstractions that allow programmers to efficiently develop unique applications. This paper extends EgoSpaces to provide a variety of such mechanisms. Previous work has explored, among other things, tuple spaces [10], reactive programming [2], data hoarding [9], and event-based interactions [3]. We revisit these coordination mechanisms, injecting them with our focus on context-awareness, ad hoc networks, and personalization. The EgoSpaces extensions generalize these coordination mechanisms and provide several programming options without sacrificing the simplicity of the high-level abstractions. Of particular interest was our ability to reduce the specialized behaviors to a single construct, the reaction.

Finally, in the resource constrained environment in which we operate, increased application responsiveness and decreased communication overhead are key concerns In providing programmers with powerful higher-level of abstractions, we target opportunities for optimization, and allow these issues to drive our protocol development.

The next section reviews EgoSpaces. Section 3 adds some advanced constructs, transactions and reactions, to the basic model. Section 4 presents the behavioral extensions and shows that they can be reduced to EgoSpaces' reactions. Section 5 directly addresses performance considerations in presenting the behaviors' implementations. Conclusions appear in Section 6.

2 EgoSpaces Review

EgoSpaces, presented in [7], introduces 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 physical and logical mobility that consist of 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. Each agent manages its own data items, stored in a local tuple space.

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 data in the network. Since one's context is relative, we use the term *reference agent* to denote the agent whose context we are considering, and *reference host* refers to the reference agent's host. 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 his quadrant on the floors adjacent to his current floor. An agent running on his PDA declares the following view:

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

Figure 1 depicts this example, the shaded circle 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 all defined 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 automatically updates to include different sensors.

EgoSpaces employs an agent-specified access control function to limit the ability of other agents to access an agent's local data. When a reference agent defines a view, it attaches a set of credentials verifying itself to other agents. Additionally, the reference agent declares the operations it intends to perform on the view. When determining the contents of a view, EgoSpaces evaluates, for each tuple that meets the view specification, the contributing agent's access control function with respect to the specified list of credentials and operations. This provides EgoSpaces with a very fine grained access control mechanism. More details on view specifications, transparent maintenance, and access control can

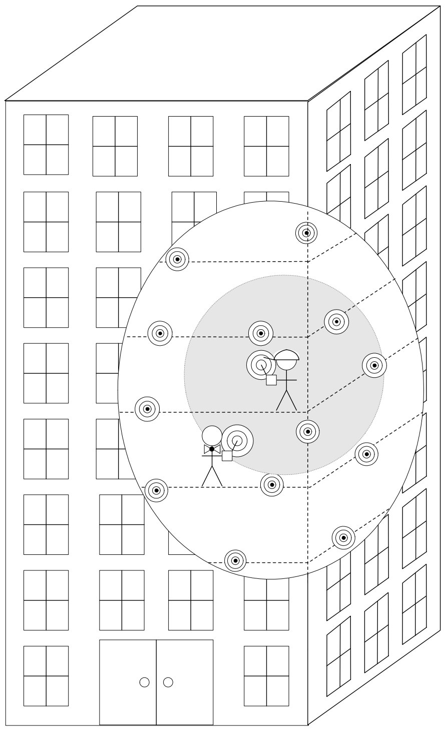

Figure 1. Example view definition.

be found in [7] and [13].

2.3 Basic Data Access Operations

EgoSpaces bases its coordination on the Linda [4] model, in which components coordinate through a global tuple space. However, like Lime [10], EgoSpaces partitions Linda's global tuple space into individual spaces distributed among mobile agents. When agents move within communication range of each other, their tuple spaces logically merge to form a single, "global" tuple space. Agents interact with tuple spaces by matching a pattern against a tuple's contents. Standard operations provided in Linda include tuple creation (**out**), tuple reading (**rd**), and tuple removal (**in**). EgoSpaces provides similar operations, but the scope of each operation is constrained to a single view.

Agents create tuples using **out** operations. A new tuple 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 exists immediately. 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 [7].

In our example, sensors use **out** operations to generate sensed information. Engineers or inspectors use **rd** and **in** operations to access data. More complicated sensors might access data, assimilate it, and generate work orders representing problems needing immediate attention. Workers can then access these work orders to obtain their tasks.

3 Advanced Constructs

All the constructs previously described involve explicit data access. If a mobile component needs to wait for a piece of data to appear before performing additional actions, it must poll. This costly and inefficient mechanism prevents the component from performing other work in the meantime. For example, all building occupants want to react to exceptional conditions (e.g., a tuple indicating a fire) so they can act accordingly. Furthermore, as described so far, EgoSpaces provides no mechanism for grouping operations in a transactional fashion. For example, a sensor may want to remove a piece of data and replace it with an update. If this piece of data is critical, the sensor needs a transaction to ensure the data's constant availability. This section introduces 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 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 including Lime [10] and MARS [2]. An EgoSpaces reaction associates a trigger (i.e., a pattern) with a set of operations. When a tuple matches the pattern the operations to execute. A reaction can read its trigger, remove its trigger from the tuple space, and output an arbitrary tuple in the reference agent's tuple space. A reaction has one of two scheduling modalities, eager or lazy, indicating when they 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. 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 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). 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 fire repeatedly. The burden of handling these cases falls on the application programmer. Reactions take the form:

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

where the local name τ is bound to the trigger; p is the reactive pattern; the keyword **remove** causes tuple removal; and the optional **out**($\mathit{tuple\_modifiers}(\tau)$) places a tuple into the reference agent's tuple space. The *tuple_modifiers* may be applied to a copy of the trigger to allow the reaction to add or remove fields in the tuple. A reference agent enables and disables a reaction using:

$$\textbf{enable}\, \rho\ \ \textbf{with}\ \mathit{sched\_modality}, \mathit{priority}\ \textbf{over}\ \nu$$

$$\textbf{disable}\, \rho\ \ \textbf{over}\ \nu$$

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 statements 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.

3.2 Transactions

An EgoSpaces transaction is a named sequence of simple actions that can include plain code, atomic or scattered probing operations, and tuple creation. Because transactions must complete, they cannot include blocking operations that could halt the transaction indefinitely. EgoSpaces prevents **out** operations from affecting the transaction by delaying them until just after the transaction's completion. Transactions are individual atomic actions; their 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:

$$\textbf{execute } T$$

3.3 Augmenting Reactions

Transactions can extend reactions to allow them to operate 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. To prevent deadlock, the trigger for this reaction must be located in the reference agent's local tuple space so that the agents involved in the transaction can be locked in order. To ensure this, EgoSpaces introduces *local views* that are restricted in scope to only the reference agent. An extended reaction has the form:

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

where T is the transaction that executes in response to the trigger. 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.

An agent may desire the same style of interaction in response to remote agents' tuples. Any tuple can trigger these more generalized reactions. 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 receives a copy (τ) of the tuple, but if the transaction attempts to read or remove it directly from the tuple space, it may not succeed. This more generalized reaction has the form:

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

The enabling mechanism for generalized reactions is identical to basic reactions.

4 Extending EgoSpaces with Behaviors

Many ad hoc and context-aware applications benefit from proactive coordination. A common paradigm in distributed programming involves data transfer from producers to consumers. In ad hoc environments, however, producers and consumers may communicate for only brief instants. Reliance on polling proves inefficient and, worse, may cause consumers to miss data. Such applications benefit from the ability to specify data items to be implicitly moved to the local repository whenever encountered. As another example, the operations covered in the previous section all focus on state. Some applications, however, require knowledge about events. In investigating application needs, we identified the following as useful coordination styles: data migration, data duplication, and event capture, and we provide them as behavioral extensions to EgoSpaces. We also leave EgoSpaces open to extension.

A reference agent attaches behaviors to views. 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—either a regular data tuple or a special tuple used for the particular behavior— identified via a pattern. Once enabled, EgoSpaces monitors both the behavior's pattern and tuples in the view and triggers the behavior whenever the pattern is matched. Like basic 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 have scheduling modalities of 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 carry a different guarantee. If the behavior remains enabled and the trigger stays present, a lazy behavior will eventually execute.

Behaviors can include tuple modifiers, which allow the reference agent to insert or remove fields in resulting local tuples. Finally, behaviors have an optional transaction executed at the behavior's completion.

In general, behaviors take the form:

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

where `act` is the name of the behavior (e.g., "migrate" or "duplicate"). Names are integral to the system and must be agreed upon to allow access control implementation. Reference agents enable and disable behaviors using:

$$\textbf{enable}\ \beta\ \ \textbf{with}\ \textit{sched\_modality}\ \textbf{over}\ \nu$$

$$\textbf{disable}\ \beta\ \textbf{over}\ \nu$$

Again, access controls must be considered. A reference agent must identify which behaviors it might attach to a view. Contributing agents consider the set of potential behaviors when evaluating access control functions.

We discuss each behavior individually, providing a brief description and syntax. We then show the behaviors' semantics by reducing them to reactions and transactions.

4.1 Data Migration

Mobile agents encounter a lot of data, but both data and agents are constantly moving. A particular agent may want to implicitly pull data towards it, without having to explicitly read each piece. Many applications require data consistency. Agents cannot make duplicates of data items and operate on them because other agents might operate on the original. A common solution is replica management, 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 offers a solution. 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.

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

$$\begin{aligned}\mathcal{M} &= \texttt{migrate}\, p \;\; \mathbf{out}(\mathit{tuple\_modifiers}(\tau)) \\ &\triangleq \rho_m = \mathbf{react\ to}\, p \;\; \mathbf{remove\ \ and\ \ out}(\mathit{tuple\_modifiers}(\tau)))\end{aligned}$$

The tuple generated is identical to the trigger tuple with the tuple modifiers applied and a new tuple id. Tuple migration may trigger reactions in the new location that have already fired for the tuple in the previous location.

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):

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

In providing behaviors, EgoSpaces uses a scheduling scheme that maximizes the number of behaviors that execute, i.e., 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.

4.2 Data Duplication

When agents want to continue to access particular pieces of data but do not want control of the originals, data duplication offers the correct solution. A duplication behavior copies tuples matching some pattern, and the copies are placed in the reference agent's local tuple space, leaving the originals unaffected. 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 application's reference 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. While some applications might desire this behavior, others may not. Again, applications can deal with these concerns individually. 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. Notice that a reference agent can also use tuple modifiers to generate entirely new tuples by removing all the fields and adding new ones.

Semantics. Duplication reduces to a reaction that does not remove the trigger and generates a new tuple in the reference agent's local tuple space:

$$\begin{aligned}\mathcal{D} &= \texttt{duplicate}\, p\ \ \mathbf{out}(\mathit{tuple\_modifiers}(\tau)) \\ &\triangleq \rho_d = \textbf{react to}\, p\ \ \textbf{and}\,\mathbf{out}(\mathit{tuple\_modifiers}(\tau)))\end{aligned}$$

A behavior with 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 occurs before migration.

4.3 Event Capture

Many coordination systems allow applications to adapt their behavior to events occurring in the system. In our system, events include the arrival of a new view contributer or another agent's data access operations. For 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 by providing a pattern over event tuples. Once registered, event notifications for events matching the pattern propagate to the reference agent. To prevent superfluous event generation, EgoSpaces generates event tuples only for specific registrations. The event's callback execution consumes the event tuple created for it, allowing multiple registrations for the same event, even by multiple agents. 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 agent after the reference agent receives the notification.

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 event registration's callback:

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

The $\oplus$ symbol indicates the tuple or template is augmented with the provided field, in this case the new event id. The generation of the event copy and the callback execution are not a single 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:

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

The first reaction (that generates a personal copy of the event) is enabled with `eager` modality and high priority. This guarantees the reference agent receives notification of the event regardless of whether the behavior is eager or lazy. The second reaction's scheduling modality corresponds to the behavior's provided 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 tuples local to the reference agent matching p augmented with an `eID` field.

This reduction assumes mechanisms exist to generate events and clean up event tuples. 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 (e.g., $p = \langle(\mathit{event\_tuple\_tag}, \mathit{string}, = \mathit{event})\rangle$). This reaction is enabled with eager modality and a priority of at least 2, guaranteeing all event copies have been generated (at priority 1):

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

5 Protocols and Implementation

View construction and maintenance protocols directly influence the implementations for the operations. Inefficient view building limits performance. Our initial efforts led to the development of network abstractions [13] that, given neighborhood restrictions requested by the reference agent, provide a list of qualifying agents.

The list corresponds to a tree constructed for the reference agent over its operating context. In any network, both hosts and links between them have attributes affecting network communication, including link bandwidth, battery power, and signal strength. A reference agent specifies which of these properties contribute to a weight for each link. Once a weight has been calculated for each link, an application-specified cost function over these weights determines the cost of network paths. We build a tree rooted at the reference host including only the lowest cost path to each host in the network. Because we aim to restrict the scope of a reference agent's view, calculating the cost to every host in the network is unreasonable. To limit the view to some manageable region, the application specifies a bound for its cost function. Agents on hosts to which the cost is less than the bound are included in the view. Once the computation reaches a host outside the bound, all hosts farther on the same path must also lie outside the bound. To guarantee this, we require the cost of a given path strictly increase with the number of hops from the reference host. As an example, an entire specification for restricting agents within a certain number of hops would be written as: all nodes which can be reached in fewer than five hops.

We developed a protocol that builds this tree, and an additional protocol maintains the tree in the face of mobility. This maintenance is required for views over which the reference agent registers persistent operations (e.g., reactions and behaviors). We also have under development a mesh-based protocol which maintains multiple qualifying paths for certain hosts and should provide more reliable communication. Other view maintenance protocols may provide some uncertainty regarding the view participants, allowing us to implement very similar operations with slightly weaker semantics.

5.1 Blocking Operations

Blocking tuple space operations are implemented as 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 reaction (or another **in** operation) removed the tuple first. This is within the operation's semantics.

Other blocking operations can be implemented using similar techniques.

5.2 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 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 non-deterministically and returned; **in** operations also remove the tuple from the tuple space. Group operations return all matches found, and **ingp** also removes the matches. It becomes obvious that a reference agent benefits from the intelligent definition of its views, as this type of operation becomes costly on views involving large numbers of agents.

5.3 Scattered Probe Operations

A variety of possible implementations for scattered probes exist. As described before, scattered probes perform a best effort search for a matching tuple. The simplest implementation polls the view's participants in order (by id) for a matching tuple. When one 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. For example, one might query the physically closest agents or the agents with the highest bandwidth connections first.

5.4 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. EgoSpaces then locks the transaction's participants (including the reference agent) in order.

If any other agent also performs a transaction, they will lock agents in the same order, avoiding deadlock. If, while an agent is locking agents for a transaction, a contributing agent moves out of the view, 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 there is enough time to complete the transaction before the agent disappears entirely from communication range. Such a guarantee can be provided using the notion of *safe distance* [12]. If a new agent moves into the view while the reference agent is in the process of locking hosts or in the middle of executing the transaction, the arrival of the agent must be delayed until the transaction completes. During its execution, the transaction's operations execute in sequence, according to the operation implementations.

5.5 Reactions

To implement reactions, 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 and the tuple that should be inserted in the tuple space when the reaction fires (if any) and the transaction that should extend or follow this reaction (if any). A reaction list entry contains the reaction's id, the owning agent's id, the reaction's pattern, the data pattern for the view, and a boolean indicating whether or not to remove the trigger. Upon reaction registration, the message propagates to all view participants (as 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 tuple) to the registering agent. If specified, the tuple is removed from the tuple space. While the reaction remains enabled, new tuples appearing 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, and the patterns of both happen to 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

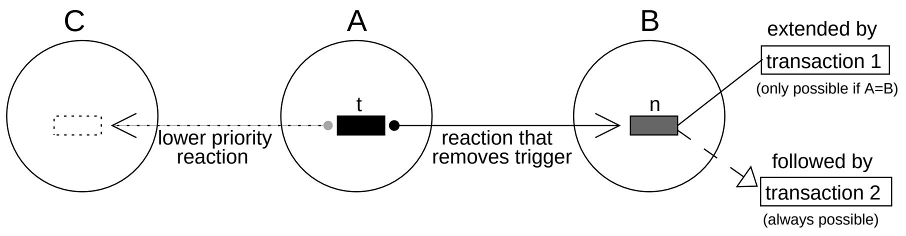

Figure 2. The Reaction Mechanism

not fire. B's reaction can be extended or followed by a transaction. The former is only allowed when the reaction is triggered by a local tuple (i.e., A=B).

Reactions are treated as persistent operations by the view building and maintenance protocols. During the view's construction, agents added to it 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 any information regarding the reference agent's registered reactions. If these agents later move back, they will receive the registrations and fire the associated reactions again if matching tuples exist.

6 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 immense variety among 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 our middleware design. The original EgoSpaces model begins to directly 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 our target environment.

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.

References

[1] 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.

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

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

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

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

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

[7] C. Julien and G.-C. Roman. Egocentric context-aware programming in ad hoc mobile environments. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, November 2002. (to appear).

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

[9] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 264–275, October 1997.

[10] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing 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.-C. Roman, Q. Huang, and A. Hazemi. Consistent group membership in ad hoc networks. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[13] G.-C. Roman, C. Julien, and Q. Huang. Network abstractions for context-aware mobile computing. In *Proceedings of the 24th International Conference on Software Engineering*, pages 363–373, May 2002.

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

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

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