Report Number: WUCSE-2003-13

2003-03-17

Declarative and Dynamic Context Specification Supporting Mobile Computing in Ad Hoc Networks

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. Previous work along these lines presumed a rather narrow definition of context that centered on resources immediately available to the component in question, e.g., communication bandwidth, physical location, etc. This paper explores context-aware computing in the setting of ad hoc networks consisting of numerous mobile hosts interacting with each other opportunistically via transient wireless interconnections. We extend a component's context to encompass awareness of a neighborhood within the ad hoc network.... **Read complete abstract on page 2.**

Declarative and Dynamic Context Specification Supporting Mobile Computing in Ad Hoc Networks

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. Previous work along these lines presumed a rather narrow definition of context that centered on resources immediately available to the component in question, e.g., communication bandwidth, physical location, etc. This paper explores context-aware computing in the setting of ad hoc networks consisting of numerous mobile hosts interacting with each other opportunistically via transient wireless interconnections. We extend a component's context to encompass awareness of a neighborhood within the ad hoc network. A formal abstract characterization of this new perspective is proposed. The result is a specification method and associated context maintenance protocol. The former enables an application to define an individualized context that extends across multiple mobile hosts in the ad hoc network. The latter delegates the continuous evaluation of the context and the performance of operations on it to some middleware operating below the application level. This relieves application development of the obligation of explicitly managing mobility and its implications on the component's behavior. While a basic approach to the specification and protocol involves constructing a tree structure on top of the dynamically changing network, a more robust approach involves a mesh construction; we explore both approaches. We also present an initial implementation of the protocol.

2003-13

Declarative and Dynamic Context Specification Supporting Mobile Computing in Ad Hoc Networks **PLEASE SEE WUCSE-2004-37**

Authors: Julien, Christine; Roman, Gruia-Catalin; Huang, Qingfeng

March 17, 2003

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. Previous work along these lines presumed a rather narrow definition of context that centered on resources immediately available to the component in question, e.g., communication bandwidth, physical location, etc. This paper explores context-aware computing in the setting of ad hoc networks consisting of numerous mobile hosts interacting with each other opportunistically via transient wireless interconnections. We extend a component's context to encompass awareness of a neighborhood within the ad hoc network. A formal abstract characterization of this new perspective is proposed. The result is a specification method and associated context maintenance protocol. The former enables an application to define an individualized context that extends across multiple mobile hosts i in the ad hoc network. The latter delegates the continuous evaluation of the context and the performance of operations on it to some middleware operating below the application level. This relieves application development of the obligation of explicitly managing mobility and its implications on the component's behavior. While a basic approach to the specification and protocol involves constructing a tree structure on top of the dynamically changing network, a more robust approach involves a mesh construction; we explore both approaches. We also present an initial implementation of the protocol.

Declarative and Dynamic Context Specification Supporting Mobile Computing in Ad Hoc Networks

CHRISTINE JULIEN, GRUIA-CATALIN ROMAN, QINGFENG HUANG
Washington University

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. Previous work along these lines presumed a rather narrow definition of context that centered on resources immediately available to the component in question, e.g., communication bandwidth, physical location, etc. This paper explores context-aware computing in the setting of ad hoc networks consisting of numerous mobile hosts interacting with each other opportunistically via transient wireless interconnections. We extend a component's context to encompass awareness of a neighborhood within the ad hoc network. A formal abstract characterization of this new perspective is proposed. The result is a specification method and associated context maintenance protocol. The former enables an application to define an individualized context that extends across multiple mobile hosts in the ad hoc network. The latter delegates the continuous evaluation of the context and the performance of operations on it to some middleware operating below the application level. This relieves application development of the obligation of explicitly managing mobility and its implications on the component's behavior. While a basic approach to the specification and protocol involves constructing a tree structure on top of the dynamically changing network, a more robust approach involves a mesh construction; we explore both approaches. We also present an initial implementation of the protocol.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms: Design, Reliability, Algorithms

Additional Key Words and Phrases: Mobility, Context-Aware, Ad Hoc Network, Scalability

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 the host's context. In ad hoc networks especially, software must adapt its behavior continuously in response to this changing context. In this environment, devices are commonly small and

Authors' address: Department of Computer Science and Engineering, Washington University, Saint Louis, MO 63130.
Authors' emails: {julien, roman, qingfeng}@cse.wustl.edu

constrained, and therefore they must often rely on other connected devices for information and computations.

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 placed in a building to monitor movement of the users and forward telephone calls to them. Xerox PARC's PARCTab system [Want et al. 1995] 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 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.

In the later 90's, generalized software built to support the development of context-aware applications began 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 uses an object oriented approach to separate the sensing, gathering, and interpretation of contextual information for each user. The Context Fabric provides a service infrastructure that focuses on decoupling context services from the chosen hardware, operating system, and programming language.

Much of the work to date [Chen and Kotz 2000] restricts its use of contextual information to only information that can be monitored directly by the host running the software. Recent work has targeted publish/subscribe systems for the ad hoc environment, specifically addressing implementation concerns of reconfiguration algorithms much needed in the highly dynamic ad hoc environment. One can easily imagine a situation where a mobile user has an interest not only in contextual information collected by his mobile unit, but also by other units, even units that are not directly connected. Because of this, mobile units in the network become part of other hosts' contexts. Additionally, the type of contextual information available to users and applications has been of limited types. For example, the guide tools define context strictly as the user's location, and most context-aware systems are built upon similarly rigid structures. To support more general context-aware applications, we will allow them to define individualized contexts; such definitions may extend beyond our current vision of context and may need to include a rich amalgamation of facets of the environment. An application in such an environment should be permitted to supply a definition of its desired context; subsequent operations issued by the application would be performed only over the subnet specified by the application. This requires mechanisms for computing an application-defined context that may include distant hosts reachable only indirectly through other hosts in the network.

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 to the information in 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's observations of his own target subjects. The context defined in this case will 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 mobile surveillance applications. Throughout the remainder of this paper, we will revisit this example and indicate how this application relates to individual pieces of the work.

Our work starts from the premise that development of mobile applications can be simplified by allowing developers to specify a context specific to their application needs and by adopting a notion of context that extends to an entire reachable set of neighboring hosts whose size and shape is under the direct control of the application. 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 an 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. Second is the issue 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 given context, as required by an application on a particular mobile host. We extend this protocol to one that uses a mesh over the ad hoc network. Context sensitive operations are carried out through a cooperative effort involving only hosts that are part of a given context, regardless of its structure.

The paper is organized as follows. Section 2 provides a more detailed problem definition. Section 3 discusses the abstractions required to create a context specification. Sample metrics are presented in Section 4. Section 5 presents a protocol that computes and maintains a specified context as a tree over a subnet of the ad hoc network. Section 6 extends this protocol to a mesh structure. In Section 7, we present our implementation. Section 9 provides some discussion of the protocol followed by conclusions in Section 10.

2. PROBLEM DEFINITION

Ad hoc mobile networks may 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. A major difference between our approach and previous work in context-aware computing is the breadth of our definition of context. Our goals include broadening the context available to a host to include not only those properties that can be measured directly by a host, but also properties of other reachable hosts and properties of links among them. This approach, however, has the potential to greatly increase the amount of contextual information available, and therefore an application running on a host should specify the precise context that interests it based on the properties of hosts and links in the network. The application should also specify a bound that defines the size of the context it chooses to operate in. 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 only in gas stations within five miles. 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.

In summary, we want to 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). In most cases these properties will not be only those of the raw hardware but also properties associated with hosts or links by virtue of the applications they support.

3. CONTEXT SPECIFICATION

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 first must 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. A simple example of a weight is for each link to have a weight of one. This will allow us to count the number of network hops between nodes in the network.

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. Continuing the network hop count example, the cost function specified by the application would be the sum of the weights of the links along a path. Because the weight of each link is one, the number of hops from the source of the path to a node determines the cost at that node. In a real network, however, multiple paths may exist between two given nodes. Therefore we will 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. Section 6 extends this concept to build a mesh over the subnet that may offer multiple satisfactory paths to nodes in the subnet.

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. For the hop count example, an entire context specification might be written as: all nodes which can be reached in fewer than five hops.

We will start with this bound on the context specification and work backwards, dissecting the tree built to see from where each step derives. We will 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. At each point, we will also explain how the specification applies to the application scenario introduced previously.

3.1 Ensuring Boundedness

We will see later how an ad hoc network can be represented as a graph, $G = (V, E)$ with weighted edges. We will create a tree rooted at a reference node that includes only the shortest paths from the reference node to each other reachable node in the network. Given this tree representation and the shortest paths, we can define a bound on the 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.

Again, in the network hop count example, hosts that are five or more hops away are not included. In the case of a field researcher needing to utilize another researcher's video information, the context will 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 minimum end-to-end requirement will be included in the context. The next section explores this bound in more detail.

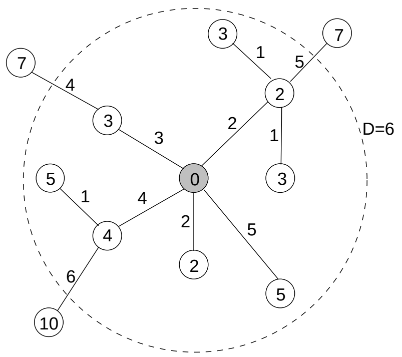


Fig. 1. The bounded shortest path tree

Figure 1 shows a tree rooted at the shaded reference node, α. The weights on the links can be used to compute the cost of a given path, shown inside each node. Mechanisms for assigning these weights and calculating the cost of the paths will be discussed in later sections. This particular example shows a shortest path tree whose cost function simply sums the weights on a path. Figure 1 also shows the bound, D, indicated by the dashed line. 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 of node i as ν_i, then $\nu_i > \nu_{i-1}$. 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 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 1, 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.

Unfortunately, an ad hoc network does not look like a shortest path tree with the cost of each path labeled on the node. In the following section, we will show how to build the shortest path tree, given the cost of individual paths. Then we will discuss how to calculate the cost of a given path between two nodes in a graph using an application specified cost function. Finally, we will introduce the network abstraction that allows us to represent contextual information from the ad hoc network as weights on edges in this graph.

3.2 The Minimum Cost Path

In the next section, we will see that, given an application specified cost function, we can determine the cost of a path between two given nodes as a function of quantified properties of links and nodes. The calculation of the cost of a path P originating at the reference node α is represented as $f_\alpha(P)$. In an arbitrary graph, however, multiple paths may exist from α to another node β each with an associated cost. For each of these nodes, β, reachable from α, one of these paths is the shortest path. We call the cost of this path $g_\alpha(\beta)$. That is, for all paths, P 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 $g_\alpha(\beta)$.

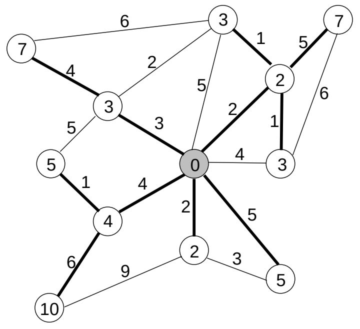


Fig. 2. The logical network and shortest path tree

Figure 2 shows the same shortest path tree as Figure 1. This time, however, the bound is omitted, and the graph from which the tree was generated is shown in its entirety. The set of nodes is identical to that in Figure 1, but now more links and their weights are included. The links from Figure 1 are darkened; these are the links from the graph that make up the shortest path tree. Though the graph contains multiple paths from the reference node to each other node, the tree includes only one of the shortest paths to each node.

3.3 The Path Cost Function

The previous sections show that if we can define the cost of every path in a graph, we can compute the shortest path tree. We can then add a bound on the path cost to generate a set of acquaintances for a reference node. Given a logical view of an ad hoc network $\overline{G} = (\overline{V}, \overline{E})$ 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:

$$f_{v_0}(P_k) = Cost(f_{v_0}(P_{k-1}), w_{k-1,k}) \tag{1}$$

$$f_{v_0}(\langle \overline{v_0} \rangle) = 0 \tag{2}$$

Figure 3 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 $\overline{v_{i-1}}$.

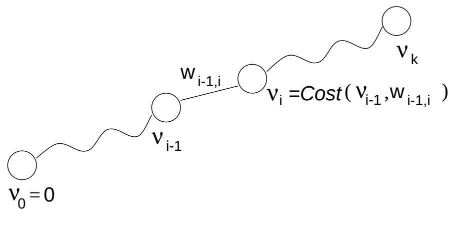


Fig. 3. The recursive cost function

We first revisit the network hop count example. Assuming the weight of a link is one, this example intends that the cost of a path be the number of hops along that path. Therefore, the associated cost function should be additive. In this case,

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

For our field research application scenario, assume the weight of each link in the network is a combination of the total latency incurred in traversing the link and the inverse of the bandwidth of the link. In this case, the cost function is additive with respect to the latency, but maximizing with respect to the inverse of the bandwidth. The entire cost function and its reasoning are presented in Section 4. Additional examples will also be presented.

The cost of the path from the reference node to each reachable node can be defined using a path cost function. In a later section, we will see what the weights on the edges mean and how they are derived from the properties of the ad hoc network. Before we can define weights, however, we need to map the physical ad hoc network to an abstract space, a graph. The weights will then allow us to quantify the reference node's context, giving us the logical network, $\overline{G}$, over which the cost function has been defined.

3.4 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 intend to fold into a single weight on each edge in our abstract graph. To do this, we map the host and link attributes to the abstract space represented by the graph G, by placing values on every vertex and edge. First, we quantify the properties of a mobile host as a value, ρ_i on the vertex $v_i \in V$ representing the mobile host in the graph. Formally, $\rho : V \rightarrow R$. The value of ρ_i can be a combination of a host's battery power, location, load, service availability, etc.

Second, we quantify the properties of a network link as a value, ω_{ij} on the edge $e_{ij} \in E$ representing the edge in the graph. Formally, $\omega : E \rightarrow \Omega$. The value of ω_{ij} can be a combination of the link's length, throughput, etc. Some examples of these weights will be given in Section 4.

3.5 Logical View of the 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. Each interpretation of the properties of the underlying physical network represents a logical view from the perspective of the corresponding application. We designate an application's logical network $\overline{G} = (\overline{V}, \overline{E})$, formed from the original mapping, G. This is the logical network on which the cost function was built previously. We use the information about node and link properties to create a topological *distance* between each pair of connected nodes in the logical network, $\overline{G}$. To do this, we combine 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 w_{ij} on the edge $\overline{e_{ij}} \in \overline{E}$ in the logical network $\overline{G}$. No host $\overline{v_i} \in \overline{V}$ in the logical network has a weight. Formally, this projection from the physical world to the virtual one can be represented as:

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

or more specifically,

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

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

As an illustration of a weight assignment, we will revisit the network hop count example. As discussed, the weight in this example can be measured simply by placing a weight of one on every edge. More formally, let ω_{ij} be one for every $e_{ij} \in E$ in the original graph. The value of ρ_i will not be used for this particular metric. Then the weight, w_{ij}, of an edge, $\overline{e_{ij}} \in \overline{E}$ in the logical network is: $w_{ij} = \omega_{ij} = 1$.

In our application scenario, each link in the logical graph should be assigned a weight defined by a combination of the total latency experienced by traffic on that link. This weight assignment is detailed in the next section.

4. SAMPLE METRICS

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

4.1 Building Floor Restriction

Application. We first introduce a simple metric that constructs a context based on the floor locations of sensors in a building. Imagine a building with a fixed infrastructure of sensors and information appliances providing contextual information. Sensors provide information regarding the building's structural integrity, the frequency of sounds, the movement of occupants, etc. Engineers and inspectors carry PDAs or laptops that provide additional context and assimilate context information. Different people have specific tasks and will therefore use information from different sensors. As an engineer moves through the building, he wishes to see structural information not for the whole building, but only for his current floor and the floors adjacent to it.

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

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

so that the value of ρ corresponds to the integer floor number where the node is located. We do not use the link weight, ω, in this case. This definition uses higher level information than the definitions for the previous metrics. To map these values to logical weights, we simply combine the floors of nodes i and j so that w_{ij} consists of the range of floors of the two nodes

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

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 also add a hop count parameter to the cost of a path at each node. 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, described previously, counts the number of hops taken in the current range of floors.

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

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

For ranges, we use the notation $\in$ to refer to the fact that the range on the left is entirely contained in the range on the right. 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, since the range has not changed. 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 has to specify the acceptable range of floors and a hop count. For the example introduced in this section, the building engineer might define a bound something like:

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

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

4.2 Network Latency

Application Next we begin developing a metric for the application scenario introduced at the beginning of this paper. Briefly, this application consists of field researchers who wish to share sensor data and video feeds. It is very 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 will build 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. To accomplish the former, we define

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

where *node packet processing latency*$_i$ is defined as 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 because using the whole number would count the node's latency twice if the node is in the middle of the path. This latency value will suffice under the assumption that the incoming latency for the node is approximately equivalent to the node's outgoing latency. We define

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

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

Possible mappings to the logical network in this case abound; the link latency and node latency can each be given a different importance by multiplying the ρ and ω values as they are listed above. For simplicity's sake, the value w_{ij} for a link $\overline{e_{ij}}$ in the logical network is defined as

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

The cost function is then:

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

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 depends on the bandwidth in addition to the network latency, we now 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:

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

where $\mathit{bandwidth}_{ij}$ is the bandwidth on the link between nodes i and j. 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".

Again, the mapping to the logical network can take many forms. We continue with a simple mapping, where the value of w_{ij} for a link $\overline{e_{ij}}$ in the logical network is defined as a pair of values:

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

To access the first and second components of w_{ij}, we use the same notation as above (e.g., $w_{ij}[0]$ 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:

$$\nu = (\mathit{latency}, \mathit{bandwidth})$$

The cost function is then:

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

We use a maximum function to compute the minimum bandwidth encountered to account for the fact that the bandwidth component of the weight is the inverse of the link's bandwidth.

Notice that this cost function is guaranteed to increase at every hop. Because the latency is completely additive, the *latency* component increases every hop. Additionally, because we take the maximum of the *bandwidth* component each hop, it is guaranteed not to decrease.

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

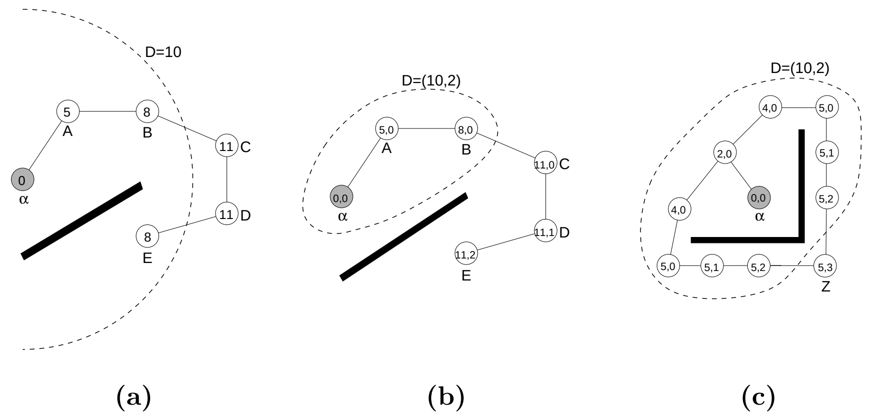
4.3 Physical Distance

Application. Finally, we present a general-purpose metric based on physical distance that will prove useful in a wide variety of application situations. 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 it wants to operate over the information that will affect its immediate trip. Therefore, this data should be restricted to information within a certain physical distance (e.g., within a mile), and as the car moves, the data contained within that area changes. A number of other everyday applications as well as military applications could 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. For this example, the weight placed on edges in the logical network reflects the distance vector between two connected nodes. That is, given two connected nodes, the weight on $\overline{e_{ij}}$ connecting them accounts for both the displacement and the direction of the displacement between the two nodes. Formally,

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

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.



$$f_{v_0}(P_k) = \begin{cases} (|f_{v_0}(P_{k-1}).\mathbf{V} + w_{k-1,k}|, f_{v_0}(P_{k-1}).C, f_{v_0}(P_{k-1}).\mathbf{V} + w_{k-1,k}) \\ \qquad\qquad \text{if } |\mathrm{f}_{\mathrm{v}_0}(\mathrm{P}_{\mathrm{k}-1}).\mathbf{V} + \mathrm{w}_{\mathrm{k}-1,\mathrm{k}}| > \mathrm{f}_{\mathrm{v}_0}(\mathrm{P}_{\mathrm{k}-1}).\mathit{maxD} \\ \\ (f_{v_0}(P_{k-1}).\mathit{maxD}, f_{v_0}(P_{k-1}).C + 1, f_{v_0}(P_{k-1}).\mathbf{V} + w_{k-1,k}) \\ \qquad\qquad \text{otherwise} \end{cases}$$

(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

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

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

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. We show below how this vector is calculated; it is used to keep track of the path's location relative to the reference host. It differs from the first two components of the cost function in that it is used simply for bookkeeping.

Specifying a bound for this cost function requires specifying a bound on both maxD and C. It is important that we also define the comparison function for this metric. A given bound has two values, and if the components of a host's ν values meet or exceed either of these values, the host is outside the bound. That is, 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. This observation allows us to treat this cost function as 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 of the number of hops for which the maximum distance did not change (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. The values shown on the nodes in the figure reflect the pair, $maxD$ and C. 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.

5. CONTEXT COMPUTATION

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 reference host α does not necessarily need to know which other hosts are part of its acquaintance list. Instead, the application needs to be guaranteed both 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 its nature, this tree defines a single route from the reference node to each other node in the acquaintance list. To send a message to the members of the acquaintance list, an application on the reference node needs only to broadcast the message over the tree.

5.1 Assumptions

The protocol presented next relies on a few assumptions regarding the behavior of the underlying system. First, it assumes that there exists a message passing mechanism for its use and that this mechanism guarantees reliable delivery with the associated acknowledgments. These acknowledgments therefore lie outside the concern of the protocol. 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 links in the network by updating these weights in response to changes in the contextual information specified by the application. Additionally, we assume that the system has calculated 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.

5.2 Protocol

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

State Information

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

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.

Tree 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.

```
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 $Cost(d, w_c) \geq D$ by calling SENDCLEANUP to c
PropagateCleanUps	–for every member of I, send a clean up message by calling SENDCLEANUP
PropagateReply(r)	–send the reply to p
AppProcessReply(r)	–application processing of the data message part of the reply

Fig. 8. Support Functions

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

Tree 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.

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

```
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

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.

```
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

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 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 ∞.

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.

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

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 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.

6. MESH-BASED EXTENSION

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.

```
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

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

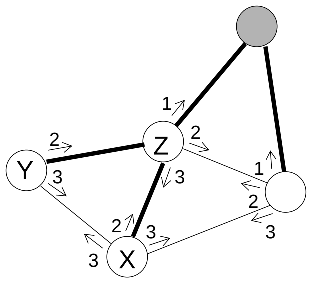


Fig. 13. Mesh reply routing example

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 for routing reply messages if their costs qualify. In this figure, the numbers on the arrows along the links refer to the shortest possible path from that link back to the reference host. Consider the host labeled X sending a reply. The host first packages the reply with a bag of three tokens (because three is the context's bound). At this point, host X can send this message to any of its neighbors because all of the paths are qualifying. Let's say X chooses host Y. Host X first updates the bag of tokens by subtracting one (the cost of every link in our example is one) and then sends the reply to Y. Y has only one choice of path to send the reply along because the message is not sent back to any previous node on its path. This prevents reply messages from cycling unnecessarily in the network. Y therefore updates the bag of tokens by subtracting one and sends the reply to Z. When the message reaches Z, the bag contains only a single token. This forces Z to consume the last token and route the reply along the direct link to the reference host. Figure 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.

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

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

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.

```
Actions

QUERYARRIVES($q$)
    ... as before

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

CLEANUPARRIVES($id$)
    ... as before

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

Fig. 15. Reply Propagation Over a Mesh

[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 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.

7. IMPLEMENTATION AND EVALUATION

The implementation of the protocol as described in Section 5 is written entirely in Java. The design of the API of the package attempts to present the flexibility of the context specification to the programmer while simplifying the programmer's task of defining and operating over his contexts. The core of the algorithm described in this paper resides in a package called `NetworkAbstractions`, and it relies on two supporting packages, `NetworkDiscovery` and `Monitor`. In all of the packages, the implementation is necessarily distributed; no host has global knowledge of the network.

7.1 The `NetworkDiscovery` Package

This package implements a basic discovery mechanism and keeps any registered listeners aware of the comings and goings of other hosts in the neighborhood. To accomplish this, the package sends periodic beacons to announce this host's presence. Additionally, the package listens for beacons sent by neighboring nodes, and, when it determines a new node has arrived, the package notifies the listeners that have registered to receive such an event.

The main component seen to a user of the `NetworkDiscovery` package is the `DiscoveryServer`, which mediates both the sending and receiving of beacons. Upon creating and initializing a `DiscoveryServer`, a user of the package (in our case, the user is the `NetworkAbstractions` package) specifies a policy to be used to determine when to "connect to" and "disconnect from" neighbors. For example, the user may want to specify that a neighbor is connected when the package receives two consecutive beacons from the neighbor and that a neighbor is disconnected if its beacon has not been received for two consecutive periods. For simplicity's sake, this policy is defined by two parameters; the first specifies the number of beacons that must be received from a host for it to be considered a connected neighbor, and the second specifies the time (in milliseconds) for which the package may not receive a beacon before a neighbor is considered disconnected. Once the server is created, the user can start and stop it using the `start()` and `stop()` methods. To receive events generated when a neighbor is added or removed, the user must define a class that implements the `DiscoveryListener` interface and must provide definitions for the `neighborAdded()` and `neighborRemoved()`. The code in these method definitions specifies the actions the user wishes to take when a neighbor is added or removed, respectively. This basically forces a rediscovery of the specified neighbor. Figure 16 shows the API of the `NetworkDiscovery` package.

7.2 The `Monitor` Package

This package provides a unified interface to different sensors and monitors connected to a host in the network. Examples of monitors include thermometers to measure temperature; GPS units to measure position, speed, or time; and a sensor that measures a link's bandwidth. For the monitor package to recognize a physical monitor that is added to the system, the user must extend the `AbstractMonitor` class and implement the methods required of a logical monitor. The user's implementation of this abstract class fills in the behavior specific to the monitor which includes the protocol for communicating with the physical monitor to acquire the

<u>DiscoveryServer</u>

DiscoveryServer(int period, InetAddress mcast, int port, int required, long disconnect)
— initializes a new `DiscoveryServer` that sends beacons every period to the multicast address defined by mcast and port a host is only added if required beacons have been received, and is removed if no beacon is received for disconnectTime

addDiscoveryListener(DiscoveryListener dl)
— adds a new component to be notified when neighbors are added or removed

removeDiscoveryListener(DiscoveryListener dl)
— remove the specified listener

start()
— starts the threads that make up the `DiscoveryServer`

stop()
— stops the threads that make up the `DiscoveryServer`

<u>DiscoveryListener</u>

discoveryEventReceived(DiscoveryEvent de)
— called to notify the application of an event's arrival; performs application level actions

Fig. 16. Network Discovery API

sensed values. A monitor is identified by a unique string that is known to any user of the monitor; it is the user's responsibility to guarantee this uniqueness.

To use this package, a user first creates a `MonitorRegistry`. Then, for all of the physical monitors on the system, the user creates an extension of the `AbstractMonitor` class specific to the sensor and adds the monitor to the `MonitorRegistry`. Any other application that can access the `MonitorRegistry` can retrieve the monitor and query for its value. In addition to offering one-time queries on the monitors, the package also provides an interface for registering listeners. Given a monitor retrieved from the `MonitorRegistry`, a user can implement the `MonitorListener` interface and then add the listener to the monitor. Whenever the monitor's value changes, it will generate an event, and this event will be passed to the user's registered listeners.

The `Monitor` package also provides a facility for distributed, or remote sensing. This allows a user on a host to receive information from monitors residing on directly connected hosts. This behavior occurs through the use of an additional class, the `RemoteMonitor` class, which also extends the `AbstractMonitor` class and provides the same interface as any other monitor. A user constructs a `RemoteMonitor` by providing the IP address of the host on which to sense (this must be a directly connected neighbor) and the unique string that identifies the desired sensor on the remote host. After creating such a `RemoteMonitor`, the user can place it in the `MonitorRegistry` and interact with it as if it were local. The provided implementations internal to the concrete `RemoteMonitor` class perform the work of connecting to the neighboring host and calling the appropriate methods on the local monitor on that machine. The user is responsible for maintaining these `RemoteMonitors`; when the referenced host becomes disconnected, the user must ensure that the `RemoteMonitor` is removed from the `MonitorRegistry`. The user can monitor the connections to its neighbors through the previously described ne`NetworkDiscovery` package. Figure 17 shows the API for the `Monitor` package.

<u>MonitorRegistry</u>

MonitorRegistry()
— constructs a registry ready to hold monitors

addMonitor(AbstractMonitor am)
— adds the provided monitor to the registry; if a monitor with the same ID already exists it is replaced

removeMonitor(AbstractMonitor am)
— removes the monitor whose object matches the provided one exactly

removeMonitor(String ID)
— removes the monitor with a unique id matching the provided ID

getMonitor(String ID)
— returns the monitor from the registry that has a matching ID

<u>AbstractMonitor</u>

AbstractMonitor(String ID, IMonitorValue value)
— initializes the class's instance variables; called by the extending classes

queryMonitor()
— allows the user to retrieve the current value of the monitor

addMonitorListener(MonitorListener ml)
— allows the user to register to be notified when the value of the monitor changes

removeMonitorListener(MonitorListener ml)
— removes the specified listener from the set of registered ones

<u>MonitorListener</u>

monitorEventReceived(MonitorEvent de)
— called to notify the application of an event's arrival; performs application level actions

Fig. 17. Monitor API

7.3 The `NetworkAbstractions` Package

This package defines the protocol described in this paper. It allows users to create contexts that span the network and to send one-time queries or register persistent queries on defined contexts. It also allows users to respond to context queries issued by other hosts in the network.

The main point of interaction between the package and the protocol is the `NetAbsMgr` class. The user has no knowledge of the `NetworkDiscovery` package described above; the package simply supports the needs of the protocol. The user does have some knowledge about the `Monitor` package, however, since the user must extend the `AbstractMonitor` class for every physical monitor it wants the `NetworkAbstractions` package to use if they do not already exist. It must then use the methods provided by the `NetAbsMgr` to register these monitors and to access them. The current implementation does not provide any built in monitors, but later extensions may provide for some standard monitors.

The `NetAbsMgr` implements the Singleton pattern [Gamma et al. 1995], which ensures that only a single instance of the class exists in a single JVM. To access the instance, a user can call the `getManager()` method in the `NetAbsMgr` class. If a user wants to use settings other than the defaults provided, the `initialize()` method, if called before the user starts to use the manager object, allows the specification of new parameters. The `NetAbsMgr` object provides methods that allow users to add and remove monitors to and from the `MonitorRegistry` the `NetAbsMgr` maintains. These monitors can then be accessed through the `getMonitor()` method.

To define a context over the ad hoc network, a user calls the `constructContext()` method. This method takes as parameters a `CostFunction` and a bound of type `Cost`. These two types are interfaces in the `NetworkAbstractions` package, and the user must create specialized classes that implement the interfaces. A `CostFunction` encapsulates the user's definition of the cost of a path. The interface contains an `evaluate()` method that the user must define. This method can access the `NetAbsMgr` located on the host where the function is being evaluated and can gain access to the necessary monitors through the manager. When the user defines the `evaluate()` method, the code can access any desired monitor by name, but when the code executes on another host, the application running on that host must have already registered the required monitors. It is the assumption of the `NetworkAbstractions` package that this is dealt with by the application. Every user-defined `CostFunction` ought to have an associated user defined `Cost`. This class will define the meaning of the cost and can be as simple as a wrapper for an integer or as complicated as the user desires. The only requirement is that any `Cost` class defines a `compareTo()` method. When successfully called, the `constructContext()` method returns to the user a unique context id (of type `NetAbsID`) that the user will provide in future queries over the context.

Once the context is constructed, the user performs queries over it through the `sendQuery()` and `registerQuery()` methods. Both of these methods take as parameters a `NetAbsID` and an application level message. The `sendQuery()` method simply propagates the application message to the hosts present in the specified context. The `registerQuery()` method does the same, but the query remains registered on all hosts that remain in the context, is removed from those that leave, and is added to any that enter. In this case, the query remains active on the hosts in the context until the registering host calls the `deregisterQuery()` method.

Responses to queries arrive in reply messages from other hosts. To receive replies to queries it sends over a context, a host must create a `ReplyMessageListener` and register it with the `NetAbsMgr`. The current implementation of the manager does not demultiplex the context replies; it sends all replies to all registered listeners, regardless of the `NetAbsID` of the context being replied to.

For the application to respond to queries issued by other hosts over their defined contexts, the user must also create and register `QueryMessageListener`s with the `NetAbsMgr`. Once registered, these listeners are notified whenever a context query arrives from another host. The application can send an application level response through the `sendReply()` method of the `NetAbsMgr`. Figure 18 shows the API for the `NetworkAbstractions` package.

7.4 Simplifications

While the APIs presented in this section provide a flexible mechanism for specifying generalized contexts, it can be overly complicated for the needs of straightforward applications. For these cases, we provide metric implementations as veneers on top

<u>**NetAbsMgr**</u>

getManager()
— allows access to the singleton instance of the manager

initialize(InetAddress mcast, int mcastPort, int queryPort, int replyPort)
— initializes the `NetAbsMgr` with the provided parameters

addMonitor(AbstractMonitor am)
— adds the provided monitor to the stored `MonitorRegistry`

getMonitor(String ID)
— returns the monitor from the `MonitorRegistry` that has a matching ID

removeMonitor(String ID)
— removes the monitor matching the provided ID

constructContext(CostFunction cf, Cost bound)
— initializes the context requested by the user; returns a context id to the caller

sendQuery(Message appMessage, NetAbsID contextID)
— sends the provided query to the context identified by the ID

registerQuery(Message appMessage, NetAbsID)
— registers the provided query on the context specified; the query remains registered until explicitly deregistered

deregisterQuery(Message appMessage, NetAbsID)
— deregisters the query associated with the specified message on the specified context

addReplyMessageListener(ReplyMessageListener rml)
— adds the specified listener to those notified when a reply to a query arrives

removeReplyMessageListener(ReplyMessageListener rml)
— removes the specified listener from those notified when a reply to a query arrives

addQueryMessageListener(QueryMessageListener qml)
— adds the specified listener to the set of those notified when another host's query arrives at this host

removeQueryMessageListener(QueryMessageListener qml)
— removes the specified listener from those notified when another host's query arrives at this host

sendReply(Message appMessage, NetAbsID contextID)
— sends the specified application message in response to a query over the context with the specified id

Fig. 18. Network Abstractions API

of these APIs. These veneers provide basic metric functions (e.g., those described in Section 4). They take as parameters the essential portions of the cost function and bound. As an example, a veneer that constructs a context based on increasing hop count implements the functionality, taking a single parameter that is the bound on the hop count.

8. RELATED WORK

Creating paths between nodes in an ad hoc network is neither a new problem nor an easy one. 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 1997] 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 our current work 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 shows the diversity present among them. The main difference between the solutions offered by these protocols and the requirements of the acquaintance list problem previously described lies in the fact that each of the ad hoc routing protocols described requires a known source and a known destination. Instead, our protocol allows a host to abstractly specify the group of hosts with which to communicate.

Communication with a subset of the nodes in a network is accomplished using multicast routing protocols. One possible solution to our problem would build a multicast tree or mesh for the acquaintance list and then send messages over this structure. Multicasting in ad hoc networks has received much attention as of late. Early approaches used the shared tree paradigm commonly seen in wired networks. Shared tree protocols have been adapted for the wireless environment to account for mobility in these systems [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]. Both the multicast tree and mesh protocols use a shared data structure approach. That is, they assume that for a given multicast group, there may be multiple senders. These senders share the tree built for the group to route their messages. While a shared approach might optimize a solution, an acquaintance list is built for a particular application running on a particular host. There is no need to create a shared data structure. Also, a sender is guaranteed that its messages will be received by all members of the multicast group, but these members must initially register with the group. While these protocols address the mobility that causes nodes to join and leave the group, the acquaintance list problem does not use a registration. Instead, a particular query should reach only the nodes that satisfy the context specification at the time of the query's life in the system.

In summary, our protocol is influenced by the unicast and multicast protocols described above because we need to address many of the same concerns as these protocols. Like them, our solution accounts for frequent mobility of nodes, the transient nature of connections, and the changing properties of both the nodes and links in the network. Our approach, however, differs in some key aspects. First, a node does not necessarily know to which other nodes a particular query will be sent. Instead, the node can specify some properties of the path to the nodes with which it wants to communicate. Second, any data structure built over the system must guarantee that the path used to communicate with a node satisfies the constraint specified by the application. Finally, the protocol does not need to search the whole network for possible paths. As described in the previous section, the nature of the context specification guarantees that once a node that does not satisfy the specification is found, any nodes farther on that path will not satisfy the specification either. This last point is the key that guarantees our protocol can reach a fixed point.

9. DISCUSSION

The abstraction allowed by the context specification is quite powerful. Through use of diverse examples, we have provided a glimpse of its expressive power. This power comes from the abstraction's ability to accommodate any property of a network that can be quantified either on an individual host or on the link between two connected hosts. In this way, the abstraction itself does not limit the definition of context but leaves it open to the application's needs. The context can be computed not just over neighboring nodes, but over all reachable nodes in the ad hoc network. Because ad hoc networks can grow very large, the application developer or user must specify reasonable contexts that can be computed and operated over efficiently. Specifying a wider context might be desirable for applications that operate in a more static environment or can sacrifice performance. A narrower context might be desirable for applications operating in a highly dynamic or densely populated environment.

The protocol presented in the previous section offers one example of a distributed implementation of the context computation. This protocol makes assumptions about atomicity guarantees of both the computation of the context and the operation over the context. One requirement is that the queries issued by the reference application are atomic with respect to each other. That is, it must be guaranteed that a query finishes before a subsequent query is issued. This guarantee is not built into the protocol but could be easily added in a number of ways. The application could compute a timeout over network properties after which it should be guaranteed that the query has propagated along the entire shortest path tree. On the other hand, the protocol might require that every member of the context reply to the reference host with a list of its "children". When all descendants have responded, the application is free to issue a new query.

10. CONCLUSION

The ideas behind this work are rooted in the notion that mobile application development could be simplified if the maintenance of contextual information were to be delegated to the software 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.

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.

Acknowledgements

This research was supported in part by the National Science Foundation under Grant No. CCR-9970939 and by the Office of Naval Research under MURI Research Contract No. N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research. The authors wish to thank Jamie Payton for her help with the implementation. This paper is a substantially revised and extended version of earlier work that appeared in "Network Abstractions for Context-Aware Mobile Computing" that appeared in *Proceedings of the 24th International Conference on Software Engineering*, (pp. 363–373, May 2002).

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, G. and Kotz, D. 2000. A survey of context-aware mobile computing research. Tech. Rep. TR2000-381, Department of Computer Science, Dartmouth College. November.

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.

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.

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

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.

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*.

Park, V. and Corson, M. 1997. A highly adaptive distributed routing algorithm for mobile wireless networks. *IEEE Infocom*.

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.

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.

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.