Report Number: WUCSE-2003-63

2003-09-24

A Software Engineering Perspective on Context-Awareness in Ad Hoc Mobile Environments

Christine Julien, Gruia-Catalin Roman, and Jamie Payton

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

A Software Engineering Perspective on Context-Awareness in Ad Hoc Mobile Environments

Christine Julien, Gruia-Catalin Roman, and Jamie Payton

Complete Abstract:

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

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-63

A Software Engineering Perspective on Context-Awareness in Ad Hoc Mobile Environments **PLEASE SEE WUCSE-04-18**

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

September 24, 2003

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

A Software Engineering Perspective on Context-Awareness in Ad Hoc Mobile Environments

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

Abstract

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

1 Introduction

Traditionally, context-aware computing refers to an application's ability to adapt to changes in its environment. For example, calendar or reminder programs [4] use time to display pertinent notifications to users. Tour guide applications [1, 3] display different information based on the user's current physical location or proximity to an attraction. Still other programs implicitly attach context information to data, for example to research notes taken in the field [10]. Each of these applications must independently gather context information from the required sensors and tailor the provision of context to its needs.

With the increasing popularity of communicating mobile devices, context-aware computing has moved from a target environment of an autonomous device gathering information single-handedly to a sophisticated network of connected devices, all providing context information to each other. A key difference between our approach and previous work lies in our extension of context to include not only sensor information but arbitrary data resources in the surrounding physical neighborhood. This supports development of powerful context-aware applications that allow complex device interactions. In this environment, an application's behavior may depend on information collected from hosts several hops away.

Given this expanded context, the range of applications a developer can provide multiplies to include applications that utilize high-level component coordination. For example, an application monitoring traffic conditions on a highway can collect traffic observations from cars in its direction of travel. This information is displayed to the driver, and it can also help the driver plan an alternate route. As another example, a set of robots exploring the surface of a planet might each be assigned an experimental task. The robots carry different sets of sensors that provide varying types of information. A single robot on the planet might utilize sensor information from other robots, selected in a context-sensitive manner.

Potential applications in many domains abound, but the difficulty of programming these applications lies in the need to manage large amounts of distributed and

transiently available context data. This challenge motivated us to develop an infrastructure that facilitates program development by hiding the details associated with mobility, distribution, and transient connectivity. We provide an application developer only that which is necessary for him to successfully operate on his needed context. Specifically, our middleware, EgoSpaces, allows applications to define individualized contexts tailored to their needs. An application may define different contexts that reflect diverse concurrent and changing needs and which encompass varying data from multiple sources. Each of these contexts may access a wide range of data, and EgoSpaces manages this information for the application. This relieves the programmer from having to open sockets between the communicating parties and manage the network disconnections common in mobile environments. The middleware's communication and management primitives facilitate rapid development of context-aware applications.

This paper presents the design, implementation, and use of an infrastructure based on the EgoSpaces model of asymmetric coordination. Section 2 briefly reviews the coordination model, highlighting how it manages sophisticated contexts. Section 3 details the mechanics of using the infrastructure to program context-aware applications. Section 4, describes sample applications, show how they rely on the infrastructure, and discuss lessons learned in their development. In Section 5, we present the middleware's design and implementation which focuses on modularity to allow reuse of components as necessary. Section 6 provides conclusions.

2 EgoSpaces Model Overview

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

The EgoSpaces model [6] uses asymmetric coordination to gather and utilize context information in an ad hoc mobile environment, i.e., each agent filters the world around it through its own unique and changing perspectives. The amount of context information ultimately available to an application may span a large network; this generates an overwhelming amount of data for the application to manage. EgoSpaces allows an individual application agent to precisely specify the context necessary for completing its tasks, and the infrastructure provides this context for the application. As the environment changes, the set of data satisfying the application's specification also changes, and the infrastructure adapts accordingly. Throughout this paper, we use the term *reference agent* to refer to the particular agent whose context we are discussing.

2.1 The View Concept

To provide scalable coordination in an ad hoc network, EgoSpaces relies on a key abstraction called a *view*. This concept is agent-centric because a reference agent defines views with respect to its individual needs for resources from and knowledge about its environment. In principle, a reference agent's context consists of all information available in the network. In practice, however, the reference agent's behavior generally relies only on information available in a region surrounding its location. An agent sees the world through a set of personalized views that it may altered at will. The software engineering gains of the view abstraction stem directly from the level of flexibility and simplicity it offers application developers.

Each view presents a projection of all data available to the reference agent. The unique properties of ad hoc mobile networks force context restriction based on attributes of the hosts and links in the network. EgoSpaces combines these network and host constraints with restrictions on the data and agents that own the data. An agent describes its personal needs through a declarative view specification. An example view specification is:

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

Network Constraints. To restrict the scope of the network, the application specifies an abstract metric over network properties. This metric calculates a logical distance from the reference host to other network hosts. The application also provides a bound over allowable distances which restricts which hosts belong to the neighborhood. The specific needs of this restriction and a protocol for providing it are detailed in [9].

Host and Agent Constraints. Host and agent constraints allow an application to restrict a view's data based on properties of the hosts and agents that hold the data. In EgoSpaces, every host and agent has a profile describing its properties. Host properties include, for example, the unique host id, the identity of the computer's owner, or services the computer

provides. Agent properties include the agent's operating host or the agent's application task. To restrict which hosts and agents contribute to its view, a reference agent provides constraints over profile properties.

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

Access Controls. In EgoSpaces, each agent specifies an individualized function that limits the ability of other agents to access its local data. From the opposite direction, when an agent specifies a view, it attaches to the view a set of credentials that verify it to other agents. The specifying agent also declares the operations it intends to perform on the view. When determining the contents of a view, EgoSpaces evaluates each contributing agent's access control function over the view's credentials and operations. The access control function is evaluated for each individual tuple, which provides a fine level of granularity.

Given these components, a view specification consists of three patterns (one over data items, one over agent profiles, and one over host profiles), the network constraints (consisting of a metric for network path costs and a bound on the metric), and an operation list and credentials that allow provision of access controls. With this information, our middleware constructs the application's desired view. Next, we discuss our use of tuple space based coordination. In the end, a reference agent's view consists of the set of tuples (data items) that satisfy the above constraints.

2.2 Tuple Space Based Coordination

In Linda [5], distributed processes use a shared tuple space to share data items, or tuples. A tuple is a list of typed fields. As discussed below, EgoSpaces extends this definition to provide more flexible coordination.

In Linda, coordinating processes interact directly with a single, centralized tuple space. Adaptations of Linda divide this tuple space to accommodate mobility of hosts and disconnected operation. MARS [2] associates a tuple space with each host in the network and allows coordination among co-located application agents. LIME [8] associates a tuple space with each agent, and the tuple space moves with the agent. In this model, the tuple space an application operates on is defined as the instantaneous union of all tuple spaces within communication range. EgoSpaces also associates tuple spaces with individual application agents because it flexibly supports both physical host mobility and logical agent mobility.

Processes place tuples in the tuple space using **out**(t) operations. Data access occurs in a content based manner by matching a tuple against a pattern, or template, constraining the values of the fields in the tuple. To provide a more flexible pattern matching mechanism, we assume a more general tuple definition.

Tuple Definition and Pattern Matching. In EgoSpaces, a tuple is an unordered set of fields, each consisting of a name, type, and value. Tuple field names must be unique; a tuple can have only one field of a given name. The use of this name field allows us to relax restrictions on tuple pattern matching.

In Linda, patterns must be the same length as the tuple, and the fields of the tuple and pattern are matched according to their order. We extend pattern matching to maintain content based matching but operate over unordered tuples. A pattern is similar to a tuple, but each field's value is replaced with a constraint that restricts the field's value. A tuple matches a pattern if, for every constraint in the pattern, there exists a field in the tuple with the same name and type. The value of the field must also satisfy the corresponding constraint function. While the matching mechanism does require that every constraint in the pattern is satisfied, it does not require that every field in the tuple is constrained, i.e., a tuple must contain exactly the fields in the pattern, but the tuple can contain additional fields.

2.3 View Operations

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

Extensions to Linda provide atomic probing operations, **rdp**(p) and **inp**(p) that carry the same atomicity guarantees as the original operations but return immediately instead of blocking. If no tuple in the view immediately matches, an empty value is returned.

Other Linda extensions utilize aggregate operations that return all matching tuples. EgoSpaces provides these operations in both blocking (**rdg**(p) and **ing**(p)) and probing (**rdgp**(p) and **ingp**(p)) forms.

Finally, in the dynamic ad hoc environment, atomic

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

Formal semantic definitions for all of these view operations can be found in [6]. Additional programming constructs such as reactive programming and behavioral extensions are also available in EgoSpaces; they are not detailed in this paper.

3 Rapid Development Potential

EgoSpaces reduces programming context-aware mobile applications to simple operations tailored to the capabilities of novice programmers. The middleware provides all network communication programming and presents the programmer with a high-level agent coordination interface. In this section, we show how EgoSpaces's programming abstractions ease programming by simplifying the programming interfaces while retaining the necessary power of coordination.

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

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

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

Figure 1. The API for the `Agent` class

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

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

Figure 2. The API for the `AgentProfile` class

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

In extending the `Agent` base class, the application agent receives two methods. The first method registers the `Agent` with the `EgoManager`, a component described in more detail in Section 5. By registering with the `EgoManager`, an application agent delegates responsibility for data management and communication. This also facilitates agent migration among hosts, which we will discuss later.

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

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

We first examine the components of the `View` constructor. The `Metric` and `Cost` allow an application to define an abstraction over the physical ad hoc network. Both of these components are part of the `NetworkAbstractions` interface. The `Metric` defines

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

Figure 3. The API for the `View` class

the costs of paths in the network based on properties of hosts and links. Based on this `Metric` and the `Cost` that defines a bound on the lengths of paths, the `NetworkAbstractions` package builds a subnet that contains exactly the hosts that satisfy the view's network constraints. EgoSpaces provides commonly used `Metric` definitions, for example, a metric based on hop count and another based on physical distance. More sophisticated applications can build their own `Metric` and `Cost` definitions by following the procedure outlined in [7]. The `HostConstraints` and `AgentConstraints` provide restrictions that hosts and agents must satisfy to contribute data to the view. Because EgoSpaces represents profiles as tuples, both types of constraints can be provided as patterns over tuples. The `DataConstraints` in a `View` specification are a pattern over data items that appear in the view.

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

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

4 Sample Applications

The best demonstration of the middleware's ability to ease context-aware application development is by example. In this section, we present two applications that take advantage of the view abstraction in networks constructed over automobiles on roadways.

4.1 Emergency Vehicle Warning System

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

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

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

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

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

Lessons Learned. The key to successful implementation of this application lies in the definition of the view. Because both the car and the emergency vehicles speeds are variable, the scope of the view depends on their velocities. Given a well-defined view, the application agent's minimal interaction with EgoSpaces

involves only simple view operations. The car is guaranteed to be notified as soon as possible of the approach of an emergency vehicle. Notification that the emergency vehicle has departed is not guaranteed to be as timely.

4.2 Subscription Music Service

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

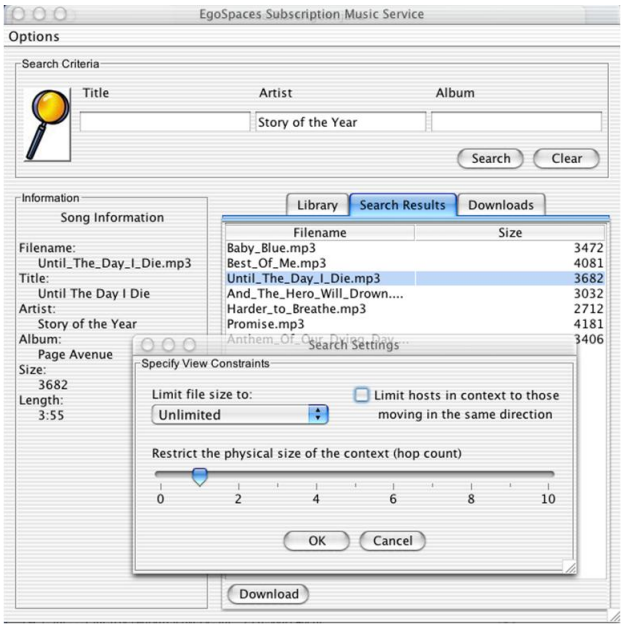

Figure 4. The subscription music service

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

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

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

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

Figure 5. Building a data constraint

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

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

Figure 6. Generating information tuples

When the user performs a search, the "Search Results" tab displays the results. The user can choose to download an available file, and the progress is displayed in the "Downloads" tab. The "Library" tab allows the user to manage his music files. When a song is selected in one of these three lists, information about the file is displayed in the "Information" section.

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

Lessons Learned. The simplified programming interface in EgoSpaces reduces applications' interactions to high-level coordination constructs, and the subscription music service takes full advantage of these novel

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

Figure 7. Accessing the view

constructs. Using the view abstraction and coordination constructs EgoSpaces allows the programmer to focus on how the music subscription application uses the information collected from the environment instead of having to explicitly discover and communicate with other agents in the network.

5 Infrastructure Design and Implementation

The programming abstractions presented in Section 2 facilitate rapid program development of applications that operate on data distributed across ad hoc networks. In this section, we demonstrate the feasibility of providing an infrastructure to facilitate rapid development of context-aware mobile applications. Figure 8 shows he high-level system architecture of the EgoSpaces middleware. The gray boxes represent components we assume to exist (message passing and the ad hoc physical network) or components the programmer provides (the application). The white boxes represent components we provide.

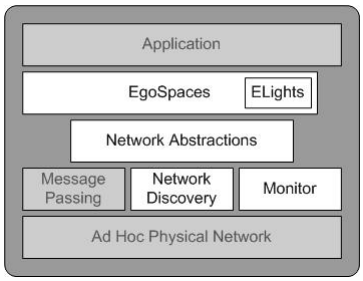

Figure 8. The system architecture

5.1 Supporting Packages

In building EgoSpaces, we designed and implemented three support packages (a network discovery package, a monitor package, and a network abstractions package) that provide lightweight implementations of services necessary for building the view abstraction. The ELIGHTS package provides the tuple matching mechanism described in Section 3.

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

Monitoring Environmental Conditions. Essential to adapting to context information is the ability to sense environmental changes. The Context Toolkit [11] uses context widgets to abstract context sensing and provide context information to applications. It allows applications to gather context information from both local and remote sensors about which the application has a priori knowledge. Our purposes require a more lightweight mechanism in which both local and neighboring environmental sensors are accessed in a context-sensitive manner. This sensor information is used to calculate the network scope restriction discussed next. EgoSpaces applications may also access monitors directly. Our monitor service provides context information by maintaining a registry of monitors available on the local host and neighboring hosts. An application tailors the monitor package to its needed capabilities. As an example, to add a location monitor, the application may provide code that interacts with a GPS monitor. New monitors must adhere to a standard monitor interface.

Defining Metrics on the Network. To provide network constraints, we use the network abstractions protocol to construct a subnet of the ad hoc network based on properties of hosts and links. `NetworkAbstractions` uses sensor information from monitors and the view's metric and bound to build a tree over the subnet of the ad hoc network that contains exactly the hosts in the network that satisfy the network constraints. The protocol also maintains the tree as the hosts in the network move and the path costs change. The network abstractions protocol provides EgoSpaces the ability to send messages to exactly the hosts in the context. EgoSpaces can also use the network abstraction interface to register operations on the context hosts. As new hosts move into the subnet defined by the network abstraction, they receive notification of any registered operations, and as hosts move out of the context, registered operations are removed. For detailed information on this protocol and its implementation, see [7].

5.2 EgoSpaces Implementation

Figure 9 depicts the middleware's details. The previous sections explained how the application agent interacts with the upper portions in this figure. In this section, we detail how the underlying components support the view abstraction while being attentive to the need for a lightweight and efficient system realization.

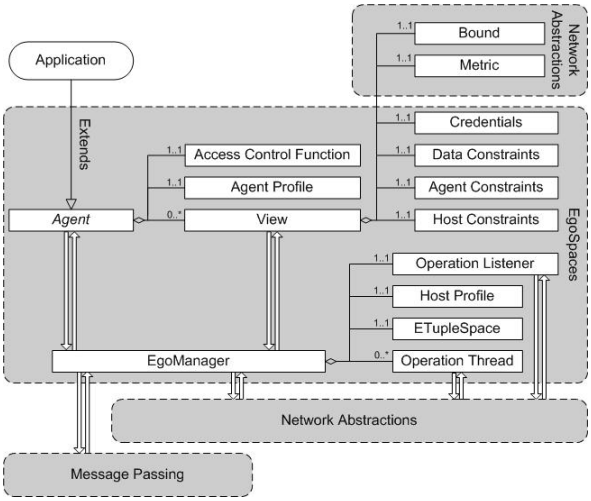

Figure 9. Internal class diagram of EgoSpaces

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

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

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

View Creation and Maintenance. Any registered agent can define views over the data available in the network. For each view, the `EgoManager` uses `NetworkAbstractions` to construct the subnet of hosts that define the network over which the view's operations will be issued. This construction is performed on-demand; the `EgoManager` only builds and maintains views when operations are issued to avoid unnecessary communication overhead.

View Operation and Agent Interaction. When the reference agent issues an operation on a `View`, the operation and view constraint information are passed to the `EgoManager`. The `EgoManager` creates a dedicated operation thread for the request. From this point, the steps necessary to implement each operation depend on the operation's semantics.

Atomic Blocking Operations. Figure 10 shows a sequence diagram describing an **in** operation. The calling thread blocks until the operation thread finds a tuple matching the operation's template.

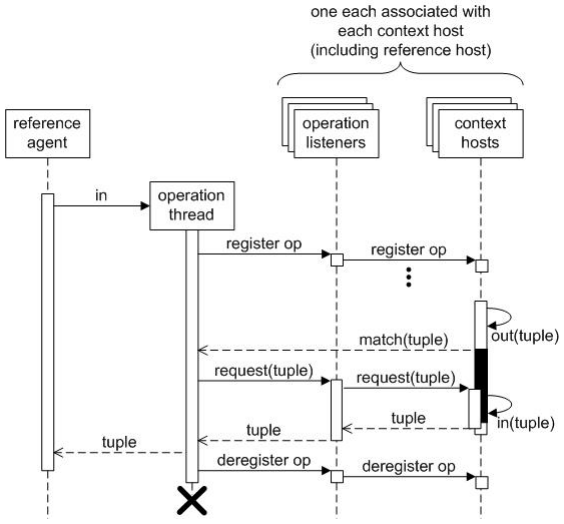

Figure 10. Sequence diagram of an in

The operation thread uses `NetworkAbstractions` to distribute a persistent query to every host in the context, and the query remains registered on those hosts until the operation thread deregisters it. If new hosts move into the context while the query remains active, they receive the query. Similarly, as hosts move out of the context, the query is removed from them.

Two things can happen when the operation is registered on a host. First, a tuple in the host tuple space may immediately match. If so, the context host notifies the operation thread. If not, the context host stores the registration and checks every tuple generated to see if it matches. When a tuple matches the request, the context host reserves the matching tuple for the requesting agent until either the operation thread requests it be removed and returned or the operation's query is deregistered (indicated as the blackened active period in Figure 10).

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

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

Atomic Probing Operations. The sequence diagram in Figure 11 shows a **rdp** operation. Again, when the reference agent issues its operation, the `EgoManager` spawns a dedicated operation thread; the reference agent remains active, waiting for a response. If, after checking each host in turn, the operation thread finds no matching tuple, it will return a null value.

The operation thread first collects the ids of hosts within the view by sending a query to the hosts defined by the view's network constraints. Every host within the context responds with its host's id and the host ids of its children in the tree. The `EgoManager` on the reference agent's host uses this information to ensure that it hears from every member of the context before continuing. At this point, the set of hosts on which the operation will be performed is fixed. If new hosts move within the constraints of the view, their addition to the context is delayed until this operation completes.

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

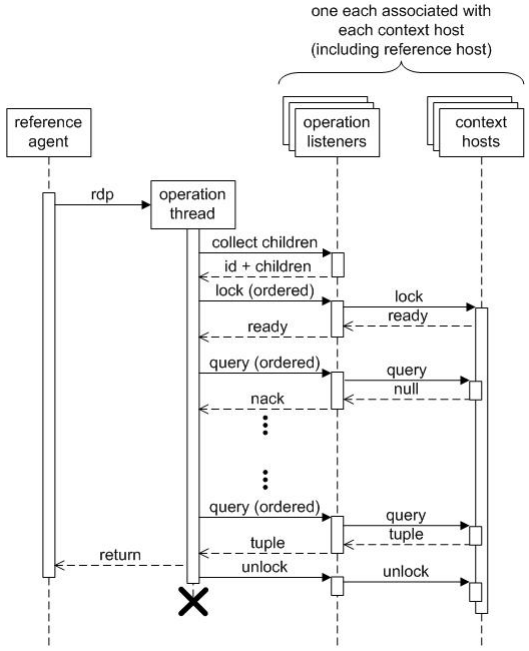

Figure 11. Sequence diagram of a `rdp`

The need for locking is not immediately obvious. Consider, however, the case shown in Figure 12, which

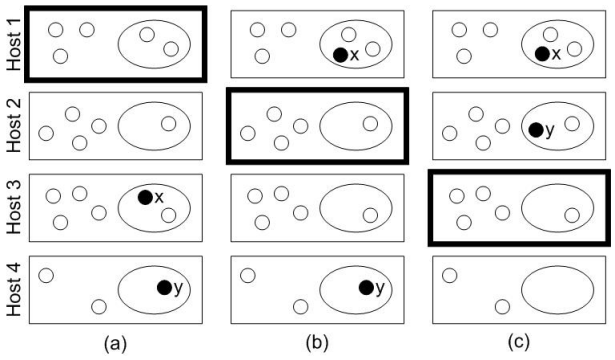

Figure 12. Locking example

shows four host tuple spaces that contain tuples in the reference agent's view. The ellipse inside each host tuple space contains the tuples that satisfy the view constraint. The black tuples also satisfy the operation's template. In this figure, the operation queries the host tuple spaces for matching tuples in order; the outlined rectangle indicates the host tuple space being queried. In Figure 12(a), the operation thread first queries Host 1. Being unsuccessful, in part (b), the operation thread then queries Host 2. At the same time, a different operation thread removes tuple x from

Host 3's tuple space to Host 1's tuple space. This is allowed because the tuple spaces are not locked. In part (c), because the operation thread did not find a matching tuple, it queries Host 3, while the tuple y is moved to Host 2. The operation thread finds no match at Hosts 3 or 4. This violates the semantics of the atomic probing operation because a matching tuple existed in the view the entire time the operation was processed.

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

Scattered Probing Operations. These operations provide weaker semantics than the previous two in that the operations are allowed to miss matching tuples in the view. That is, the case shown in Figure 12 is acceptable. The weakened semantics of these operations allows more efficient implementations that do not require locking. The sequence of events in executing a scattered probing operation follows those of an atomic probing operation, without the need to lock the context hosts. Thus, context hosts are active only while responding directly to the operation thread.

6 Conclusions

EgoSpaces is one of the very first attempts to develop a coordination model providing comprehensive support for the development of context-aware applications in ad hoc networks. Software engineering considerations played an important role in the formulation of the model. EgoSpaces introduces the notion of asymmetric coordination, which proves essential in expansive networks entailing large amounts of data. The selection and definition of key constructs such as the view were explicitly determined by the desire to simplify programming and by our earlier experiences with developing other coordination models and applications in ad hoc settings. This paper demonstrates the feasibility of providing asymmetric interactions among components in ad hoc network environments. We also report encouraging results obtained from the deployment of several initial applications which demonstrate that using EgoSpaces to develop context-aware mobile applications ease the development task.

ACKNOWLEDGMENTS

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

References

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

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

[3] 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.

[4] A. K. Dey and G. D. Abowd. Cybreminder: A context-aware system for supporting reminders. In *Proceedings of the 2^{nd} International Symposium on Handheld and Ubiquitous Computing*, pages 172–186, 2000.

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

[6] C. Julien and G.-C. Roman. Egocentric context-aware programming in ad hoc mobile environments. In *Proceedings of the 10^{th} International Symposium on the Foundations of Software Engineering*, November 2002.

[7] C. Julien and G.-C. Roman. A protocol supporting context provision in wireless mobile ad hoc networks. Technical Report WUCSE-03-57, Washington University, 2003.

[8] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the 21^{st} International Conference on Distributed Computing Systems*, pages 524–533, 2001.

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

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

[11] 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.