Short Title: Mobile Context-Aware Middleware Julien, D.Sc. 2004

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUPPORTING CONTEXT-AWARE APPLICATION DEVELOPMENT IN AD HOC MOBILE NETWORKS

by

Christine Julien, M.S.

Prepared under the direction of Professor Gruia-Catalin Roman

---

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

August, 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

SUPPORTING CONTEXT-AWARE APPLICATION DEVELOPMENT IN AD HOC MOBILE NETWORKS

by Christine Julien

ADVISOR: Professor Gruia-Catalin Roman

August, 2004
Saint Louis, Missouri

Some of the most dynamic systems being built today consist of physically mobile hosts and logically mobile agents. Such systems exhibit frequent configuration changes and a great deal of resource variability. Applications executing under these circumstances need to react continuously and rapidly to changes in operating conditions and must adapt their behavior accordingly. Applications with these capabilities are referred to as context-aware. Much of the current work on context-aware computing relies on information directly available to an application via context sensors on its local host, e.g., user profile, host location, time of day, resource availability, and quality of service measurements. The work reported in this dissertation starts by building a new perspective on context-awareness, in which the context includes, in principle, any information available in the ad hoc network but is restricted, in practice, to specific projections of the overall context. This work reports on the design and implementation of a middleware model that brings this notion of context to the application programmer. Another important aspect of the software engineering process is the ability to reason formally about the programs we create. This dissertation details initial steps to create formal reasoning mechanisms dedicated to the needs of context-aware applications. The results of this work simplify application development in ad hoc mobile networks from a design and implementation perspective and through formal reasoning.

to my family

Contents

List of Figures . . . . **vii**

Acknowledgments . . . . **ix**

1 Introduction **1**
- 1.1 Ad Hoc Network Application Development 2
- 1.2 Context-Aware Computing 5
- 1.3 A New Perspective on Context-Awareness 7
- 1.4 Organization of the Dissertation 8

2 A Conceptual Model of Ad Hoc Mobile Applications **9**
- 2.1 Computational Model 9
- 2.2 The View Concept 10
  - 2.2.1 Declarative View Specification 11
  - 2.2.2 Data Representation 12
  - 2.2.3 View Specification Mechanics 13
- 2.3 Formal View Definition 16
- 2.4 Access Control Provision 17
- 2.5 Transparent View Maintenance 20
- 2.6 View Programming 21
  - 2.6.1 Basic Operations 21
  - 2.6.2 Consistency Concerns 24
- 2.7 Making Views Reactive 26
  - 2.7.1 Reactions 27
  - 2.7.2 Transactions 29
  - 2.7.3 Augmenting Reactions 30
- 2.8 Capturing Natural Interactions in Active Views 32
  - 2.8.1 Data Migration 33
  - 2.8.2 Data Duplication 35
  - 2.8.3 Event Capture 36

2.9 Chapter Summary 38

3 EgoSpaces: Bringing Context-Awareness to Ad Hoc Mobile Applications 40

3.1 Why Middleware? 40
3.2 Rapid Development Potential 42
  3.2.1 Agent Extension 43
  3.2.2 View Definition and Use 44
3.3 EgoSpaces Middleware Architecture and Implementation 45
  3.3.1 Supporting Packages 46
  3.3.2 EgoSpaces Implementation 48
3.4 Application Development in EgoSpaces 56
  3.4.1 Emergency Vehicle Warning System 56
  3.4.2 Subscription Music Service 57
  3.4.3 Collaborative Puzzle Game 60
3.5 Chapter Summary 62

4 Supporting Context-Sensitive Communication in Ad Hoc Networks . . 64

4.1 Communication in Ad Hoc Networks 64
4.2 Network Abstractions for Context Provision 66
  4.2.1 The Physical Network 67
  4.2.2 Logical View of the Network 68
  4.2.3 The Path Cost Function 68
  4.2.4 The Minimum Cost Path 69
  4.2.5 Ensuring Boundedness 70
4.3 Sample Metrics 71
  4.3.1 Building Floor Restriction 72
  4.3.2 Network Latency 73
  4.3.3 Physical Distance 75
4.4 Context Calculation and Maintenance Protocols 78
  4.4.1 Assumptions 78
  4.4.2 Protocol Foundations 78
  4.4.3 Protocol State Information 79
  4.4.4 Context Building 80
  4.4.5 Context Maintenance 83
  4.4.6 Reply Propagation 87
4.5 Practical Protocol Implementation and Use 90
  4.5.1 Demonstration System 95
  4.5.2 Example Usage 95
4.6 Simulation Evaluation 96

4.6.1 Simulation Settings 97
4.6.2 Simulation Results 98
4.7 Chapter Summary 103

5 A Formal Model of Context-Awareness 105
5.1 Formalizing Context-Awareness 107
5.1.1 Foundational Concepts 107
5.1.2 Context Specification 112
5.2 Proof Logic 119
5.3 Patterns of Context-Awareness 121
5.3.1 Simple context interactions 121
5.3.2 Security-constrained context interactions 123
5.3.3 Tailored context definitions 125
5.3.4 Uniform context definition 126
5.4 Chapter Summary 127

6 Conclusions 128

References 131

Vita 137

List of Figures

1.1 Automobile application 3

1.2 Building engineer application 4

2.1 The computational model 10

2.2 View used by a collision warning agent on car X 11

2.3 View dynamics. Data items visible to reference agent `a1` located on host `h1` before and after `h3` moves into `h1`'s range. Hosts, agents, and data items with darkened borders contribute to the view, while ones with lighter borders do not satisfy the specification. 21

3.1 The API for the `Agent` class 43

3.2 The API for the `AgentProfile` class 44

3.3 The API for the `View` class 45

3.4 The EgoSpaces system architecture 46

3.5 The GPSMonitor Class 48

3.6 Internal class diagram of EgoSpaces 49

3.7 Sequence diagram of an **in** 51

3.8 Sequence diagram of a **rdp** 53

3.9 Locking example 54

3.10 The Reaction Mechanism 55

3.11 The subscription music service 58

3.12 Building a data constraint 59

3.13 Generating information tuples 59

3.14 Accessing the view 59

3.15 Credentials for the music sharing agent 60

3.16 Access control function for music sharing agent 60

3.17 Two views of a puzzle game 61

3.18 Seeing only edge pieces 61

4.1 The recursive cost function 69

4.2 The logical network and shortest path tree 70
4.3 The bounded shortest path tree 71
4.4 (a) Physical distance only; (b) Physical distance with hop count, restricted due to distance; (c) Physical distance with hop count, restricted due to hop count; (d) The correct cost function 76
4.5 The Components of a Query 79
4.6 State Variables 80
4.7 Context Computation 81
4.8 Support Functions 82
4.9 Context Computation and Maintenance 84
4.10 Context Computation, Maintenance, and Clean Up 86
4.11 The Components of a Reply 87
4.12 Complete Context Computation Protocol 88
4.13 Mesh reply routing example 89
4.14 The Components of a Reply 90
4.15 Reply Propagation Over a Mesh 91
4.16 The `Cost` interface 91
4.17 The `DistanceCost` Class 92
4.18 The `Cost` interface 93
4.19 The `DistanceMetric` Class 94
4.20 The Network Abstractions interface 95
4.21 Screen capture of demonstration system 96
4.22 Average number of neighbors for varying transmission ranges 96
4.23 Percentage of context members receiving the message for contexts of varying sizes 99
4.24 Settling time for contexts of varying sizes 100
4.25 Percentage of broadcasts that reached new context members for contexts of varying sizes 101
4.26 Percentage of context members receiving context messages for varying network loads 102
4.27 Maximum time for last context recipient to receive notification for varying network loads 104

5.1 A Context UNITY Specification 108
5.2 A Context-Aware System for Acquaintance Maintenance 110
5.3 Variable Components 111
5.4 A simple guide system in Context UNITY 122
5.5 An example security-constrained context-aware application in Context UNITY. 124

Acknowledgments

I would like to express my gratitude to numerous people who have supported me throughout my graduate studies. First, I would like to thank my advisor, Catalin Roman for four years of guidance. By example he has taught me the ins and outs of a successful academic career and its balance with a personal life. I would also like to thank the other members of my thesis committee, Dan Fuhrmann, Chris Gill, Cindy Grimm, and Aaron Stump, for their feedback, both of the early stages of this work and of the final product.

I would like to thank the Mobilab members, Octav Chipara, Liang Fok, Radu Handorean, Ali Hazemi, Qingfeng Huang, Jamie Payton, and Rohan Sen, whose lively research discussions challenged each of us to produce quality work. I specifically thank Jamie Payton for her collaborations on the EgoSpaces implementation, access control mechanisms, and Context UNITY. Tom Elgin assisted significantly on an initial prototype of EgoSpaces. Other undergraduate students, including Bilal Allawala, Greg Hackmann, Shannon Lieberg, Clint Scott, and Brian Thrall, implemented applications for our middleware systems. Finally, I would like to offer a special thanks to Amy Murphy and Gian Pietro Picco who contributed to the research discussions that motivated much of the work in this thesis.

I would also like to thank other members of the department for helping me keep my sanity. In particular, the office staff, Peggy Fuller, Jean Grothe, Myrna Harbison, Sharon Matlock, and Stella Sung, know absolutely everything about everything and have served as a valuable resource. Thanks also to Richard Souvenir, Tilman Wolf, and others, who organized social activities for the CSE graduate students.

For financial support, I would like to thank the National Science Foundation for the Graduate Research Fellowship Program, which supported three years of my graduate work.

Outside of the university, my friends here in Saint Louis have provided a much needed support structure. Thanks to the many of them who have provided necessary diversions, especially Matt Hampton and Tyler and Trish Small. I would also like to thank my family, from whom I have received immense inspiration and encouragement. My parents, Craig and Susan, taught me respect for myself and my education and always believed in my success. Finally, I thank my husband for his support throughout my years as a graduate student.

Christine Julien

Washington University in Saint Louis
August 2004

Chapter 1

Introduction

With the increasing popularity of mobile computing devices, software users find themselves in environments characterized by the ability to coordinate with a multitude of wirelessly networked resources. In the most extreme of cases, this network is completely disconnected from a wired infrastructure. Imagine a network that forms among wirelessly enabled vehicles on a highway, in which the cars communicate directly with one another. This type of network, in which mobile components communicate directly with each other using only wireless radio signals, is commonly referred to as an *ad hoc network*.

Ad hoc networks form opportunistically and change rapidly in response to the movement of the networked devices, or *mobile hosts*. These networks present an environment in which the network topology is both dynamic and unpredictable. The lack of a static infrastructure requires the mobile hosts themselves to serve as routers for messages in the network. In addition, because communicating parties may be constantly moving, their interactions are inherently transient in nature. Two communicating parties may be only briefly connected and may never encounter each other again.

Much work on supporting applications in ad hoc network environments builds on the foundation of ad hoc routing protocols that create and maintain communication pathways between senders and receivers. As the topology of the ad hoc mobile network changes due to host mobility, the protocols adjust the pathways to maintain end-to-end connectivity. The creation and development of these protocols are motivated largely by the desire to support the style of end-to-end communication common in Internet applications. These applications send messages destined for a known IP address, and the goal of the protocols is to maintain at least one route between the sender and specified receiver.

The increased pervasiveness of small communicating devices heightens end-users' demands for tailored applications. As users understand the capabilities of their devices, they require applications with slight variations from other users, demanding personalization at the application level. A second added difficulty for application development stems from the fact that the types of interactions between these clusters of devices depart from the traditional notion of a network application. Building applications using currently available tools and protocols is not only time consuming because of the necessity to understand the fundamentals of network programming, but the applications do not necessarily have the information the current communication protocols require.

Before embarking on an effort to support the large-scale development of applications for ad hoc networks, we first attempt to generalize the communication and interaction patterns these applications utilize. By investigating applications from a variety of domains, we hope to understand the general needs of a mobile application and avoid restricting our solutions to a particular set of applications. We will revisit the example applications we discuss in the next section throughout the dissertation to explain and motivate concepts and abstractions we introduce.

1.1 Ad Hoc Network Application Development

Communicating computing devices fill many roles in our everyday environments. As support for these devices develops, the potential for applications in many domains abounds. Soldiers on a battlefield communicate with each other, a field station, aerial monitors, satellites, etc. Cars on interstate highways can coordinate their movements, share data, or communicate with gas stations. Autonomous robots exploring an uninhabited planet can share data gathered from sensors attached to nearby robots. A structural engineer can monitor information appliances throughout a building to gauge structural problems or changes. The goal of this dissertation is to provide a communication and coordination infrastructure that supports the rapid development of such applications without itself being tailored to a specific domain. We highlight a few of these applications from different domains as motivation for the decisions taken in building the infrastructure.

Consider a network formed from automobiles on the highway. An individual driver in this evolving network might desire to perform several types of information queries over the environment. He might first want to keep track of all cars likely to collide with him. If another car comes too close, a light on the dashboard warns the driver, and he can attempt to avoid the collision. The driver might also want to monitor traffic conditions for the upcoming portion of his route. For example, he might want to be aware of any traffic jams on the next two miles of the highway in the direction he is moving. A pictorial representation of the needs of a particular driver is shown in Figure 1.1. In the figure, the driver of the car marked with an “X” keeps track of the cars likely to collide (the circular region) and cars that carry traffic information for his route (the triangular region).

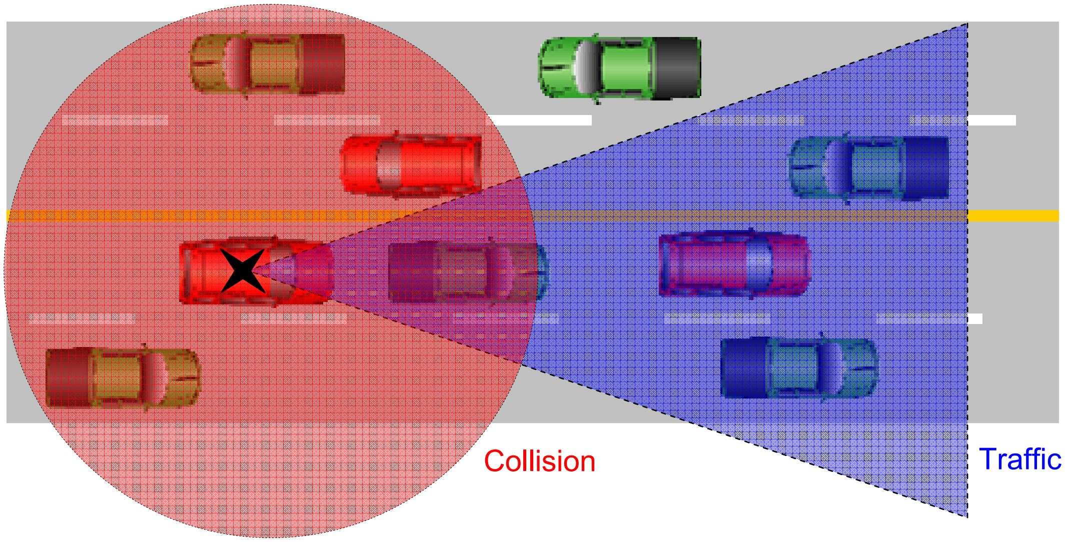


Figure 1.1: Automobile application

As a different scenario, imagine a cooperative work environment in which several participants come and go and yet wish to work together to accomplish a difficult, long-lived task. The task could be writing a research paper, exchanging conference reviews and settling on accepted papers, or even assembling a puzzle. Participants share the necessary information across an ad hoc network that morphs as others join and leave.

Imagine field researchers studying the behavioral patterns of a group of animals. Each researcher is assigned a particular animal or animals to monitor and take notes about. The researchers also use temperature and location information to add to their notes. It is possible that not every researcher carries a thermometer, but temperature information sensed by another researcher within a certain distance will suffice. Therefore, one would define a context to extend just as far as temperature information is valid and use the information contained in that area. Extending this particular example even further, each researcher might carry a camera that automatically records their observations. If one researcher's subject moves behind a boulder, the researcher can no longer see it from his location, but he can use another's camera feed to observe the target. This use of another's camera information does not interfere with the other researcher's observations of his own target subject. The usable cameras might be bounded by network latency; only cameras within a certain end-to-end latency can provide a camera feed with a high enough frame rate to be useful. It is easy to see how this particular example might extrapolate to more generalized surveillance applications.

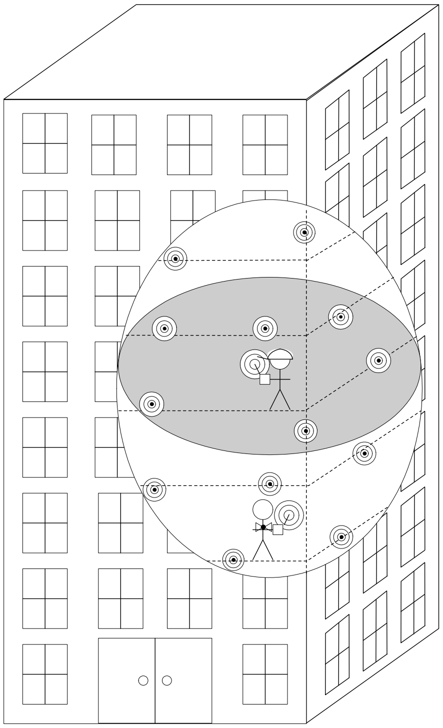


Figure 1.2: Building engineer application

As a final example, imagine a building with a fixed infrastructure of sensors that provide information about the building's structural integrity, frequency of sounds, movements of occupants, etc. Engineers and inspectors carry PDAs that allow them to assimilate information. As an engineer moves through the building, he wishes to see certain sets of structural information depending on his task or location. He may also want to respond to events in the building, e.g., the fact that an inspector has entered the building. A single engineer interacting with an information-rich building is shown in Figure 1.2. In the example shown in the figure, the engineer (in the hard hat) is performing a task that requires utilizing information provided by sensors on his current floor and adjacent floors.

Common to all these applications is the need to allow mobile entities to communicate with other unknown entities in their environments. For example, in the last application, the building engineer needs to discover the sensors in the building and collect information from them. The work in this dissertation bridges the gap between the current provisions for communication in ad hoc networks and the needs of applications in this highly dynamic environment. Specifically, we see two fundamental limitations in application development for ad hoc networks. First, the currently available communication primitives do not fully address the challenges of ad hoc networks previously outlined. While ad hoc routing protocols handle topology changes due to host mobility, applications in ad hoc networks like those discussed above are not likely to have advanced knowledge of the IP addresses of the parties with whom they want to communicate. These applications instead require communication constructs that support the transient connections they encounter in ad hoc networks and facilitate opportunistic interactions with a changing set of participants. Second, the existing communication primitives do not provide a sufficient level of abstraction for enabling rapid development and dissemination of applications in ad hoc networks. Application developers must still understand and operate at the network level, a significant undertaking for any novice programmer. The appropriate abstractions of the ad hoc network, on the other hand, will ease the development task of mobile applications.

Specifically, this work explores the application of the context-aware computing paradigm to the challenges encountered in ad hoc mobile networks. While context-aware computing is a well-known field, its application to wireless devices has been limited to simple applications involving limited communication and no significant coordination between devices. As we will see later, this work explores an extension of the notion of context to define interactions among entities within an ad hoc network. For example, the structural engineer's context is defined by the sensor information available in his building. Before exploring this new notion of context in more detail, we first review current trends in context-aware computing.

1.2 Context-Aware Computing

Context-aware computing refers to a computing paradigm in which the behavior of individual components is determined by the circumstances in which they find themselves. The extent with which these applications interact with their environments greatly exceeds the typical system/environment interaction pattern common to most modern computing. The environment has an exceedingly powerful impact on a particular application component either because the latter needs to adapt in response to changing external conditions or because it relies on resources whose availability is subject to continuous change.

Context-aware computing first came to the forefront in the early 1990's with the introduction of small mobile devices. Olivetti's Active Badge [64] used infrared communication between badges worn by users and sensors placed in a building to monitor users' movements and forward telephone calls to them. Xerox PARC's PARCTab [65] also used infrared communication between users' palm top devices and desktop computers to allow an application to adapt to its user's environment. These applications perform activities ranging from simply presenting information to the user about his current location to attaching a file directory to a room for use as a blackboard by users in the room. More recent work [29] in building such ubiquitous computing environments uses CORBA and operates over a wired network infrastructure that supports both localization and communication. These systems require extensive infrastructures which need constant maintenance. They also rely on wired communication and do not address the issues inherent in ad hoc networks, including the need to scale to large and unpredictable networks.

More recent context-aware applications, e.g., Cyberguide [1] and GUIDE [14], serve as tour guides by presenting information about the user's current environment. Fieldwork tools [45] automatically attach contextual information (e.g., time) to notes taken by researchers in the field. Memory aids [49] record notes about the current context that might later be useful to the user. These applications each collect their own context information and focus on providing a specific type of context. While these applications have proven useful in their target environments, the applications commonly demanded by ad hoc networks share some characteristics that set them apart from previous context-aware applications. Most specifically, existing applications have advanced knowledge about the parties with which they interact. The new breed of applications described in the previous section instead relies on opportunistic interactions. For example, an application for vehicles on a highway interacts with other cars locally (e.g., in the same area of a city) to collect traffic information. A particular driver has no advance knowledge about which cars will provide the traffic information; instead the driver knows to collect traffic information from other nearby cars.

Generalized software built to support the development of context-aware computing in mobile environments has also become a focus of much research. Among the best known systems are the Context Toolkit [55] and the Context Fabric [32]. The Context Toolkit provides abstractions for representing context information through the use of context widgets. These widgets collect low-level sensor information and aggregate it into higher-level information more easily handled by application developers. These widgets form a library that developers can use when constructing context-aware applications. The Context Fabric attacks a similar problem but uses an infrastructure approach. While the Context Toolkit and Context Fabric offer developers much needed building blocks for constructing context-aware applications, even those for collecting information from distributed sets of sensors, these systems do not explicitly address the needs of applications in ad hoc networks to dynamically discover and operate over a constantly changing context.

Ubiquitous computing environments build on traditional approaches to context-aware computing. The GAIA project [51] introduces the notion of *Active Spaces* as immersive computing environments for context-aware applications. Users move from one Active Space to another, seamlessly extracting from one space and integrating into a new one. GAIA addresses the needs of context-aware applications in small networked environments where the available resources in the space can be centrally managed by a kernel. This type of approach does not map well to large-scale context-aware applications in completely wireless environments that necessitate an entirely decentralized solution.

The CORTEX project [63] proposes an infrastructure for context-awareness in nomadic mobile environments that combine mobile entities with a wired infrastructure. This project focuses on quality of service guarantees that can be provided within a region of the network and uses gateways to connect these various regions. Similarly, Solar [11] provides an infrastructure to support context acquisition and operation for nomadic wireless networks. The goals of these systems are in line with our goals—to support large-scale mobile computing—but the target environment differs in that the concerns apparent in ad hoc networks require specialized solutions that are not applicable in nomadic networks.

1.3 A New Perspective on Context-Awareness

From the review of current context-aware systems, it becomes apparent that context-aware computing provides abstractions similar to those that would be useful for supporting ad hoc network applications like those discussed previously. These applications, however, share a relatively narrow view of what it means to be context-aware that requires extension to support these new applications. Existing applications take individualized views of context, each tailored to their own needs. To support context-awareness in general, we must take an approach somehow similar to that of the Context Toolkit in generalizing applications' interactions with their environment. In addition, the context that affects a traditional context-aware application's behavior is sensed directly by that application and generally uses information gathered by that application's host. We instead want to be able to gather information from a wide variety of sources spread through a region of the ad hoc network. While some systems, e.g., the Context Toolkit already allow us to do this to a modest extent, these systems require detailed a priori information about the communicating parties, a luxury we do not have in the ad hoc mobile environment. Therefore, the distinguishing characteristics of ad hoc networks and the applications likely to be desired in these dynamic environments necessitate a redefinition of what it means to be context-aware. The key ideas of this new perspective on context-awareness are:

- Context should be generalized so that applications interact with different types of context (e.g., location, bandwidth, etc.) in a similar manner.
- Different applications require contexts tailored to their individual and changing needs.
- In an ad hoc network, an application's context includes information collected from a distributed network of devices surrounding the application's host. An application must be able to specify this region without having a priori knowledge about the identities of other participants.

- Due to the large-scale nature of the environment, applications require a decentralized solution for interacting with their contexts.
- High-level abstractions of this distributed context ease the programming burden.

This dissertation uses this new definition of context-awareness to design and develop a middleware infrastructure that supports rapid application development. Specifically, this redefinition of context-awareness allows us to address the problems inherent in the ad hoc networking environment. We also use this notion of context-awareness to develop a formal model for reasoning about a variety of context-aware applications and systems.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 begins by building a conceptual model for supporting applications destined for deployment in ad hoc networks. The various components of the conceptual model are motivated by the needs of the specific applications, several of which were described in this introduction.

A main goal of this work is to simplify the task of application developers. Chapter 3 describes how we use the conceptual model described to define the requirements of a middleware system, EgoSpaces. This chapter provides details of the system design and implementation and discusses the development of three applications for the middleware.

A key component of the middleware is the ability to selectively communicate with the appropriate set of hosts in the ad hoc network without adversely affecting the performance of other hosts who need not participate in the communication. Chapter 4 discusses a network protocol that supports the middleware described in Chapter 3 by providing an underlying support for communication. In addition to describing the protocol, this chapter also provides some initial simulation evaluations of a sample implementation of the protocol.

Finally, an investigation into the ability to reason about context-aware systems immediately encounters a lack of formal models that allow such reasoning. Chapter 5 presents a formal model that elevates context to an essential component of a reasoning system. The resulting model, Context UNITY, provides a mechanism to formally express applications, protocols, and systems that entail context-awareness.

Chapter 6 summarizes the work discussed, provides a brief overview of future work and concludes this dissertation.

Chapter 2

A Conceptual Model of Ad Hoc Mobile Applications

Armed with this new perspective on context-awareness, we developed a conceptual model of applications in ad hoc networks. The goal of this conceptual model is to describe the general behavior of these applications and to provide support for their rapid development. The latter we accomplish through the use of abstraction and the provision of natural programming constructs to the application developer.

2.1 Computational Model

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

Since the notion of context defined in the previous chapter is always relative to a particular application on a particular host, we use the term *reference agent* to denote the agent whose context we are considering, and we will refer to the host on which the agent is located as the *reference host*. In principle, the context associated with a given agent in the ad hoc network consists of all the information available in that network. Of course, such broad access to information is generally costly to implement and may be undesirable, especially in scenarios where the ad hoc network grows very large. Consider, for example, the application in which a driver on a highway collects traffic information. Automobiles may be transitively connected for hundreds of miles, but only local traffic information is of interest to the driver. For these reasons, we believe it is important to structure the context in terms of fine-grained units that we call views.

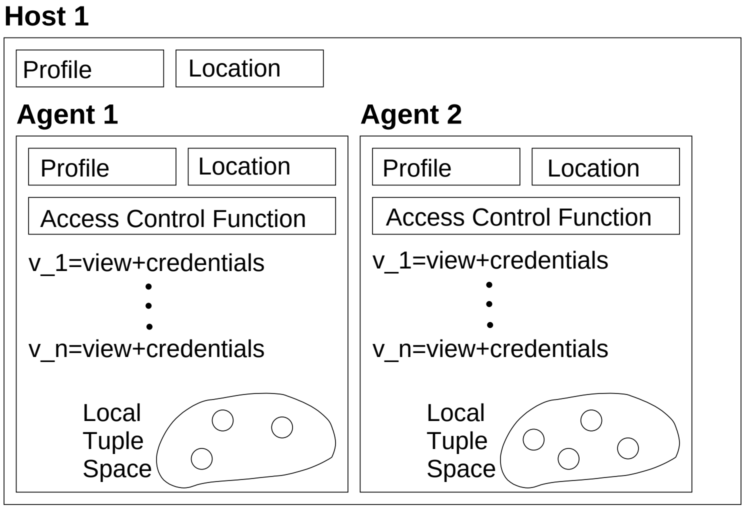


Figure 2.1: The computational model

Figure 2.1 shows our computational model in detail. We give an overview of the components of the figure here; they are discussed in more detail throughout this chapter. A host, the outer rectangle in the figure, has a physical location and a profile describing its properties. Each host contains mobile agents, the smaller rectangles in the figure. Each agent also stores its properties in a profile and has a logical location, the host on which it is located. Additionally, agents can define views which consist of the view specification, described in detail below, and the credentials provided for that view. For evaluating the view specifications of other agents, each agent also defines an access control function. Finally, every agent owns a local tuple space that contains its data items.

2.2 The View Concept

A *view* is a projection of everything available to the reference agent together with an interpretation that defines the rules of engagement between the agent and the particular view. An agent can define and use one or more views (which can be redefined over time as needs or expectations change) and can operate on each view in a manner compatible with the view's definition. The actual *contents* of the view may be visible directly or indirectly to the agent depending on the abstract interpretation associated with that view. For the purposes of

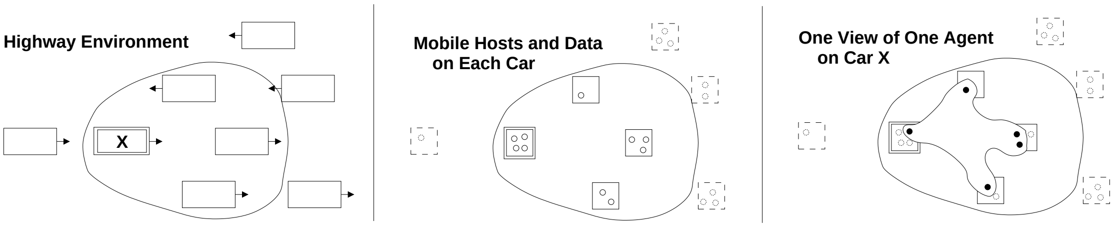

Highway Environment
X
Mobile Hosts and Data on Each Car
One View of One Agent on Car X

Figure 2.2: View used by a collision warning agent on car X

this dissertation, we assume that the view contains discrete data items that the reference agent accesses using various data access operations.

2.2.1 Declarative View Specification

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

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

Figure 2.2 shows an evaluation of the declarative view specification. The figure shows cars on a highway; the arrows indicate their approximate movement patterns. In the figure, the rectangle labeled "X" represents the reference agent's car. To simplify this picture, we assume only a single agent per car. In the picture on the left, the reference agent provides a restriction of the cars that participate in the view. The center picture shows how hosts and data items (circles in the picture) map to cars. Because the reference agent is interested only in location data (represented by blackened circles on the picture on the right), the actual view contains only these data items.

2.2.2 Data Representation

The manner in which each agent perceives and accesses data has ramifications for the ease of programming and the efficiency of operations over data in the view. Different application needs as well as a programmer's expertise contribute to the choice of a mechanism for data representation. Therefore, we separate the contents of the view from the presentation of the view to the application agents. That is, we assume a single data representation as a basis for coordination. Other forms of interaction can be easily swapped in for our choice; the investigation of the variety of such context-sensitive data structures lies outside the scope of this dissertation.

Within our conceptual model of mobile applications, we choose to present an application's context using a tuple space based abstraction. Thus, the network appears to applications as an underlying database of tuples. Tuple space representations based on the Linda model [25] enjoy a great deal of popularity due to the content-based manner in which data is accessed. In mobile computing specifically, several systems have found success using shared tuple spaces. MARS [7] focuses on logical mobility, or the movement of application agents over physically stationary hosts. This system uses a tuple space located at each host to allow coordination among co-located agents. LIME [41] combines support for logical mobility with support for physical mobility and relies on transient sharing of tuple spaces among agents and hosts within communication range. We reuse this notion of transient sharing of tuple spaces, combine it with a more flexible tuple representation, and allow an agent to use a declarative view specification to indicate with which other components it wants to share data.

To support our tuple space based coordination needs, we developed a flexible tuple space model, ELIGHTS. In this version of tuple space coordination, a tuple is an unordered set of triples of the form:

$$\langle (\textit{name},\ \textit{type},\ \textit{value}),\ (\textit{name},\ \textit{type},\ \textit{value}), \ldots \rangle$$

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

$$\langle (\textit{name},\ \textit{type},\ \textit{constraint}),\ (\textit{name},\ \textit{type}\ ,\textit{constraint}), \ldots \rangle$$

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

$$\begin{aligned}\mathcal{M}(\theta, p) \equiv \langle \forall c : c \in p ::\\ &\langle \exists f : f \in \theta \wedge f.name = c.name \wedge\ f.type\ \texttt{instanceof}\ c.type\\ &\quad :: c.constraint(f.value)\rangle\rangle\ ^1\end{aligned}$$

This matching function ($\mathcal{M}$) requires that, for every constraint c in the pattern, there must be a corresponding field f in the tuple with the same name, the same type or a derived type, and a value that satisfies the constraint. While the function does require that each constraint is satisfied, it does not require that every field in the tuple is constrained, i.e., a tuple must contain exactly the fields contained in the pattern, but the tuple can contain additional fields. Because a field's constraint is a function evaluated over the field's value, it allows both positive and negative constraints. We will see by example later in this section the forms that both tuples and patterns can take.

We found the need for such a tuple space package, even outside the effort reported in this dissertation. In general, tuple space packages hold true to the original Linda model, which can be limiting when used for coordination in ad hoc mobile networks. We changed two specific aspects of the pattern matching mechanisms, which in turn affords us a significant increase in the flexibility of tuple space based coordination. First, in models based on the original Linda model, a pattern can only match a tuple if every field in the tuple is matched by a field in the pattern. We found this restrictive because it requires the applications to have a great deal of knowledge about the pieces of data they will encounter. By allowing agents to select tuples based on a subset of the fields, we reduce the amount of knowledge needed. In the original Linda tuple space coordination models, selection of tuples required the pattern to exactly match the field types. In ELIGHTS we instead allow matching on subtypes of the field's type as well.

2.2.3 View Specification Mechanics

In our computational model, properties of hosts, agents, and data all contribute to the definition of the operational context. In providing a view specification, an agent indicates

---

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

the specific data that should comprise its view. This view, therefore, consists of a subset of all the data available on the reachable hosts in the ad hoc network. The data, the agents owning the data, the hosts where the agents are located, and the paths to those hosts must all satisfy the application agent's view specification.

Network Constraints

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

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

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

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

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

Because we aim to restrict the scope of an application's context, calculating the lowest cost to every node in the network is not reasonable. To limit the context specification, we require the application to specify a bound for its cost function. Nodes to which the cost is less than the bound are included in the context. Because cost functions must strictly increase, once the computation reaches a node that lies outside the bound, all nodes farther on the same path must also lie outside the bound. By combining the previous components of this network abstraction, we see that in providing this piece of the context specification, the application agent must include three things: the mechanism for calculating the weight of a link, the cost function used to determine the cost of the path, and a bound on that cost function. For the hop count example, an entire context specification might be written as: all nodes which can be reached in fewer than five hops. The evaluation of this context specification results in a tree rooted at the reference node and spanning a subnet of the entire ad hoc network. The path to every node in this tree satisfies the restrictions imposed by the context specification's cost function and bound, and this tree is maintained by the underlying system as long as necessary for supporting the application. That is, as hosts move in the network, the properties defining this tree change, thus changing both the contents and the topology of the tree.

Chapter 4 details the motivations, design, and implementation of the network abstractions protocol that provides this abstraction of the network for our purposes. It also shows several application-driven uses of the metric, discussing how particular application needs can be satisfied through a variety of definitions of network abstractions.

Host Constraints

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

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

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

$$\begin{aligned}&\langle (service,\ enumeration,\ printer),\\ &(printer\ type,\ enumeration,\ laser),\\ &(color,\ boolean,\ true),\\ &(quality,\ integer,\ 1200)\rangle.\end{aligned}$$

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

$$\begin{aligned}&\langle(\mathit{service},\ \mathit{enumeration},\ \mathit{printer}),\\&(\mathit{color},\ \mathit{boolean},\ \mathit{true}),\\&(\mathit{quality},\ \mathit{integer},\ >800)\rangle.\end{aligned}$$

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

Agent Constraints

Every agent defines a profile similar to a host profile, containing agent properties instead of host properties. Providing constraints over agent profiles allows application agents to restrict the set of agents that contribute data to the view. Because agents are mobile pieces of code, an obvious agent property is the host on which the agent is located. Other, more application specific properties are also useful. For example, in the building inspection domain discussed in Section 1.1, some application agents may sense air quality throughout the building, while other agents on the same devices monitor physical vibrations. Restricting operations to one type of agent or another increases the efficiency with which coordination can occur by decreasing the number of parties involved.

Data Constraints

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

2.3 Formal View Definition

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

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

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

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

2.4 Access Control Provision

As dynamic components become increasingly pervasive, security concerns become of paramount importance. Given the coordination model described previously, an agent assumes responsibility for mediating access to its data. The ability to control access in this manner is fundamental because it allows the access policies to reflect an agent's instantaneous needs. This is especially important in the highly dynamic mobile environment where mobile agents want to constantly adjust their behavior to adapt to a changing context that can include communicating with unpredictable parties. To achieve flexible access control in this environment, each agent specifies an individualized access control function.

We allow an agent to restrict which other agents access its data and the manner in which the access occurs. To accomplish the former, a requesting agent must provide credentials identifying itself. To accomplish the latter, the access policy accounts for the operation being performed. In the end, each agent defines a single access control function that takes as parameters a tuple, a set of credentials identifying the requesting agent, the operation being performed, the pattern used in the operation, and the owning agent's profile (defined next). This function returns a boolean indicating whether the requested access is allowed.

Profiles. Before describing the access control function in more detail, we introduce a profile to maintain properties of each agent, which we represent as a tuple. Particular applications or coordination systems may require specific attributes in this profile. In general, we assume a profile contains at least a unique host id identifying the agent's host and a unique agent id. In Section 3.2.1, we discuss the provision of these profiles in more detail.

Parameters. An access control function takes five parameters: the credentials, operation, tuple, pattern, and the owner's profile.

Credentials. As shown in Figure 2.1 a reference agent attaches credentials to each of its view definitions. These credentials allow an agent that is requesting access to convey information about itself. In simple cases, they can be a standard set of attributes, e.g., the agent's id or a third-party authentication. When an agent has a priori knowledge of the access requirements, credentials can be more complicated, e.g., a password. When constructing credentials, an agent must take care not to give away too much information, e.g., if the agent has multiple passwords, it should use only the correct one. This identification is especially necessary in open and dynamic mobile environments, where it is often not possible to know a priori exactly which agents can access restricted information. Instead, agents must prove they have required privileges. Credentials are a subset of the agent profile and are presented as a tuple of attributes, which allows the access control function to use pattern matching to evaluate credentials. The credentials and their transmission with an operation request are assumed to be private. This security is outside the scope of this paper but could be accomplished using cryptography schemes already under development.

Operation. The access control function can also account for the operation requested. Often, some data should be restricted to read-only access, yet current systems do not inherently allow this restriction. Considering the operation when determining access allows a dynamic application to permit one set of operations for some agents, but different operations for others.

Requested Tuple. Because we focus on tuple space models, the access control function can operate over the tuple to be returned from an operation. Pattern-matching allows this portion of the access control function to be easily defined while remaining flexible.

Pattern. A powerful component of the access control function is its ability to account for the pattern used in the content-based operation. The pattern provides information about an application's prior knowledge of the data. The owning agent may allow access only to agents that know the "correct" way to access the data (e.g., providing a wild card pattern that matches any tuple may not be acceptable). Some knowledge of the structure of the requested tuple might indicate that the requesting agent shares common application goals.

Owner's Profile. The access control function also considers the owner's current state. Because the access policy is determined dynamically, access can be granted based on context information. In some cases, data may never be sent wirelessly between devices unless they are within a secure physical environment where eavesdropping is known to be impossible.

Access Control Function. The access control function takes the five parameters described above and determines whether or not to allow the requested access. Formally, this function can be represented as:

$$\mathrm{ACF} : T \times C \times O \times P \times \Pi \rightarrow \{0, 1\}$$

where T is the universe of tuples, C is the universe of credentials, O is the finite set of operations, P is the universe of patterns, and Π is the universe of profiles. The access control function (ACF) maps the values of the parameters to a boolean indicating the access decision. The function can also be represented as:

$$\mathit{access} = \mathrm{ACF}(\mathit{credentials}_r, \mathit{op}, \mathit{tuple}, \mathit{pattern}, \mathit{profile}_o)$$

where r is the requesting agent and o is the tuple's owner.

We will briefly discuss the expressive power of this construct later. For now we consider what it *cannot* easily represent. Access decisions cannot be based on properties of the requesting agent not included in its credentials. Therefore the requesting agent must carefully construct the credentials it sends with each operation request. Also, the access decision can also not rely on arbitrary environmental properties. For example, an agent cannot base a decision on the number of copies of a tuple.

The access control function lends itself well to the mobile environment because it allows access policies to adapt to the context. Access decisions are transparent to requesting agents; if access is denied, a requester does not even know that the matching tuple existed. The fact that the access control function is evaluated on an individual basis for each tuple adhering to the view constraints provides a very fine level of granularity of the access control decision. The definition of the view becomes dependent on the evaluation of these access control functions. The following definition shows the previous formal view definition augmented to account for the credentials (κ) of the reference agent, the operations that will be performed on the view (ops), and the access control function for agent α ($\alpha.\mathit{acf}$):

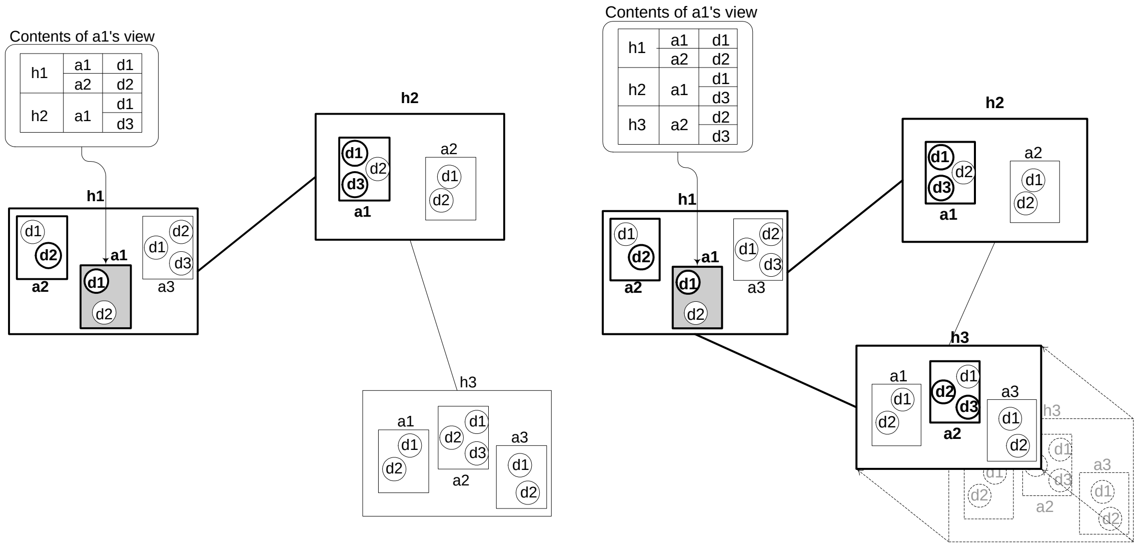
$$
\begin{aligned}
&\mathsf{view}_{\mathsf{r}}(n, h, a, d, \kappa, ops) \triangleq \\
&\quad \langle \mathsf{set}\ \eta, \gamma, \alpha, \theta : \eta \subseteq \mathit{Closure}(\mathsf{r}) \wedge \mathit{tree}(\eta) \\
&\qquad\qquad \wedge\ \mathsf{r} \in \eta \wedge \eta\ \mathtt{sat}\ n \\
&\qquad\qquad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

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

2.5 Transparent View Maintenance

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

The dynamic nature of the view definition is illustrated in Figure 2.3, where the depicted view of agent `a1` changes as the distance between hosts `h1` and `h3` decreases. Agent `a1` is grayed to indicate that it is the agent specifying the view. Hosts, agents, and data items that contribute to the view are shown with darkened borders. In part (a) of the figure, due to `a1`'s specification, only hosts `h1` and `h2` qualify to contribute agents to the view. Because of the restrictions on agent and data properties, only certain data items on certain agents on these hosts appear in the view. The balloon pointing to `a1` shows a table of the hosts, agents, and data items contributing to `a1`'s view. As part (b) shows, when host `h3` moves closer to `h1`, it satisfies the view's constraints. Again, only certain data items on certain agents appear in the view. Exactly which hosts, agents, and data items contribute is determined by the application-provided view specification.



Contents of a1's view

h1	a1	d1
	a2	d2
h2	a1	d1
		d3

Contents of a1's view

h1	a1	d1
	a2	d2
h2	a1	d1
		d3
h3	a2	d2
		d3

(a) **(b)**

Figure 2.3: View dynamics. Data items visible to reference agent `a1` located on host `h1` before and after `h3` moves into `h1`'s range. Hosts, agents, and data items with darkened borders contribute to the view, while ones with lighter borders do not satisfy the specification.

2.6 View Programming

An agent interacts with the world by specifying views that define projections of the set of all available tuples. In our conceptual model, we present the contents of the defined views to the application as a tuple space, and the application can use traditional tuple space operations to interact with the view's contents. An agent operates over the ad hoc network through a set of such views, each with its own view specification. These views can overlap, i.e., two views can contain the same pieces of data, but an agent operates over only a single view at a time. Because the view concept is agent-centric, each agent defines the views needed for its successful operation. Given the specification, the underlying system constructs and manages all of these views transparently from the application's perspective.

2.6.1 Basic Operations

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

Tuple Generation

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

$$\boxed{\mathsf{out}(T, t)}$$

$$T := T \cup \{(\texttt{ID}, \mathit{tuple\ id}, \mathit{newId}()) \oplus t\}$$

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

On-Demand Tuple Access

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

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

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

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

The two basic types of operations allowed mirror the access operations in Linda. As in Linda, these operations are blocking, meaning that they return immediately upon finding a matching tuple in the view; if a matching tuple does not exist upon issuance of the operation, the operation blocks until one does exist. The first type, a rd operation, returns a copy of a tuple in the specified view that matches the provided pattern. A rd copies a tuple by selecting one nondeterministically from the matchingSet and returning a duplicate of it. The nondeterministic selection of a tuple from the $\mathsf{matchingSet}$ uses the *nondeterministic assignment statement* [2]. A statement $x := x'.Q$ assigns to x a value x' nondeterministically selected from among the values satisfying the predicate Q. If such an assignment is not possible, the statement aborts.

$$\boxed{t := \mathsf{rd}(\nu, p)}$$

$$\begin{aligned}&\langle \textbf{await } \mathsf{matchExists}(\nu, p)\\ &\quad \rightarrow t := t'.(t' \in \mathsf{matchingSet}(\nu, p))\rangle^2\end{aligned}$$

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

The second type of operation, in, returns a tuple in the specified view that matches the provided pattern. Unlike rd, however, in deletes the returned tuple from the network. Removal of a tuple is accomplished by nondeterministically selecting a tuple from the pattern's $\mathsf{matchingSet}$, removing the tuple from the tuple space, and returning it.

$$\boxed{t := \mathsf{in}(\nu, p)}$$

$$\begin{aligned}&\langle \textbf{await } \mathsf{matchExists}(\nu, p)\\ &\quad \rightarrow t := t'.(t' \in \mathsf{matchingSet}(\nu, p))\\ &\qquad || \langle || \alpha : t \in \alpha.T :: \alpha.T := \alpha.T - \{t\}\rangle\rangle^3,\end{aligned}$$

where α is the agent owning the tuple t.

Several common extensions of the Linda primitives [19, 34, 41, 58] include probing operations. These operations differ from the blocking operations by returning immediately, even if a matching tuple does not exist in the view. As an example, rdp returns a copy of a matching tuple if one exists; otherwise it returns ϵ.

---

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

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

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

The definition of inp is the same but deletes the tuple.

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

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

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

2.6.2 Consistency Concerns

All operations discussed thus far act over the view atomically. This requires a transaction over all participants in the view. In some application instances, this transactional behavior is required. For example, if money is transferred from one party in the network to another, a transaction is required to guarantee that the same amount of money disappears from one participant and appears at another. We will discuss in Section 3.3.2 how, under certain assumptions, we can provide transactional guarantees, even in the face of mobility. As the number of participants increases, providing these transactional guarantees can become costly. From a different perspective, the previously discussed operations come with strict guarantees—if a matching tuple (or tuples) exists in the view, it (or they) will be returned. To more efficiently accommodate applications that do not require these strong guarantees, we introduce *scattered probes* that provide a weaker consistency because they are allowed to miss a matching tuple in the view. Scattered probes provide a best-effort solution and return ϵ (or an empty set) if they do not find a matching tuple.

Several different implementations of scattered probes might apply in different application scenarios. The general intuition behind the operations, however, is a simple one at a time polling of the agents contributing to the view. The operation keeps track of which agents have been polled, and if it has covered all contributing agents without finding a matching tuple, the operation returns ϵ (or an empty set). To define these operations more formally, we first define a helper macro that builds the set of agents contributing to the view:

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

We must also provide $\mathsf{matchExists}$ and $\mathsf{matchingSet}$ functions constrained to specific agents.

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

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

These functions assume $\alpha \in \mathsf{contrib}(\nu)$.

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

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

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

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

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

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

2.7 Making Views Reactive

Using the previous constructs, to wait for a piece of data an agent must either block or poll, which prevents it from performing other work in the meantime. Furthermore, these primitives provide no mechanism for grouping operations transactionally. We introduce reactions and transactions to the model to address these concerns. We provide definitions of these new constructs that show how one uses them within the framework of the conceptual model already introduced. Throughout this section, we also provide programming examples that show how the constructs can be used in the context of the building engineer application example.

2.7.1 Reactions

Like other models of mobile systems [7, 41], we find the ability to react to the presence of particular data items essential for providing adaptation mechanisms. A similar style of interaction is also found in publish-subscribe systems [9, 17]. Our reaction associates a pattern with actions to perform when a tuple in the view matches the pattern. A reaction fires once for every matching tuple in the view. Disabling and re-enabling a reaction causes it to fire again for all matches. Similarly, disconnection followed by reconnection causes reactions to re-fire. A reaction can delete its trigger and/or output the trigger modified in some way. This modification occurs through a tuple_modifiers subroutine that can add, remove, or change the tuple's fields. For example, if an agent with id `ID1` retrieves the following tuple holding a temperature reading:

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

and wants to change the time stamp, remove the destination, and add an owner, it defines the following tuple_modifiers:

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

The `newID` method allows the tuple's new owner to give it a new, unique id. An example resulting tuple is:

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

If the tuple_modifiers add a field that already exists, the field's value is replaced. The tuple generated has the same id (unless it is changed by the tuple_modifiers), and therefore, any registered reactions will not fire again for this same tuple.

A reaction has one of two scheduling modalities. The need for two modalities is driven by the same concerns as the creation of the scattered probing operations: some applications require transactional guarantees yet others find better performance in relaxing the guarantees. Eager reactions occur immediately following the appearance of a matching tuple. Only other eager reactions can preempt them and prevent them from firing. A lazy modality brings a much weaker guarantee—the reaction eventually fires 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 also have a priority that arranges a hierarchy within each scheduling modality. Within each modality, reactions with higher priorities fire before reactions with lower priorities (the highest priority being 1). If more than one reaction with the same modality and same priority exists, the one fired first is chosen nondeterministically. If the first reaction removes the trigger, the second reaction will not fire. Reactions have the form:

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

where the local name τ is bound to the trigger tuple; p is the reactive pattern; the optional keyword **remove** causes tuple removal; and the optional **out**(tuple_modifiers(τ)) outputs the trigger tuple with the tuple_modifiers applied. A reference agent enables and disables reactions using:

$$\textbf{enable}\, p\, \textbf{with}\; \textit{sched\_modality},\; \textit{priority}\; \textbf{over}\; \nu$$

$$\textbf{disable}\, p\, \textbf{over}\; \nu$$

where *sched_modality* is either eager or lazy, and *priority* is an integer. Reactions affect contributing agents' access controls; when specifying a view, the reference agent indicates if it intends to register reactions on it. Therefore, when a reaction occurs, the agent owning the trigger tuple has implicitly given permission for the reference agent to react to the particular tuple.

Triggering a reaction and executing the associated actions occur as a single atomic step. If used, the out places a tuple in the reference agent's local tuple space at the completion of the reaction's execution. This tuple may trigger additional reactions.

Programming Example

Consider a scenario in which the original sensors placed in the building generated Fahrenheit temperatures, but most have been replaced by Celsius sensors. For standardization, an agent in the Celsius sensors reacts to the Fahrenheit readings, converts the values, and replaces the readings. Without the reactive construct, a programmer could use code similar to:

```
$\nu$ = [temperature data on this floor and adjacent ones]
p = $\langle(tempType, string, = \text{"Fahrenheit"})\rangle$
while(true)
   sleep(time)
   data[] = $\nu$.rdgp(p)
   if data $\neq$ null
      for i=1 to data.length
         $\nu$.inp(data[i])
         data[i].changeField(tempType, "Celsius")
         data[i].changeField(tempValue, convert(oldT))
         out(data[i])
```

This code is slightly simplified because it refers to the Fahrenheit temperature as "`oldT`", but this value must really be retrieved from the tuple (`data[i]`). The programmer must manage this code independent of the agent's other operations. The agent creates and executes the thread to "enable" the reaction, and stops it to "disable" the reaction. In this example, the thread awakens periodically to check the reactive condition. The thread first reads all tuples matching p from the tuple space and executes the actions for the tuple.

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

```
$\nu$ = [temperature data on this floor and adjacent ones]
p = $\langle(tempType, string, = \text{"Fahrenheit"})\rangle$
$t\_m(t) = \{t$.changeField(tempType, "Celsius"),
         $t$.changeField(tempValue, convert(oldT))$\}$
$\rho$ = react to p remove and out $t\_m(\tau)$
enable $\rho$ with eager, 1 over $\nu$
```

In this example, the programmer enables a high priority, eager reaction. Not only does this reactive construct simplify the code, it adds subtle, useful semantics. Instead of polling as in the example without a built-in construct, the implementation of this reactive behavior uses a style similar to a publish-subscribe system that evaluates the registration at each remote host, returning matches when they appear. As such, the reaction is guaranteed to fire immediately following the insertion of a matching tuple unless another eager reaction fires and removes the tuple. In the first example, tuples may be inserted and removed before the thread awakens to check for matches. Finally, the application programmer has encapsulated the reaction and can reuse it on other views if desired.

2.7.2 Transactions

For an agent, performing several operations sequentially is not atomic because other operations can interleave. For example, if an agent performs a successful `rdp` operation and immediately attempts to `in` the same tuple, it may be unsuccessful if another agent has, in the meantime, removed the tuple. At times, an application may want a sequence of operations to be atomic with respect to other operations on the involved views. For example, if an application wants to replace a piece of data with an update, but does not want it to ever appear that the data is unavailable, it needs to group the removal and replacement as an atomic step. To support this, we introduce *transactions* to the model.

A transaction is a named sequence of actions that can include plain code, probing operations, and tuple creation. Because transactions must complete, they cannot include blocking operations that could halt the transaction indefinitely. Transactions are atomic actions; their intermediate results are not visible to the outside world. When creating a transaction, the reference agent provides a view restriction listing the involved views and serving as a contract between the agent and the system. Any attempt by the transaction to perform operations outside this view restriction generates an exception. As discussed in more detail in Section 3.3.2, under certain reasonable assumptions, the view restriction makes a deadlock-free implementation possible. A transaction has 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 to perform as part of this transaction. An agent executes a transaction using:

$$\textbf{execute}\, T$$

2.7.3 Augmenting Reactions

In the first reactive construct we discussed, an agent can only remove the trigger and output a modified trigger. We augment reactions to allow them to execute a transaction in response to a matching tuple. This allows applications to respond to the presence of particular data items with arbitrary behaviors that can interact with various other facets of the environment. If the tuple triggering the reaction is local (i.e., in the reference agent's tuple space), the triggering of the reaction and the execution of the transaction can be grouped as a single atomic step. In this case, we refer to the augmented reaction as an *extended reaction*, with the form:

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

An agent enables an extended reaction using:

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

The system verifies that ν_l is a local view that is restricted in scope to the reference agent.

When the trigger tuple is not local, trigger, removal, and notification are a single atomic action, while the execution of the associated transaction is a separate action. The most important ramification of this subtle difference is that the trigger might not be available when the transaction executes because other operations can interleave with the reaction's triggering and the transaction. The transaction does, however, receive a copy (τ) of the trigger tuple. This type of reaction, a *followed reaction*, has the form:

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

The word **followed** indicates the separation of the transaction's execution. The enabling mechanism is the same as above but not limited to a local view.

Programming Example

Imagine an agent that replaces temperature readings generated on the current floor over the past hour with an average. Without reactions, a programmer writes something like:

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

With the built-in construct the code consists of defining a reaction:

```
$\nu$ = [Celsius temperature data on current floor]
p = $\langle(timestamp, time,\ \text{minutes} = \text{:00})\rangle$
$T(\tau) =$ transaction over $\nu$
        begin
            p$_1$ = $\langle(tempValue, \mathsf{any}, \mathsf{any}), (timestamp, time,\ \text{[within past hour]})\rangle$
            temps[] = $\nu$.inpg(p$_1$)
            avg = average(temps[])
            average = [tuple with average information]
            out(average)
        end
$\rho =$ react to p followed by $T(\tau)$
enable $\rho$ with eager, 1 over $\nu$
```

The programmer's declaration of the views over which its transaction will act allows the system to provide atomicity guarantees associated with the execution of the operations; the transaction executes as a single atomic step, while in the hand-coded case, each operation may interleave with other operations.

2.8 Capturing Natural Interactions in Active Views

The reactive constructs discussed above make programming applications for ad hoc networks more flexible and provide powerful semantics. In many instances applications, even those from differing application domains use reactions to construct similar behaviors. Capturing these behaviors as programming constructs reduces the programming burden in common cases. We classify three such behaviors and express their operational semantics using reactions. The system is also open to extension to include additional behaviors.

A reference agent attaches behaviors to views, and, as long as the behavior is enabled, encountering certain conditions triggers an automatic action. In general, a behavior responds to a trigger, identified via a pattern. Like reactions, behaviors respond once to each matching tuple. If tuples leave the view and return or the behavior is disabled and re-enabled, the behavior executes again.

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

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

In general, `act` is the behavior's name (e.g., "migrate" or "duplicate"). The operation list in a view specification includes behaviors, and contributing agents consider this set when evaluating access control functions. Reference agents enable and disable behaviors using:

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

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

The *sched_modality* of each behavior can be either eager or lazy. To further simplify the interface, we do not provide priorities for creating hierarchies of behaviors. Instead, as we will see, the semantics of the behaviors themselves arrange them appropriately. We discuss each behavior individually, providing a brief description and syntax. We then show the behaviors' semantics with respect to the previously defined reactive constructs. To motivate each behavior, we will refer to the application scenario discussed in Section 1.1 involving a structural engineer operating within a building containing networked sensors.

2.8.1 Data Migration

Mobile agents encounter a lot of data, and an agent may want to collect certain data without explicitly reading each piece. When data consistency is important, a common solution is data replication and the associated replica management, where copies of data items are kept consistent with each other. This solution is impractical in ad hoc environments where agents carrying originals and duplicates meet unpredictably. Transparent data migration allows an agent to collect data matching a provided pattern. Since only one copy of the data item exists, replica management is unnecessary. For example, building engineers might respond to work orders generated by distributed components. A single engineer should take responsibility for each work order because if multiple engineers pick up the same job, work will be wasted.

A migration behavior automatically moves all tuples in the view that match the provided pattern to the reference agent's local tuple space. Because access controls are evaluated 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. If desired, a migration uses tuple modifiers to change migrated tuples. An engineer collecting work orders can mark the migrated tuples as "assigned" to prevent the work orders from migrating again.

Semantics

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

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

If the programmer supplies tuple modifiers, the tuple placed in the tuple space is the trigger tuple with the tuple modifiers applied. Otherwise, the tuple is exactly the trigger. Even though the migrated tuple is the same tuple (unless the tuple modifiers change the ID), tuple migration may trigger reactions in the new location that have already fired for the tuple in the previous location. Enabling a migration reduces to enabling the reaction using the migration's scheduling modality and a low priority (e.g., 10):

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

where ν_r is ν with the reference agent eliminated. This prevents the "migration" of local tuples. The priority scheme maximizes the number of behaviors that execute, e.g., it ensures duplication occurs before migration. A migration's low priority allows other reactions and behaviors of the same modality to trigger first. If these actions remove the tuple, however, the migration will not occur.

Programming Example

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

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

The tuple output has the same id as the one read, but the "assigned" field has been set to true. This implementation might miss matching tuples. To ensure tuples are not infinitely migrated, the programmer must explicitly define ν_r, or the remote portion of a view ν. The use of ν_r prevents tuples in the local tuple space (e.g., work orders created by this engineer that other engineers should perform) from being "migrated" to their current host.

The built-in migration behavior hides the declaration of ν_r.

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

Because this behavior is integrated with the system, we can guarantee, for eager migrations, that tuples are migrated if they appear in the reference agent's view, conditional on no other reactive constructs removing the tuple first.

2.8.2 Data Duplication

Under different circumstances, data availability is more important than consistency, and an application would rather duplicate data to make it available upon disconnection, with the knowledge that duplicates will not remain consistent with the originals. A duplication behavior copies tuples matching a pattern and leaves the originals unaffected. In our example application, the building engineer may collect sensor data for processing off-site. The engineer does not want to remove the data because others may need it.

Duplicated tuples may match the view specification and be infinitely duplicated. They may also appear in other agents' views. As with migration, applications deal with these concerns individually, e.g., by tagging all duplicates and preventing duplication of the tagged tuples. Because replica management proves too costly, duplicates do not remain consistent with originals, even if both persist in the view.

Semantics

Duplication reduces to a reaction that does not remove the trigger and generates a new tuple (with a unique id):

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

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

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

$$\textbf{enable}\ \mathcal{D}\ \textbf{with}\ \textit{sched\_modality}\ \textbf{over}\ \nu$$
$$\triangleq \textbf{enable}\ \rho_d\ \textbf{with}\ \textit{sched\_modality}, 1\ \textbf{over}\ \nu$$

A high priority ensures a duplication occurs before other actions, e.g., migration.

Programming Example

Using only the EgoSpaces primitive operations, an engineer duplicating structural integrity data uses code similar to:

```
ν = [structural agent data on this floor and adjacent ones]
p = ⟨(strain, number, any), (acoustic emission, number, any),...⟩
seenTuples = new Vector()
while(true)
   sleep(time)
   data[] = ν.rdgp(p)
   if data ≠ null
      for i=1 to data.length
         data[i].newID()
         out(data[i])
         seenTuples.add(data[i])
```

Using `seenTuples` prevents the agent from re-duplicating the same data.

Using the built-in duplication behavior reduces to defining a view, creating a duplication behavior, and enabling it on the view:

```
ν = [structural agent data on this floor and adjacent ones]
p = ⟨(strain, number, any), (acoustic emission, number, any),...⟩
$\mathcal{D}$ = duplicate $p$
enable $\mathcal{D}$ with eager over $\nu$
```

This eager behavior is guaranteed to duplicate all matching tuples that appear in the view without missing any, while the hand-coded example may miss some. A lazy duplication has semantics identical to the hand-coded example.

2.8.3 Event Capture

The previously described programming constructs, including reactions and the two previous behaviors, operate over the system's state. Many applications also benefit from reacting to events raised in the system. For example, an agent might want to be notified when another party has accessed a piece of data. In our ad hoc network system, events include an agent's arrival, another agent's data access operations, etc.

To handle event registrations and notifications, we must first introduce an event generation mechanism to our model. This mechanism is discussed in detail in Section 3.3.2.

We then represent events as special tuples. An agent registers its interest in an event via patterns over such tuples. Once registered, event notifications matching the pattern propagate to the reference agent. To prevent superfluous event generation, we raise events only for specific registrations, and the event's callback execution consumes the event tuple created for it. This allows multiple registrations for the same event and guarantees that all registered parties receive notification. A reference agent uses a transaction to specify the event's callback.

Semantics

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

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

As before, the $\oplus$ indicates that the provided field, in this caused the new event id, is added to the tuple. The generation of the event copy and the callback execution are not an atomic action. However, the reference agent can prevent other agents from stealing its event tuples using its access control function.

Enabling an event behavior reduces to enabling the two reactions:

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

The first reaction has eager modality and high priority, guaranteeing the reference agent is notified of the event. The second reaction's scheduling modality corresponds to the behavior's modality and also executes at high priority. This reaction is enabled on a local view (ν_l) that contains only local event tuples.

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

This reduction assumes mechanisms exist to generate events and clean up event tuples. The former is discussed in Section 3.3.2; the latter is accomplished by another reaction:

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

where p matches any event tuple. This eager reaction with a priority of at least 2 executes after all event copies have been generated (at priority 1):

$$\textbf{enable}\ \rho_{gc}\ \textbf{with}\ \mathsf{eager}, 2\ \ \textbf{over}\ \nu_e$$

This reaction is defined and enabled on every agent's event view, so an agent need not define it each time it enables an event behavior.

Programming Example

Because event capture requires an event generation mechanism, there is no way to accomplish this behavior using the initial EgoSpaces operations. Assume that a tuple indicating a host's arrival is represented with an event tuple similar to the following:

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

If the building engineer wants to receive notification of an inspector's arrival on adjacent floor, his application agent has the following code:

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

The null view restriction indicates that the transaction does not use any views.

2.9 Chapter Summary

In this chapter, we presented a model for application development in mobile ad hoc networks. This conceptual model represents a first effort at using the abstractions provided by context-awareness specifically for simplifying the application development task in mobile ad hoc networks. This conceptual model represents the data available in the network as the application's context and thereby reduces all interactions between mobile components to interactions with an ever-changing context. In this way, application developers are able to use natural constructs for coordination, and we have abstracted away the difficulties associated with communication in this highly dynamic network structure. In the next chapter, we will use this conceptual model of applications to build a middleware infrastrcture that provides the necessary underlying system support for application developers.

Chapter 3

EgoSpaces: Bringing Context-Awareness to Ad Hoc Mobile Applications

The previous chapter introduced a model of application development that aims to abstract the complexities associated with ad hoc mobile networks into a data structure form that novice programmers can interact with naturally. In this chapter we discuss how we made this conceptual model concrete through the implementation of a middleware system that provides the context-aware programming constructs previously discussed. We also discuss how programmers use the middleware to build applications and report on the use of the middleware to build some of the applications discussed in Section 1.1.

3.1 Why Middleware?

Middleware supports the application development task by enhancing the level of abstraction associated with the programming effort and by adding mechanisms and services which are more specialized than those provided by the operating system. Distributed object systems, agent systems, and coordination systems each address different abstractions useful for dynamic mobile applications.

Distributed Object Systems.
In systems such as the Object Management Group's CORBA (Common Object Request Broker Architecture) [22], Microsoft's COM (Common Object Model) [22], and Sun's Jini [21], objects serve as the main abstraction of the distribution in the network [60]. Key features include a lookup mechanism to identify the objects associated with particular services and

common interfaces to enable interactions and to hide the distribution of components. For example in Jini, service objects register with a centralized lookup-service which plays the role of matchmaker between clients and services. After a client finds a service, all interactions are performed in a location-transparent manner and without the aid of the lookup-service. Typically, object-based systems assume that a connection between a client and a service object is long-lasting, and therefore these systems do not address the possibility of disconnection. The DENO (Decentralized Network Objects) [37] system begins to address this issue in the context of mobile and unreliable networks, adding replication of the objects to increase efficiency, availability, and fault tolerance.

Mobile Agent Systems.

Mobile code and mobile agents can further enhance the programming model, increasing program performance by optimizing the placement of executing code during the application lifetime. One of the primary points of comparison for mobile agent systems is their support for either strong or weak mobility [24]. In weak mobility, a mobile agent migrates, carrying only its code and data state, and abandoning its execution state including the program counter and stack variables. This model is by far the most common as it requires little additional support from the runtime environment. Java-based systems such as Aglets [62] and mobile code systems such as μCode [48] support weak mobility. In strong mobility, the execution state is included when the agent migrates. The TCL scripting language, supported by D'Agents [26] offers strong mobility, and recent work has been done to enhance the Java VM [59] or modify Java byte code [61] to allow strong mobility in the Java language.

Other work aims to combine the mobile agent paradigm with distributed object systems. The ORB/OS Task Force [42] and SOMA (Secure and Open Mobile Objects) [57] address scalability and interoperability by exploiting CORBA interfaces and adding services to support mobile objects. Because the objects (or agents) in these systems adhere to the CORBA standards, they can interact with other agents only according to those specifications. That is, to locate one another, they must coordinate with a centralized server. Other extensions allow objects to maintain references when clients move [5, 54], but, in most cases, this is supported by a straightforward proxy and does not generalize to the mobile ad hoc environment. The FarGo framework [31] provides for relocation of objects at runtime, to follow user-specified semantics. This system, however, provides limited support to locate objects and no support to automatically rebind to a new object after a reference is dropped.

Coordination Systems.

Coordination abstracts the behavior of the mobile units and focuses on high level communication protocols. The Linda tuple space coordination model [25] was introduced and popularized in the mid 1980's to support parallel computation. More recently, the distributed computing community has realized the power of this model and has produced several client-server tuple space implementations including IBM's TSpaces [34] and Sun's JavaSpaces [58]. Because these systems rely on a centralized tuple space, they do not adapt well to mobile systems where support for transient connections, decreased access to resources, and dynamic changes in context are needed. The MARS [8] and TuCSoN [43] systems exploit programmable Linda tuple spaces for coordination of logically mobile agents, essentially providing a tuple center on each host to support interaction among co-located agents. Applications can "program" these tuple spaces by installing actions that respond the the presence of particular data items. The LIME [41] middleware introduces the notion of transiently shared tuple spaces, a model which does not rely on any central server and supports both physical and logical mobility. In LIME, each agent is permanently bound to a tuple space, and when two agents can communicate, their tuple spaces are logically merged. PEERWARE [18] similarly exploits the notion of transiently shared data structures in peer to peer networks, focusing on data organized as a forest of trees and providing an event distribution mechanism over the trees. This event distribution mechanism extends the JEDI [17] distributed event model. Both PEERWARE and LIME have a symmetric sharing model which is not always appropriate, especially in large ad hoc networks. Symmetric coordination, in which all of the coordinating parties form a consistent group and every member is aware of every other members do not scale well to networks that consist of hundreds or thousands of hosts. Asymmetric coordination, on the other hand, provides applications the opportunity to reduce this overhead by selecting a subset of the entire network to interact with.

In general, these systems share the goal of simplifying the programming task for application developers. They use some abstraction of a difficult programming space to reduce the complexity programmers encounter. The EgoSpaces middleware discussed next has the same goal, but is specifically targeted to the domain of ad hoc mobile computing, an area that has received some attention. The existing middleware efforts in this area tend to lack the ability to scale to ad hoc networks containing large numbers of mobile components. In the remainder of this chapter, we discuss the design, implementation, and use of the EgoSpaces middleware infrastructure, which uses the conceptual model of mobile applications discussed in the previous chapter.

3.2 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 the development task 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 developer extends the `Agent` class, which allows the application access to the view specification mechanics and communication capabilities.

3.2.1 Agent Extension

Figure 3.1 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 3.1: 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 3.2 to modify its profile's contents.

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 its

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

Figure 3.2: The API for the `AgentProfile` class

`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 3.3.2. By registering with the `EgoManager`, the application agent delegates responsibility for data management and communication. This also facilitates agent migration among hosts, which we 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 for their views or their current location. If an agent moves to a new host, all its data moves with it.

3.2.2 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 use views. Figure 3.3 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. These two components represent the network constraints and are part of the `NetworkAbstractions` interface. The `Metric` defines the cost 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 application developers can build their own `Metric` and `Cost` definitions by following the procedure outlined in Chapter 4. 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

```
public class View {
  public View(Metric m, Cost bound,
           HostConstraints hc, AgentConstraints ac,
           DataConstraints dc, 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 3.3: The API for the `View` class

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' `AccessControlFunctions` 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 3.3 to access data. Each operation takes a pattern, or template, over a tuple which allows an application to provide a final restriction that any returned tuple must satisfy. The operations shown in the figure have the semantics described in the previous chapter.

3.3 EgoSpaces Middleware Architecture and Implementation

The programming abstractions presented in Chapter 2 facilitate rapid development of applications in ad hoc networks. Figure 3.4 shows the high-level system architecture of the EgoSpaces middleware that provides these abstractions to application programmers. 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 pieces of the architecture we provide.

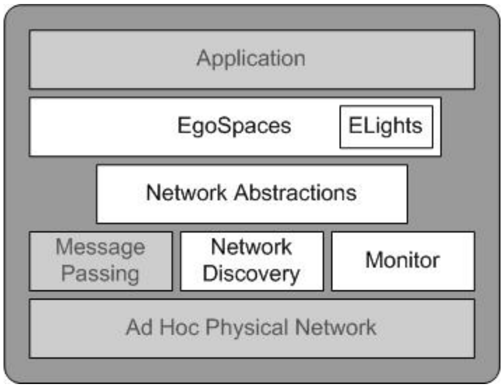


Figure 3.4: The EgoSpaces system architecture

3.3.1 Supporting Packages

To build EgoSpaces, we implemented three support packages (a discovery package, a monitor package, and a network abstractions package) that provide lightweight implementations of services necessary for building the view abstraction and providing operations over an application's defined views. The eLights package provides the tuple matching mechanism described in Section 2.2.2.

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. The algorithm provided in this package allows a host to keep track of the set of neighbors it can directly communicate with. The error associated with this knowledge is directly dependent on the beaconing period, i.e., there is a period of time related to the beaconing period that a host may have inconsistent information about its current set of neighbors. Given information about a target environment, the beaconing period can be adjusted to partially account for the rate of change.

Monitoring Environmental Conditions

Essential to adapting to context information is the ability to sense environmental changes. The Context Toolkit [20] 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. The ad hoc network requires a more lightweight mechanism in which both local and neighboring environmental sensors are accessed in a context-sensitive manner with no use of such advance knowledge. This sensor information is used to calculate the link weights needed in the network abstractions protocol discussed in detail in Chapter 4.

The monitor service we provide maintains a registry of monitors available on the local host and neighboring hosts (within one hop). The former are referred to as *local monitors* and the latter as *remote monitors*. An application tailors the monitor package to its needed capabilities. As an example, to add a location monitor, the application provides code that interacts with, for instance, a GPS device. In general, a monitor contains its current value in a variable (e.g., the value of a GPS monitor might be represented by a variable of type `Location`) and contains methods that allow applications to access the value (through the `queryValue()` method) or react to changes in the value (through the `MonitorListener` interface). This functionality is contained in an abstract base class called `AbstractMonitor`. When a programmer extends the monitoring package to add a new monitor, he must extend the `AbstractMonitor` class. This extending class is responsible for ensuring that the class's `value` variable is kept consistent with the current state of the environment. Changes to this variable should be performed through the `setValue()` method in the base class to ensure that any listeners registered for changes to the variable are notified. The programmer should also add his defined monitors to the monitor registry at run-time.

Figure 3.5 shows the code that a programmer must write to extend the monitor package by showing the code for a class that extends `AbstractMonitor` to collect information from a GPS device. From the perspective of our package, the important pieces are how the extending class interacts with the base class. The details of communicating with a particular GPS device are omitted; their complexity is directly dependent on the individual device and its programming interface.

To monitor context information on neighboring hosts, the monitor registry creates instances of the class `RemoteMonitor` that connect to concrete monitors on the remote host. These `RemoteMonitor`s serve as proxies to the actual monitors; when the values change on the monitor on the remote host, the `RemoteMonitor`'s value is also updated. To gain access to `RemoteMonitor`s, the application provides the id of the host (which can be retrieved from the discovery package) and the name of the monitor (e.g., "GPSMonitor"). The monitor registry creates the proxy on the local host, connects it to the remote monitor, and returns a proxy handle to the application. The application interacts with the remote monitor in

```
public class GPSMonitor extends AbstractMonitor{
  public GPSMonitor(...){
    //call the AbstractMonitor constructor
    super();
    //set up serial connection to GPS receiver
    ...
  }
  public void serialEvent(SerialPortEvent event){
    //handle periodic events from GPS receiver
    ...
    //turn GPS event into a GPSLocation object
    ...
    //set local value variable, notify listeners
    setValue(gpsLocation);
  }
}
```

Figure 3.5: The GPSMonitor Class

the same manner as with a local monitor (e.g., by calling the `queryValue()` method or registering listeners for changes in the value).

Defining Metrics on the Networks

To provide network constraints, we use the network abstractions protocol, discussed in detail in Chapter 4, to construct a subnet of the ad hoc network based on properties of hosts and links. The network abstractions protocol uses sensor information from monitors and the view's metric and bound to build a tree over the subnet of the ad hoc network. This tree contains exactly the hosts in the network that satisfy the network constraints. When the application accesses data items in the view, the system routes over this tree to service the application's view queries. The network abstractions protocol can also maintain the tree as the hosts in the network move and the path costs change. The protocol provides EgoSpaces the ability to send messages to exactly the hosts that are in the context, i.e., those hosts that can contribute data to the view. As new hosts move into the subnet defined by the network abstraction, they receive notification of any registered operations (e.g., reactions), and as hosts move out of the context, registered operations are removed.

3.3.2 EgoSpaces Implementation

Figure 3.6 depicts the middleware's details. The previous section 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

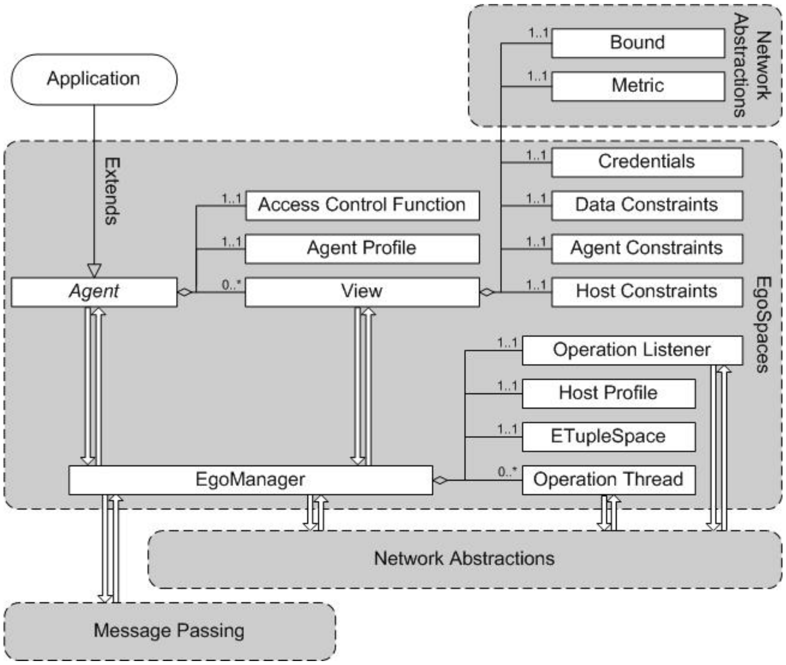


Figure 3.6: Internal class diagram of EgoSpaces

for a lightweight and efficient system realization.

Agent Registration and Migration

When an agent is created, a data structure is initialized within the agent to hold any tuples the agent creates. EgoSpaces hides this data structure from the extending class. 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. This reduces the overhead of remote operations. The justification for this choice 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 deregistration, the application agent's code and state are 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. This is important to ensuring as efficient an implementation as possible. The mechanisms for the protocols to build and maintain views are discussed in depth in Chapter 4.

Access Control

An agent defines a view as a set of constraints: network constraints, host constraints, agent constraints, data constraints, and credentials. Each agent also defines an access control function. The application agents use the `EgoManager` to define and interact with their views. An agent issues content-based retrieval operations on its views. These operations are actually serviced by the `EgoManager` with which the agent is registered. The `EgoManager` uses the pattern provided to select tuples that match the pattern provided with the operation and evaluates each tuple individually to determine whether or not the tuple satisfies the view and is a viable candidate for return to the requesting agent. In evaluating each tuple, the `EgoManager` extracts information about the agent (properties of the host the agent resides on, properties of the agent, and the agent's access control function) that is providing the tuple and compares this information with the constraints defined in the requesting agent's view, *including the credentials*. The latter is key to the access control function in EgoSpaces. If the tuple satisfies the view's constraints *and* the requesting agent's credentials satisfy the tuple owner's access control function, then the tuple can be returned to the requesting agent.

An important aspect of the access control mechanism in EgoSpaces is the fact that it relies on the mechanisms inherent to tuple space based systems to mediate access. Tuples are used to describe credentials, and access control functions can be described by a set of access policies defined as patterns, or templates, over tuples. Implementing credentials and access control functions in this way provides a number of benefits. First, the pattern matching mechanisms already provided by the tuple space system can be used to check the credentials against an access control function. Second, we allow the programmer to construct credentials and access control functions in a familiar way. Third, using tuples and templates allows for flexibility and adaptation, since adding and removing fields from existing tuples and patterns is relatively simple. Finally, the use of tuples and patterns allows for expressive access control functions and credentials. Access control may be expressed according to any property of the interacting agents, as long as the properties can be captured in tuple and template form.

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 3.7 shows a sequence diagram describing an `in` operation. The calling thread blocks until the operation thread finds a tuple matching the operation's template. The matching function used in this example is the same as that discussed in Section 2.2.2. 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.

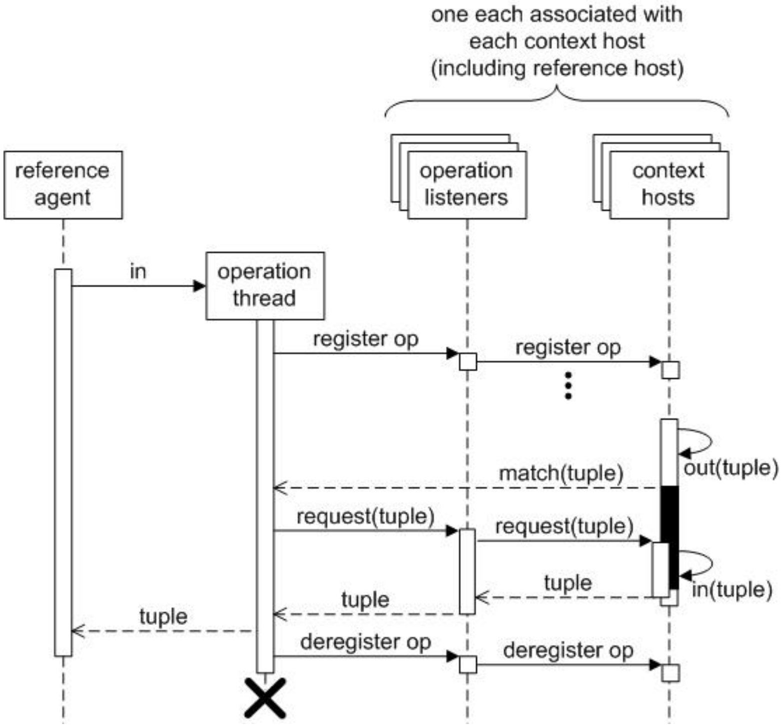


Figure 3.7: Sequence diagram of an **in**

Two things can happen when the operation is registered on a host. First, a tuple in the host's 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 3.7). A match may also be triggered by a new host with a matching tuple moving into the view. The registration of the operation on this arriving host (as well as deregistration from any departing host) is handled implicitly by the `NetworkAbstractions` protocol.

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 nondeterministically. Once the operation is ready to return, the persistent operation query is deregistered from all of 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, to complete the operation.

Atomic Probing Operations. The sequence diagram in Figure 3.8 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 of the 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 thread, then returns positively. The operation thread waits to hear from each context host before locking the next host.

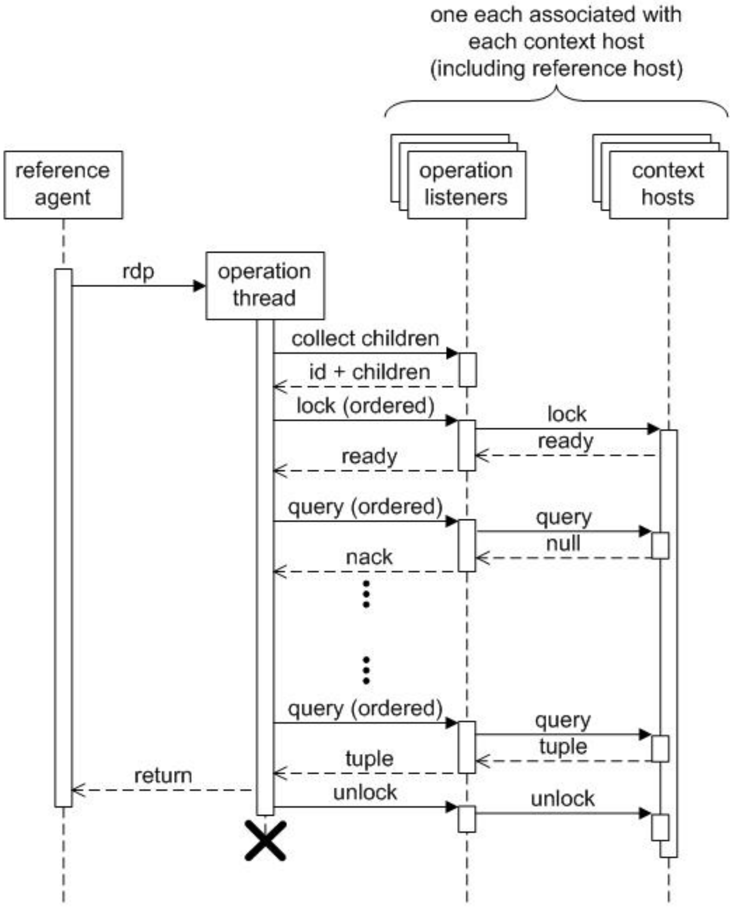


Figure 3.8: Sequence diagram of a **rdp**

The need for locking is not immediately obvious. Consider, however, the case shown in Figure 3.9, which 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 3.9(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 moves 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.

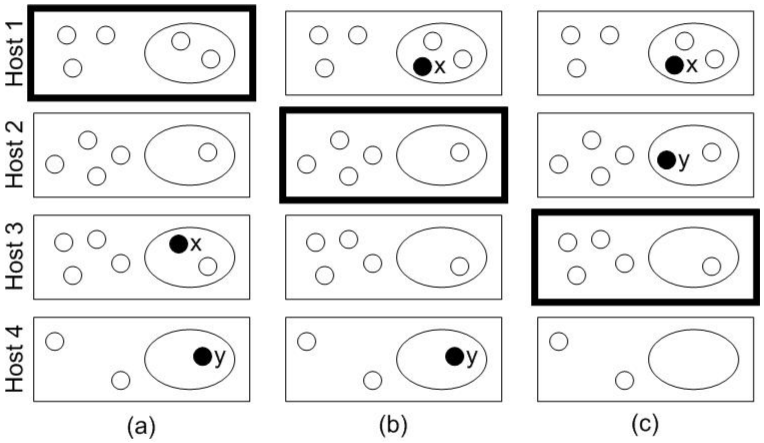


Figure 3.9: Locking example

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 3.9 is acceptable. The weakened semantics of these operations allow 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.

Transactions. A transaction operates over several views. As such, transactions are inherently costly. EgoSpaces reduces this cost by requiring a reference agent to explicitly declare which other agents need to be locked for the transaction by providing a list of views. Because the agents contributing to each view are known, EgoSpaces can lock the transaction's participants (including the reference agent) in order (by id). If any other agent also performs a transaction, it locks agents in the same order, avoiding deadlock. If a contributing agent moves out of the view while a transaction is locking agents, it is unlocked before departing. If the transaction's operations are already executing, the agent's departure must be delayed until the transaction completes. If a new agent moves into the view while a transaction is in progress, its arrival is ignored until the transaction completes.

We guarantee enough time to complete the transaction before the agent disappears from communication range using *safe distance* [33].

Reactions. Because reactions are the core of the EgoSpaces extensions, an efficient implementation is essential. Each agent keeps a reaction registry (containing all reactions it has registered) and a reaction list (containing all reactions this agent should fire on behalf of other agents, including itself). A reaction registry entry contains a reaction's id, the tuple to output when the reaction fires (if any), and the transaction that extends or follows this reaction (if any). A reaction list entry contains the reaction's id, the reaction issuer's id, the reaction's pattern, the view's data pattern, and a boolean indicating whether or not to remove the trigger. Upon registration, the reaction propagates to all view participants and is inserted in each participant's reaction list. For all matching tuples in the view, the reaction fires, sending a notification (containing a copy of the trigger) to the registering agent. If specified, the tuple is removed from the tuple space. As long as the reaction remains enabled, new tuples in the view 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 schedules any associated transaction. In Figure 3.10, agents B and C register reactions, which both match `t`. The reaction with the highest priority (B's reaction) fires first, generating notification `n` for B. Because this reaction removes `t`, C's lower priority reaction will not fire. B's reaction can be extended or followed by a transaction. The former is only allowed when the trigger is local (i.e., A=B).

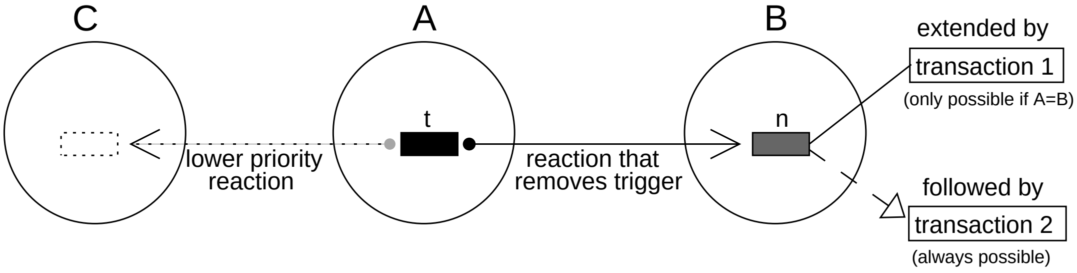


Figure 3.10: The Reaction Mechanism

During the view's construction, agents receive the reaction registration and add it to their reaction list. As new agents move into the view's scope, they receive any registered reactions. As agents move out of the view, they remove information regarding registered reactions. If these agents return, they receive the registrations and fire the associated reactions again for matching tuples.

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

Event Generation. To successfully implement event capture, we add an event raising mechanism to EgoSpaces. Event types include host arrival and departure, agent arrival and departure, and data access operations. Each type of operation has a defined type string (e.g., *hostArrival*) and some secondary information (e.g., the *HostID* for a host arrival or departure event). The event generation mechanism raises an event only if an agent has registered for the event. Upon generation, special event tuples are created for each registered agent, and these tuples are transmitted to the agent. The event's callback then executes according to the registration's modality (eager or lazy).

3.4 Application Development in EgoSpaces

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

3.4.1 Emergency Vehicle Warning System

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 the ability to notify the car in time for it to give way for the emergency vehicle. The car's view constraints are:

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

Agent Interaction

Only the emergency vehicle generates tuples. An emergency vehicle creates a tuple when it turns its siren on and removes the tuple when it turns its siren off. The access controls for the emergency vehicle prevent any other agent from removing the warning tuple 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, e.g., at one second intervals) 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 cars' and the emergency vehicles' speeds are variable, the scope of the view depends on their velocities. Given a well-defined view, the application agent's minimal interaction with EgoSpaces involves only simple view operations. The car is guaranteed to be notified as soon as possible of the approach of an emergency vehicle. Notification that the emergency vehicle has departed is not guaranteed to be as timely. This latter behavior could be further accomplished using the reactive extensions to the middleware.

3.4.2 Subscription Music Service

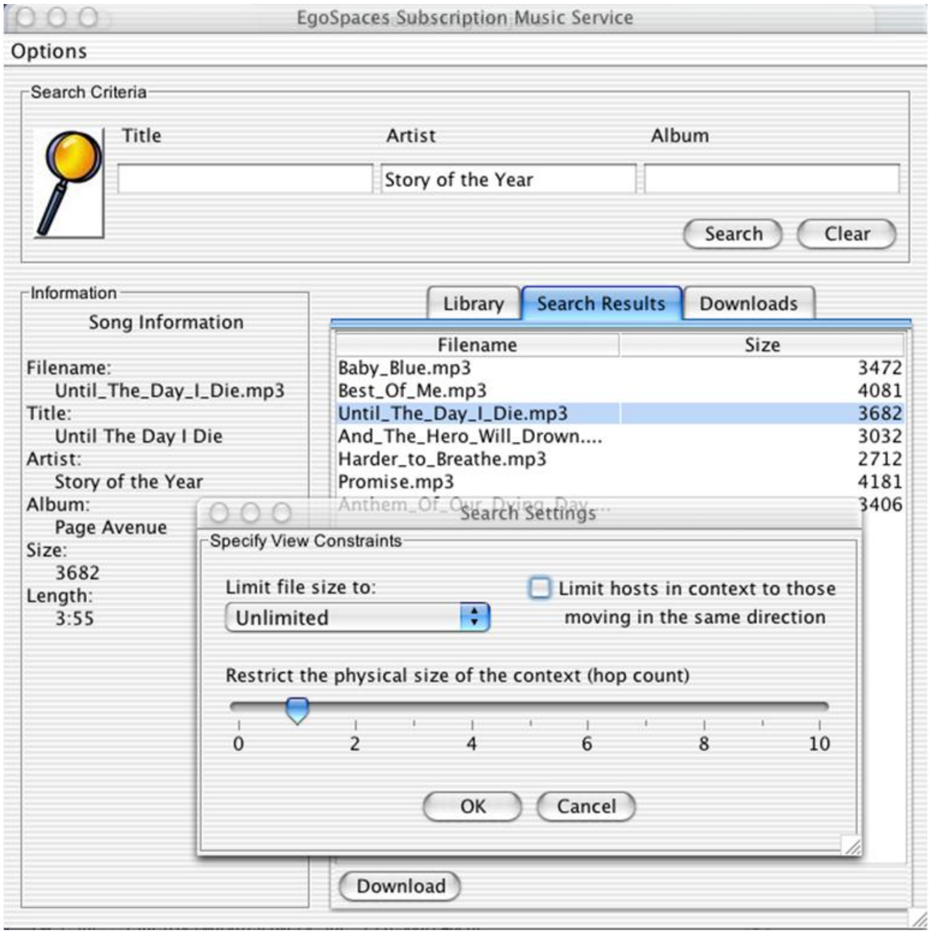
The second application enables music sharing on a network of cars and requires more sophisticated agent coordination. Users subscribe to a music file sharing service which allows them to manage their music and share music with other subscribers they meet on the highway. The application allows a user 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 picks up where it left off and completes. Figure 3.11 shows the user interface.

View Definition

The dialog box in Figure 3.11 allows the user to change his view's constraints. The constraints the user can manipulate are:

- *Network constraint.* The span of the view is defined by 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 potential downloads based on file size.

As one example, Figure 3.12 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



Title:

Artist: Story of the Year

Album:

Song Information

Filename: Until_The_Day_I_Die.mp3
Title: Until The Day I Die
Artist: Story of the Year
Album: Page Avenue
Size: 3682
Length: 3:55

Filename	Size
Baby_Blue.mp3	3472
Best_Of_Me.mp3	4081
Until_The_Day_I_Die.mp3	3682
And_The_Hero_Will_Drown....	3032
Harder_to_Breathe.mp3	2712
Promise.mp3	4181
Anthem_Of_Our_Dying_Day...	3406

Limit file size to: Unlimited

☐ Limit hosts in context to those moving in the same direction

Restrict the physical size of the context (hop count)

0 2 4 6 8 10

Figure 3.11: The subscription music service

than `maxSize`.

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 3.13 shows the application code used to generate an information tuple. This code is part of the `FileShareAgent`, which extends the `Agent` base class.

When the user performs a search, the "Search Results" tab displays the results. The user can choose to download a file, and the progress appears in the "Downloads" tab. The "Library" tab allows the user to manage his music files.

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,

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

Figure 3.12: Building a data constraint

```
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 3.13: Generating information tuples

or album. Because a music subscription service does not require atomicity guarantees, we use scattered probing operations. Figure 3.14 shows the code for querying the view.

Access Control

We also use this application as a test bed for the implementation of our access control mechanism. Because this represents a subscription music service, the application requires that any agent that downloads music proves that it has subscribed to the service. Each agent requesting music, therefore, must provide credentials that demonstrate this fact. These credentials include a unique agent id and a known phrase encrypted with a shared password provided in the user’s official registration from the music service. This password encrypted phrase authenticates the user as a subscriber. Since users share music only with others subscribed to the service, the agent also provides an access control policy which specifies that a requesting agent must have an agent id and must have the correct phrase encrypted with the subscription password. Successful decryption of the phrase by the

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

Figure 3.14: Accessing the view

```
Credentials c =
  new Credentials(getAgentID());
c.addProperty("Passphrase", encryptedPhrase);
```

Figure 3.15: Credentials for the music sharing agent

```
AccessControlPolicy policy =
  new AccessControlPolicy();
policy.addPropertyConstraint("Passphrase",
    String.class,
    new EquivalencyConstraintFn(encryptedPhrase));
policy.addPermittedOperation(Operations.RDP);
acf.addPolicy(policy)
```

Figure 3.16: Access control function for music sharing agent

receiving agent implies that the requesting agent holds the correct password. The code to define the credentials within the application is shown in Figure 3.15.

To build the access control policy, the agent defines the policy and adds it to the access control function as shown in Figure 3.16.

Lessons Learned

The subscription music service takes full advantage of the simplified programming interface in EgoSpaces. Using the view abstraction and coordination constructs, EgoSpaces allows the programmer to focus on how the music subscription application uses the information collected instead of having to explicitly discover and communicate with other agents in the network.

3.4.3 Collaborative Puzzle Game

The final application demonstrates how the EgoSpaces coordination model can be applied to cooperative work applications. In this example, several users collaborate to complete a puzzle whose pieces are distributed throughout the ad hoc network. Figure 3.17 shows the screens of two puzzle participants.

View Definition

This application uses the view constraints to limit the amount of data displayed based on properties of the puzzle to be solved. This view is more logical in nature and can be as simple as to contain only data constraints. The specific constraints used depend on a

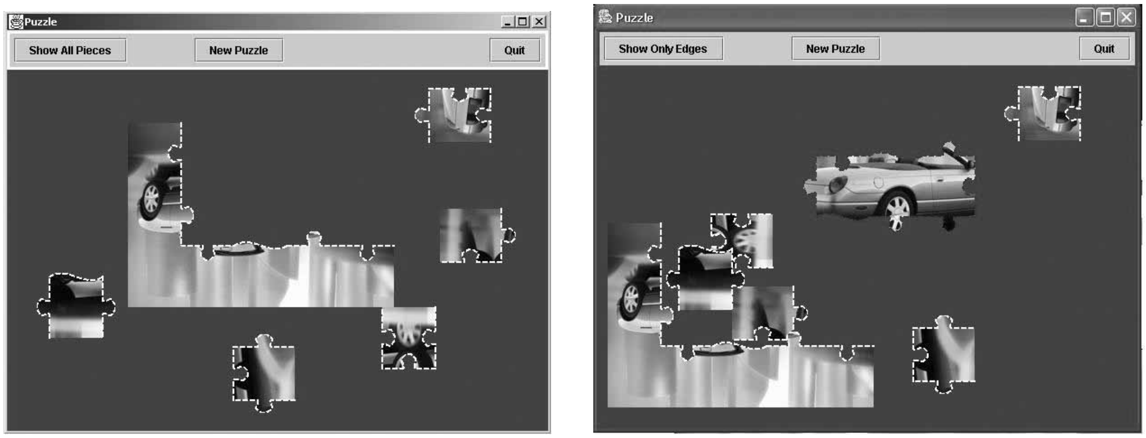


Figure 3.17: Two views of a puzzle game

particular user's goals; as one example, the view might be defined to contain only edge pieces, or only pieces of a certain color. An example of the data constraint required to define the former is shown in Figure 3.18. It makes use of the `EqualConstraint` function included in EgoSpaces that requires the field's value to equal a designated value.

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

Figure 3.18: Seeing only edge pieces

Puzzle players may find many different view definitions useful. If player agents have an idle status, a player might define a view that contains only pieces owned by idle players. If a player is facing a hole of a certain shape, he might specify his view to contain only the partially assembled piece he is working on and any pieces that are the correct shape for the hole. In the puzzle application, choices for defining these views are provided through a series of menus and dialog boxes.

Agent Interaction

One player in the game initializes the puzzle by loading an image. The pieces of the puzzle are represented by tuples in the data space of the agent initializing the puzzle. Each agent (representing a single player in the puzzle game), can define views that determine which puzzle pieces are displayed at a given time. Each agent initially starts with the maximal view, i.e., the view contains all pieces owned by any connected agents. As new agents connect, they too define this view and can see the puzzle pieces available in the system. A user can select a piece by clicking on it. When the user does so, the tuple corresponding to the puzzle piece is removed from its owner and placed in the user's local data space. To all users, this change appears as a change in the color of the border of the displayed puzzle piece. Players can assemble their pieces, and these changes are also reflected in the displays of connected agents.

When a user defines a different view of the puzzle pieces, the display changes appropriately. For example, if the user defines a view to contain only edge pieces, the player will see only these pieces, and all of the interior pieces are hidden. This is the view seen by the agent on the left-hand side of Figure 3.17. The player on the right has the default view and sees all the pieces. Changes made by the player on the left are displayed to the player on the right, but the reverse is not necessarily true. This is because the player on the right may make changes that affect only interior pieces not included in the other player's view.

Lessons Learned

In the previous two application scenarios, the view definitions were based on obvious notions of distance and relative location. With the puzzle game example, on the other hand, we see that the same abstractions can be used to define more logical views in perhaps smaller scale networks where a user wants to interact with a subset of all of the available data. While the particular subset was determined partially by the data's location in the previous examples, in the puzzle game only properties of the data or agents matter. Other applications that involve cooperative work by distributed parties can be implemented in a similar way. If the collaborative project does span a large-scale network, the application can be extended to account for the relative locations of the data items, in much the same way as in the music sharing example.

3.5 Chapter Summary

In this chapter, we realized the conceptual model from Chapter 2 in a middleware implementation. The interfaces this middleware offers to the application developer provide natural coordination constructs by representing the application's context as a data structure that changes as the environment changes. With this representation, it is natural for the middleware to keep the data structure consistent with the data available in the environment, and the application developer merely has to specify the context and interact with a well-known type of data structure, the tuple space. We also discussed the details of the implementation of the middleware, and gave examples of its uses by describing how we built a few applications using the middleware. One of the most novel aspects of this middleware systems revolves around the asymmetric coordination among connected parties. To fully provide this style of communication within an ad hoc network, we had to reexamine the available communication protocols. We discuss this process in the next chapter and present a new protocol that specifically allows asymmetric coordination.

Chapter 4

Supporting Context-Sensitive Communication in Ad Hoc Networks

To support the middleware described in the previous chapter, we must facilitate communication among the hosts in the ad hoc network. In this section, we first explore the capabilities of currently available communication protocols. We will explain why these protocols do not suffice for our purposes and will discuss an algorithm and associated protocol for mediating the style of communication among hosts needed to support the view abstraction for the EgoSpaces middleware. Finally, we provide an initial analysis of the protocol through a simulation evaluation.

4.1 Communication in Ad Hoc Networks

Most protocol work in ad hoc networks has focused on creating routing algorithms tailored to this novel environment. Because a protocol for gathering context information from a surrounding subnet requires communicating with a set of nodes in the ad hoc network, it will either use an existing ad hoc routing algorithm or make use of similar interactions in defining its own specialized behavior. In this section, we review existing work in developing ad hoc routing protocols and examine applying these techniques directly to context acquisition in the style discussed in the previous section.

Routing protocols for traditional wired networks do not function well in the ad hoc environment because of the special conditions encountered in this new type of network. Hosts in mobile ad hoc networks are constantly moving, and hosts that are encountered once may never be encountered again. Ad hoc routing protocols can generally be divided into two categories. Table-driven protocols, such as Destination Sequenced Distance Vector (DSDV) routing [46] and Clusterhead Gateway Switch Routing [15] mimic traditional routing protocols because they maintain consistent up-to-date information for routes to all other nodes in the network [53]. This class of algorithms is based on modifications to the classical Bellman-Ford routing algorithm [13]. Maintaining routes for every other node in the network can become quite costly, but applications gain decreased latency in response time. Performance comparisons [6] have shown that, while the overhead of DSDV is predictable, the protocol can be unreliable. The overhead can be lessened by utilizing routing protocols from the second class, source initiated on-demand routing protocols. By choosing protocols from the second class, an application trades off latency for decreased bandwidth utilization. On-demand routing creates routes only when requested by a particular source and maintains them only until they are no longer wanted. Ad-Hoc On-Demand Distance Vector (AODV) routing [47] builds on the DSDV algorithm but minimizes routing overhead by creating routes on demand. Dynamic Source Routing (DSR) [36] requires that nodes maintain routes for source nodes of which they are aware in the system. Finally, the Temporally Ordered Routing Algorithm (TORA) [44] uses link reversal to present a loop-free and adaptive protocol. It is source initiated, provides multiple routes, and has the ability to localize control messages to a small set of nodes near the occurrence of a topological change. Another type of routing that relates well to the work presented here is Distributed Quality of Service Routing [12]. In this scheme, routes are chosen from the source to the destination based on network resources available along that path.

While this is not an exhaustive survey of the current ad hoc routing protocols, it highlights the well-known and fundamental approaches to routing in ad hoc networks. Collectively, these protocols show the diversity available among solutions tailored to the ad hoc mobile environment. These solutions, however, do not meet the communication requirements of an individualized context-aware application in an ad hoc network. The main gap between the services provided by these protocols and the needs of a context-aware application lies in the fact that each of the ad hoc routing protocols described requires a known source and a known destination. Instead, context-aware programs as described in the previous section require the ability to abstractly specify the group of hosts with which to communicate.

Such communication with a subset of nodes in a network is commonly accomplished using multicast routing protocols. One possible solution for context-aware applications would build a multicast tree or mesh for the neighborhood defining the context and then send messages over this structure. Multicasting in ad hoc networks has received much attention. Early approaches used the shared tree paradigm commonly seen in wired networks, adapting these protocols to account for mobility [16, 27]. More recent work in ad hoc multicasting has realized that maintaining a multicast tree in the face of a highly mobile environment can drastically increase the network overhead. These research directions have led to the development of shared mesh approaches in which the protocol builds a multicast mesh instead of a tree [4, 39]. Both the multicast tree and mesh protocols use a shared data structure approach. That is, they assume that, for a given multicast group, there may be multiple senders. These senders share the tree built for the group to route their messages. While a shared approach might optimize a solution, the personalized needs of context-aware applications require individualized structures. There is no need to create a shared data structure. Also, a sender is guaranteed that its messages will be received by all members of the multicast group, but these members must initially register with the group. While these protocols address the mobility that causes nodes to join and leave the group, context-aware applications cannot use such a registration mechanism because they inherently have no knowledge of the identities of their communication partners. Instead, a particular query should reach only the nodes that satisfy the context specification at the time of the query's live in the system.

In summary, we build on the unicast and multicast protocols described above because we must address many of the same concerns as these protocols. Like them, our solution must account for frequent mobility of hosts, the transient nature of connections, and the changing properties of both the hosts and links in the network. Our approach must also address additional challenges. First, a host does not necessarily know the ids of the hosts with which it wishes to communicate. Instead, the host knows something about the properties of the paths that are acceptable to use for communication. Only hosts that lie on these paths (e.g., on a path that does not exceed a certain physical distance). Second, any data structure built over the network must guarantee that the path used to communicate with a host in the context satisfies the requirements for inclusion in the context. Finally, we do not need to search the whole network for possible paths. Instead, the search for hosts with which to communicate can be localized to a region of the network satisfying the needs of a particular context-aware application.

4.2 Network Abstractions for Context Provision

Extending the availability of contextual information beyond a host's immediate scope is facilitated by an abstraction of the network topology and its properties. Without this facility, the programmer must explicitly program at the socket level to find and connect to all of the desired hosts. Additionally, he must directly access the sensors that provide context information, and he must know how to interact with each different type of sensor. By abstracting these properties, we provide the programmer with a more logical view of the available resources and unify his interactions with different types of context sensors. After specifying some constraints that include the application's specific definition of distance and a maximum allowable distance, an application on the reference host would like a qualifying list of acquaintances to be generated. That is:

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

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

Once a weight has been defined and calculated for each link in the network, a cost function specified by the application can be evaluated over these weights to determine the cost of a particular path in the network. In a real network, multiple paths are likely to exist between two given nodes. Therefore, we build a tree rooted at the reference host that includes only the lowest cost path to each node in the network. We will see later in this section and in the subsequent sections that this tree and the paths composing it have several nice properties that we will take advantage of in building and maintaining the tree. Because we aim to restrict the scope of an application's context, calculating the lowest cost to every node in the network is not reasonable. To limit the context specification, we require the application to specify a bound for its cost function. Nodes to which the cost is less than the bound are included in the context. In the remainder of this section, we provide formal descriptions of the weights, cost function, and bound for the cost function. Throughout these descriptions, we will revisit the hop count example as a tool for understanding the definitions.

4.2.1 The Physical Network

Different properties of the physical network may interest different applications. Because each application individually specifies which properties of hosts and links to use in its context specification, each application has its own interpretation of the physical network. To begin mapping the ad hoc network to an abstract space, we represent the entire network as a graph $G = (V, E)$ where mobile hosts are mapped to V, the graph's vertices, and the communication links between hosts are mapped to E, the graph's edges. In the ad hoc network, every host and link has attributes that we map to the abstract space represented by the graph G by placing values on every vertex and edge. We quantify relevant properties of a mobile host (chosen based on an application's needs) as a value ρ_i on the vertex $v_i \in V$

representing the mobile host in the graph. Formally, $p : v \rightarrow R$. The value of ρ_i can combine a host's battery power, location, load, service availability, etc. We quantify the properties of a network link as a value ω_{ij} on the edge $e_{ij} \in E$ representing the edge in the graph. Formally, $\omega : E \rightarrow \Omega$. The value of ω_{ij} can combine values representing a link's length, throughput, etc. Further examples of these weights will be given in Section 4.3.

4.2.2 Logical View of the Network

Each application creates a logical view of the network based on the context data that interests it. We designate an application's logical network $\overline{G} = (\overline{V}, \overline{E})$, formed from the original mapping G. We use the information about node and link properties to create a topological *distance* between each pair of connected nodes in the logical network $\overline{G}$ by combining the quantifications of node properties and link properties into weights on edges in $\overline{G}$. Given an edge $e_{ij} \in E$ from the original mapping G and the two nodes it connects $v_i, v_j \in V$, the weights of the two nodes ρ_i and ρ_j are combined with the weight of the edge ω_{ij}, resulting in a single weight m_{ij} on the edge $\overline{e_{ij}} \in \overline{E}$ in the logical network $\overline{G}$. No host $\overline{v_i} \in \overline{V}$ in the logical network has a weight. Formally, this projection from the physical world to the virtual one can be represented as:

$$\Gamma : R \times R \times \Omega \rightarrow M$$

or more specifically:

$$m_{ij} = \Gamma(\rho_i, \rho_j, \omega_{ij}).$$

The value of m_{ij} is defined only if nodes v_i and v_j are connected as we assume $m_{ij} = \infty$ for missing edges.

4.2.3 The Path Cost Function

Given the logical view of an ad hoc network in which each edge has a weight, we need to assign a cost from the reference node $\alpha \in \overline{V}$ to any reachable node $\beta \in \overline{V}$. An application running on the reference node specifies a cost function providing instructions to the lower layer on calculating the cost of a given path in the logical network $\overline{G}$. A path $p = \langle \overline{v_0}, \overline{v_1}, \cdots, \overline{v_k} \rangle$ indicates the path originating at the reference host, now referred to as $\overline{v_0}$, traversing nodes $\overline{v_1}$ through $\overline{v_{k-1}}$ and terminating at $\overline{v_k}$. As a shorthand, we introduce the notation p_n to indicate the portion of the path P from $\overline{v_0}$ to $\overline{v_n}$ where $\overline{v_n}$ is one of the nodes on the path. Using this notation, $P_k = P$.

Given a path in $\overline{G}$, the topological cost of the path from the reference node $\overline{v_0}$ to a host $\overline{v_k}$ can be defined recursively using a path cost function *Cost*, specified by the reference host's application. The cost of the path from $\overline{v_0}$ to $\overline{V_k}$ along a particular path P_k

is represented by $f_{v_0}(P_k)$. The recursive evaluation to determine this value is:

$$f_{v_0}(P_k) = Cost(f_{v_0}(P_{k-1}, m_{k-1,k})$$

$$f_{v_0}(\langle\overline{v_0}\rangle) = 0$$

Figure 4.1 shows the recursive cost function pictorially. The figure shows that the cost of, or distance to, host $\overline{v_i}$, represented by ν_i results from the evaluation of the application-specified cost function over the weight of edge $\overline{e_{i-1,i}}$ and the cost of, or distance to, host $\overline{v_{i-1}}$ Figure 4.1

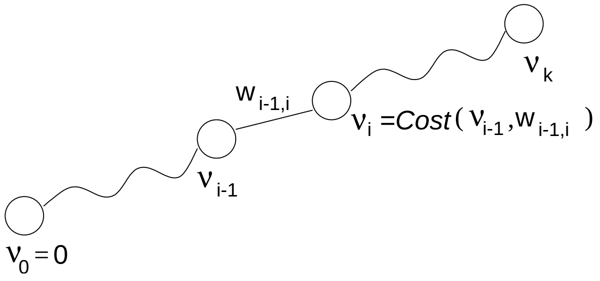


Figure 4.1: The recursive cost function

For the field research application scenario discussed in Section 1.1, assume the weight of each link in the network is a combination of the total latency incurred in traversing the link and the inverse of the bandwidth of the link. In this case, the cost function is additive with respect to the latency, but maximizing with respect to the inverse of the bandwidth. The entire cost function and its reasoning are presented in Section 4.3. Additional examples will also be presented.

4.2.4 The Minimum Cost Path

In an arbitrary graph multiple paths may exist from α to another node β each with an associated cost. For each of these nodes, β, reachable from α, one of these paths is the shortest path. We call the cost of this path $g_\alpha(\beta)$. That is, for all paths from α to β,

$$g_\alpha(\beta) = \min_{\textit{over all } P \textit{ from } \alpha \textit{ to } \beta} f_\alpha(P)$$

There is a shortest path tree T spanning the graph representing the ad hoc network rooted at the reference node α. For all nodes β in this tree, the path from α to β in T has cost $g_\alpha(\beta)$. Figure 4.2 shows the the shortest path tree built over an example network of nodes. The numbers near each edge in the graph represent the weight (m_{ij}) on the link in the logical network. The cost function used in this example simply adds the weights of the links

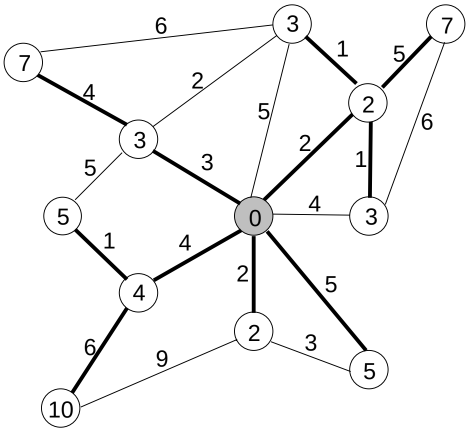


Figure 4.2: The logical network and shortest path tree

along the path. The links that form the shortest path tree are darkened. Though the graph shown contains multiple paths from the reference node to each other node, the tree includes only one of the shortest paths to each node.

4.2.5 Ensuring Boundedness

Given a shortest path tree constructed over an ad hoc network, we define a bound on nodes included in the context. Any nodes for which the cost of the shortest path is greater than the bound are not included in the set of acquaintances. In the case of a field researcher needing to utilize another researcher's video information, the context might be bounded by a combination of the tolerable latency of the video program and the required bandwidth. Therefore, only hosts to which the latency is less than some maximum while the bandwidth satisfies some end-to-end requirement will be included in the context. The next section explores this bound in more detail.

Figure 4.3 shows the shortest path tree from Figure 4.2. This time, however, only the shortest cost paths are shown, with the cost of the shortest path inside each node. The figure shows the bound D, indicated by the dashed circle. Nodes inside the dashed circle are part of host α's acquaintance list Q_α, while nodes outside the dashed circle are not part of this list and will not be included in queries over Q_α.

Notice that this bound is useful only if the value of the cost of the shortest path is strictly increasing as the path extends away from the reference node. That is, if we number the nodes on a path $\langle 1, 2, \ldots, i, \ldots, n \rangle$ and designate the value of the cost to node i as ν_i, we require that $\nu_i > \nu_{i-1}$. This guarantees that a parent in the tree is always topologically closer to the root than its children, i.e., that the cost of the path to the parent is always less than the cost to the child. If the cost of a path in the tree strictly increases as the distance

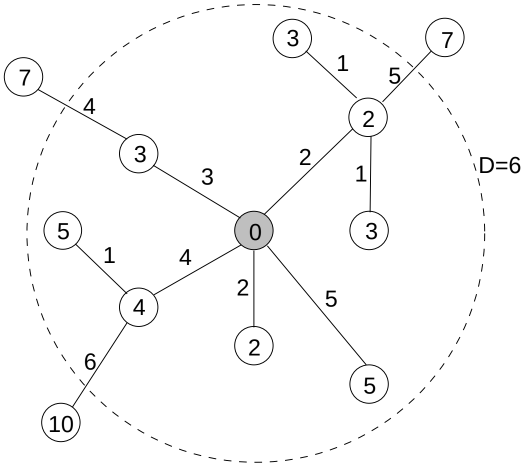


Figure 4.3: The bounded shortest path tree

from the reference node grows, the application can enforce a topological constraint over the search space by specifying the bound D over the value ν returned by the cost function. The lower level protocol can stop propagating context building messages once it reaches a node on the path that has a distance (cost) greater than D. In the particular case shown in Figure 4.3, context building messages are no longer propagated once a node with a cost greater than 6 is reached. This strictly increasing requirement is necessary to prevent an infinite number of nodes on a path having the same cost, resulting in a context that cannot be bounded.

Defining the properties that contribute to a link's weight and constructing cost functions that use these weights is the most important aspect of this network abstraction. In the next section we show how the use of the metric concept allows the definition of a variety of network abstractions in a simple, expressive, and flexible manner.

4.3 Sample Metrics

In this section, we explore more sophisticated metrics and relate them to the application environments in which they may be useful. The most basic metric consists of link weights of one and a cost function that adds the weights on the links. This particular metric allows the application to restrict its context based on the network hop count; only nodes within a specified number of hops will contribute to the context. Most context-aware applications, however, have more complicated reasons for restricting their operating context.

4.3.1 Building Floor Restriction

We first introduce a simple metric that constructs a context based on the floor locations of sensors in a building. The building has a fixed infrastructure of sensors and information appliances providing contextual 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 example, as an engineer moves through the building, he wishes to see structural information not for the whole building, but only for his current floor and the floors adjacent to it.

Metric

In comparison to the other metrics we will present, this one is more logical in nature. The weight on link $\overline{e_{ij}}$ connecting nodes i and j accounts for the floors of the nodes. We define

$$\rho_i = \textit{node floor \#}$$

so that the value of ρ corresponds to the integer floor number where the node is located. We do not use the link weight, ω, in this case. To generate logical weights, we combine the floors of nodes i and j so that m_{ij} consists of the range of floors of the two nodes

$$m_{ij} = \{\rho_i, \rho_{i+1}, \ldots, \rho_{j-1}, \rho_j\}.$$

For example, if nodes on floors 2 and 4 are directly connected, the weight on the link between them will be the range $\{2, 3, 4\}$.

Using a cost function based only on this property, however, does not guarantee that the metric will increase. For this reason, we add a hop count parameter. In this case, the count measures the number of network hops the path has taken without moving to a new floor (i.e., a floor that the path has not traversed in the past). The cost function's value ν at a given node consists of two values:

$$\nu = (r, c).$$

The first of these values, r, is the range of floors covered by the network path. The second value, c, counts the number of hops taken in the current range of floors.

Formally, the cost function generates a cost for each node according to:

$$f_{v_0}(P_k) = \begin{cases} (f_{v_0}(P_{k-1}).r, f_{v_0}(P_{k-1}).c + 1) & \text{if } m_{k-1,k} \in f_{v_0}(P_{k-1}).r \\ (f_{v_0}(P_{k-1}).r \cup m_{k-1,k}, 0) & \text{otherwise} \end{cases}$$

For ranges, we use the notation $\in$ to refer to the fact that one range is entirely contained in another. The union of two ranges ($\cup$) refers to the range that exactly covers the two input ranges. The first case in the cost function above corresponds to the situation when the current link does not move to a new floor. In this case, the range of floors for the path is equal to the range of floors at the previous node. The hop count is incremented by one. The second case corresponds to the case where the current link does move to a new floor. The range of floors for the path is the union of the previous node's range with this link's range. The counter is reset to 0. Note that this cost function is guaranteed to increase at every hop because either the range expands or the hop count is incremented.

To specify a bound on this cost function, the application specifies the acceptable range of floors and a hop count. For the example introduced in this section, the building engineer might define the bound:

$$bound = (\{f-1, f, f+1\}, 10)$$

where f is the number of the engineer's current floor, and this context contains only hosts on his current floor or adjacent ones. As he moves throughout the building, his f changes, and his context changes to reflect this. The use of 10 as a hop count is fairly arbitrary; the engineer's application will choose something large enough to ensure that he includes as many nodes as possible while ensuring that performance does not degrade.

4.3.2 Network Latency

Next we design a metric for the application scenario introduced previously Briefly, this application consists of field researchers who share sensor data and video feeds. It is likely that the context requirements for each of these tasks will be different due to differences in data being gathered. For each such task, the researcher builds a network abstraction to define the particular context. Here we focus on the video transmission.

Metric

In this abstraction, the weight on link $\overline{e_{ij}}$ connecting two nodes accounts for the node-to-node latency. We will show later how this metric can be extended to account for the bandwidth in addition to the latency. These are not the only network measurements that might affect video transmissions; more complicated metrics could account for additional constraints. To create this metric, we define

$$\rho_i = \frac{node\ packet\ processing\ latency_i}{2}$$

where $node\ packet\ processing\ latency_i$ is the average time between when node i receives a packet and when it propagates the packet (i.e., the time node i takes to process the packet, if any). We use only half of this number to avoid counting the node's latency twice if the node is in the middle of the path. This latency value suffices under the assumption that the incoming latency for the node is approximately equivalent to its outgoing latency. We define

$$\omega_{ij} = \textit{link latency}_{ij}$$

where *link latency* is the time it takes for a message to travel from node i to node j.

Possible mappings to the logical network abound; the link latency and node latency can each be given a different importance by weighting the ρ and ω values. For simplicity's sake, the value m_{ij} in the logical network is defined as

$$m_{ij} = \rho_i + \rho_j + \omega_{ij}$$

The cost function is then:

$$f_{v_0}(P_k) = f_{v_0}(P_{k-1}) + m_{k-1,k}$$

This cost function is guaranteed to increase at every hop because it is additive and each latency term must be strictly positive. A bound on this cost function is defined by a bound on the total latency.

Metric Extension

Because the usefulness of the video feed might also depend on the bandwidth in addition to the network latency, we show how the previous metric can be easily extended to include a bandwidth component. In this case, the ρ values remain the same, but the ω values are extended to a pair of values, where the second number in the pair relates to the link's bandwidth:

$$\omega_{ij} = (\textit{link latency}_{ij}, \frac{1}{\textit{bandwidth}_{ij}})$$

We treat this pair of values as an array; to access the latency component, we use the notation: $\omega_{ij}[0]$, and to access the bandwidth component, we use the notation: $\omega_{ij}[1]$. It is reasonable to use the inverse of the bandwidth because a connection with a higher bandwidth can be considered "shorter," while one of lower bandwidth "longer."

We continue with a simple mapping to the logical network, where the value of m_{ij} is defined as a pair of values:

$$m_{ij} = ((\rho_i + \rho_j + \omega_{ij}[0]), \omega_{ij}[1])$$

To access the first and second components of m_{ij}, we use the same notation as above (e.g., $m_{ij}[0]$ refers to the total latency component of the weight).

The cost function computes a pair of values for each node's cost in the network. The first value corresponds to the total latency experienced on the path to the node. The second value stores the minimum bandwidth as yet encountered:

$$\nu = (latency, bandwidth)$$

The cost function is then:

$$f_{v_0}(P_k) = (f_{v_0}(P_{k-1}).latency + m_{k-1,k}[0], \mathbf{max}(f_{v_0}(P_{k-1}).bandwidth, m_{k-1,k}[1]))$$

We use a maximum function to compute the minimum bandwidth encountered to account for the fact that the bandwidth component of the weight is the inverse of the link's bandwidth. Notice that this cost function is guaranteed to increase at every hop. Because the latency is completely additive, the *latency* component increases every hop. Additionally, because we take the maximum of the *bandwidth* component each hop, it is guaranteed not to decrease.

A bound on this cost function consists of two components: a bound on the total latency, and a bound on the bandwidth. When either of the cost function components increases beyond its corresponding bound, the path's cost is no longer satisfactory, and nodes further along the path are not included in the context.

4.3.3 Physical Distance

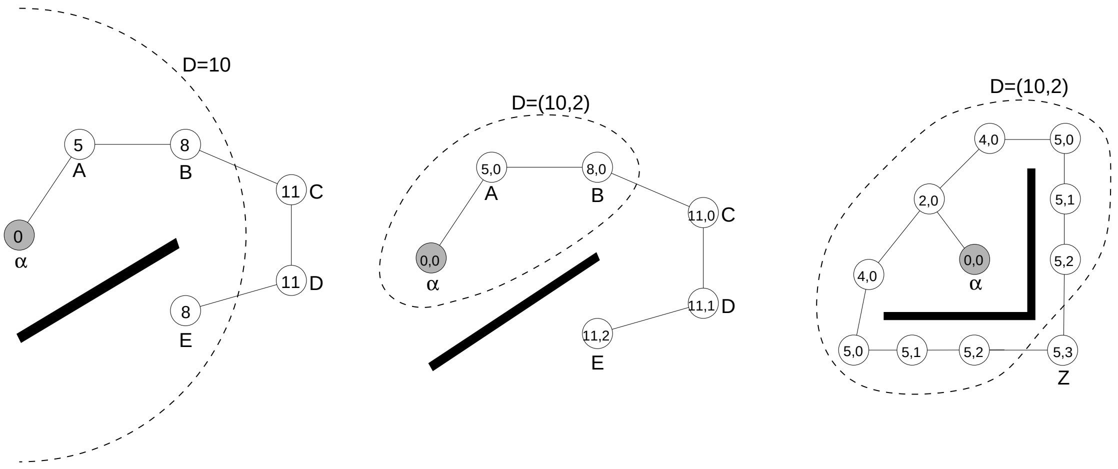
Finally, we present a general-purpose metric based on physical distance. Imagine, for example, a network consisting of vehicles on a highway. Each vehicle gathers information about weather conditions, highway exits, accidents, and traffic patterns. As a car moves through this environment, the driver wants to gather information that will affect his immediate trip. This data should be restricted to information within a certain physical distance (e.g., within a mile). A number of other everyday applications as well as military applications also benefit from this type of context specification.

Metric

As the application description intimates, the calculated context should be based on the physical distance between the reference host and other reachable hosts. The weight placed on edges in the logical network reflects the distance vector between two connected nodes and accounts for both the displacement and the direction of the displacement between two connected nodes:

$$m_{ij} = \vec{\mathrm{IJ}}$$

Figure 4.4a shows an example network where specifying distance alone causes the cost function to violate the requirement that the function be strictly increasing. The figure



(a) **(b)** **(c)**

$$f_{v_0}(P_k) = \begin{cases} (|f_{v_0}(P_{k-1}).\mathbf{V} + m_{k-1,k}|, f_{v_0}(P_{k-1}).C, f_{v_0}(P_{k-1}).\mathbf{V} + m_{k-1,k}) \\ \qquad\qquad \text{if } |\mathrm{f_{v_0}(P_{k-1})}.\mathbf{V} + \mathrm{m_{k-1,k}}| > \mathrm{f_{v_0}(P_{k-1})}.maxD \\ \\ (f_{v_0}(P_{k-1}).maxD, f_{v_0}(P_{k-1}).C + 1, f_{v_0}(P_{k-1}).\mathbf{V} + m_{k-1,k}) \\ \qquad\qquad \text{otherwise} \end{cases}$$

(d)

Figure 4.4: (a) Physical distance only; (b) Physical distance with hop count, restricted due to distance; (c) Physical distance with hop count, restricted due to hop count; (d) The correct cost function

shows the shaded reference host, α, and the results of its specified cost function. The numbers on each node indicate the node's calculation of its cost, given the reference host's cost function. The cost function shown in this figure simply assigns as the cost of a node the distance to the reference. The bound the application specified in this example is $D = 10$. Notice that nodes C and D are outside the context while E should be placed inside the context. In this case, node A cannot communicate directly with node E due to some obstruction (e.g., a wall) between them. When the cost of the path is strictly increasing, host C knows that no hosts farther on the path will qualify for context membership. In this example, this condition is not satisfied, however, and no limit can be placed on how long context building messages must be propagated.

To overcome this problem, we base the cost function on both the distance vector and a hop count. The cost function's value ν at a given node consists of three values:

$$\nu = (maxD, C, \mathbf{V})$$

The first value, $maxD$, stores the maximum distance of any node seen on this path. This may or may not be the magnitude of the distance vector from the reference to this host. The second value, C, keeps the number of consecutive hops for which $maxD$ did not increase previously along the path. The final value, $\mathbf{V}$, is the distance vector from the reference host to this host; it is used to keep track of the path's location relative to the reference host.

Specifying a bound for this cost function requires specifying a bound on both $maxD$ and C. A given bound has two values, and if a host's ν values meet or exceed either of these values, the host is outside the bound. Specifically, a host is in the specified context only if both its $maxD$ and C are less than the values specified in the bound. As will become clear with the definition of our cost function, neither the value of $maxD$ nor the value of C can ever decrease. Also, if one value remains constant for any hop, the other is guaranteed to increase, therefore this cost function is strictly increasing.

Figure 4.4d shows the cost function for this example. In the first case, the new magnitude of the vector from the reference host to this host is larger than the current value of $maxD$. In this case, $maxD$ is reset to the magnitude of the vector from the reference to this host, C remains the same, and the distance vector to this host is stored. In the second case, $maxD$ is the same for this node as the previous node. Here, $maxD$ remains the same, C is set to its old value incremented by one, and the distance vector to this host is stored.

Figure 4.4b shows the same nodes as Figure 4.4a. In this figure, however, the cost function from Figure 4.4c assigns the path costs shown. The application specified bound shown in Figure 4.4b is $D = (10, 2)$ where 10 is the bound on the maximum distance ($maxD$) and 2 is the bound on the maximum number of hops for which the maximum distance did not change (C). The values shown on the nodes in the figure reflect the pair $maxD$ and C. As the figure shows, because the cost function includes a hop count and is based on maximum distance instead of actual distance, node C can correctly determine that no host farther on the path will satisfy the context's membership requirements. In this case, nodes C, D, and E lie outside of the bound due to the maximum distance portion of the cost function. Figure 4.4c shows the same cost function applied to a different network. In this case, while the paths never left the area within distance 10, node Z still falls outside the context because the maximum distance remained the same for more than two hops.

4.4 Context Calculation and Maintenance Protocols

The protocol we developed for computing the context based on the tree structure described above takes advantage of the fact that an application running on a reference host α does not necessarily need to know which other hosts are part of the acquaintance list. Instead, the application needs to be guaranteed that, if it sends a message to its acquaintance list, the message is received only by hosts belonging to the list and that all hosts belonging to the list receive the message. The protocol described here builds a tree over the network corresponding to a given application's acquaintance list. By nature, this tree defines a single route from the reference node to each other node in the acquaintance list. To send a message only to the members of the acquaintance list, an application on the reference node needs only to broadcast the message over the tree.

4.4.1 Assumptions

The protocol presented next relies on a few assumptions regarding the behavior of the underlying system. First, it assumes that there exists a message passing mechanism and that this mechanism guarantees reliable delivery with the associated acknowledgments. These acknowledgments therefore lie outside the concern of this protocol. Providing this type of guarantee in the highly dynamic ad hoc network can prove difficult and has been much studied. Work on building consistent group membership [33], for example, ensures stable communication given information about hosts' positions, relative velocities, and properties of the wireless network in use.

The protocol also assumes that disconnection is detectable, i.e., when a link disappears, both hosts that were connected by the link can detect the disconnection. Finally, the protocol requires that all configuration changes and an application's issuance of queries over the context are serializable with respect to each other. In the case of this particular protocol, a configuration change is defined as the change in the value of the metric at a given link and the propagation of those changes through the tree structure.

More specifically to our protocol, we assume that the underlying system maintains the weights on the links in the network by updating these weights in response to changes in the contextual information requested by the application. Additionally, we assume that the system calculates the weight for each link and that this weight information is available to our protocol. For each link it participates in, a host should have access to both the weight of the link and the identity of the host on the other side of the link.

4.4.2 Protocol Foundations

As intimated in the introduction to this section, our protocol takes advantage of the fact that an application running on a reference host specifies the context over which it would like to operate, but the application does not need to know the identities of the other hosts in this context. Therefore, the context computation can operate in a purely distributed fashion, where responses to data queries are simply sent back along the path from whence they came. The protocol is also on-demand in that a shortest path tree is built only when a data query is sent from the reference node. The context specification and the information necessary for its computation are piggy-backed on this data message.

Query, q

$q.initiator$	the initiator's id
$q.num$	the application sequence number of q
$q.s$	the sender of this copy of q, NOT necessarily the reference node
$q.sd$	the distance from the reference to $q.s$
$q.d$	the distance from the reference to the host at which the query is arriving
$q.D$	the bound on the cost function
$q.Cost$	the cost function
$q.data$	the application level data associated with this query

Figure 4.5: The Components of a Query

Figure 4.5 shows the components of a query. The query's sequence number allows the protocol to determine whether or not this query is a duplicate. This prevents a particular host from responding to the same query multiple times. As discussed later, the host's response contains application-level data for the reference host.

It should be noted here that we will talk about a query's sender. This is not a term used interchangeably with the query's reference. The reference for a query is the host running the application for which the context is being constructed. The sender of a query is the most recent host on the path to the current host.

The detailed explanation of the protocol is divided into three sections: tree building, tree maintenance, and reply propagation. After the presentation of the building of the shortest path tree, it will be easy to add maintenance to the algorithm. The subsequent description of reply propagation is fairly straightforward. Before we describe the algorithm itself, however, we present the information that a given host needs to remember about a single context specification.

4.4.3 Protocol State Information

Figure 4.6 shows the state variables that a host participating in a context computation must hold. This is the information for a host β that is part of α's acquaintance list. This shows only the information needed for participation in α's acquaintance list; in general, an

State

id	this host's unique identifier
num	application sequence number, initialized to -1
d	the distance from the reference node, initialized to ∞
p	this host's parent in the tree
pd	parent's distance (or cost) from reference node
D	bound on the cost function
$Cost$	cost function
C	set of connected neighbors, the weight of the link to each, and the cost of the path to the neighbor. As a shorthand, we refer to the weight of a link to neighbor c as w_c and the cost of the path to c as d_c.
I	a subset of C containing the connected neighbors that are in the reference's context, initially empty. These will be used later to clean up memory used for the protocol

Figure 4.6: State Variables

individual host would be participating in multiple acquaintance lists and would therefore have a set of these variables for each such list.

Most of the state variables are self-explanatory. One worth discussing is the set C, which holds the list of all connected neighbors. Each of these neighbors has a link to it from this host; the weight of that link is stored in C and is referred to as w_c for some $c \in C$. This set is also used to store other paths to this host. If a host receives a query from host c that would give it a cost $d_c < D$ that it does not use as its shortest path, it remembers c's cost, and associates it with c in C. When we discuss maintenance of the tree later, this information will prove useful in quickly finding a new shortest path to replace a defunct path.

4.4.4 Context Building

Any information that a particular host requires for computing another host's context arrives in a query; there is no requirement for a host to keep any information about a global state. Because the protocol services queries on-demand, it does not build the tree until a request is made. To do this most efficiently, the information for building and maintaining the tree is packaged with the application's data queries. An application with a data query ready to send bundles the context specification with the query and sends it to all its neighbors that also lie within the context. When such a query arrives at a host in the ad hoc network, it brings with it the cost function and the bound which together define the context specification. It also brings the cost to this host.

Any query a host receives is guaranteed to be within the context's bound because the sending node determines the destination node's cost before sending it the query. Only neighbors that fall within the bound are sent the message. The first query that arrives at a host is guaranteed to have a cost lower than the one already stored because the cost is initialized to ∞. Subsequent copies of the same query are disregarded unless they offer a lower cost path. As shown in the second **if** block of the QueryArrives action in Figure 4.7, when a shorter cost path is found, the cost of the new path, the new parent, and the new parent's cost are all stored. Also, the query is propagated to non-parent neighbors whose distance will keep them inside the context specification's bound. This is done through the *PropagateQuery* function, described with the protocol's other support functions in Figure 4.8. For each non-parent neighbor, c, this host applies the cost function to its own distance and the weight of the link to c. If this results in a cost less than the context specification's bound, D, the host propagates the query to c. A host must propagate a query with a lower cost even if its application has already processed it from a previous parent because this shorter path might allow additional downstream hosts to be included in the context. Finally, upon reception of any query, the host adds the information about the parent to the set C.

```
Actions

QueryArrives($q$)
    Effect:
        if $q.num = num + 1$ then
            save query specific information ($Cost := q.Cost, D := q.D$)
            clear $C$
            record information ($d := q.d, p := q.s, pd := q.sd$)
            $Propagate\ Query(q)$
            $AppProcessQuery(q)$
            save the sequence number ($num := q.num$)
        else if $q.d < d$ then
            record information ($d := q.d, p := q.s, pd := q.sd$)
            $PropagateQuery(q)$
        end    update $C$ ($d_{q.s} := q.sd$)
```

Figure 4.7: Context Computation

When a host receives a query that it has not seen before (i.e., the sequence number of the arriving query is one more than the stored sequence number), the application automatically processes it regardless of whether or not it arrived on the currently stored shortest path. A host does not wait for more additional copies of a query to come *only* from its parent because it is possible that the path through the parent no longer exists

Support Functions

PropagateQuery(q)	–for each non-parent neighbor, c, send the query to c if $Cost(d, w_c) < D$ by calling SendQuery to c after setting $q.d = Cost(d, w_c)$ and $q.s = id$ in the query; update I to include exactly those c to which the query was propagated
AppProcessQuery(q)	–application processing of the data message part of the query
SendCleanUps	–for each non-parent neighbor, c, send a clean up message to c if $Cost(d, w_c) \geq D$ by calling SendCleanUp to c
PropagateCleanUps	–for every member of I, send a clean up message by calling SendCleanUp
PropagateReply(r)	–send the reply to p
AppProcessReply(r)	–application processing of the data message part of the reply

Figure 4.8: Support Functions

or that its cost has increased. If the path does still exist and is still the shortest path, the query will eventually arrive along that path, causing the cost to be updated and the effects to be propagated to the children. Upon receiving a new query, the host stores the cost of the query, the new parent, the new parent's cost, and the sequence number, then it propagates the query in the manner described above. Finally, the host sends the data portion of the query to the application for processing using the *AppProcessQuery* support function described in Figure 4.8.

Earlier, we introduced an example application in which a field researcher may need to collect temperature data to be associated with some field notes, but the researcher himself may not carry a thermometer. Other researchers in the field, however, may have thermometers whose data could be used. Once the researcher defines a context to include some thermometers (e.g., a context based on physical distance or thermometer accuracy), he issues a variety of queries over his context, depending on his needs. For example, he might use a one-time query if he simply needs to attach a single piece of temperature data to a note. On the other hand, if the surveillance of the target subject is an ongoing process and the temperature data needs to be constantly correlated with notes regarding the subject's behavior, the researcher needs a longer-lasting query. For example, the researcher may want to know when the temperature fluctuates a given number of degrees. Next, we classify various types of operations and show how our protocol is modified to handle these long lasting queries through tree maintenance.

4.4.5 Context Maintenance

An application can perform two different types of operations: transient and persistent. A transient operation is a one-time query or instruction. For example, in the traditional children's card game, *Go Fish*, a player A's request "Do you have a six?" would represent a transient query. All other players, if they are part of the context, can easily respond "yes" or "no" and move on. In a modified version of the game, player A might request to be notified when another player finds a six. This is an example of a persistent operation because the other players have to remember that another player asked for a six. As long as player A still wants a six, all players that enter the context have to be notified of the persistent operation. An application issues a persistent operation with an initial registration query. As long as the persistent operation remains registered, the associated query propagates to new hosts that enter the context. If a host moves out of the context, the persistent operation is deregistered at that host. When an application wants to deregister a persistent operation from the entire context, it issues a deregistration query which effectively deletes the operation from each host in the context.

The protocol presented in Figure 4.7 is sufficient if the specifying application issues only transient operations over its context. In this case, the context needs to be recomputed only if a new query is issued. Because the protocol propagates each query to all included neighbors of a host, the shortest path will be computed each time, even if the weights of the links have changed between the queries.

For transient operations alone, the protocol essentially rebuilds the shortest path tree each time a query is issued, on-demand. For these purposes, the only state a host needs to remember for a given context specification is its own current shortest distance, its parent, and the sequence number. It uses its distance to compare against other potentially shorter paths and the identity of its parent to return messages to the reference along the current shortest path. The need for the remaining state variables in Figure 4.6 becomes clear only when we introduce tree maintenance to the protocol. Because the protocol in Figure 4.7 does no maintenance on the tree, there is also no way for a host to recover the memory used by context specification's issued by hosts that have disconnected never to return.

At times, an application needs to register persistent operations on other hosts in its context. These persistent operations should remain registered at all hosts in the context until such time that the reference host deregisters them. An initial query over the context serves to register the persistent operation, and a later query, issued and propagated in a similar fashion, deregisters the operation. In such cases, the reference host's context needs to be maintained, even when no new queries are issued over it. The tree requires maintenance whenever the topology of the ad hoc network changes. Any topology change that affects the current context specification directly reflects as a change in at least one link's weight.

We assume that the underlying system brings such a change to the attention of both hosts connected by the link. That is, if weight, w_{ij} changes, then hosts v_i and v_j are both notified. Hosts whose costs grow as a result of a network topology change may have to be removed from the acquaintance list, while hosts that enter the context after the persistent query has been issued should be notified of the query. To do this, the system needs to react to changes in weights on links and recalculate the shortest paths if necessary. Again, we assume that topology changes are atomic with respect to the application's operations. In the case of persistent operations, this means that a topology change and the propagation of its effects are atomic with respect to the registration and de-registration of the persistent operations and the transmission of the results for these operations.

```
Actions

QueryArrives($q$)
    ... as before

WeightChangeArrives($wnew_{id}$)
    Effect:
        if $id = p$ then
            calculate the cost ($d := Cost(pd, wnew_{id})$)
            if $wnew_{id} > w_p$ then
                calculate shortest path not through $p$ ($minpath := \min_c Cost(d_c, w_c)$)
                if $minpath < d$ then
                    reset the cost ($d := minpath$)
                    assign new parent
                end
            end
            set the query fields ($q := \langle num, id, d, D, Cost \rangle$)
            $PropagateQuery(q)$
        else if $wnew_{id} < w_{id}$ then
            if $Cost(d_{id}, wnew_{id}) < d$ then
                recalculate cost ($d := Cost(d_{id}, wnew_{id})$)
                reset the parent ($p := id$)
                set the query fields ($q := \langle num, id, d, D, Cost \rangle$)
                $PropagateQuery(q)$
            end
        end
        store the new weight ($w_{id} := wnew_{id}$)
```

Figure 4.9: Context Computation and Maintenance

Because both hosts connected by the link are notified of any change, both can take measures to recalculate the shortest path tree. Figure 4.9 shows the same protocol presented in Figure 4.7. A new action, WEIGHTCHANGEARRIVES has been added to deal with the dynamic topology. This action is activated when the notification of a weight change arrives at a host. The weight changes are divided into two categories: the weight of the link to the parent has changed, and any other weight has changed.

In the first case, the path through the parent has either lengthened or shortened. If the length of the path through the parent has increased, then it is possible that the shortest path to this node from the reference node is through a different neighbor. The node sets its cost to be the minimum of the cost through the old parent and the shortest path through any other neighbor. To find the shortest path through a non-parent neighbor, the host accesses the information stored in the state variable, C. On the other hand, if the length of the path through the parent has shortened, the node should still be included in the context, and the shortest path to it from the reference should still be through the same parent. In either case, the node recalculates its distance and propagates the information to its neighbors, using the support function, *PropagateQuery*. The neighbors will then process the weight change information using the already discussed QUERYARRIVES action.

If the weight change has occurred on a link to a non-parent neighbor, then the change interests this host only if it causes the path through the neighbor to be shorter than the path through the parent. For this to be the case, the link's weight must have decreased. Because this host is storing distance information for all of its neighbors, however, it can simply calculate what the new distance would be, compare it to the stored cost, and reset its values if they have changed. If these calculations change the cost to the node, it should package the current context values in a query and propagate that query using the *PropagateQuery* support function.

The protocol presented in Figure 4.9 still does not free the memory used to store information about the reference host's context specification. For example, as a car moves across the country, it leaves information about its specified contexts on every other car it encounters. The car may never come back, so each car that was part of one of these contexts would like to recover its memory when it is no longer part of the context specified. We can build a clean up mechanism into the protocol as shown in Figure 4.10. Whenever it is possible that a change has pushed a host that was in the context out of the context, the parent should notify the child that its context information is no longer useful and should be deleted. There are two places in the algorithm where a change might push another node out of the context. The first is when a weight changes and the path through the parent becomes longer. Not only might this node be pushed out of the context, any of its descendants in the tree might also be pushed out. First, after calculating its new cost, the node should verify that it is still within the bound, D. If not, it should clean up its own storage. If this node is still within the bound, it propagates a copy of the current query to its neighbors that will remain within the bound and sends a message to the neighbors that are not within the bound instructing them to clean up this context specification's information if they know about it.

The other change required to the protocol occurs in the QueryArrives action. When a query arrives with a new sequence number, it is possible that the shortest path has increased in cost, thereby pushing neighbors out of the context. To account for this, after propagating the query to all neighbors within the bound, D, the host should also send a clean up message to all neighbors not within D.

```
Actions

QueryArrives($q$)
    ... as before

WeightChangeArrives($wnew_{id}$)
    ... as before

CleanUpArrives($id$)
    Effect:
        if $id = p$ then
            calculate shortest path not through $p$ ($minpath := \min_c Cost(d_c, w_c)$)
            if $minpath < D$ then
                reset the cost ($d := minpath$)
                reset the parent
                set the query fields ($q := \langle num, id, d, D, Cost \rangle$)
                $PropagateQuery(q)$
                $SendCleanUps$
            else
                $PropagateCleanUps$
                clean up local memory
            end
        else
            update $d_{id}$ in $C$
        end
```

Figure 4.10: Context Computation, Maintenance, and Clean Up

A new action, CleanUpArrives has been added to the protocol shown in Figure 4.10 to deal with the arrival of the clean up messages. If the clean up message comes from the parent, it is an indication that there no longer exists a path to the reference that satisfies the context specification's constraints. In this case, a new shortest path is selected using the information in C and the information propagated. If no qualifying shortest path exists, the local memory is recovered. In both cases some clean up messages are sent. If the clean up message comes from a node other than the parent, the state variable C needs to be updated to reflect that the cost to the source is ∞.

4.4.6 Reply Propagation

The previous discussions explain how an initiator's context is constructed over the network. Most applications will require responses from the hosts in their contexts. To guarantee the application's bound requirements, these responses must traverse the shortest cost path back to the initiator, constructed as part of the tree. Not only does the reply contain information requested by the initiator's application, it also contains protocol specific data to help the reply find the correct path home. Figure 4.11 shows the components of a reply.

Reply, r

$r.\mathit{initiator}$	the initiator's id
$r.\mathit{num}$	the application sequence number of the query the reply is in response to
$r.\mathit{id}$	the replying node's id
$r.\mathit{cost}$	the cost from the initiator to the replying node
$r.\mathit{data}$	the application level data associated with this reply

Figure 4.11: The Components of a Reply

The initiator's id and the sequence number allow the initiator to differentiate replies that correspond to different queries. The replying node's id and its cost are also for use by the initiator when the reply arrives.

The information needed to propagate this query back to the initiating node is already contained within the network before the replying node sends a response. As shown in Figure 4.6, for each context request, a node in the network (other than the initiator) maintains a variable p that stores the identity of the next hop back along the shortest path. When host receives a reply message, it checks the destination of the reply, i.e., the initiator. If this host is the destination, the protocol passes the reply to the application level using the support function, *AppProcessReply* listed in Figure 4.8. If this host is not the destination, the protocol propagates the reply back through this host's parent in the context's tree. The entire protocol, including the new action to deal with the arrival of replies appears in Figure 4.12.

The abstraction and protocol presented in the previous sections build a routing tree rooted at the initiator of a context query. This tree contains exactly the minimum cost paths to every node that qualifies for membership in the specified context. However, a single host in the network may be transitively connected to the context initiator by multiple paths that satisfy the bound requirement of the context specification. Forcing the routing of replies

```
Actions

QueryArrives($q$)
  Effect:
    save information from $q$
      ($Cost := q.Cost, D := q.D$)
    update $C$ ($d_{q.o} := q.sd$)
    if $q.num = num + 1$ then
      record information
        ($d := q.d, p := q.s, pd := q.sd$)
      $Propagate\ Query(q)$
      $AppProcessQuery(q)$
      save the sequence number
        ($num := q.num$)
      $SendCleanUps$
    else if $Cost(q.d, w_{q.s}) < d$ then
      record information
        ($d := q.d, p := q.s, pd := q.sd$)
      $PropagateQuery(q)$
    end

CleanUpArrives($id$)
  Effect:
    if $id = p$ then
      calculate shortest path not thru $p$
        ($minpath := \min_c Cost(d_c, w_c)$)
      if $minpath < D$ then
        reset the cost ($d := minpath$)
        reset the parent
        set the query fields
          ($q := \langle num, id, d, D, Cost \rangle$)
        $PropagateQuery(q)$
        $SendCleanUps$
      else
        $PropagateCleanUps$
        clean up local memory
      end
    else
      update $d_{id}$ in $C$
    end

WeightChangeArrives($wnew_{id}$)
  Effect:
    if $id = p$ then
      calculate the cost
        ($d := Cost(pd, wnew_{id})$)
      if $wnew_{id} > w_p$ then
        calculate shortest path not thru $p$
          ($minpath := \min_c Cost(d_c, w_c)$)
        if $minpath < d$ then
          reset the cost ($d := minpath$)
          reset the parent
      end
      if $d < D$ then
        set the query fields
          ($q := \langle num, id, d, D, Cost \rangle$)
        $PropagateQuery(q)$
        $SendCleanUps$
      else
        $PropagateCleanUps$
        clean up local memory
      end
    else if $wnew_{id} < w_{id}$ then
      if $Cost(d_{id}, w_{id}) < d$ then
        recalculate cost
          ($d := Cost(d_{id}, w_{id})$)
        reset the parent ($p := id$)
        set the query fields
          ($q := \langle num, id, d, D, Cost \rangle$)
        $PropagateQuery(q)$
      end
    end
    store the new weight ($w_{id} := wnew_{id}$)

ReplyArrives($q$)
  Effect:
    if $id = r.initiator$ then
      $AppProcessReply(r)$
    else
      $PropagateReply(r)$
    end
```

Figure 4.12: Complete Context Computation Protocol

back to the initiator only along the links present in the shortest path tree ignores using links that have the capability of performing useful work. This section outlines the changes needed in the protocol to successfully route the context reply messages over a mesh instead of a tree.

This extension is based on the observation that multiple paths with a cost within the bound are likely to exist to many nodes within the context. Routing reply messages back to the initiator only over the shortest of these paths unnecessarily overloads the links on the shortest path while possibly leaving capable links without any work. Most of the information necessary for this mesh routing is already stored at each intermediate node in the state variable C, described in Figure 4.6. The necessary changes arise in the structure of the reply message itself and in the behavior of the nodes.

The guarantee an application requires is that every response to a particular context query travels a path whose total cost is less than the bound. This path can be the shortest path to the node, but, in the case that there are multiple paths connecting the reference node to a responding node, the reply can travel any qualifying path. We accomplish this on a mesh by starting each reply message with a bag of tokens. Because the reply has not yet traveled on any link and therefore has not yet incurred any cost, the initial number of tokens in the bag is equal to the context query's bound. As the reply travels toward the initiator, tokens are removed from the message based on the cost of the links traveled.

Figure 4.13 shows an example network with a mesh for routing reply messages built on it. The shaded host is a reference host that has defined a context to include all nodes within three hops. The shortest path tree constructed for this specification is shown with darker links. The other links are additional links that can be used for routing reply messages if their costs qualify. In this figure, the numbers on the arrows along the links refer to the shortest possible path from that link back to the reference host. Consider the host labeled X sending a reply. The host first packages the reply with a bag of three tokens (because three is the context's bound). At this point, host X can send this message to any of its neighbors because all of the paths are qualifying. Let's say X chooses host Y. Host X first updates the bag of tokens by subtracting one (the cost of every link in our example is one) and then sends the reply to Y. Y has only one choice of path to send the reply along because the message is not sent back to any previous node on its path. This prevents reply messages from cycling unnecessarily in the network. Y therefore updates the bag of tokens by subtracting one and sends the reply to Z. When the message reaches Z, the bag contains only a single token. This forces Z to consume the last token and route the reply along the direct link to the reference host. Figure 4.14 shows the modified reply. The only changes from the previous section are the addition of a bag for the reply's tokens and the addition of the path traversed by the reply.

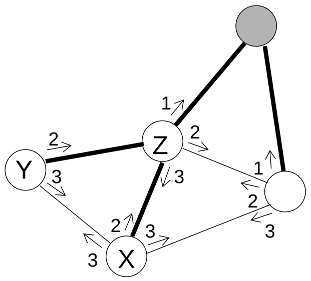


Figure 4.13: Mesh reply routing example

Reply, r

$r.initiator$	the initiator's id
$r.num$	the application sequence number of the query the reply is in response to
$r.tokens$	the number of tokens remaining for this reply to use, i.e., initially equal to the context query's bound
$r.path$	the hosts that this reply has passed through so far
$r.id$	the relying node's id
$r.cost$	the cost from the initiator to the replying node

Figure 4.14: The Components of a Reply

The protocol changes slightly within the ReplyArrives(r) action. Now, instead of sending the reply back along only the shortest cost path, the node chooses a host from C through which the cost back to the initiator is less than the tokens carried by the reply. Recall that for every connected neighbor $c \in C$, a host also stores the cost of the shortest path from the initiator to c (call it $c.cost$) and the weight of the link between this host and c (call it $c.weight$). Our protocol will use these values to choose a path and update the reply's tokens. Assuming that a node does not always choose the same path for replies, this method will increase the performance of the reply propagation by spreading the network traffic to previously unused links. Figure 4.15 shows this updated action. This action uses a support function *SendReply*(r, c) which sends the reply r to the connected neighbor identified by c.

4.5 Practical Protocol Implementation and Use

We describe the implementation of the protocol in two phases. First, we discuss what an application programmer must do to use this implementation of the Network Abstractions protocol, both in terms of the classes the programmer must define and the interface to the protocol that allows sending queries. We then move on to describe the underpinnings of the protocol that are transparent to the application.

Before defining a context, an application must create the components that build a distance metric. This includes two pieces: a `Cost` that defines the components of the costs of paths and a `Metric` that provides that algorithm for computing these costs from a previous hop cost and a link weight.

The `Cost` interface is simple; it requires the extending class to implement a single method that allows two instances of the derived `Cost` to be compared. This interface is shown in Figure 4.16.

```
Actions

QueryArrives($q$)
    ... as before

WeightChangeArrives($wnew_{id}$)
    ... as before

CleanUpArrives($id$)
    ... as before

ReplyArrives($r$)
    Effect:
        if $id = r.initiator$ then
            $AppProcessReply(r)$
        else
            Choose a host to send the reply through
                $(c := c'.(c' \in C \wedge c'.cost + c'.weight < r.tokens \wedge c' \notin r.path))$[1]
            Update the reply $(r.tokens := r.tokens - c.weight, r.path.\text{append}(c))$
            $SendReply(r, c)$
        end
```

Figure 4.15: Reply Propagation Over a Mesh

An extending class must first define any instance variables needed to store the state of the particular cost. It must then provide a definition for the `compareTo()` method. It can provide any other methods that its corresponding `Metric` class may require, which is likely to include access methods for the instance variables. As an example, consider the `Cost` class a programmer must define if he wants to build the distance-based metric described in the previous section. The code for this `Cost` class is shown in Figure 4.17.

The `Metric` base class is more complicated than the `Cost` because it defines how the costs are generated along paths in the context. The API for this base class is shown in Figure 4.18. For an extending class, the tricky parts to adhering to these requirements

int compareTo(Cost cost)
— compares two instances of the cost and returns whether the passed cost is equal to, greater than, or less than the stored cost.

Figure 4.16: The `Cost` interface

```
public class DistanceCost implements Cost{
  private double maxD;
  private int c;
  private DistanceVector v;
  public DistanceCost(double in_d, int in_c,
                    DistanceVector in_v){
    //initialize the variables
    maxD = in_d; c = in_c; v = in_v;
  }
  public int compareTo(Cost cost){
    //compare each variable
    ...
  }
  public double getD(){ return maxD; }
  public int getC(){ return c; }
  public DistanceVector getV(){ return v; }
}
```

Figure 4.17: The `DistanceCost` Class

include correctly implementing the logic of the cost function and precisely identifying the monitors whose values are important. To continue the distance-based example, Figure 4.19 shows the code the programmer must define to create this metric.

The programmer extending the metric class does not have to worry about how these methods are called; the Network Abstractions protocol, when invoked, will take the `Cost` and `Metric` that define a context and call the necessary methods as appropriate.

To define a context using the Network Abstractions protocol, an application programmer creates a `Cost` and `Metric` as discussed above and passes them to the Network Abstractions protocol. The basic interface the protocol presents to the application is detailed in Figure 4.20. The first method, `createContext()` allows the application to notify the Network Abstractions protocol of its intention to operate over a context defined by the provided `Metric` and bound (of type `Cost`). Once this context is defined, the application can use it to send and register queries. As will be discussed next, the protocol only maintains contexts that have persistent queries registered.

When an application sends a one time query over a defined context (via the `sendQuery()` method), the protocol layer uses information provided by the neighbor discovery and environmental monitoring services to determine which neighbors must receive the message, if any. If neighbors exist that are within the context's bound, the local host packages the application's data with the context information and broadcasts the entire packet to its qualifying neighbors.

private String[] monitorNames
— this instance variable holds the names of the monitors whose values affect this metric. This information is used when a context is maintained to ensure the weight values are correct. An extending class should take care to ensure this variable is initialized.

public void setMonitorNames(String[] names)
— allows the extending class to set the names of the monitors that affect this metric.

public abstract Cost wFunction(HostID otherHost)
— the implementation of this method should use information about the local host (gathered through the monitor registry if necessary) and information about the remote host (identified by the host id) to calculate the weight on the link between the hosts.

public abstract Cost costFunction(Cost currentD, Cost weight)
— the implementation of this method should take the cost at the previous hop and the cost on the subsequent link and calculate the new cost.

Figure 4.18: The `Cost` interface

Upon receiving a one-time context query, the receiving host stores the previous hop, and repeats the propagation step, forwarding the packet to any of its neighbors within the bound. It also passes the packet's data portion to application level listeners registered to receive it. If this same query (identified by a sequence number) is received from another source, the new information is remembered and propagated only if the cost of the new path is less than the previous cost.

An application can also reply to a data packet. The protocol uses the stored previous hop information to route the reply back to the reference host. Because this reply is asynchronous and the context for a one-time query is not maintained, it is possible that the route no longer exists. In these cases, the reply is dropped. To provide a stronger guarantee on a reply's return, an application should use a persistent query which forces the protocol to maintain the context.

The structure of a persistent query differs slightly from a one-time query in that it must include the entire path. This information is used to overcome the count-to-infinity problem encountered as the links in the network change. The distribution of the query is the

```
public class DistanceMetric extends Metric{
  public DistanceMetric(){
    String[] monitors = {''GPSMonitor''}
    setMonitorNames(monitors);
  }
  public Cost wFunction(HostID otherHost){
    //calculate the weight on the link
    (the DistanceVector from this host to otherHost)
    DistanceVector vec = ...
    //store it in a DistanceCost object
    DistanceCost weight = ...
    return weight;
  }
  public Cost costFunction(Cost currentD,
                           Cost weight){
    //implement the function from Figure 4.4(d)
    DistanceCost newCost = ...
    ...
    return newCost;
  }
}
```

Figure 4.19: The `DistanceMetric` Class

same as above, but the actions taken upon query reception vary slightly. The receiving host must remember the entire path back to the reference host. When the same query arrives on multiple paths, the host remembers every qualifying path. If the currently used path breaks, the protocol can replace it with a viable path. To keep both the current path and the list of possible paths consistent, the protocol monitors the aspects of the context that contribute to distance definition (through the monitor package); if these values change, the cost at this host or its neighbors could also change. The protocol reacts to such changes and updates its cost information locally. It also propagates these changes to affected neighbors. Therefore local changes to the metric do not affect the entire context, only from the point of change out to the bound. Before replacing a path, the protocol checks that the path is loop-free.

Replies to persistent queries propagate back towards the reference host along the paths maintained by the protocol. A reply is not guaranteed to reach the reference. Our practical experience shows, however, that, in reasonably sized networks with a fair amount of mobility, the delivery assumption is likely to hold. Section 4.6 provides an empirical evaluation of this assumption.

public NetAbsID createContext(Metric m, Cost b)
— initializes a context according to the provided metric and bound. the bound defines the maximum allowed cost that belongs to the context. this method returns a handle to the application that it can use to access the context.

public void sendQuery(NetAbsID id, Query q)
— this method sends the provided query to all members of the context identified by id.

public Ticket registerQuery(NetAbsID id, Query q)
— this method registers the provided query on all members of the context identified by id. the method returns a ticket to the application that it can use to deregister the query.

public void deregisterQuery(Ticket t)
– removes the persistent query identified by the provided ticket.

Figure 4.20: The Network Abstractions interface

4.5.1 Demonstration System

Figure 4.21 shows a screen capture of our demonstration system. In this example, each circle depicts a single host running an instance of the protocol. The demonstration system uses the network for communication, which allows this system to display information gathered from actual mobile hosts. This figure shows a single context defined by a host (the gray host in the center of the white hosts). This context is simple; it includes all hosts within one hop. When a host moves within the context's bound, it receives a query registered on the context that causes the node to turn its displayed circle white. When the node moves out of the context, it turns itself black. The demonstration system provides simulations using a variety of mobility models, including a markov model, a random waypoint model [6], and a highway model. This system is particularly useful because it allows us to visually evaluate what kinds of contexts match what styles of mobility. This gives us some intuition into what our goals should be before we start extensive simulation or implementation of a complex application.

4.5.2 Example Usage

The protocol implementation described here is currently in use to support the ongoing implementation of a middleware model for ad hoc mobile computing. In this system, called

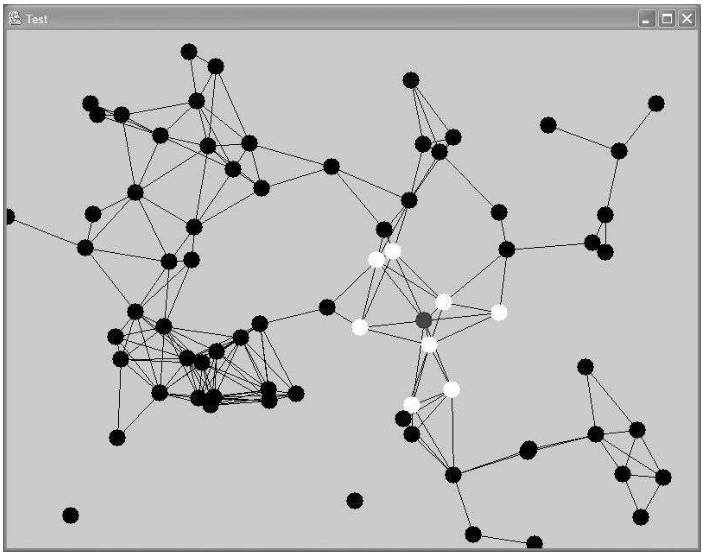


Figure 4.21: Screen capture of demonstration system

EgoSpaces, application agents operate over projections (*views*) of the data available in the world. EgoSpaces addresses the specific needs of individual application agents, allowing them to define what data is to be included in a view by 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. This protocol provides the latter in a flexible manner, and EgoSpaces uses the Network Abstractions protocol to deliver all communication among agents.

4.6 Simulation Evaluation

Range (m)	50	75	100	125	150	175	200	225	250
Neighbors	1.09	2.47	4.21	6.38	9.18	12.30	15.51	19.47	23.89

Figure 4.22: Average number of neighbors for varying transmission ranges

To examine the definitions of contexts on real mobile ad hoc networks, we used the ns-2 network simulator, version 2.26. This section provides simulation results for context dissemination. These simulations are a first step in analyzing the practicality of the protocol we have implemented. Not only do they serve to show that it is beneficial to define contexts in the manner described in ad hoc networks, the measurements also provide information to application programmers about what types or sizes of contexts should be used under given mobility conditions or to achieve required guarantees. All of the simulations we describe in this section implement a context defined by the number of hops from the reference node. Because this is the simplest type of context to define using the Network Abstractions protocol, this provides a baseline against which we can compare simulations of more complex or computationally difficult definitions. Before providing the experimental results, we detail the simulation settings we used.

4.6.1 Simulation Settings

We generated random 100 node ad hoc networks that use the random waypoint mobility model. The simulation is restricted to a 1000x1000m^2 space. We vary the network density (measured in average number of neighbors) by varying the transmission range. We measured the average number of neighbors over our simulation runs for each transmission range we used; these averages are shown in Figure 4.22. While the random waypoint mobility model suffers from “density waves” as described in [52], it does not adversely affect our simulations. An average of 1.09 neighbors (i.e., 50m transmission range) represents an almost disconnected network, while an average of 23.89 neighbors (i.e. 250m transmission range) is extremely dense. While the optimal number of neighbors for a static ad hoc network was shown to be the “magic number” six [38], more recent work [52] shows that the optimal number of neighbors in mobile ad hoc networks varies with the degree of mobility and mobility model. The extreme densities in our simulations lie well above the optimum for our mobility degrees.

In our simulations, we used the MAC 802.11 standard [35] implementation built in to ns-2. Our protocol sends only broadcast packets, for which MAC 802.11 uses Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) [2]. This broadcast mechanism is not reliable, and we will measure our protocol’s reliability over this broadcast scheme in our simulations. We implemented a simple “routing protocol” on top of the MAC layer that, when it receives a packet to send simply broadcasts it once but does not repeat it.

We also tested our protocol over a variety of mobility scenarios using the random waypoint mobility model with a 0s pause time. In the least dynamic scenarios, we use a fixed speed of 1m/s for each mobile node. We vary the maximum speed up to 20m/s while holding a fixed minimum speed of 1m/s to avoid the speed degradation described in [67].

---

[2] In CSMA/CA a node ready to send senses the medium for activity and uses a back off timer to wait if the medium is busy. When the node senses a clear medium, it broadcasts the packet but waits for no acknowledgments.

4.6.2 Simulation Results

The results presented evaluate our protocol for three metrics in a variety of settings. The first metric measures the context's consistency, i.e., the percentage of nodes receiving a context notification given the nodes that were actually within the context when the query was issued. Using this method to evaluate a proposed context definition, we can give an application using the protocol an idea of how successful it will be in reaching the members of its contexts. Applications can use this information to tailor their context definitions to the combination of their needs and requirements. For example, an application that relies on strong guarantees (e.g., the application transfers money or measures safety criticality) will have to define contexts that have an extremely high level of consistency. At the other end of the spectrum, many applications can accept a best-effort style of interaction, and can therefore define wider contexts that provide weaker guarantees.

The second metric measures the context notification's settling time, i.e., the time that passes between the reference host's issuance of a context query and the time that every node in the context that will receive the query has received it. This is the first step in providing applications with information about how long they should wait for responses from differently sized contexts before timing out and resending a query if necessary. This metric also gives us, as protocol implementers, some information about how long a single context definition is utilizing network resources.

The third metric evaluates the protocol's efficiency through the rate of "useful broadcasts", i.e., the percentage of broadcast transmissions that reached nodes that had not yet received the context query. As we will see in the discussion of the results, this measurement provides us insight into under what conditions (e.g., high speeds, densities, or loads) the protocol might require tailoring in the dynamic ad hoc network.

For these first simulations, we rely on hop count as our measure of path cost. That is, every link in the network has a weight of one, and the cost function in additive. While this is the simplest metric we can build using the network abstractions protocol, it is useful to many applications, it is a building block for many more complex metrics (as discussed in Section 4.3, and it provides a nice starting point from which we can guage future measurements. We provide simulation results in this initial evaluation only for the dissemination of queries using the network abstractions protocol. We do not include the return of replies to the queries or the maintenance portion of the protocol. The first set of results compare context definitions of varying sizes, specifically, definitions of one, two, three, and four hop contexts. We then evaluate our protocol's performance as network load increases, specifically as multiple nodes define contexts simultaneously. Unless otherwise specified, nodes move with a $20m/s$ maximum speed. These contexts based on hop counts do not directly relate to any particular application. As a guideline, however, we can imagine that the smaller contexts (e.g., one and two hops) relate to the collision detection application discussed in

Section 1.1, while the large contexts (e.g., three and four hops) relate to traffic collection.

Reasonably Sized Contexts Have Good Consistency Guarantees

In comparing contexts of varying sizes, we found that as the size of the context (measured in this example in the number of hops) increases, the consistency of the context decreases slightly. Results for different context sizes are shown in Figure 4.23. These results show a single context definition on our 100 node network. The protocol can provide localized contexts (e.g., one or two hops) with near 100% consistency. With broader context definitions, the percentage of the context notified can drop as low as 94%. The disparity between large and small context definitions becomes most apparent with increasing network density. At large densities, the extended contexts contain almost the entire network, e.g., at a transmission range of 175m, a four hop context contains ∼80% of the network's nodes. In addition, the number of neighbors is 12.3, leading to network congestion when many neighboring

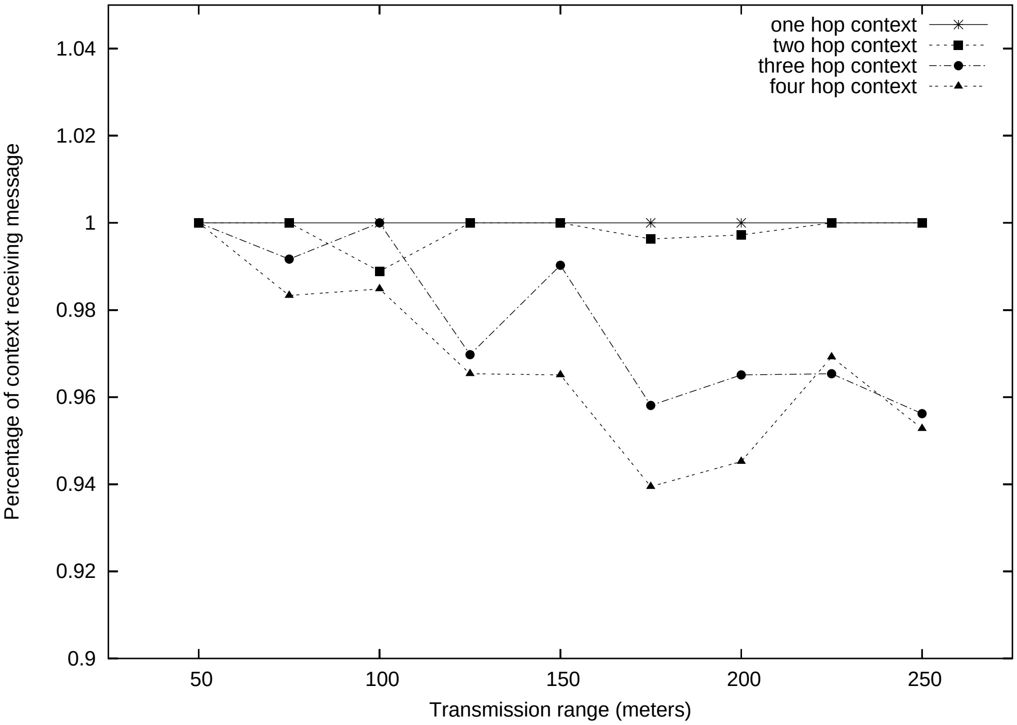


Figure 4.23: Percentage of context members receiving the message for contexts of varying sizes

nodes rebroadcast. This finding lends credence to the idea that applications should define contexts which require guarantees (e.g., automobile collision detection) as more localized, while contexts that can tolerate some inconsistency (e.g., traffic information collection) can cover a larger region. In addition, small modifications to the protocol that address the fact that neighboring nodes should not rebroadcast simultaneously may positively benefit performance. We discuss this problem (called the "broadcast storm") and some possible solutions in the next section.

Context Building Settles Quickly

As the size of the defined context increases, more time is required to notify all the context members. Figure 4.24 shows the settling times for contexts of varying sizes defined on networks of increasing density. For a two hop context with a reasonable density (9.18 neighbors at 150m transmission range), the maximum time to notify a context member was 20.12ms. The settling times for different sized networks eventually become similar as network density

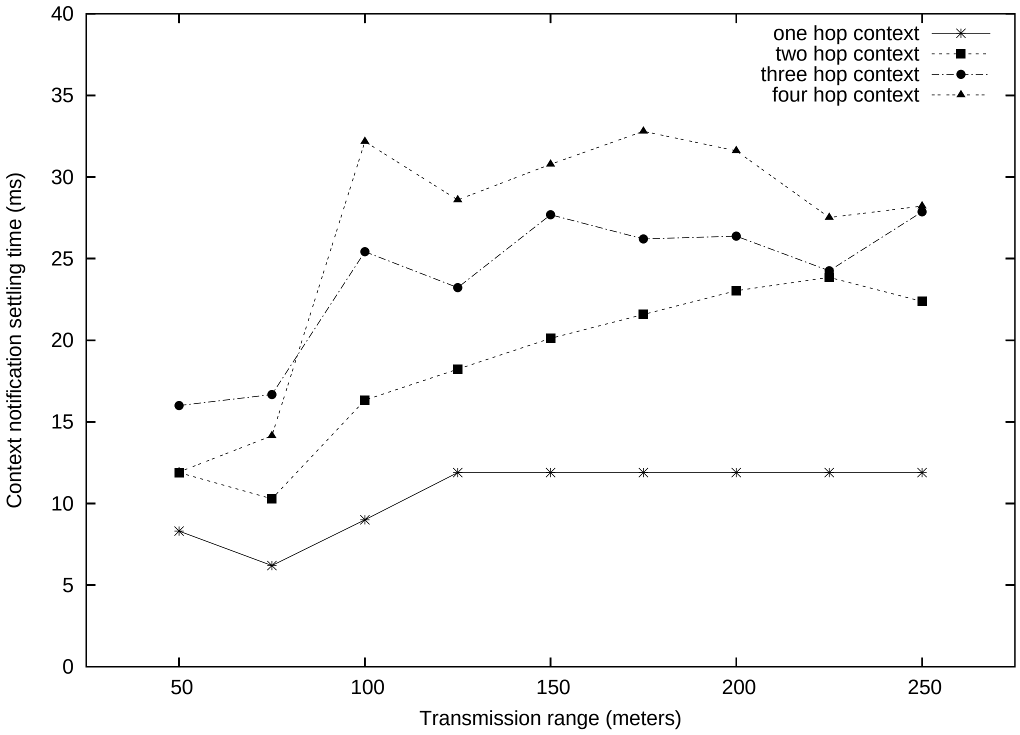


Figure 4.24: Settling time for contexts of varying sizes

increases. This is due to the fact that even though the context is defined to be four hops, all nodes are within two hops of each other, effectively rendering a four hop context definition a two hop context.

Efficiency Decreases Almost Linearly with Increasing Density

Figure 4.25 shows the protocol's efficiency versus density for different sized contexts. First, notice that the efficiency for a one hop network is always 100% (assuming no MAC level errors) because only one broadcast (the initial one) is ever sent. For larger contexts, the efficiency is lower and decreases with increasing density. Most of the lower efficiency and the descending nature of the curve results from the fact that rebroadcasting neighbors are likely to reach the same set of additional nodes. This becomes increasingly the case as the density of the network increases. Even at high densities, however, a good number (> 20%)

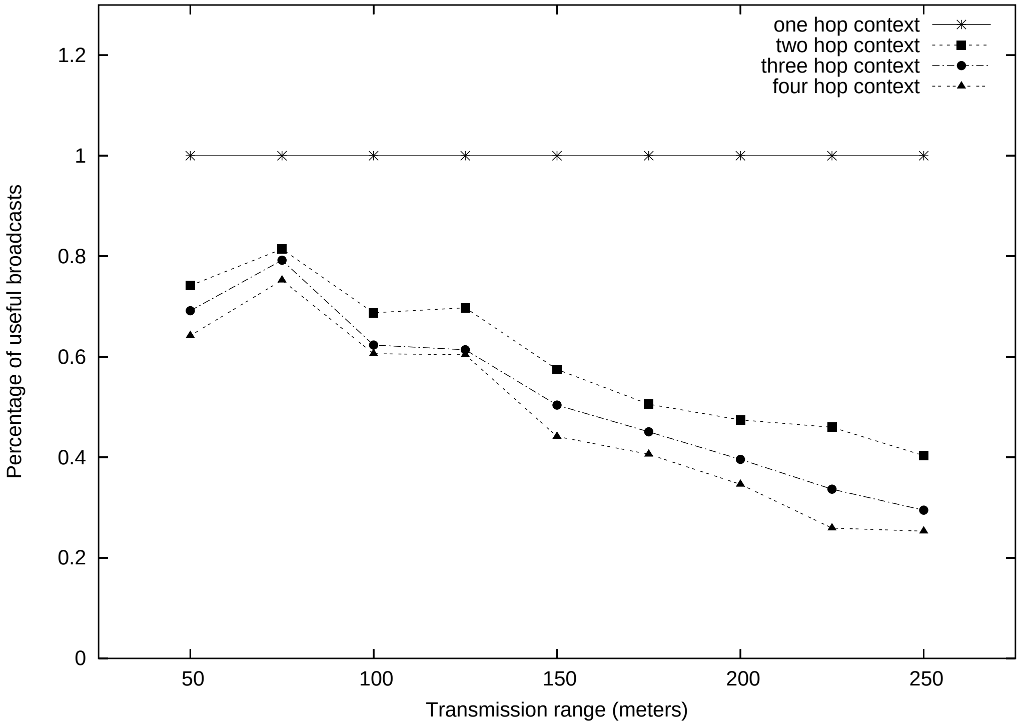


Figure 4.25: Percentage of broadcasts that reached new context members for contexts of varying sizes

of the broadcasts reach additional context members. In the next section, we discuss possible solutions to increase the performance of the protocol in these cases.

Consistency Remains above 80% with Increased Network Load

The remainder of the analysis focuses on an increasing load in the network, caused by multiple simultaneous context definitions by multiple nodes in the network. We show only results for four hop contexts because they are the largest and have the worst behavior; results for smaller contexts are discussed in comparison. As Figure 4.26 shows, five context definitions have no significant impact on the consistency as compared to a single definition. For ten definitions, the atomicity starts to decrease, but remains above ∼80% at all transmission ranges. With more registrations, especially at the larger densities, the different context messages interfere significantly with each other. Two factors contribute to this observation. The first is that the broadcast messages collide and are never delivered. The second results from the fact that MAC 802.11 uses CSMA/CA. Because the medium is busier (more

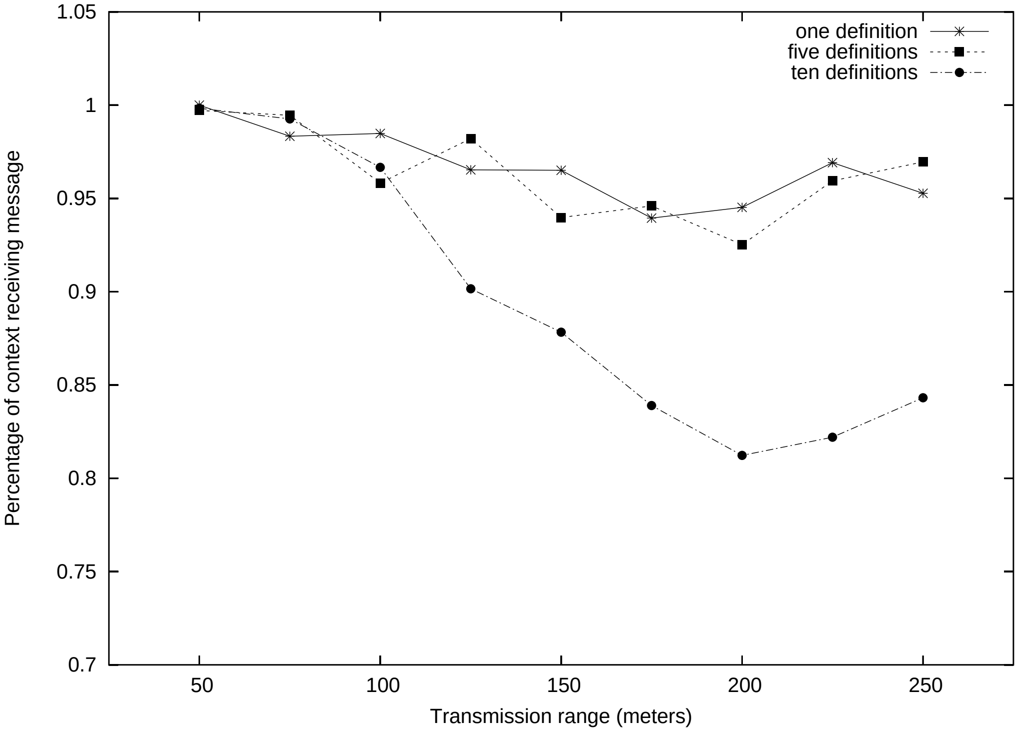


Figure 4.26: Percentage of context members receiving context messages for varying network loads

neighboring nodes are broadcasting), nodes are more likely to back off and wait their turn to transmit. During this extended waiting time, the context members are moving (at a maximum speed of $20m/s$). Because the hosts are moving rapidly, context members that were in the context initially move out of it before the query can traverse the entire context. These effects decrease significantly with smaller context sizes, e.g., at a transmission rate of $175m$, ten definitions on a two hop context can be delivered with ∼97% consistency, and twenty can be delivered with ∼89.5% consistency. This type of information informs applications that, in extremely mobile, dense, or active networks, contexts that span a smaller set of nodes are likely to be more consistent with respect to delivery guarantees. Applications can use this information to determine which types of contexts are appropriate in different environments.

Increased Network Load Increases Settling Time at High Densities.

Given the previous results, it is not surprising that increasing the network load to five context definitions does not increase settling time. As shown in Fig. 4.27, however, increasing the network load to ten definitions increases settling times of networks with high densities. Again, when the network density is large and multiple nodes are building contexts, the dispersions of their contexts queries interfere with each other, causing the broadcasting nodes to use their back off timers. This increased back off causes a longer delay in the delivery of context messages, especially to outlying context members.

Changing Speed has No Impact on Context Notification

In our analysis of this protocol, we tested scenarios with a wide variety of network speeds. We found that even the consistency of context message delivery is not greatly affected by the speed of the nodes. As mentioned earlier in this section, future work will include the simulation of the return of replies to queries and of the maintenance portion of the protocol. It is likely that in these cases, the speed of the hosts involved will have a greater impact on the consistency and efficiency measurements, and it will be necessary to revisit the speed concerns under these new constraints. Simulation results for this portion of the protocol lie outside the scope of this dissertation and are planned as future work.

4.7 Chapter Summary

In this chapter, we highlighted the reasons why current communication protocols for ad hoc networks fall short of the needs of asymmetric coordination. Specifically, we envision applications whose communications specifically target a region of the network that defines a neighborhood surrounding the network host. This neighborhood is often defined by measures of distance including physical distance, bandwidth, throughput, or latency.

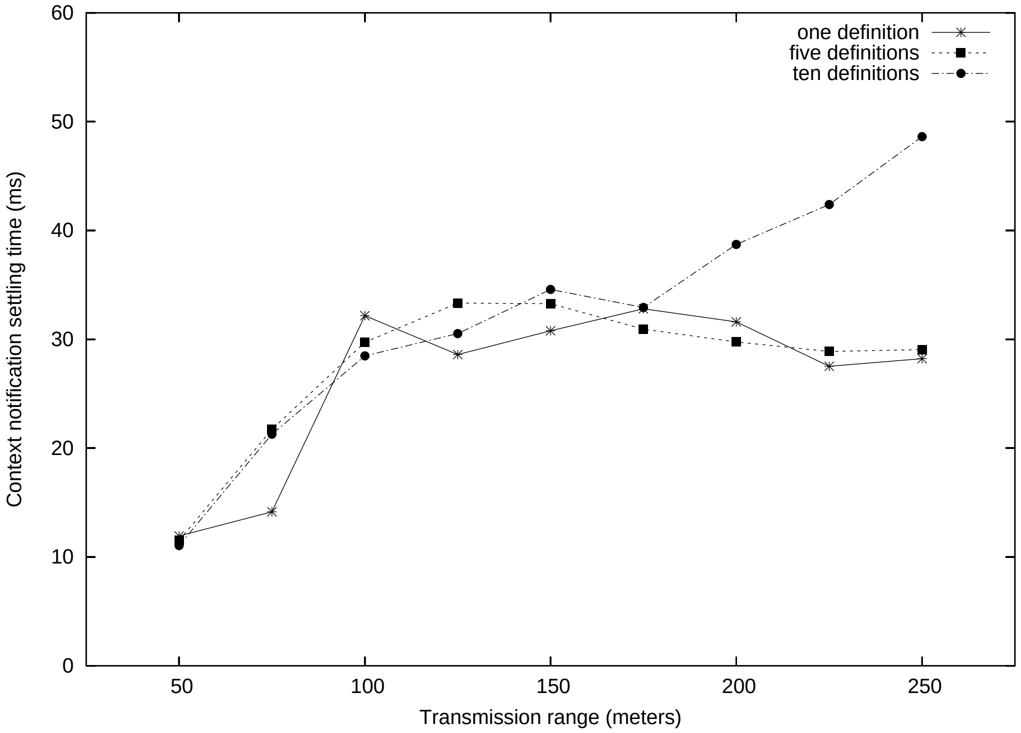


Figure 4.27: Maximum time for last context recipient to receive notification for varying network loads

We evaluated existing protocols and build a novel protocol that constructs an application's desired communication partners based on an application-provided specification. We presented a starting implementation for the protocol and evaluated its performance through simulation. We feel that this protocol and associated evaluation provide a significant improvement to the current state of ad hoc network communication and a good starting point for continuing investigations into context-sensitive communication.

In the next chapter, we take a step back from the low-level implementation details we have been focusing on in the last two chapters and examine the formal reasoning needs of application developers. Even when using a protocol like network abstractions, we require the ability to provide formally stated guarantees and explanations of behavior. A focus on context-aware formalization is currently lacking in the literature, and we examine the creation of a formal model of reasoning that applies to a wide variety of context-aware applications and systems.

Chapter 5

A Formal Model of Context-Awareness

In the previous sections, we described a model of application development that utilizes the notion of context-aware computing to simplify the development task for ad hoc mobile networks. As a complement to this programming environment, application developers often require the ability to formally reason about the programs they write. With regard to context-aware applications, the literature supplies few appropriate models. In the case of mobile computing, we can use existing formal models of mobility to begin the reasoning task, but our applications quickly lose the appearance of being context-aware, which was a key abstraction in the development process. In this chapter, we explore the creation of a formal model tailored for use in reasoning about context-aware systems. Because of the lack of currently available models, we take care not to tailor our model specifically to mobile computing and instead focus on the notion of context-awareness. The goal is to create a model that applies to a wide variety of context-aware applications and systems regardless of the operating environment.

A model of context-awareness must be *expansive*, i.e., it must recognize the fact that distant entities in the operational environment can affect an agent's behavior [56]. This requirement states that one should not place a priori limits on the scope of the context being associated with a particular agent. While specific instantiations of the model may impose restrictions due to pragmatic considerations having to do with the cost of context maintenance or the nature of the physical devices, application needs are likely to evolve with time. As a consequence, fundamental assumptions about the model could be invalidated. To balance out the expansive nature of the model and to accommodate the need for agents to exercise control over the cost of context maintenance, we also require the model to support a notion of context that exhibits a high degree of *specificity*. In other words, it must be possible for context definitions to be tailored to the needs of each individual agent.

Furthermore, as agents adapt, evolve, and alter their needs, context definitions also should be amenable to modification in direct response to such developments.

Expansiveness and specificity are central to achieving generality. They are necessary but not sufficient features of the context-aware computing paradigm that consider the way in which the operational environment relates to an agent's notion of context, i.e., the distinction between potentiality and relevance. They fail to consider the manner in which the agent forms and manipulates its own notion of context. The only way an agent can exercise control over its context is to have an *explicit* notion of context. This gives the agent the power to define its own context and to change the definition as it sees fit. It also formalizes the range of possible interactions between the agent and its operational environment. Consequently, context definition has to be an identifiable element of the proposed model and must capture the essential features of the agent/context interaction pattern. Separation of concerns suggests that an agent's context specification be *separable* from its behavior specification. The agent behavior may result in changes to the definition of context, but the latter should be readily understood without one needing to examine the details of the agent behavior. This requirement rules out the option of having to derive context from the actions of the agent. This distinction is important because many systems interact with and learn about their operational environment without actually employing the context-aware paradigm. Finally, context maintenance must be *transparent*. This implies that the definition of context must be sufficiently abstract to free the agent of the operational details of discovering its own context and sufficiently precise for some underlying support system to be able to determine what the context is at each point in time.

In this chapter, we explore the development of an abstract formal model for context-aware computing. Because our ultimate goal is to achieve a better understanding of the essence of the context-aware computing paradigm, we seek to achieve minimality of concepts and elegance of notation while remaining faithful to the formative assumptions that define our perspective on context-awareness. The resulting model is called Context UNITY and has its roots in earlier formal work on Mobile UNITY [50, 40]. Context UNITY assumes that the universe (called a system) is populated by a bounded set of agents whose behaviors can be described by a finite set of program types. At the abstract level, each agent is a state transition system, and context changes are perceived as spontaneous state transitions outside of the agent's control. However, the manner in which the operational environment can affect the agent state is an explicit part of the program definition. In this way, the agent code is local in appearance and totally decoupled from that of all the other agents in the system. The context definition is an explicit part of the program type description, is specific to the needs of each agent as it changes over time, and is separate from the behavior exhibited by the agent. The design of the Context UNITY notation is augmented with an assertional style proof logic that facilitates formal reasoning about context-aware programs.

5.1 Formalizing Context-Awareness

Context UNITY represents an application as a community of interacting agents. Each agent's behavior is described by a program that serves as the agent's prototype. To distinguish agents from each other, each has a unique identifier. Because we aim to model context-aware systems, an agent must access its environment, which, in Context UNITY, is defined by the values of the variables other agents in the system are willing to expose. As described in the previous section agents require context definitions tailored to their individualized needs. In Context UNITY, agents interact with a portion of the operational environment defined through a unique set of variables designed to handle the agent's context needs.

A central aspect of Context UNITY is its representation of program state. Three categories of variables appear in programs; they are distinct in the manner in which they relate to context maintenance and access. First, a program's *internal variables* hold private data that the agent uses but does not share with the other agents in the system. They do not affect the operational environment of any other agent. *Exposed variables* store the agent's public data; the values of these exposed variables can contribute to the context of other agents. The third category of variables, *context variables*, represent the context in which the particular agent operates. These variables can both gather information from the exposed variables of other agents and push data out to the exposed variables of other agents. These actions are governed by context rules specified by each agent and subject to access control restrictions associated with the exposed variables.

In the remainder of this section, we first detail the structure of a Context UNITY system. We then show how programs use context variables to define a context tailored to the needs of each particular agent and the mechanics that allow an agent to explicitly affect its operational environment. Throughout we provide examples using the model to reinforce each concept.

5.1.1 Foundational Concepts

Context UNITY represents an application as a *system specification* that includes a set of *programs* representing the application's component types. Figure 5.1 shows the Context UNITY representation of a **System**. The first portion of this definition lists programs that specify the behavior of the application's individual agents. Separating the programs in this manner encapsulates the behavior of different application components and their differing context needs. The **Components** section of the system declares the instances of programs, or agents, that are present in the application. These declarations are given by referring to program names, program arguments, and a function (*new_id*) that generates a unique id for each agent declared. Multiple instantiations of the same program type are possible;

```
System SystemName
  Program ProgramName (parameters)
    declare
      internal — internal variable declarations
      exposed — exposed variable declarations
      context — context variable declarations
    initially — initial conditions of variables
    assign — assignments to declared variables
    context
      definitions affecting context variables—they can pull information from and
          push information to the environment
  end
  . . . additional program definitions . . .
  Components
    the agents that make up the system
  Governance
    global impact statements
end SystemName
```

Figure 5.1: A Context UNITY Specification

each resulting agent has a different identifier. The final portion of a system definition, the **Governance** section, captures interactions that are uniform across the system. Specifically, the rules present in this section describe statements that can impact exposed variables in all programs throughout the system. The details of an entire system specification will be made clearer through examples later in this section. First we describe in detail the contents of an individual Context UNITY program.

Each Context UNITY program lists the variables defining its individual state. The declaration of each variable makes its category evident (internal, exposed, or context). A program's **initially** section defines what values the variables are allowed to have at the start of the program.

The **assign** section defines how variables are updated. These assignment statements can include references to any of the three types of variables. Like UNITY and its descendants, Context UNITY's execution model selects statements for execution in a weakly-fair manner—in an infinite execution, each assignment statement is selected for execution infinitely often. In the assignment section, a program can use simple assignment statements, transactions, or reactions. A *transaction* is a sequence of simple assignment statements which must be scheduled in the specified order with no other (non-reactive) statements interleaved. They capture a form of sequential execution whose net effect is a large-grained atomic state change. In the **assign** section of a program, a transaction uses the notation: $\langle s_1; s_2; \ldots; s_n \rangle$. A *reaction* allows a program to respond to changes in the state of the system. A reaction is triggered by an enabling condition Q and has the form s **reacts-to** Q. As in Mobile UNITY, Context UNITY modifies the execution model of traditional UNITY to accommodate reactions. Normal statements, i.e., all statements other than reactions, continue to be selected for execution in a weakly-fair manner. After execution of a normal statement, the set of all reactions in the system, forming what we call a *reactive program*, executes until it reaches *fixed-point*. During the reactive program's execution, the reactive statements are selected for execution in a weakly-fair manner while all normal statements are ignored. When the reactive program reaches a fixed-point, the weakly-fair selection of normal statements continues.

In Context UNITY, an agent's behavior is defined exclusively through its interaction with variables. To handle context interactions, Context UNITY introduces context variables and a special **context** section that provides the rules that manage an agent's interaction with its desired context. Specifically, the **context** section contains definitions that sense information from the operational environment and store it in the agent's context variables. The rules can also allow the agent to affect the behavior of other agents in the system by impacting their exposed variables. The use of this special **context** section explicitly separates the management of an agent's context from its internal behavior.

Two prototypical uses of the **context** section lie at the extremes of sensing and affecting context. First, a program's context definition may only read the exposed variables of other programs but not affect the variables' values. When used in such a way, we refer to the context variables as *sentient variables* because they only gather information from the environment to build the agent's context. In the other extreme case, a program can use its context variables to disperse information to components of the environment. From the perspective of the reference agent, this affects the context for other agents, and we refer to context variables used in this manner as *impact variables.* While these two extremes capture the behavior of context-aware systems in the most common cases, the generality of Context UNITY's context specification mechanism allows it to model a variety of systems that fall between these two extremes. The examples discussed in Section 5.3 demonstrate this in more detail.

The acquaintance list application introduced in the previous section provides a list of nearby coordination participants. Several context-aware systems in the literature, e.g., Limone [23], use this data structure as a basis for more sophisticated coordination mechanisms. The acquaintance list is defined by dynamically changing needs of a reference agent. Figure 5.2 shows a Context UNITY specification for an application that relies on the usage of an acquaintance list. This system consists of three agents of two differing types. Each agent stores its unique agent id in an exposed variable named `agent_id` that is available to other programs. Because we are modeling systems that entail agent mobility, each agent also has a variable named `location` that stores its location. The movement of the agent is outside this example; it could occur through local assignment statements to the location variable (in the **assign** section of the individual program) or even by a global controller (via the

```
System AcquaintanceManagement
 Program Agent1
  declare
   exposed id ! agent_id : agent_id
          λ ! location : location
   context Q : set of agent_id
  assign
   . . . definition of local behavior . . .
  context
   define — define Q based on desired properties of acquaintance list members
 end
 Program Agent2
  declare
   exposed id ! agent_id : agent_id
          λ ! location : location
   context Q : set of agent_id
  assign
   . . . definition of local behavior . . .
  context
   define — define Q based on different restrictions
 end
 Components
  Agent1[new_id], Agent1[new_id], Agent2[new_id]
end AcquaintanceManagement
```

Figure 5.2: A Context-Aware System for Acquaintance Maintenance

Governance section of the system). Both *id* and λ are local handles for built-in variables whose names are agent_id and location, respectively. We discuss these built-in variables in more detail later in this section. Each program type has individualized behavior defined via the **assign** section that may use additional context variables or definitions. In this example, we are most concerned with the maintenance of the acquaintance list. Each agent declares a context variable Q of type set that will store the contents of the acquaintance list. Different program types (in this case, *Agent1* and *Agent2*) employ different eligibility qualification criteria for the members of the acquaintance list, exemplified in the **context** section of each program type. This example shows a high-level definition of a context variable. In the acquaintance management specification, each program's **context** section contains a rule that describes how the context variable Q is updated. Later in this section we will show exactly what this rule entails. First however, we expound on the structure of Context UNITY exposed variables.

Exposed Variables Revisited

In UNITY and many of its descendants, variables are simply references to values. In Context UNITY, both internal and context variables adhere to this standard. However, references to exposed variables appearing in the program text are references to more complex structures

ι	the variable's unique id
π	the id of the owner agent
η	the name
τ	the type
ν	the value
α	the access control policy

Figure 5.3: Variable Components

needed to support context-sensitive access within an unknown operational environment. These handle names have no meaning outside the scope of the program. A complete semantic representation of exposed variables is depicted in Figure 5.3. Each exposed variable has a unique id ι—uniqueness could be ensured by making each variable unique within an agent and combining this with the unique agent id. This unique id is used in the context interaction rules to provide a handle to the specific variable. The agent owning the exposed variable, π or type `agent_id`, also appears in the semantic structure and allows an exposed variable to be selected based on its owner. An exposed variable's name, η, provides information about the kind of data the variable contains; the name of an exposed variable can be changed by the program's assignment statements. The type τ reflects the exposed variable's data type and is fixed. An exposed variable's value, ν, refers to the data value held by the variable. Programs refer to the value when assigning to the variable or when accessing the value the variable stores. The value of an exposed variable can be assigned in the **assign** section or can be determined by a different program's impact on its environment. The program can control the extent to which its exposed variables can be modified by others using the access control policy described below.

Modeling Access Control

The final component of an exposed variable in Context UNITY, α, stores the variable's access control policy. Because many context-aware systems and applications use some form of access restriction, Context UNITY provides a generalized mechanism for modeling access control. An access policy determines access based on properties of the particular agent accessing the variable. The access control policy determines both the readability and writability of the particular variable on a per-agent basis. The function α takes as arguments credentials provided by the reference agent and returns the set of allowable operations on this variable, e.g., {r, w} signifies permission to both read and write the particular exposed variable. Because Context UNITY associates an access control policy with each variable, it models the finest-grained access restrictions possible in a context-aware application. This model can be tailored to the needs of current context-aware systems, including those that utilize a trusted third party for authentication.

Built-in Variables

To ease representation of context-aware interactions, Context UNITY programs contain four built-in exposed variables. In Context UNITY, these variables are automatically declared and have default initial values. An individual program can override the initial values in the program's **initially** section and can assign and use the variables throughout the **assign** and **context** sections. The first of these variables has the name "location" and facilitates modeling mobile context-aware applications by storing the location of the program owning the variable. This variable is exposed and available to other programs to use. An example use of this variable was shown in the system in Figure 5.2. The definition of location can be based on either a physical or logical space and can take on many forms. This style of modeling location is identical to that used in Mobile UNITY. The second of Context UNITY's built-in variables is also exposed and has the name "type", and its value is the program's name (e.g., "*Agent1*" or "*Agent2*" in the example system). As we will see, the use of this variable can help context variables select programs based on their general function. The third of the built-in variables has the name "agent_id" and holds the unique identifier assigned to the agent when the agent is instantiated in the **Components** section. The final built-in variable is internal and has the local handle "*credentials*". It is used in Context UNITY interactions to support access control restrictions. Specifically, the variable stores a profile of attributes of the program that are provided to the access control policies of the exposed variables of other programs. These credentials are available to access control policies when determining whether or not this program has access to a particular exposed variable.

5.1.2 Context Specification

Context-aware applications rely on conditions in the environment for adaptation. Context UNITY facilitates specification of context interactions through the use of context variables that use the exposed variables of other agents to provide exactly the context that a reference agent requires. In a Context UNITY program, the **context** section of a program contains the rules that dictate restrictions over the operational environment to define the context over which an agent operates. Additionally, the rules in the **context** section allow the agent to feed back information into its context. Structuring the **context** section as a portion of each program allows agents to have explicit and individualized interactions with their contexts.

As indicated in the beginning of this section, due to the unpredictable nature of the dynamic environments in which context-aware agents operate, their context definitions require a mechanism to handle their lack of a priori knowledge about the operational environment. In Context UNITY, we introduce *non-deterministic assignment statements* to the definition of context. Specifically, the non-deterministic assignment statement $x := x'.Q$ assigns to x a value x' non-deterministically selected from all values satisfying the condition Q [3]. A program's context rules define how an agent can access and interact with the exposed variables of other agents. It can select which other agents' variables affect its behavior by employing non-deterministic assignments and existential quantification. The flexibility of this selection mechanism allows agents that contribute to the context to be selected based on attributes defined in their exposed variables. For example, in a mobile context-aware application, an agent can use the built-in Context UNITY `location` variable to store its current physical location. Whenever the component moves, the agent updates the location variable using an assignment statement in the local **assign** section. Another agent can use relative distance to identify which other agents are to contribute to its context. We refer to this selection of agents based on their properties as *context-sensitive program selection.*

Context UNITY wraps the use of non-deterministic assignment in a specialized notation for handing context-aware interactions. To manage its interaction with context information, a program uses statements of the following form in its **context** section:

```
c uses    quantified variables
  given   restrictions on variables
  where   c becomes expr
          expr_1 impacts exposed variable_1
          expr_2 impacts exposed variable_2
          ...
  [reactive]
```

This expression, which we refer to as a *context rule*, governs the interactions associated with the context variable c. A context rule first declares existentially quantified dummy variables to be used in defining the interactions with the exposed variables that relate to the context variable c. The scope of these dummy variables is limited to the particular context rule that declares them. The expression can refer to any exposed variables in the system, but referring to other programs' exposed variables explicitly requires the program to have advance knowledge about the other components it will encounter over time, which programs rarely have. Typically, context-aware applications rely on opportunistic interactions that cannot be predetermined. To capture this style of interaction in Context UNITY, the exposed variables that contribute to the context rule are selected in a context-sensitive manner using the restrictions provided in the rule's definition. As one example, because a wireless context-aware application contains many agents that may or may not be connected, the restrictions used in a context rule for a particular application must account for the connectivity restrictions imposed by the operational environment.

Given the set of exposed variables selected in accordance with the restrictions, the context rule can define an expression, *expr*, over the exposed variables and any locally declared variables (internal, exposed, or context). The result of evaluating this expression is assigned to the context variable. The context rule can also define how this context variable impacts the operational environment.

The execution of each context rule can optionally be declared **reactive**, which dictates the degree of consistency with which the context rule reflects the environment. If a context rule is declared **reactive**, it becomes part of the system's reactive program that is executed to fixed-point after the execution of each normal statement. Using a reaction guarantees that the context information expressed by the rule remains consistently up to date because no normal statements can execute until the reactive program reaches fixed-point. If not declared **reactive**, the context rule is a normal, unguarded statement and part of Context UNITY's normal execution model.

Within a context rule, if no explicit restrictions are placed on the referenced exposed variables, two restrictions are automatically assumed. The first requires that the variable referenced be an exposed variable in its owner program since only exposed variables are accessible from other programs. The second implicit restriction requires that the program whose context uses a particular exposed variable must satisfy the variable's access control policy. Consider the following simple context rule that pulls the value out of some exposed variable, places the value in the context variable c, and deletes the value from the exposed variable used. The statement is a reactive statement that is triggered when a is larger than the value of some local variable x:

$$\begin{array}{ll} c\ \textbf{uses} & a \\ \quad \textbf{given} & a > x \\ \quad \textbf{where} & c\ \textbf{becomes}\ a \\ & 0\ \textbf{impacts}\ a \\ \quad \textbf{reactive} & \end{array}$$

This reactive construct makes the rule part of the system's set of reactive statements. This context rule corresponds to the following formal definition, which includes the two implicit restrictions on the exposed variable a as discussed above:

$$\begin{array}{l} \langle a : \ a = a'.(\mathsf{var}[a'] > x \wedge \{r, w\} \subseteq \mathsf{var}[a'].\alpha(\mathit{credentials})) \\ \quad :: (c := \mathsf{var}[a].\nu \ || \ \mathsf{var}[a].\nu := 0)\ \textbf{reacts} - \textbf{to}\ \mathit{true} \\ \rangle \end{array}$$

In this definition, we introduce var, a logical table that allows us to refer to all variables in the system, referenced by the unique variable id. When selecting the variable a from the table, the statement above really selects its variable id, which serves as a reference to a specific entry in the table var. In this statement, for instance, the exposed variable a is non-deterministically selected from all exposed variables whose access control policies allow this agent access to read and write the exposed variable that the dummy variable a refers to. The latter is determined by applying the variable's access control policy to this agent's credentials. The set returned by this application can contain any combination of r and w, where the presence of the former indicates permission to read the variable, and the presence of the latter indicates permission to write the variable. After selecting the particular exposed variable to which a refers, the rule contains two assignments. The first assigns the value stored in a (i.e., $\mathsf{var}[a].\nu$) to the context variable c. The second assignment captures the fact that the context rule can also impact the environment, in this case by zeroing out the exposed variable used.

The power of the context-sensitive selection of exposed variables becomes apparent only when the restrictions within the context rules are used. Within the restrictions, the context rule can select exposed variables to be used based on the exposed variables' names, types, values, owning agent, or even based on properties of other variables belonging to the same or different agents. To simplify the specification of these restrictions, we introduce a few new pieces of notation. Referring to the system-wide table of variables (i.e., var) is cumbersome and confusing because the table is both virtual and distributed. For this reason, context rules refer directly to indexes in the table instead. Specifically, in this notation, we allow the variable id a to denote the value of the variable in var for entry a, i.e., $\mathsf{var}[a].\nu$. To access the other components of the variable (e.g., name), we abuse the notation slightly and allow $a.\eta$ to denote $\mathsf{var}[a].\eta$. Because a common operation in context-sensitive selection relies on selecting variables from the same program, we also introduce a shorthand for accessing a variable by the combination of name and program. To do this, when declaring dummy variables, a context rule can restrict both the names and relative owners of the variables. For example, the notation: $x\ !\ \mathsf{name}_1, y\ !\ \mathsf{name}_2\ \mathit{in}\ p; z\ !\ \mathsf{name}_3\ \mathit{in}\ q$ refers to three variables, one named name_1 and a second named name_2 that both belong to the same agent whose agent_id can be referenced as p. The third variable, z, must be named name_3 and located in program q. q may or may not be the same as p, depending on further restrictions that might be specified. Additional variables can be listed in this declaration; they are grouped by program and separated by semicolons. If no combination of variables in the system satisfies the constraints, then the dummy variables are undefined, and the rule reduces to a skip.

As a simple example of a context rule, consider a program with a context variable called c that holds the value of an exposed variable with the name data and located on an agent at the same location as the reference. This context variable simply represents the context, and it does not change the data stored on the agent owning the exposed variable. To achieve this kind of behavior, the specification relies on the existence of the built-in exposed variable with the name $\mathsf{location}$, locally referred to as λ. The context rule for the context variable c uses a single exposed variable that refers to the data that will be stored in c. In this example, we leave the rule unguarded, and it falls into the set of normal statements that are executed in a weakly-fair manner.

$$\begin{array}{rl} c\ \textbf{uses} & d\,!\,\mathsf{data}, l\,!\,\mathsf{location}\ \mathit{in}\ p \\ \textbf{given} & l = \lambda \\ \textbf{where} & c\ \textbf{becomes}\ d \end{array}$$

Formally, using the above notation is equivalent to the following expression:

$$\begin{array}{l} \langle d, l : (d, l) = (d', l').(\{r\} \subseteq \mathsf{var}[d'].\alpha(\mathsf{credentials}) \wedge \{r\} \subseteq \mathsf{var}[l'].\alpha(\mathit{credentials}) \wedge \\ \qquad\qquad\qquad\qquad \mathsf{var}[d'].\eta = \mathsf{data} \wedge \mathsf{var}[l'].\eta = \mathsf{location} \wedge \\ \qquad\qquad\qquad\qquad \mathsf{var}[d'].\pi = \mathsf{var}[l'].\pi \wedge \mathsf{var}[l'].\nu = \lambda.\nu) \\ \quad :: c := \mathsf{var}[d].\nu \\ \quad \rangle \end{array}$$

Because the expression assigned to the context variable c is simply the value of the selected exposed variable, the most interesting portion of this expression is the non-deterministic selection of the exposed variables. The formal expression non-deterministically selects a variable to pull data from that satisfies a set of conditions. These conditions rely on the selection of a second exposed variable that stores the program's location. The first line of the non-deterministic selection checks the access control function for each of the variables to ensure that this agent is allowed read access given its credentials. The second line restricts the names of the two variables. The variable d being selected must be named data, according to the restrictions provided in the rule. The location variable is selected based on its name being $\mathsf{location}$. The final line in the non-deterministic selection deals with the locations of the two variables. The first clause ensures that the two variables (d and l) are located in the same program. The second clause ensures that the agent that owns these two variables is at the same location as the agent defining the rule.

To show how these expressions can be used to facilitate modeling real-world context-aware interactions, we revisit the acquaintance list example from earlier in the section. More extensive examples will be discussed in Section 5.3.

In Figure 5.2, we gave only a high level description of the context rules required to define an agent's acquaintance list. To define the membership qualifications exactly, the agent uses a context rule that adds qualifying agents to the context variable Q that stores the acquaintance list. In this particular case, assume that the program wants to restrict the acquaintance list members to other agents within some predefined range. This range is stored in a local variable whose local handle is referred to as *range*. The acquaintance list context variable can be defined using the following rule:

$$\begin{array}{lll} Q & \textbf{uses} & l \,!\, \textsf{location}\ \textit{in}\ a \\ & \textbf{given} & |l - \lambda| \leq \textit{range} \\ & \textbf{where} & Q\ \textbf{becomes}\ Q \cup \{a\} \\ & \textbf{reactive} & \end{array}$$

This expression uses the two handles *range* and λ to refer to local variables that store the maximum allowable range and the agent's current location, respectively. This statement adds agents that satisfy the membership requirements to the acquaintance list Q one at a time. Because it is a reactive statement that is enabled when an agent is within range, the rule ensures that the acquaintance list remains consistent with the state of the environment. As a portion of the reactive program that executes after each normal statement, this context rule reaches fixed-point when the acquaintance list contains all of the agents that satisfy the requirements for membership. An additional rule is required to eliminate agents that might still be in Q but are no longer in range:

$$\begin{array}{lll} Q & \textbf{uses} & l \,!\, \textsf{location}\ \textit{in}\ a \\ & \textbf{given} & |l - \lambda| > \textit{range} \\ & \textbf{where} & Q\ \textbf{becomes}\ Q - \{a\} \\ & \textbf{reactive} & \end{array}$$

Governing Universal Behaviors

Figure 5.1 showed that the final portion of a Context UNITY system specification is a **Governance** section. It contains rules that capture behaviors that have universal impact across the system. These rules use the exposed variables available in programs throughout the system to affect other exposed variables in the system. The rules have a format similar to the definition of a program's local context rules except that they do not affect individual context variables:

$$\begin{array}{ll} \textbf{use} & \textit{quantified variables} \\ \textbf{where} & \textit{restrictions on quantified variables} \\ & \textit{expr}_1\ \textbf{impacts}\ \textit{exposed variable}_1 \\ & \textit{expr}_2\ \textbf{impacts}\ \textit{exposed variable}_2 \\ & \ldots \end{array}$$

As a simple example of governance, imagine a central controller that, each time its governance rule is selected, non-deterministically chooses an agent in the system and moves it, i.e., it models a random walk. This example assumes a one-dimensional space in which agents are located; essentially the agents can move along a line. Each agent's built-in $\mathsf{location}$ variable stores the agent's position on the line, and another variable named $\mathsf{direction}$ indicates which direction along the line the agent is moving. If the value of the $\mathsf{direction}$ variable is $+1$, the agent is moving in the positive direction along the line; if the value of the $\mathsf{direction}$ variable is -1, the agent is moving in the negative direction. We arbitrarily assume the physical space for movement is bounded by 0 on the low end and 25 on the upper end. The governance rule has the following form:

$$\begin{array}{ll}\textbf{use} & d\,!\,\mathsf{direction}, l\,!\,\mathsf{location}\ \mathit{in}\ p\\ \textbf{where} & l+d\ \textbf{impacts}\ l\\ & (\mathit{if}\ l+d=25\vee l-d=0\ \mathit{then}\ -d\ \mathit{else}\ d)\ \textbf{impacts}\ d\end{array}$$

The non-deterministic selection clause chooses a d and l from the same program with the appropriate variable names. The first of the impact statements moves the agent in its current direction. The second impact statement switches the agent's direction if it has reached either boundary. The rules placed in the **Governance** section can be declared reactive, just as a local program's context rules are. The formal semantic definition of context rules in the **Governance** section differs slightly from the definition outlined above in that the governance rules need not account for the access control policies of the referenced exposed variables. This is due to the fact that the specified rules define system-wide interactions that are assumed, since they are provided by a controller, to be safe and allowed actions. As an example, the formal definition for the rule described above would be:

$$\begin{array}{ll}\langle d,l: & (d,l)=(d',l').(\mathsf{var}[l'].\eta=\mathsf{location}\wedge\mathsf{var}[d'].\eta=\mathsf{direction}\wedge\\ & \qquad\qquad\qquad \mathsf{var}[l'].\pi=\mathsf{var}[d'].\pi)\\ \quad :: & \mathsf{var}[l].\nu:=\mathsf{var}[l].\nu+\mathsf{var}[d].\nu\\ & ||\mathsf{var}[d].\nu:=-\mathsf{var}[d].\nu\ \mathit{if}\ l+d=25\vee l+d=0\\ & \rangle\end{array}$$

Using the unique combination of independent programs, their context rules, and universal governance rules, Context UNITY possesses the ability to model a wide-variety of applications in the area of context-aware computing. We demonstrate this in Section 5.3 by providing snippets of Context UNITY systems required to model applications taken from the context-aware literature. In the next section, we briefly overview the proof logic associated with the Context UNITY model.

5.2 Proof Logic

Context UNITY has an associated proof logic largely inherited from Mobile UNITY [50], which in turn builds on the original UNITY proof logic [10]. Program properties are expressed using a small set of predicate relations whose validity can be derived directly from the program text, indirectly through translation of program text fragments into Mobile UNITY constructs, or from other properties through the application of inference rules. In all of these systems, the fundamental aspect of proving programs correct deals with the semantics of individual program statements. UNITY contains only standard conditional multiple assignment statements, while both Mobile UNITY and Context UNITY extend this model with reactive statements and transactions. Context UNITY also adds nondeterministic assignment statements. In all of these models, proving individual statements correct starts with the use of the *Hoare triple* [30].

For the normal statements in UNITY, a property such as:

$$\{p\}s\{q\} \textbf{ where } s \textbf{ in } P$$

refers to a standard conditional multiple assignment statement s exactly as it appears in the text of the program P. By contrast, in a Mobile UNITY or Context UNITY program, the presence of reactive statements requires us to use:

$$\{p\}s^*\{q\} \textbf{ where } s \in \mathcal{N}$$

where $\mathcal{N}$ denotes the normal statements of P while s^* denotes a normal statement s modified to reflect the extended behavior resulting from the execution of the reactive statements in the reactive program $\mathcal{R}$ consisting of all reactive statements in P. The following inference rule captures the proof obligations associated with verifying a Hoare triple in Context UNITY under the assumption that s is not a transaction:

$$\frac{\{p\}s\{H\}, H \mapsto (FP(\mathcal{R}) \wedge q) \text{ in } \mathcal{R}}{\{p\}s^*\{q\}}$$

The first component of the hypothesis states that, when executed in a state satisfying p, the statement s establishes the intermediate postcondition H. This postcondition serves as a precondition of the reactive program $\mathcal{R}$, that, when executed to fixed-point, establishes the final postcondition q. The "in $\mathcal{R}$" must be added because the proof of termination is to be carried out from the text of the reactive statements, ignoring other statements in the system. This can be accomplished with a variety of standard UNITY techniques. It is required that the predicate H leads to a fixed-point and q in the reactive program $\mathcal{R}$.

This proof obligation (i.e., $H \mapsto (FP(\mathcal{R}) \wedge q)$ in $\mathcal{R}$) can be proven with standard techniques because $\mathcal{R}$ is treated as a standard UNITY program.

For transactions of the form $\langle s_1; s_2; \ldots; s_n \rangle$ we can use the following inference rule before application of the one above:

$$\frac{\{a\}\langle s_1; s_2; \ldots s_{n-1}\rangle^*\{c\}, \{c\}s_n^*\{b\}}{\{a\}\langle s_1; s_2; \ldots s_n\rangle\{b\}}$$

where c may be guessed at or derived from b as appropriate. This represents sequential composition of a reactively-augmented prefix of the transaction with its last sub-action. This rule can be used recursively until we have reduced the transaction to a single sub-action. Then we can apply the more complex rule above to each statement. This rule may seem complicated, but it represents standard axiomatic reasoning for ordinary sequential programs, where each sub-statement is a predicate transformer that is functionally composed with others.

Finally, Context UNITY introduces the notion of non-deterministic assignment to the Mobile UNITY proof logic. The proof obligation of these non-deterministic assignments differs slightly from that of the standard assignment statements. Given the property $\{p\}s\{r\}$ in UNITY, if the statement s is a non-deterministic assignment statement of the form $x := x'.Q(x')$, then the inference rule describing the associated proof obligation for the statement s has the form:

$$\frac{\{p \wedge \exists x' :: Q(x')\}s\{\forall x' : Q(x') :: r\}}{\{p\}s\{r\}}$$

Special care must be taken to translate Context UNITY context rules from both the local program **context** sections and the **Governance** section to standard notation (i.e., to the appropriate normal or reactive statements) before applying the proof logic outlined here. Once translated as described in the previous section, proof of the system can be accomplished directly by applying the rules outlined above.

To prove more sophisticated properties, UNITY-based models use predicate relations. Basic safety is expressed using the **unless** relation. For two state predicates p and q, the expression p **unless** q means that, for any state satisfying p and not q, the next state in the execution must satisfy either p or q. There is no requirement for the program to reach a state that satisfies q, i.e., p may hold forever. Progress is expressed using the **ensures** relation. The relation p **ensures** q means that for any state satisfying p and not q, the next state must satisfy p or q. In addition, there is some statement in the program that guarantees the establishment of q if executed in a state satisfying p and not q. Note that the **ensures** relation is not itself a pure liveness property but is a conjunction of a safety and a liveness property; the safety part of the **ensures** relation can be expressed as an **unless** property. In UNITY, these predicate relations are defined by:

$$p \textbf{ unless } q \equiv \langle \forall s : s \textbf{ in } P :: \{p \wedge \neg q\} s \{p \vee q\} \rangle$$

$$p \textbf{ ensures } q \equiv (p \textbf{ unless } q) \wedge \langle \exists s : s \textbf{ in } P :: \{p \wedge \neg q\} s \{q\} \rangle$$

where s is a statement in the program P. Mobile UNITY and Context UNITY use the same definitions since all distinctions are captured in the verification of the Hoare triple. Additional relations may be derived to express other safety (e.g., **invariant** and **stable**) and liveness (e.g., **leads-to**) properties.

5.3 Patterns of Context-Awareness

Much published research acknowledges the need for applications that rapidly adapt to changes in resource availability and the operational environment. As a result, a number of researchers sought to provide context-aware software systems designed to function in a variety of operating scenarios. These systems vary in their approaches to managing context; models that underlie context-aware systems range from a simple client-server model in which servers provide context information directly to clients, to sophisticated tuple space coordination models in which the details of communicating context information is transparent to the application. In this section, we examine a representative set of context-aware systems found in the literature, abstract their key features, and suggest ways to model them in Context UNITY.

5.3.1 Simple context interactions

Initial work in context-aware computing resulted in the development of applications that use relatively simple context definitions. Such systems often separate concerns related to providing and using context. Many systems consist of *kiosks*, entities which provide context information to *visitors*, which use context and state information to adapt their behavior.

Applications exhibiting the characteristics of the simple kiosk-visitor interaction pattern include context-aware office applications such as Active Badge [28] and PARCTab [65]. In these systems, personnel carry devices that periodically communicate a unique identifier via a signal to fixed sensors, allowing the location of the carrier to be known. An application uses the location information to adapt the office environment accordingly in response to the changing location of the carrier, e.g., by forwarding phone calls to the appropriate office or changing the applications available on a workstation. Another type of context-aware applications that use simple context interactions relate to the development of tour guides, e.g., Cyberguide [1] and GUIDE [14]. In these applications, tourists carry mobile devices equipped with context-aware tour guide software. As a tourist moves about in a guide-friendly area, his display is updated according to locally stored preferences combined with context information provided by stationary access points located at points of interest.

In all of the context-aware applications described above, a particular type of entity provides context information and another type reads and uses the provided information. Generally, one of the parties is stationary, while the other is mobile. We can readily capture this style of interaction in Context UNITY. Agents providing context information to other agents in the Context UNITY system do so through the use of *exposed* variables. Agents obtain the provided context information through the use of *context* variables, the values of which are defined by values of selected exposed variables of context-providing agents.

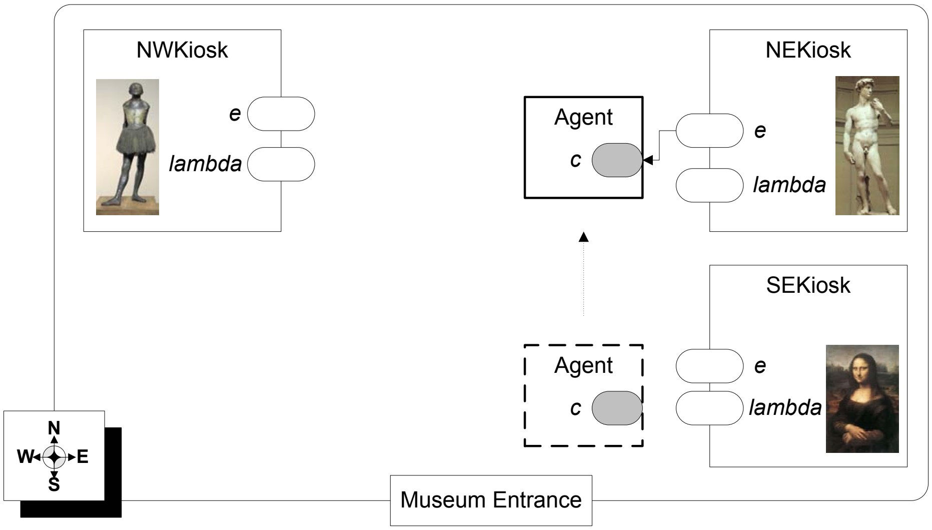


Figure 5.4: A simple guide system in Context UNITY

Figure 5.4 illustrates the interaction between a visitor and kiosks in a simple museum guide system. In this system, each stationary museum kiosk provides information about an exhibit at its location using an exposed variable. A kiosk in the southeast corner of the museum gives information about a painting through its exposed variable e named “painting” with a textual description of the painting as the variable’s value. The kiosks in the northeast and northwest corners of the museum each provide information about a certain sculpture by naming its exposed variable e “sculpture,” and assigning to the variable a short textual description of the work of art at that location. As a particular visitor moves around the room, his context variable, c, defined to contain a co-located sculpture exhibit, changes in response to the available context. If new context information about a sculpture is available, the visitor’s display is updated to show the information. The figure depicts what happens when a visitor walks around the museum. The initial position of the visitor agent is depicted by the dashed box labeled “Agent.” As the visitor moves around the museum in the path indicated by the dotted arrow, the context variable c is updated. Specifically, when the visitor reaches the northeast corner of the museum, the context variable c is updated to contain information about the sculpture at that location. Such an application can be specified in the Context UNITY notation, as shown below. For brevity, we show only the most interesting aspect of the system specification, which is a visitor's context rule:

$$\begin{array}{rl} c\ \mathbf{uses} & e\,!\,\textsf{sculpture}, l\,!\,\textsf{location}\ \mathit{in}\ p \\ \mathbf{given} & l = \lambda \\ \mathbf{where} & c\ \mathbf{becomes}\ e \end{array}$$

More complex patterns of interaction are frequently utilized in the development of context-aware systems. In some systems, for instance, kiosks provide context information to a stationary context manager, and the context manager communicates directly with visitors to adapt their behavior accordingly given the current conditions of the environment. An instance of this pattern of interaction is found in the Gaia operating system [51], which manages *active spaces*. An active space is a physical location in which the physical and logical resources present can be adapted in response to changes in the environment. A typical interaction in an active space is as follows: a user enters the active space and registers with the context manager, which uses information about the user and the environment to perform appropriate actions, e.g., turn on a projector and load the user's presentation. Such a system can be modeled in Context UNITY similarly to those systems described above that exhibit simple context interactions: users are providing context information to the context manager through the use of exposed variables, and the context manager uses context variables to obtain context information and react accordingly.

5.3.2 Security-constrained context interactions

Security is a major concern in the development of all modern software systems, including those supporting context-awareness. In several systems, multi-level security mechanisms are provided through the use of *domains*. A domain provides a level of security and isolates the available resources according to the level of security offered. Agents authorized to operate within that domain have the ability to act upon all resources within a domain, and a domain may have an authorizing authority that grants and revokes entering and exiting agents' access rights. Examples of systems exhibiting such characteristics include the Gaia file system [51] and the multi-level access control proposed by Wickramasuriya and Venkatasubramanian [66].

Figure 5.5 illustrates an example use of such an interaction style. In the example, a patient at a doctor's office must provide information about himself in order to receive treatment. Some of the information provided is fairly public knowledge and can be viewed by

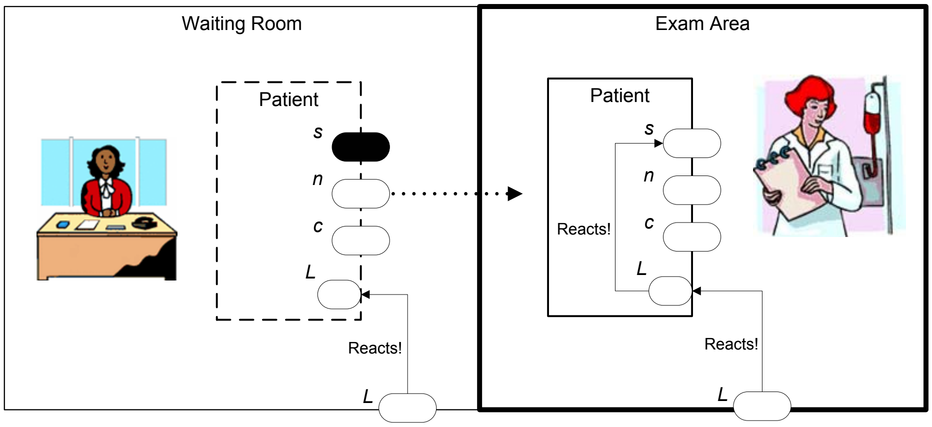


Figure 5.5: An example security-constrained context-aware application in Context UNITY.

the receptionist and other patients, e.g., name and contact information. Other information is highly sensitive and personal, e.g., health history, and should only be shared with a doctor. To facilitate this kind of interaction, the doctor's office is divided into two areas that provide different levels of privacy: the waiting room and the exam area. The waiting room is a public space (low-security), since the receptionist and other patients in the waiting room can view the information provided therein. The exam area is private (high-security), since only the patient and doctor can view the information. In the figure, in the waiting room domain, which offers a low level of security in its exposed variable L, the patient's sensitive information about symptoms is protected from inclusion in the domain by the symptom variable's access control function. The shading on the oval labeled s indicates that the symptom variable is not accessible to anyone in the environment. As the patient moves to the exam area domain offering high level security, the patient's domain security level is updated immediately, as indicated by the arrow labeled "reacts." As a result of the changed security level, a second reaction is triggered whose effect is to alter the access control function of the symptom variable s to allow the value to be available to those in the exam area domain.

To describe such applications in Context UNITY, domains could reveal their security level using an exposed variable L named "security level." Each patient agent uses a context rule for its context variable L to discover the level of security offered by the domain in which it is located. Because the definition is built to be strongly consistent using a reactive statement, the agent's perception of the security level offered by its current domain is guaranteed to be accurate and up to date. Each patient provides his name, contact information, and symptoms through the use of exposed variables n, c, and s. A patient controls how his information is made available through the use of each variable's access control function.

This access control function can be changed during the execution of the program to reflect the agent's changing data protection needs. Using a reaction, it is possible to ensure that the access control function is immediately changed to reflect a change in the security level as soon as a new domain (and hence, a new level of security) is entered.

5.3.3 Tailored context definitions

Often, the amount of context information available to a context-aware agent grows large and unmanageable. To avoid presenting an agent with an overwhelming amount of context in such a scenario, it is desirable to limit the amount of context information that the agent "sees" based on properties of its environment. The EgoSpaces middleware discussed in detail in Chapter 3 is an obvious example of a context-aware system that does just this. At the heart of EgoSpaces is the *view* concept, which restricts an agent's context according to the agent's individualized specification. A view consists of constraints on network properties, the agents from which context is obtained, and the hosts on which such agents reside. These constraints are used to filter out unwanted items in the operational environment and results in presenting the agent with a context (view of the world) tailored to its particular needs.

In a general sense, systems such as EgoSpaces consist of possibly mobile agents that are both providers and users of context, and a context management strategy that is performed on a per-agent basis. An individualized context is managed on behalf of each agent by matching items from the entire operational environment against the restrictions provided in the view definition, and presenting the result to the agent as its context. Such a system can be readily expressed in Context UNITY. To act as a context provider, an agent generates pieces of context information and places them in an exposed variable, a tuple space, in the case of EgoSpaces, i.e., a data repository consisting of tuples that the agent wishes to contribute as context. An agent provides information about itself and properties about the host on which it resides in exposed variables named "agent profile" and "host profile," respectively. They allow other agents to filter the operational environment according to the host and agent constraints in their view definitions. To act as a context user, we model an agent's view using a rule for a context variable v named "view." The value of v is defined to be the set of all tuples present in exposed tuple space variables of other reachable agents for which the exposed agent profile properties, exposed host profile properties, and exposed network properties of hosts match the reference agent's constraints. An example context rule that establishes a view v for an agent with id i to "see" can be described as follows:

$$\begin{array}{ll} v\ \mathbf{uses} & \mathit{lts}\ !\ \mathsf{tuple\_space},\ a\ !\ \mathsf{agent\_profile},\ h\ !\ \mathsf{host\_profile}\ \mathit{in}\ i \\ \quad\mathbf{given} & \mathit{reachable}(i) \wedge \mathit{eligibleAgent}(a) \wedge \mathit{eligibleHost}(h) \\ \quad\mathbf{where} & v\ \mathbf{becomes}\ v - (v \uparrow i)\ \cup\ \mathit{lts} \\ \quad\mathbf{reactive} & \end{array}$$

The function *reachable* encapsulates the network constraints that establish whether an agent should or should not be considered based on network topology data. The notation $v \uparrow i$ indicates a projection over the set v that contains tuples owned by the agent i. It is possible to obtain such a projection since we assume that each generated tuple has a field which identifies the owner of the tuple using the generating agent's unique id. In order for an agent to perform changes to the view v and have them propagate to the correct tuple space *lts* additional context rules are needed.

5.3.4 Uniform context definition

Coordination models offer a high degree of decoupling, an important design characteristic of context-aware systems. In many distributed computing environments, tuple spaces are permanently attached to agents or hosts. In some models, these pieces merge together to logically form a single shared tuple space in a manner that takes into consideration the connectivity among agents or hosts. An agent interacts with other agents by employing content-based retrieval (`rd(pattern)` and `in(pattern)`), and by generating tuples (`out(tuple)`). Often, the traditional operations are augmented with reactions that extend their effects to include arbitrary atomic state transitions. Systems borne out of such a tuple space coordination paradigm can be considered context-aware; an agent's context is managed by the tuple space system in the form of tuples in a logically shared tuple space.

Examples of such context-aware systems are TSpaces [34], JavaSpaces [58], MARS [7], and LIME [41]. A common characteristic of all these systems is the fact that agents that enter in a sharing relation have the same definition of context, i.e., the context rules are uniform and universally applied. Among the systems we cite here, LIME is the most general, as it allows both physical mobility of hosts and logical mobility of agents, and provides tuple space sharing in the most extreme of network environments — the ad hoc network. In LIME, agents are units of execution, mobility, and data storage, while hosts are simply containers of agents. Hosts may be mobile, and agents can migrate from host to host. Agents may be associated with several local tuple spaces, distinguished by name. Since it is a passive entity, a host has no tuple space. A LIME agent's relevant context is determined by the logically merged contents of identically named tuple spaces held by mutually reachable agents.

To capture the essential features of context-aware systems having the characteristics described above in Context UNITY, it suffices to endow an agent with one exposed variable named $\mathsf{localTS}$ that offers its local tuple space for sharing and a second exposed variable named $\mathsf{sharedTS}$ that should provide access to all the tuples making up the current context. The value of the latter is the union of tuples contained in exposed local tuple space variables belonging to connected agents. Connectivity can be defined based on various properties of the network, e.g., network hops, physical distance, etc. In MARS, only agents residing on the same host are *connected*. In LIME, agents are *connected* when residing on the same host or on connected hosts.

A final and important point to note about the modeling of such systems is that since the shared tuple space definition is uniform across all agents, we can capture it in the **Governance** section of a Context UNITY system. While it is possible to define an agent's context locally in its program description, using the **Governance** section highlights the fact that connected agents share a symmetric context. In addition, it is more economical for a programmer to write a single context definition since it applies to the entire system. The resulting context rule included in the **Governance** section is as follows:

$$\begin{array}{ll}
\textbf{use} & ts_c \,!\, \mathsf{sharedTS}\ \mathit{in}\ a; ts_l \,!\, \mathsf{localTS}\ \mathit{in}\ b \\
\textbf{given} & \mathit{connected}(a, b) \\
\textbf{where} & ts_c \ - \ (ts_c \uparrow b) \ \cup \ ts_l \ \textbf{impacts} \ ts_c \\
\textbf{reactive} &
\end{array}$$

The result of this context rule is a tuple space shared among connected agents.

This brings to an end our discussion on how Context UNITY relates to some of the existing models of context-awareness. The most striking observation about this informal evaluation of the model is the simplicity exhibited by each of the context rules that were generated in this section.

5.4 Chapter Summary

In this chapter, we outlined the requirements of a model for reasoning about context-awareness. We built such a model in Context UNITY, which provides state-based reasoning for programs by representing the context of a program within state variables that the program itself can interact with. To provide an initial evaluation of the expressiveness of Context UNITY, we used this basic model to represent a variety of context-aware systems, including those not developed by our research group. Initial results show the model to be quite expressive. We also provided a brief look into a proof logic associated with the formal representations. Further work on Context UNITY will extend the proof logic to allow direct reasoning about context interactions.

Chapter 6

Conclusions

This dissertation describes a concerted effort to simplify the application development process for programmers in ad hoc mobile networks. The investigation began with a careful study of the nature of potential applications and the classification of the communication mechanisms employed by these applications into a general asymmetric style of coordination. Given the lessons learned from this exploration, we built a conceptual model of applications for mobile ad hoc networks.

The conceptual model is founded on the belief that context-awareness offers the potential to greatly reduce the development effort by abstracting the data available in the ad hoc network into a data structure. The interaction with this data structure is natural for any novice programmer and reduces the need for complex and error-prone network programming. This use of context-awareness strictly within mobile computing and for the purpose of simplifying development for novice programmers is the first of its kind. It is coupled with the concept of asymmetric coordination, also first introduced in this work. The asymmetry is founded on the observation that applications in ad hoc networks tend to be egocentric in that they define their needs from the environment from a perspective within that environment and independent of the needs of other applications. We merge context-awareness and asymmetric coordination into a unified model and provide that conceptual model for application development within a middleware called EgoSpaces.

The application development benefits of the work presented in this dissertation are not limited to the EgoSpaces middleware. We also evaluated the application of formal reasoning models to the emerging field of context-aware computing in general. While context-aware applications can, in many cases, still be formally expressed and reasoned about using existing models, the resulting formalizations lack the direct expression of context and a program's interaction with it. To bridge this gap, we have developed Context UNITY, a state-based reasoning model dedicated to the expression of context-aware applications and systems. We have shown that this model is expressive enough to capture context-aware systems in a variety of environments, including: wired networks, mixed wired and wireless networks, and ad hoc wireless networks.

While the efforts of this dissertation are directed to the software engineer, the work also impacts other fields. At the network level, we expressed the need for a rethinking of the paradigms of communication, especially within ad hoc mobile networks. While much previous research has focused on providing the same mechanisms of communication as were available in wired networks, we discovered that the very nature of ad hoc network applications demands an entirely different style of communication. We have generalized the requirements of a set of such applications and taken steps towards mediating their communication requirements. We have accomplished this through the network abstractions protocol. Second, in the area of middleware, EgoSpaces serves as a proof of concept middleware environment detailing the careful construction of a compartmentalized architecture for use in ad hoc mobile networks. We have built several components within this architecture that prove useful outside of EgoSpaces in building slimmer versions of middleware for ad hoc mobile systems.

This work has also opened several avenues that build directly on the work presented here. Further work at the lower level would benefit both existing systems and those to come. The network abstractions protocol stemmed from an effort to explicitly express the needs of ad hoc network applications. We required a protocol to support our middleware, so we developed the protocol discussed in this dissertation. As the evaluation we performed shows, this protocol and its implementation have several areas for possible improvement, learning from existing methods targeted both for communication in ad hoc networks and in sensor networks.

Second, the EgoSpaces model and middleware serve as a stable foundation for extension into various forms of coordination. One conceivable extension builds on this work to provide different abstractions of the context in place of EgoSpaces's tuple space. One can imagine, for example, a priority queue where the elements of the context are ordered according to some application-specified ordering function. We could further envision a case where the information accessed is not data but references to objects. Applications can use these references to interact with other parties in the network. A more advanced coordination system could allow applications to specify context-sensitive bindings to these objects within the context that are automatically updated as the objects available in the world change. For example, an application might request that it always be connected to the closest printer, and as the host moves through a building, the printer at the other end of the connection is automatically updated.

Another area of future work at a high-level of abstraction lies in the need to formally classify the semantics of consistency within ad hoc mobile networks. For the purposes of building EgoSpaces, we relied on the ability to perform transactions for several of the constructs. While we provide a protocol that allows us to ensure that a transaction can complete, the costs associated with it are often prohibitive. On the other hand, throwing caution to the wind and using constructs with absolutely no guarantees is often unacceptable for an application. While work in wired networks has focused on the formal semantics of transactions, the same area in wireless networks is relatively untouched. Discovering and providing a range of consistency guarantees between the two extremes has the potential to greatly impact application development and usefulness in ad hoc networks.

References

[1] G. Abowd, C. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: A mobile context-aware tour guide. *ACM Wireless Networks*, 3(5):421–433, October 1997.

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

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

[4] S. Bae, S.-J. Lee, W. Su, and M. Gerla. The design, implementation, and performance evaluation of the On-Demand Multicast Routing Protocol in multihop wireless networks. *IEEE Network, Special Issue on Multicasting Empowering the Next Generation Internet*, 14(1):70–77, January/February 2000.

[5] A. Bakre and B. R. Badrinath. Reworking the RPC paradigm for mobile clients. *ACM-Baltzer Journal on Mobile Networks and Applications (Special Issue on Mobile Computing - System Services)*, 1(4):371–385, December 1996.

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

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

[8] G. Cabri, L. Leonardi, and F. Zambonelli. Engineering mobile agent applications via context- dependent coordination. *IEEE Transactions on Software Engineering*, 28(11):1040–1056, November 2002.

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

[10] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, NY, USA, 1988.

[11] G. Chen and D. Kotz. Solar: An open platform for context-aware mobile applications. In *Proceedings of the 1st International Conference on Pervasive Computing*, pages 41–47, March 2002.

[12] S. Chen and K. Nahrstedt. Distributed quality-of-service routing in ad-hoc networks. *IEEE Journal on Selected Areas in Communications*, 17(8):2580–2592, August 1999.

[13] C. Cheng, R. Riley, S. Kumar, and J. Garcia-Luna-Aceves. A loop-free extended Bellman-Ford routing protocol without bouncing effect. In *Proceedings of the ACM SIGCOMM*, pages 224–236, September 1989.

[14] 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 the 6th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 20–31, August 2000.

[15] C. Chiang and M. Gerla. Routing and multicast in multihop, mobile wireless networks. In *Proceedings of IEEE International Conference on Universal Personal Communications*, pages 546–551, October 1997.

[16] C. Chiang, M. Gerla, and L. Zhang. Adaptive shared tree multicast in mobile wireless networks. In *Proceedings of GLOBECOM '98*, pages 1817–1822, November 1998.

[17] G. Cugola, E. Di 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, September 2001.

[18] G. Cugola and G. P. Picco. PEERWARE: Core middleware support for Peer to Peer and mobile systems. Technical report, Politecnico di Milano, 2001.

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

[20] A. K. Dey, D. Salber, and G. D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human Computer Interaction*, 16(2–4):97–166, 2001.

[21] K. Edwards. *Core JINI.* Prentice Hall, 1999.

[22] W. Emmerich. *Engineering Distributed Objects*. John Wiley and Sons, Ltd., 2000.

[23] C.-L. Fok, G.-C. Roman, and G. Hackmann. A lightweight coordination middleware for mobile computing. In *Proceedings of the 6^{th} International Conference on Coordination Models and Languages*, February 2004.

[24] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

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

[26] R. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 154–187. 1998.

[27] S. Gupta and P. Srimani. An adaptive protocol for reliable multicast in mobile multi-hop radio networks. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 111–122, February 1999.

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

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

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[31] O. Holder, I. Ben-Shaul, and H. Gazit. System support for dynamic layout of distributed applications. In *Proceedings of the 19^{th} International Conference on Distributed Conputing*, pages 403–411, May–June 1999.

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

[33] Qingfeng Huang, Christine Julien, and Gruia-Catalin Roman. Relying on safe distance to achieve strong partitionable group membership in ad hoc networks. *IEEE Transactions on Mobile Computing*, 3(2):192–205, April–June 2004.

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

[35] IEEE Standards Department. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE standard 802.11-1999, 1999.

[36] D. Johnson and D. Maltz. Dynamic Source Routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[37] P. Keleher and U. Cetintemel. Consistency management in Deno. *Mobile Networks and Applications*, 5(4):299–309, December 2000.

[38] L. Kleinrock and J. Silvester. Optimum transmission radii in packet radio networks or why six is a magic number. In *Proceedings of the IEEE Nationa. Telecommunications Conference*, pages 4.3.1–4.3.5, December 1978.

[39] E. Madruga and J. Garcia-Luna-Aceves. Scalable multicasting: The core assisted mesh protocol. *ACM/Baltzer Mobile Networks and Applications, Special Issue on Management of Mobility*, 6(2):151–165, April 1999.

[40] P. J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, February 1998.

[41] 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, April 2001.

[42] OMG. Orb and object services task force. http://www.omg.org/homepages/orbos/, 2001.

[43] A. Omicini and F. Zambonelli. TuCSoN: A coordination model for mobile information agents. In *Proceedings of the 1st International Workshop on Innovative Internet Information Systems*, pages 177–187, June 1998.

[44] 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.

[45] J. Pascoe. Adding generic contextual capabilities to wearable computers. In *Proceedings of the 2nd International Symposium on Wearable Computers*, pages 92–99, October 1998.

[46] C. Perkins and P. Bhagwat. Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers. In *ACM SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, October 1994.

[47] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[48] G. P. Picco. μCode: A lightweight and flexible mobile code toolkit. In *Proceedings of the 2nd International Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 160–171, September 1998.

[49] B. Rhodes. The wearable remembrance agent: A system for augmented memory. In *Proceedings of the 1st International Symposium on Wearable Computers*, pages 123–128, October 1997.

[50] G.-C. Roman and P. J. McCann. A notation and logic for mobile computing. *Formal Methods in System Design*, 20(1):47–68, 2002.

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

[52] E. Royer, P. Melliar-Smith, and L. Moser. An analysis of the optimum node density for ad hoc mobile networks. In *Proceedings of the IEEE Conference on Communications*, June 2001.

[53] E. Royer and C.-K. Toh. A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications*, 6(2):46–55, April 1999.

[54] R. Ruggaber and J. Seitz. A transparent network handover for nomadic CORBA users. In *Proceedings of the The 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 499–508, April 2001.

[55] D. Salber, A. Dey, and G. Abowd. The Context Toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 434–441, May 1999.

[56] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 85–90, December 1994.

[57] Soma. http://www-lia.deis.unibo.it/Software/SOMA, 2001.

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

[59] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, and G. A. Hill. Strong mobility and fine-grained resource control in NOMADS. In *Proceedings of the 2nd International Symposium on Agent Systems and Applications and the 4th International Conference on Mobile Agents*, number 1882 in LNCS, pages 2–15, September 2000.

[60] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002.

[61] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, and W. Joosen. Portable support for transparent thread migration in Java. In *Proceedings of the 2^{nd} International Symposium on Agent Systems and Applications and the 4^{th} International Conference on Mobile Agents*, number 1882 in LNCS, pages 29–43, September 2000.

[62] B. Venners. Under the hood: The architecture of aglets. *JavaWorld*, 2(4), 1997.

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

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

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

[66] J. Wickramasuriya and N. Venkatasubramanian. A middleware approach to access control for mobile concurrent objects. In *Proceedings of the International Symposium on Distributed Objects and Applications*, October 2002.

[67] J. Yoon, M. Liu, and B. Noble. Random waypoint considered harmful. In *Proceedings of INFOCOM*, pages 1312–1321, April 2003.

Vita

Christine Julien

Date of Birth December 8, 1977

Place of Birth Washington, D.C.

Degrees
D.Sc. Computer Science, Washington University, August 2004
M.S. Computer Science, Washington University, May 2003
B.S. Summa Cum Laude, Computer Science and Biology, Washington University, May 2000

Honors NSF Graduate Research Fellow, 2001-2004

Experience
Mobile Computing Lab, Washington University, 2000-2004
bioMérieux, Saint Louis, MO, 1999

Publications

Qingfeng Huang, Christine Julien, and Gruia-Catalin Roman. Relying on Safe Distance to Achieve Strong Partitionable Group Membership in Ad Hoc Networks, *IEEE Transactions on Mobile Computing*, 3, No. 2, April–June 2004, pp. 192–205.

Christine Julien, Gruia-Catalin Roman, and Jamie Payton. Context-Sensitive Access Control for Open Mobile Agent Systems, in *Proceedings of the 3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'2004)*, May 2004, pp. 42–28.

Jamie Payton, Gruia-Catalin Roman, and Christine Julien. Context-Sensitive Data Structures Supporting Software Development in Ad Hoc Mobile Settings, in *Proceedings of the 3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'2004)*, May 2004, pp. 34–41.

Gruia-Catalin Roman, Christine Julien, and Jamie Payton. A Formal Treatment of Context-Awareness (invited paper). In *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, volume 2984, pages 12–16, March 2004.

Christine Julien and Gruia-Catalin Roman. Active Coordination in Ad Hoc Networks. In *Proceedings of the 6^{th} International Conference on Coordination Models and Languages*, pages 199–215, February 2004.

Christine Julien, Jamie Payton, and Gruia-Catalin Roman. Reasoning About Context-Awareness in the Presence of Mobility. In *Proceedings of the 2^{nd} International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA03)*, September 2003.

Radu Handorean, Jamie Payton, Christine Julien, and Gruia-Catalin Roman. Coordination Middleware Supporting Rapid Deployment of Ad Hoc Mobile Systems. In *Proceedings of the 1^{st} ICDCS Workshop on Mobile Computing Middleware (MCM03)*, pages 362–368, May 2003.

Gruia-Catalin Roman, Christine Julien, and Amy L. Murphy. A Declarative Approach to Agent-Centered Context-Aware Computing in Ad Hoc Wireless Environments (extended version). In A. Garcia et al, editors, *Software Engineering for Large-Scale Multi-Agent Systems*, Lecture Notes in Computer Science, volume 2603, pages 94–109, April 2003.

Christine Julien and Gruia-Catalin Roman. Egocentric Context-Aware Programming in Ad Hoc Mobile Environments. In *Proceedings of the 10^{th} International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 21–30, November 2002.

Gruia-Catalin Roman, Christine Julien, and Qingfeng Huang. Network Abstractions for Context-Aware Mobile Computing. In *Proceedings of the 24^{th} International Conference on Software Engineering (ICSE'02)*, pages 393–373, May 2002.

Gruia-Catalin Roman, Christine Julien, and Amy L. Murphy. A Declarative Approach to Agent-Centered Context-Aware Computing in Ad Hoc Wireless Environments. In *the 1^{st} International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'2002)*, May 2002.

Gruia-Catalin Roman, Christine Julien, and Qingfeng Huang. Formal Specification and Design of Mobile Systems. In *Proceedings of the 7^{th} International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'2002)*, April 2002.

August 2004