Washington University in St. Louis

[Washington University Open Scholarship](#)

Report Number: WUCSE-2004-37

2004-06-23

Network Abstractions for Simplifying Mobile Application Development

Christine Julien, Gruia-Catalin Roman, and Qingfeng Huang

Context-aware computing is characterized by the ability of a software system to continuously adapt its behavior to a changing environment over which it has little or no control. This style of interaction is imperative in ad hoc mobile networks that consist of numerous mobile hosts coordinating with each other opportunistically via transient wireless interconnections. In this paper, we provide a formal abstract characterization of a host's context that extends to encompass a neighborhood within the ad hoc network. We provide an application in an ad hoc network a specification mechanism for defining such contexts that allows individual applications to... **Read complete abstract on page 2.**

Network Abstractions for Simplifying Mobile Application Development

Christine Julien, Gruia-Catalin Roman, and Qingfeng Huang

Complete Abstract:

Context-aware computing is characterized by the ability of a software system to continuously adapt its behavior to a changing environment over which it has little or no control. This style of interaction is imperative in ad hoc mobile networks that consist of numerous mobile hosts coordinating with each other opportunistically via transient wireless interconnections. In this paper, we provide a formal abstract characterization of a host's context that extends to encompass a neighborhood within the ad hoc network. We provide an application in an ad hoc network a specification mechanism for defining such contexts that allows individual applications to tailor their operating contexts to their personalized needs. We describe a context maintenance protocol that provides this context abstraction in ad hoc networks through continuous evaluation of the context. This relieves the application developer of the obligation of explicitly managing mobility and its implications on behavior. The software engineering gains resulting from the use of this abstraction are measured through its expressiveness and simplicity of use. We also characterize the performance of this protocol in real ad hoc networks through simulation experiments. Finally, we describe an initial implementation of the abstraction and provide real world application examples demonstrating its use.

Network Abstractions for Simplifying Mobile Application Development

CHRISTINE JULIEN and GRUIA-CATALIN ROMAN
Washington University in Saint Louis
and
QINGFENG HUANG
Palo Alto Research Center (PARC)

Context-aware computing is characterized by the ability of a software system to continuously adapt its behavior to a changing environment over which it has little or no control. This style of interaction is imperative in ad hoc mobile networks that consist of numerous mobile hosts coordinating with each other opportunistically via transient wireless interconnections. In this paper, we provide a formal abstract characterization of a host's context that extends to encompass a neighborhood within the ad hoc network. We provide an application in an ad hoc network a specification mechanism for defining such contexts that allows individual applications to tailor their operating contexts to their personalized needs. We describe a context maintenance protocol that provides this context abstraction in ad hoc networks through continuous evaluation of the context. This relieves the application developer of the obligation of explicitly managing mobility and its implications on behavior. The software engineering gains resulting from the use of this abstraction are measured through its expressiveness and simplicity of use. We also characterize the performance of this protocol in real ad hoc networks through simulation experiments. Finally, we describe an initial implementation of the abstraction and provide real world application examples demonstrating its use.

---

1. INTRODUCTION

The ubiquity of mobile computing devices opens the user's operating environment to a rapidly changing world where the network topology, or the physical connections between hosts in the network, must be constantly recomputed. These network connections link a host to information that can be provided by other members of the computing environment—information we refer to as a host's context. In ad

---

C. Julien and G.-C. Roman are with the Department of Computer Science and Engineering, Washington University, Saint Louis, MO 63130. Email: {julien, roman}@wustl.edu

Q. Huang is with Palo Alto Research Center (PARC), 3333 Coyote Hill Road, Palo Alto, CA 94304. Email: qhuang@parc.com

hoc networks especially, software must adapt its behavior continuously in response to this changing context. In this environment, devices are commonly small and constrained and must often rely on other connected devices for information and computations. Because of this, mobile units in the ad hoc network become part of other hosts' contexts. In general, ad hoc mobile networks contain many hosts and links, all with meaningful physical characteristics. These hosts, links, and their properties define the context for an individual host in the network. The behavior of an adaptive application running on such a host depends on this continuously changing context.

The next section discusses the current trends in context-aware computing, which often limit a host's context to what it can immediately sense or limit the type of information used to define a context to specific information (e.g., location). To support more general context-aware applications, an individual application must be allowed to define an individualized context; such definitions extend beyond the current notion of context and may need to include a rich amalgamation of facets of the environment. In addition, previous work tends to limit the breadth of context collection to single nodes in the network. Our goals include broadening the context available to a host to include not only properties that can be sensed directly by a host, but also properties of other reachable hosts and properties of the links among them. This approach has the potential to greatly increase the amount of context information available, and therefore an application running on a host should specify exactly the context that interests it. For example, an ad hoc network on a highway might extend for hundreds or even thousands of miles. A driver in a particular car, however, may be interested in only gas stations within five miles. An application in such an environment should supply a definition of its desired context; subsequent operations issued by the application should be performed only over the context specified by the application. Because we aim to provide both a manner for an application to specify its context and a protocol that computes and maintains the context according to this specification, we need to allow the context specification to remain as general and flexible as possible while ensuring the feasibility and efficiency of the protocol to dynamically compute the context.

Many application scenarios will benefit directly from the use of such declarative context specifications. 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 information 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 the constructed subnet. 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 subjects. The context defined in this case 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. Throughout this paper, we will revisit this example and indicate how this application relates to individual pieces of the work.

To ease the software engineering task for this highly dynamic environment, we provide this network abstraction to the application developer through a simple interface that allows definition of expressive metrics over the ad hoc network. Specifically, we provide an application running on a particular host, henceforth called the reference host, the ability to formally specify a context that spans a subnet of the ad hoc network in existence at a given time. Abstractly, one can view the context as a subnet around the reference host and the properties of that subnet's components (hosts and links). The work starts from the premise that the development of mobile applications can be simplified by allowing developers to specify such individualized contexts. We address both specification and implementation concerns relating to context definition and maintenance. First is the question of how to facilitate a formal specification of context that is general, flexible, and amenable for use in ad hoc settings. The solution maps all nodes in the ad hoc network to points in a abstract multi-dimensional space and defines context as the set of all points whose distance from the point of reference (i.e., the point denoting the host carrying the application of interest) does not exceed some bound that can change throughout the lifetime of the application. We will show that a number of useful contexts can be defined in this manner. The second issue is one of being able to maintain the specified context and to operate on it. The protocol presented in this paper constructs and dynamically maintains a tree over a subnet of neighboring hosts and links whose attributes contribute to the definition of a give context, as required by an application on a particular mobile host. Context-sensitive operations are carried out through a cooperative effort involving only hosts that are part of a given context. The final concern is of the presentation of this construct to the application developer. We build a simple programming API that not only includes built-in general-purpose metrics but also provides a usable mechanism for creating additional network abstraction metrics.

This paper is organized as follows. Section 2 details previous work in context-aware computing and discusses the need for our new perspective on context. In Section 3, we provide background on routing in ad hoc networks, which relates directly to our need to distribute messages among hosts in ad hoc networks. We present our abstraction of the network into a context in Section 4 and provide some example applications using this abstraction in Section 5. Section 6 describes the protocol for context computation and maintenance in detail. In Section 7, we describe our initial implementation and discuss the software engineering concerns associated with using the implementation. Section 8 provides simulation results for the protocol. Finally, Section 9 contains some conclusions.

2.   CONTEXT-AWARE COMPUTING

Context-aware computing first came to the forefront in the early 1990's with the introduction of small mobile computing devices. Initial investigations at Olivetti

Research Lab and Xerox PARC laid the foundation for the development of more recent context-aware software. Olivetti's Active Badge [Want et al. 1992] uses infrared communication between badges worn by users and sensors places in a building to monitor movement of the users and forward telephone calls to them. Xerox PARC's PARCTab [Want et al. 1995] system also uses infrared communication between users' palm top devices and desktop computers. It uses location information to allow applications to adapt to the user's environment. Applications developed for PARCTab 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 [Harter et al. 2002] in building ubiquitous computing environments for supporting similar applications uses CORBA and operates over wired network infrastructure that supports both localization and communication. These systems require extensive infrastructures which require constant maintenance. They also rely on wired communication and do not address the issues inherent to ad hoc networks, including the need to scale to large networks that span more than single buildings.

More recent context-aware applications serve as tour guides by presenting information about the user's current environment. Cyberguide [Abowd et al. 1997] from Georgia Tech, and GUIDE [Cheverst et al. 2000] from the University of Lancaster are examples of two such systems. Fieldwork tools [Pascoe 1998] automatically attach contextual information, e.g., location and time, to notes taken by a researcher in the field. Memory aids [Rhodes 1997] 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, e.g., the guide tools use location information they gather from localization sensors.

Generalized software built to support the development of context-aware computing in mobile environments has begun to be developed. Among the best known systems are the Context Toolkit [Salber et al. 1999] and the Context Fabric [Hong and Landay 2001]. 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 is founded on the observation that an infrastructure approach to providing context information has advantages over a toolkit approach. While the Context Toolkit and Context Fabric offer developers much needed building blocks for constructing context-aware applications, even collecting information from distributed sets of sensors, these systems do not explicitly address the needs of applications in large scale ad hoc networks to dynamically discover and operate over a constantly changing context. The work presented in this paper satisfies the needs of applications to limit their operating contexts to a portion of the context data provided by low level sensors or even widget-like components.

The GAIA project [Roman et al. 2002] introduces the notion of *Active Spaces* as immersive computing environments for context-aware applications. In this infrastructure, users move from one Active Space to another, seamlessly extracting from one space and integrating into a new one. This work 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 in each Active Space. This type of approach does not map well to large-scale context-aware applications in completely wireless environments.

The CORTEX project [Verissimo et al. 2002] 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. The goals of the CORTEX system 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. In addition, the work presented in this paper focuses on generalized context provision, while CORTEX is tailored to concerns related to quality of service guarantees.

From this review, it becomes apparent that supporting software engineering of context-aware applications in large-scale ad hoc networks requires a redefinition of the notion of context. The key components of this new context definition are:

(1) The definition of context should be generalized so that applications interact with different types of context (e.g., location, bandwidth, etc.) in a similar manner.

(2) Different applications require contexts tailored to their individual needs.

(3) In an ad hoc network, an application's context includes information collected from a distributed network of sensors surrounding the application's host.

(4) Due to the large-scale nature of the application environment, applications require a decentralized solution to computing and operating over their needed contexts.

(5) Abstractions of context collection from sensors ease the programming burden developers experience.

The provision of such an abstraction of the operating context to application developers will significantly reduce the development burden. Instead of focusing on network communication programming, the application developer can interact with a context that satisfies its personalized needs.

Constructing and operating over this dynamic context in large-scale ad hoc networks requires protocols for supporting the definition of these contexts. Because this information must be gathered from the surrounding network, this protocol must coordinate with surrounding devices to provide the context-awareness applications need.

3.   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 ap-

plying 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 are likely never to 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 [Perkins and Bhagwat 1994] and Clusterhead Gateway Switch Routing [Chiang and Gerla 1997] mimic traditional routing protocols because they maintain consistent up-to-date information for routes to all other nodes in the network [Royer and Toh 1999]. This class of algorithms is based on modifications to the classical Bellman-Ford routing algorithm [Cheng et al. 1989]. Maintaining routes for every other node in the network can become quite costly. Performance comparisons [Broch et al. 1998] have shown that, while the overhead of DSDV is predictable, the protocol can be unreliable. Additionally, the overhead can be lessened by utilizing routing protocols from the second class, source initiated on-demand routing protocols. This type of 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 [Perkins and Royer 1999] builds on the DSDV algorithm but minimizes routing overhead by creating routes on demand. Dynamic Source Routing (DSR) [Johnson and Maltz 1996] requires that nodes maintain routes for source nodes of which they are aware in the system. Finally, the Temporally Ordered Routing Algorithm (TORA) [Park. and Corson 1998] uses the concept of 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 [Chen and Nahrstedt 1999]. 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 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.

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 [Chiang

et al. 1998; Gupta and Srimani 1999]. 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 [Bae et al. 2000; Madruga and Garcia-Luna-Aceves 1999] with improved reliability in a mobile environment. Both the tree-based and the mesh-based multicast protocols use a shared data structure approach. That is, they assume that all members in the group need/have the same view about the membership of the multicast group. However, in context-aware applications, each node may need an egocentric view regarding relevant context and in turn an egocentric view of the membership in its context group. As a result, the shared structure paradigm in conventional multicast protocols is not applicable for our context provision and context maintenance purposes.

In summary, while previous research in unicast and multicast in mobile ad hoc networks clearly points out many issues a context provision and maintenance protocol needs to address, such as the mobility of hosts, the transient nature of connections, and the changing properties of both the hosts and links in the network, existing protocols do not apply well to the context provision and maintenance problem. Our work aims to provide such a solution. The distinctive features of our solution from the communication standpoint include: (1) Property-based communication specification: instead of asking a host to explicitly name which other hosts it wishes to communicate with, the host specifies some properties of the path to the communicating parties; (2) context-aware network structure: any data structure built over the network must guarantee that the path used to communicate with a host in the context satisfies the contextual property specified; (3) good scalability: the context space is formed under relatively localized communication catering to the needs of a particular context-aware application thus avoiding a network-wide search.

4.   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.

4.1 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 connections, or 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, $p : v \to R$. 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, $\omega : E \to \Omega$. 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 5.

4.2 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:

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

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.

4.3 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 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 1

For the field research application scenario discussed in Section 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 5. Additional examples will also be presented.

4.4 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

Fig. 1.   The recursive cost function

is the shortest path. We call the cost of this path $g_\alpha(\beta)$. That is, for all paths from α to β,

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

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 2 shows the the shortest path tree built over an

Fig. 2.   The logical network and shortest path tree

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 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.

4.5   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 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.
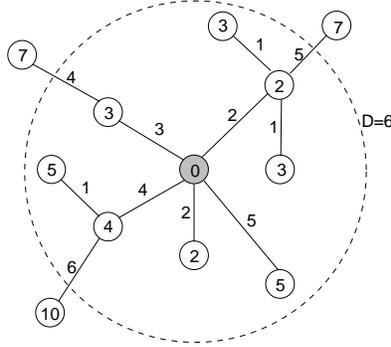
Fig. 3.   The bounded shortest path tree

Figure 3 shows the shortest path tree from Figure 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 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 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.

5.   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.

5.1 Building Floor Restriction

Application. 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

$$\rho_i = node\ floor\ \#$$

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.

5.2   Network Latency

Application 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

$$\omega_{ij} = link\ latency_{ij}$$

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:

$$\omega_{ij} = (link\ latency_{ij}, \frac{1}{bandwidth_{ij}})$$

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.

5.3   Physical Distance

Application. 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:

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

(a)   (b)   (c)

$$f_{v_0}(P_k) = \begin{cases} (|f_{v_0}(P_{k-1}).\mathbf{V} + m_{k-1,k}|, f_{v_0}(P_{k-1}).C, f_{v_0}(P_{k-1}).\mathbf{V} + m_{k-1,k}) \\ \quad \text{if } |f_{v_0}(P_{k-1}).\mathbf{V} + m_{k-1,k}| > f_{v_0}(P_{k-1}).maxD \\[2ex] (f_{v_0}(P_{k-1}).maxD, f_{v_0}(P_{k-1}).C + 1, f_{v_0}(P_{k-1}).\mathbf{V} + m_{k-1,k}) \\ \quad \text{otherwise} \end{cases}$$

(d)
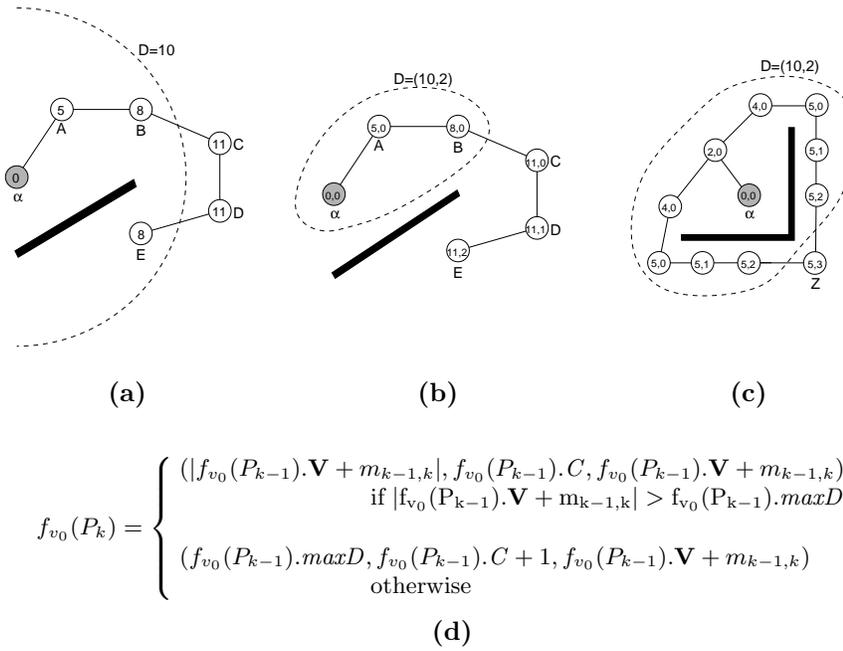
Fig. 4. (a) Physical distance only; (b) Physical distance with hop count, restricted due to distance; (c) Physical distance with hop count, restricted due to hop count; (d) The correct cost function

Figure 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 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 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 4b shows the same nodes as Figure 4a. In this figure, however, the cost function from Figure 4c assigns the path costs shown. The application specified bound shown in Figure 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 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.

6.   CONTEXT COMPUTATION AND MAINTENANCE

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 lost, an application on the reference node needs only to broadcast the message over the tree.

6.1   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 mechanisms and that this mechanism guarantees reliable delivery with the associated acknowledgements. These acknowledgements 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 [Huang et al.], 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.

6.2   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 simple 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.

Piggy-backed on this data message are the context specification and the information necessary for its computation.

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

Fig. 5.   The Components of a Query

Figure 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.

6.3   Protocol State Information

State Information

Figure 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 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.

```
┌──────────────────────────────────────────────────────────────────────────────┐
│ 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                                 │
│                                                                                │
└──────────────────────────────────────────────────────────────────────────────┘
```

Fig. 6.   State Variables

6.4   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. 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 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 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.

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

```
 ┌─────────────────────────────────────────────────────────────────────┐
 │ 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)                           │
 └─────────────────────────────────────────────────────────────────────┘
```

Fig. 7.   Context Computation

```
 ┌──────────────────────────────────────────────────────────────────────────────┐
 │ 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) ≥ 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 │
 └──────────────────────────────────────────────────────────────────────────────┘
```

Fig. 8.   Support Functions

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 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 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.

6.5 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 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 6 becomes clear only when we introduce tree maintenance to the protocol. Because the protocol in Figure 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 $(w_{id} := wnew_{id})$

Fig. 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 9 shows the same protocol presented in Figure 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 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 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.

A new action, CleanUpArrives has been added to the protocol shown in Figure 10 to deal with the arrival of the clean up messages. If the clean up message

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

Fig. 10.   Context Computation, Maintenance, and Clean Up

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 ∞.

6.6   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 11 shows 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 6, for each context request, a node in the network (other than the

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

Fig. 11.   The Components of a Reply

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 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 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 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 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 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

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

Fig. 12.   Complete Context Computation Protocol

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
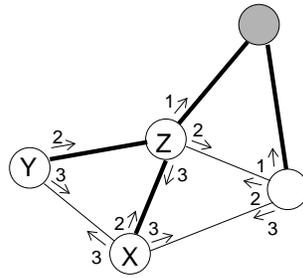
Fig. 13.   Mesh reply routing example

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 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.

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

Fig. 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 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.

---

[1]The nondeterministic selection of a host from the set C uses the nondeterministic assignment statement [Back and Sere 1990]. A statement $x := x'.Q$ assigns to x a value x' nondeterministically

```
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' ∈ C ∧ c'.cost + c'.weight < r.tokens ∧ c' ∉ r.path))¹
            Update the reply (r.tokens := r.tokens − c.weight, r.path.append(c))
            SendReply(r, c)
        end
```

Fig. 15.   Reply Propagation Over a Mesh

7.   PROTOCOL IMPLEMENTATION

The implementation of the protocol as described in Section 6 is written entirely in Java. This decision is driven by the fact that we aim to ease the software engineering burden associated with application development in ad hoc networks, which means placing control over the context in the hands of novice programmers. It is imperative that we provide a flexible protocol that an application developer can tailor to his needs. Applications must be able to define individualized distance metrics and add new environmental monitors to the system to increase the flexibility of link weight definition.

The implementation allows issuance of both one-time and persistent queries and maintains contexts which have persistent queries. Our system includes built-in metrics (e.g., hop count) but also provide a general framework for defining new metrics. Our implementation uses the support of two additional packages; one for neighbor discovery and one for environmental monitoring. We first describe these two packages before detailing the protocol implementation.

The architecture of a system using the Network Abstractions protocol is shown in Figure 16. The components in white we assume to exist, i.e., we assume an ad hoc physical network, and we assume a message passing mechanism for exchanging messages. The gray components, we provide as support packages to the Network Abstractions protocol implementation. We discuss these packages in more detail below. The designs of the APIs of these package attempt to present the flexibility

---

selected from among the values satisfying the predicate Q. If such an assignment is not possible, the statement aborts. In this case, the statement should always be possible, because, as long as our assumptions hold, our protocol guarantees that there is always at least one path back to the context initiator from any node within the context.
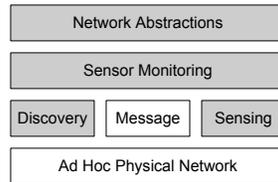
Fig. 16.   The System Architecture

of the context specification to the programmer while simplifying the programmer's task of defining and operating over his contexts. In all of the packages, the implementation is necessarily distributed; no host has global knowledge of the network.

7.1   Support Packages

7.1.1   *Neighbor Discovery.* 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. A node in the Network Abstractions protocol receives knowledge of its neighbors from a discovery service. This service uses a periodic beaconing mechanism that can be parameterized with policies for neighbor addition and removal (e.g., a neighbor is only added when its beacon has been heard for two consecutive beacon periods, and a neighbor is removed when it has not been heard from for 10 seconds).

7.1.2   *Environmental Monitoring.* Essential to adapting to context information is the ability to sense environmental changes. The Context Toolkit [Dey et al. 2001] 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. This sensor information is used to calculate the link weights needed in the Network Abstractions protocol.

Two components of the architecture shown in Figure 16 contribute to monitoring environmental conditions. First, the sensing component allows software on a host (in this case the sensor monitoring component) to interface with sensors connected to that host. For example, on a host with a GPS unit, this component makes the GPS information accessible to applications on the host. In general, a monitor contains it's 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.

Figure 17 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.

```
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);
  }
}
```

Fig. 17.   The GPSMonitor Class

The higher-level sensor monitoring component 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*. As described above, the former are created on a particular host to make the services available on that host accessible to applications. 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 handle to the proxy to the application. The application interacts with the remote monitor in the same manner as with a local monitor (e.g., by calling the `queryValue()` method or registering listeners for changes in the value).

7.2 Protocol Implementation

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 18.

---

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.

Fig. 18. The `Cost` interface

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 19.

```
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; }
}
```

Fig. 19. The `DistanceCost` Class

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 20. For an extending class, the tricky parts to adhering to

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.

Fig. 20.   The `Cost` interface

these requirements 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 21 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 22. 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.

```
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(d)
    DistanceCost newCost = ...
    ...
    return newCost;
  }
}
```

Fig. 21.   The `DistanceMetric` Class

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.

Fig. 22.   The Network Abstractions interface

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.

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 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 8 provides an empirical evaluation of this assumption.

7.3   Demonstration System

Figure 23 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 [Broch et al. 1998], 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.
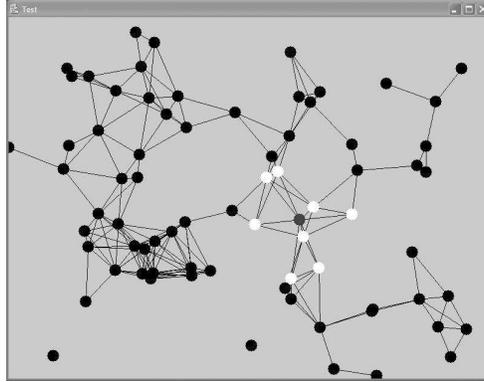
Fig. 23.   Screen capture of demonstration system

This gives us some intuition into what our goals should be before we start extensive simulation or implementation of a complex application.

7.4 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 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.

8.   ANALYSIS AND EXPERIMENTAL RESULTS

The previous sections have described a protocol for context-aware communication in ad hoc mobile networks. We have also discussed the implementation of this protocol and how application developers can use the implementation to create context-aware applications. In this section, we further motivate the use of this package by providing some performance measurements. Ideally a suite of such measurements will be used by application developers in determining which context definitions are appropriate for different needs or situations.

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

Fig. 24.   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.

8.0.1 *Simulation Settings.* We generated random 100 node ad hoc networks that use the random waypoint mobility model. The simulation is restricted to a $1000\text{x}1000m^2$ 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 24. While the random waypoint mobility model suffers from "density waves" as described in [Royer et al. 2001], 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 [Kleinrock and Silvester 1978], more recent work [Royer et al. 2001] 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 [IEEE Standards Department 1999] 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 [Yoon et al. 2003].

8.0.2 *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

---

[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 acknowledgements.

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.

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.

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 25. These results show a single context definition on our 100 node network. As for all of the results presented throughout this section, the x-axis shows the transmission range of the nodes in our simulated networks. As discussed above, this quantity is a measure of network density, which increases from left to right. 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 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
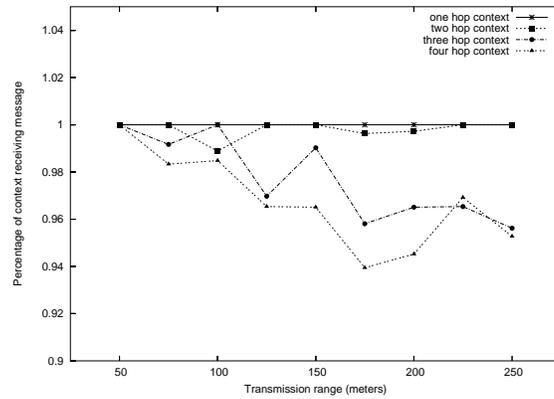
Fig. 25.   Percentage of context members receiving the message for contexts of varying sizes

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 26 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 $150m$
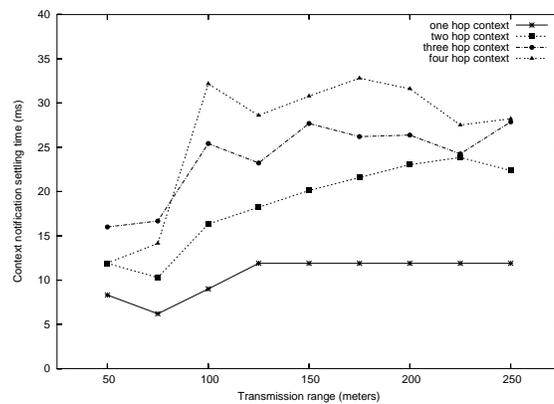
Fig. 26.   Settling time for contexts of varying sizes

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 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 27 shows the protocol's efficiency versus density for different sized contexts. First, notice that the efficiency for a one hop network is always 100% because only one broadcast (the initial one) is ever sent. For larger contexts, the efficiency is
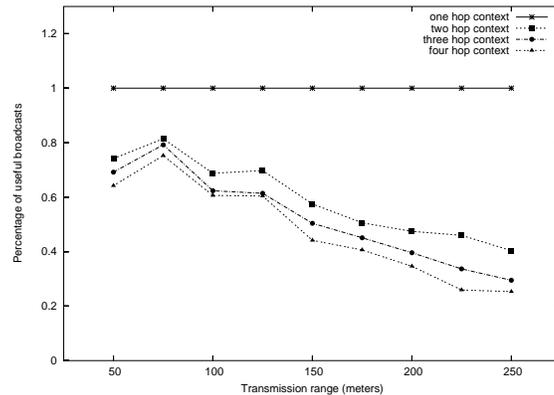
Fig. 27. Percentage of broadcasts that reached new context members for contexts of varying sizes

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 ($> 20\%$) 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 28 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 neighboring nodes are broadcasting), nodes are more likely to back off and
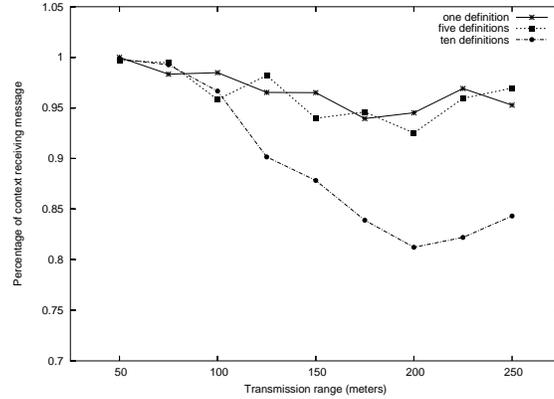
Fig. 28.   Percentage of context members receiving context messages for varying network loads

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. 29, 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.

We do not present any results for efficiency with changing network load, since network load seems to have no real effect on the percentage of useful broadcasts.

Changing Speed has Little 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. It is likely that, were we to analyze transmission of replies to queries, we would find that routes are somewhat less likely to hold up for scenarios with higher node speeds. Such concerns are addressed by the maintenance
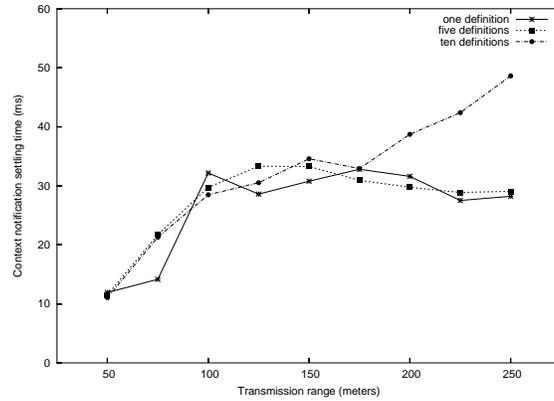
Fig. 29. Maximum time for last context recipient to receive notification for varying network loads
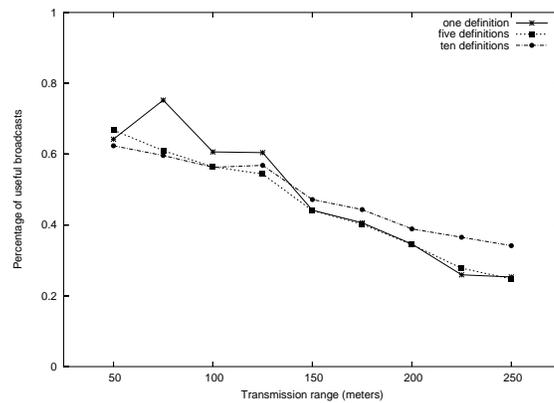
Fig. 30.　Percentage of broadcasts that reached new context members for varying network loads

protocol, and we would expect further simulation of this portion of the protocol to be more significantly impacted by changing speeds.

8.1　Discussion

To ensure application-level data consistency, we make assumptions about the atomicity of network topology changes and their propagation through the network for rebuilding the context. Future work will explore ways to relax these assumptions by weakening the required guarantees on both context maintenance and the operations performed on that context. For example, with some knowledge about the system (e.g., radio transmission range, maximum node speed, etc.) [Roman et al.

2001] shows that a node can predict a "safe distance" for a link. Incorporating a similar idea may allow us to redefine applications' contexts on the fly to essentially replace a context specification like "all nodes within two hops" with one like "all nodes guaranteed to remain within two hops for $20ms$".

Several of our results show that increased network congestion negatively affects our protocol. Specifically, Figure 25 showed that the consistency of context dissemination decreases as more neighboring hosts rebroadcast, and Figure 27 showed that the efficiency of the broadcast mechanism decreased with increasing density. This results from a commonly known problem called a "broadcast storm". [Ni et al. 1999] describes this problem in mobile ad hoc networks and quantifies the additional coverage a broadcast gains. Several alternative broadcasting mechanisms have been proposed, many of which are compared in [Williams and Camp 2002]. Such methods include using probabilistic methods or knowledge about the environment or neighbor set to determine when to rebroadcast. Integrating these or similar intelligent broadcast mechanisms may increase the resulting consistency and efficiency of context notification. Figure 28 showed that the consistency of context notification tends to fall off when network load increases. Future work includes investigating ways to handle this undesirable effect. This could include reusing information available about already constructed contexts to limit the amount of work required to construct another context for a new node. Also, one-time context distributions may be able to use information stored on nodes servicing persistent queries over maintained contexts.

9. CONCLUSIONS

The ideas behind this work are rooted in the notion that software engineering in ad hoc networks could be simplified if the maintenance of contextual information were to be delegated to a support infrastructure without loss of flexibility and generality. This paper demonstrates the feasibility of such an approach and outlines a novel technical solution for context specification, coupled with a presentation of our implementation of the protocol to compute the context. The notion of context is broadened to include, in principle, the entire ad hoc network, yet it can be conveniently limited in scope to a neighborhood whose size and scope is determined by the specific needs of each application as it changes over time.

This work implements and analyzes a protocol for providing contexts in mobile ad hoc networks. The protocol provides a flexible interface that gives the application explicit control over the expense of its operation while maintaining ease of programming by making the definition of sophisticated contexts simple. This protocol generalizes the notion of distance to account for any property, allowing an application to adjust its context definitions to account for its instantaneous needs or environment. Most importantly, the protocol explicitly bounds the computation of the application's context to exactly what the application needs. In general, these interactions will be localized in the neighborhood surrounding the host of interest, and therefore the host's operations do not affect distant nodes. This bounding allows an application to tailor its context definitions based on its needed guarantees The implementation presented here is currently in use by the EgoSpaces middleware system. This will provide further evaluation and feedback for protocol refinement.

We also presented an initial analysis of our protocol over a variety of networks and situations, showing that it is practical in reasonable situations.

REFERENCES

ABOWD, G., ATKESON, C., HONG, J., LONG, S., KOOPER, R., AND PINKERTON, M. 1997. Cyberguide: A mobile context-aware tour guide. *ACM Wireless Networks 3*, 421–433.

BACK, R. J. R. AND SERE, K. 1990. Stepwise refinement of parallel algorithms. *Science of Computer Programming 13*, 2–3, 133–180.

BAE, S., LEE, S.-J., SU, W., AND GERLA, M. 2000. 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 (January/February), 70–77.

BROCH, J., MALTZ, D., JOHNSON, D., HU, Y.-C., AND JETCHEVA, J. 1998. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the ACM/IEEE MobiCom*. 85–97.

CHEN, S. AND NAHRSTEDT, K. 1999. Distributed quality-of-service routing in ad-hoc networks. *IEEE Journal on Selected Areas in Communications 17,* 8 (August), 2580–2592.

CHENG, C., RILEY, R., AND KUMAR, S. 1989. A loop-free extended Bellman-Ford routing protocol without bouncing effect. In *Proceedings of the ACM SIGCOMM*. 224–236.

CHEVERST, K., DAVIES, N., MITCHELL, K., FRIDAY, A., AND EFSTRATIOU, C. 2000. Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In *Proceedings of MobiCom*. ACM Press, 20–31.

CHIANG, C. AND GERLA, M. 1997. Routing and multicast in multihop, mobile wireless networks. In *Proceedings of IEEE International Conference on Universal Personal Communications*. 546–551.

CHIANG, C., GERLA, M., AND ZHANG, L. 1998. Adaptive shared tree multicast in mobile wireless networks. In *Proceedings of GLOBECOM '98*. 1817–1822.

DEY, A. K., SALBER, D., AND ABOWD, G. D. 2001. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human Computer Interaction 16,* 2–4, 97–166.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns*. Addison-Wesley.

GUPTA, S. AND SRIMANI, P. 1999. An adaptive protocol for reliable multicast in mobile multi-hop radio networks. In *IEEE Workshop on Mobile Computing Systems and Applications*. 111–122.

HARTER, A., HOPPER, A., STEGGLES, P., WARD, A., AND WEBSTER, P. 2002. The anatomy of a context-aware application. *Mobile Networks 8,* 2/3, 187–197.

HONG, J. AND LANDAY, J. 2001. An infrastructure approach to context-aware computing. *Human Computer Interaction 16*.

HUANG, Q., JULIEN, C., AND ROMAN, G.-C. Relying on safe distance to achieve strong partitionable group membership in ad hoc networks. *IEEE Transactions on Mobile Computing*. (to appear).

IEEE STANDARDS DEPARTMENT. 1999. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE standard 802.11-1999.

JOHNSON, D. AND MALTZ, D. 1996. Dynamic Source Routing in ad hoc wireless networks. In *Mobile Computing*, Imielinski and Korth, Eds. Vol. 353. Kluwer Academic Publishers.

KLEINROCK, L. AND SILVESTER, J. 1978. Optimum transmission radii in packet radio networks or why six is a magic number. In *Proc. of the IEEE Nat'l. Telecommunications Conf.* 4.3.1–4.3.5.

MADRUGA, E. AND GARCIA-LUNA-ACEVES, J. 1999. Scalable multicasting: The core assisted mesh protocol. *ACM/Baltzer Mobile Networks and Applications, Special Issue on Management of Mobility*.

NI, S.-Y., TSENG, Y.-C., CHEN, Y.-S., AND SHEU, J.-P. 1999. The broadcast storm problem in a mobile ad hoc network. In *Proc. of MobiCom*. 151–162.

PARK., V. AND CORSON, M. S. 1998. Temporally-ordered routing algorithm (TORA) version 1: functional specification. Internet Draft. IETF Mobile Ad Hoc Networking Working Group.

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

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

Perkins, C. and Royer, E. 1999. Ad hoc on-demand distance vector routing. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*. 90–100.

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

Roman, G.-C., Huang, Q., and Hazemi, A. 2001. Consistent group membership in ad hoc networks. In *Proceedings of the 23rd International Conference on Software Engineering*.

Roman, M., Hess, C., Cerqueira, R., Ranganat, A., Campbell, R., and Nahrstedt, K. 2002. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing 1,* 4, 74–83.

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

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

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

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

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

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

Williams, B. and Camp, T. 2002. Comparison of broadcasting techniques for mobile ad hoc networks. In *Proc. of MobiHoc*. 194–205.

Yoon, J., Liu, M., and Noble, B. 2003. Random waypoint considered harmful. In *Proc. of INFOCOM*.