Network Abstractions for Context-Aware Mobile Computing

Authors: Gruia-Catalin Roman, Christine Julien, 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, one that was 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 that interact with each other opportunistically via transient wireless interconnections. We extend the context to encompass awareness of an entire 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, one that extends across multiple mobile hosts in the ad hoc network. The latter makes it possible to delegate the continuous reevaluation of the context and the performance of operations on it to some middleware operating below the application level. This... Read complete abstract on page 2.
Network Abstractions for Context-Aware Mobile Computing

**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, one that was 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 that interact with each other opportunistically via transient wireless interconnections. We extend the context to encompass awareness of an entire 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, one that extends across multiple mobile hosts in the ad hoc network. The latter makes it possible to delegate the continuous reevaluation 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.

This technical report is available at Washington University Open Scholarship: [http://openscholarship.wustl.edu/cse_research/266](http://openscholarship.wustl.edu/cse_research/266)
Network Abstractions for Context-Aware Mobile Computing

Gruia-Catalin Roman, Christine Julien and Qingfeng Huang

WUCS-01-28

October 2001

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130
Network Abstractions for Context-Aware Mobile Computing

Gruia-Catalin Roman, Christine Julien, and Qingfeng Huang
Department of Computer Science
Washington University
Saint Louis, MO 63130
{roman, julien, qingfeng}@cs.wustl.edu

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, one that was 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 that interact with each other opportunistically via transient wireless interconnections. We extend the context to encompass awareness of an entire 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, one that extends across multiple mobile hosts in the ad hoc network. The latter makes it possible to delegate the continuous reevaluation 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.

Keywords
Mobile computing, ad hoc network, context-aware computing, specification

1. INTRODUCTION

The ubiquity of mobile computing devices opens the user’s computing environment to a rapidly changing world where the network topology, or the physical connections between hosts in the network, must be constantly recomputed. Software must adapt its behavior continuously in response to the changing context.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2001 ACM X-XXXX-XX-XXXX/...$5.00.
application in such an environment should be permitted to
supply a definition of its desired context; subsequent opera-
tions issued by the application would be performed only in
the subset of the network satisfying the application’s defini-
tion. This requires mechanisms for computing an applica-
tion defined context that may include distant hosts reach-
able only indirectly through an ad hoc routing protocol.

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’s needs and
by adopting a notion of context that extends to an entire
reachable set of neighboring hosts. The research issues we
pose in this paper address both specification and implemen-
tation concerns relating to context definition and mainte-
nance. First is the question of how to facilitate a formal
specification of context, one 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
such points whose distance from the point of reference (i.e.,
that denoting the host carrying the application of interest)
do 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 carry out operations on it. The protocol presented
in this paper builds upon ideas from on demand ad hoc rout-
ing but constructs and dynamically maintains a tree over a
subset of neighboring hosts and links whose attributes con-
tribute to the definition of a given context, as required by
an application on a particular mobile host. Context sensi-
tive operations are carried out through a cooperative effort
involving only hosts that are part of a given context.

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. Sec-
section 4 presents a protocol that computes and maintains
a specified context. Section 5 provides some discussion of
the protocol followed by conclusions in Section 6.

2. PROBLEM DEFINITION

Ad hoc mobile networks may contain many hosts and links
with varying properties. These hosts and links and their
properties define the context for an individual host in the
network. The behavior of an adaptive application running
on a host in an ad hoc network depends on this continu-
ously 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
broadcasting 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 prop-
erties of links among them. However this 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 appli-
cation 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. Be-
cause we aim to provide both a manner for an application to
specify its context and a protocol that computes and main-
tains the context according to this specification, we need to
allow the context specification to remain as general and flex-
able 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, the ability
to formally specify a context that spans a subset of the ad
hoc network in existence at any given point in time. Ab-
stractly, the context can be viewed as a subnet around the
reference host and the properties of that subnet’s compo-
ents (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.

Throughout the discussion of the context specification
and the protocol that achieves it, we will refer to the follow-
ing scenario. A family on a cross country vacation would like
information about their current environment. That is, while
they are driving, they would like to be constantly aware of
the points of interest in their near vicinity. When the chil-
dren are hungry or the car needs gas, they would like to be
able to quickly discover restaurants or gas stations nearby.
The context they will operate on is defined as the five miles
around them. As we build the context specification and the
protocol, we will indicate how each piece would be achieved
for this scenario.

3. CONTEXT SPECIFICATION

Extending the availability of contextual information be-
ond a host’s immediate scope requires an abstraction of the
network topology and its properties. After specifying some
constraints including the application’s specific definition of
distance and a maximum allowable distance, an application
on the reference host would like a qualifying list of acquaint-
ances to be generated. That is:

Given a host, \( \alpha \), in an ad hoc network, and a
positive value, \( D \), find the set of all hosts, \( Q_\alpha \),
such that all hosts in \( Q_\alpha \) are reachable from \( \alpha \),
and for all hosts, \( \beta \), in \( Q_\alpha \), the cost of the shortest
path from \( \alpha \) to \( \beta \) is less than \( D \).

To build this list we first must define a shortest path and
how 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 in many different ways the
communication in the network. We abstract these prop-
erties 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 applica-
tion has the freedom to specify which properties define
the weights of links. 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 two nodes in the network.

Once a weight has been defined and calculated for each
link in the network, a cost function specified by the appli-
cation can be evaluated over these weights to determine the
cost of a particular path in the network. Continuing the
network hop count example, the cost function specified by
the application would be the sum of the weights of the links
along a path. Because the weight of each link is one, the
number of hops from the source of the path to that node determines the cost at that node. 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 will include only the lowest cost path to each node in the network. We will see later in this section and in the next section that this tree and the paths composing it have several nice properties that we will take advantage of in building and maintaining the tree.

Because we aim to restrict the scope of an application's context, calculating the lowest cost to every node in the network is not reasonable. To limit the context specification, we require the application to specify a bound for its cost function. Nodes to which the cost is less than the bound are included in the context. 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. We will also introduce more complex and realistic examples to demonstrate the power and generality of the approach. At each point, we will also explain how the specification applies to the family vacation scenario.

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. 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.

The vacationing family also specifies a bound on their context. If they are currently driving through Minnesota, museums in Washington D.C do not interest them. Therefore, they restrict their context to be only other communicating nodes in the vicinity, specifically, within five miles.

![Figure 1: The bounded shortest path tree](image)

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, Qα, while nodes outside the dashed circle are not part of this list and will not be included in queries over Qα.

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 (i, 2, ..., n) and designate the value of the cost of node i as vi, then vi > vi-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 logical constraint over the search space by specifying the bound, D, over the value, vi, 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.

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. The calculation of the cost of a path, P, originating at the reference node, α, will be represented as fα(P). In an arbitrary graph, however, multiple paths may exist from a node, 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 length of this path gα(β). That is, for all paths, P from α to β,

$$g_α(β) = \min_{\text{over all } P \text{ from } α \text{ to } β} f_α(P)$$

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 gα(β).

Figure 2 shows the same shortest path tree as shown in 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 shown 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 the shortest path 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, $T$. 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 would like to assign a cost from the reference node, $\alpha \in \overline{V}$, to any other reachable node, $\beta \in \overline{V}$. An application running on the reference node specifies a cost function that provides instructions to the lower layer on calculating the cost of a given path in the logical network, $\overline{G}$. A path, $P = (\overline{w_0}, \overline{w_1}, \ldots, \overline{w_k})$ indicates the path originating at the reference host, $\overline{w_0}$, traversing nodes $\overline{w}_i$ through $\overline{w}_{k-1}$ and terminating at $\overline{w}_k$. As a shorthand, we introduce the notation, $P_k$, to indicate the piece of the path, $P$, from $\overline{w}_0$ to $\overline{w}_k$, where $\overline{w}_k$ is one of the nodes along the path. Using this notation, $P_k = P$.

Given a path in $\overline{G}$, the topological cost of the path from the reference node, $\overline{w}_0$, to a host, $\overline{w}_k$, can be defined recursively using the path cost function, Cost, specified by the reference host's application. The cost of the path from the reference host, $\overline{w}_0$, to node $\overline{v}_k$ along a particular path, $P_k$, is represented by $f_{\overline{v}_0}(P_k)$. The recursive evaluation to determine this value is:

$$f_{\overline{v}_0}(P_k) = \text{Cost}(f_{\overline{v}_0}(P_{k-1}), w_{k-1,k})$$

(1)

$$f_{\overline{v}_0}(\overline{w}_0) = 0$$

(2)

Figure 3 shows the recursive cost function. The figure shows that the cost of, or distance to, host $\overline{v}_i$, represented by $\nu_i$, 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 $\overline{v}_{i-1}$.

The family on vacation would like the path cost function to reflect the physical distance from their car to other communicating nodes. We will see at the end of this section how we mathematically define a cost function to accomplish this. It is more complicated than it first appears because of the requirement that the cost of a path be strictly increasing as the number of hops from the reference grows. We do provide two other cost functions here.

We first revisit the network hop count example. Assume that the weight of a link is one. Then 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,

$$f_{\overline{v}_0}(P_k) = f_{\overline{v}_0}(P_{k-1}) + w_{k-1,k}$$

While useful, the hop count example is a bit too simplistic. Here we introduce a second example that we will carry through the explanations. Using bandwidth as a measure of cost should indicate that a path that traverses a link of lower bandwidth costs more (i.e., takes longer to send the same data) than a path with higher bandwidth links. For this example to conform to the requirement that the cost of a path strictly increases as the distance from the reference host grows, we make the simplifying assumption that the bandwidth decreases as the number of hops increases. At the end of this section we will show a mechanism for accommodating a cost function of this type without these assumptions. Assume that the weight on a link is the inverse of the bandwidth; lower bandwidth links have higher weights. The reasoning behind this weight assignment will be explained later. Because the cost of the path should reflect the lowest bandwidth encountered on that path, the cost function for a particular node indicates the minimum bandwidth, or highest weight, on the path. In this case,

$$f_{\overline{v}_0}(P_k) = \max\{f_{\overline{v}_0}(P_{k-1}), w_{k-1,k}\}$$

The cost of the path from the reference node to each reachable node can be defined using this 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 the mapping of 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 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 first map the host and link attributes to the abstract space represented by graph, $G$, by placing values on every vertex and edge. First, we quantify the properties of a mobile host as a value, $\rho_i$ on the vertex $v_i \in V$ representing the mobile host in the graph. Formally, $\rho : V \rightarrow P$. The value of $\rho_i$ 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, $\omega_{ij}$ on the edge $e_{ij} \in E$ representing the edge in the graph. Formally, $\omega : E \rightarrow \Omega$. The value of $\omega_{ij}$ can be a combination of the link's length, throughput, etc.
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 for the corresponding application. We designate an application’s logical network \( \mathcal{G} = (\mathcal{V}, \mathcal{E}) \). This is the logical network on which the cost function was built previously. We use the information about the node and link properties to create a topological ‘distance’ between each pair of connected nodes in the logical network, \( \mathcal{G} \). To do this, we combine the quantifications of node properties and link properties into only weights on edges in \( \mathcal{G} \). Given an edge, \( e_{ij} \in \mathcal{E} \) from the original mapping, \( \mathcal{G} \), and the two nodes it connects, \( v_i, v_j \in \mathcal{V} \), the weights of the two nodes, \( \rho_i \) and \( \rho_j \), are combined with the weight of the edge, \( \omega_{ij} \), resulting in a single weight, \( \omega_{ij} \), on the edge \( e_{ij} \in \mathcal{E} \) in the logical network. No host, \( v_i \), \( v_j \) in the logical network has a weight. Formally, this projection from the physical world to the virtual one can be represented as:

\[
\Gamma : P \times P \times \Omega
\]

or more specifically,

\[
\omega_{ij} = \Gamma(\rho_i, \rho_j, \omega_{ij}).
\]

The value of \( \omega_{ij} \) is defined only if nodes \( v_i \) and \( v_j \) are connected.

As an example of a weight assignment, we will first 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 \( \omega_{ij} \) be one for every \( e_{ij} \in \mathcal{E} \) in the original graph. The value of \( \rho_i \) will not be used. Then the weight, \( \omega_{ij} \), of an edge, \( e_{ij} \in \mathcal{E} \) in the logical network is: \( \omega_{ij} = \omega_{ij} = 1 \).

While this example can serve as a useful measure in a real network application, it does not demonstrate the power of the abstraction being provided. For the bandwidth example introduced in the previous section, the weight on an edge in the graph representing the logical network reflects the inverse of the bandwidth available between two nodes. Let \( \rho_i \) in the original graph, \( \mathcal{G} \), be the maximum bandwidth at which host \( i \) is capable of transmitting. Then the weight of an edge, \( \overline{e_{ij}} \), in the logical network graph can be calculated as:

\[
\omega_{ij} = \frac{1}{\min(\rho_i, \rho_j)} \quad \text{if } e_{ij} \in \mathcal{E}
\]

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’.

For our vacation scenario, we will assign weights on edges to reflect the distance vectors between connected hosts. The next section describes the assignment of these weights and defines the cost function used by the vacationing family. It also serves as an example for overcoming the strictly increasing requirement on path costs while still using distance as a metric.

3.6 A Complete Example

As mentioned when the bandwidth example was introduced, the path cost function does not satisfy the requirement that the costs along a path be strictly increasing unless we assume that the bandwidth of the hops decreases as the number of hops from the reference grows. Because this is not a reasonable assumption, we now introduce a complete example that circumvents these assumptions by combining a metric similar to the bandwidth with hop count.

This new metric will satisfy the requirements of the vacationing family that wishes to perform queries over its changing environment as their car travels across the country. The
family would like to know the locations of and information about points of interest within five miles. This requires that the calculated context be based on this physical distance between the family's car and other reachable hosts. For this example, the weight placed on edges in the logical network reflects the distance vector between connected nodes. That is, given two connected nodes, the weight on $\mathbf{ij}$ connecting them accounts for both the displacement and the direction of the displacement between the two nodes. Formally,

$$w_{ij} = \mathbf{ij}$$

Figure 4a shows an example network where specifying distance alone causes the cost function to not satisfy the requirement that the function be strictly increasing. The figure shows the shaded reference host, $\alpha$, and the results of its specified cost function. The cost function shown in this figure simply assigns as the cost of a node the physical distance to the reference. The bound the application specified in this example is $D = 10$. Notice that nodes $C$ and $D$ are outside of the context, while $E$ should be placed inside the context. When the cost of the path is strictly increasing, host $C$ knows that no hosts farther on the path will qualify for context membership. In this example, this condition is not satisfied, however, and no limit can be placed on how long context building messages must be propagated.

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

$$\nu = (\max D, \max C, V, c)$$

The first value, $\max D$, 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 host to this host. The second value, $\max C$, keeps the maximum number of consecutive hops for which the value of the cost function remained the same at any point previously on the path. The next value, $V$, is the distance vector from the reference host to this host. We will show below how this vector is calculated. The final value, $c$, indicates the number of hops for which $\max D$ has not changed. This may be less than or equal to $\max C$.

Specifying a bound for this cost function requires specifying a bound on both $\max D$ and $\max C$. It is also important that we define the comparison function for this metric. A given bound has two values, and if a host's cost function 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 $\max D$ and its $\max C$ are less than the values specified in the bound. It is important to notice that neither the value of $\max D$ nor the value of $\max C$ can ever decrease. Also, if one value remains constant for any period of time, the other is guaranteed to eventually increase. This observation allows us to treat this cost function as strictly increasing.

Figure 4c shows the cost function for this distance example. In the first case shown in the example, the new magnitude of the vector from the reference host to this host is larger than the current value of $\max D$. In this case, $\max D$ is reset to the magnitude of the vector from the reference to this host, $\max C$ remains the same, the distance vector to this host is stored, and $c$ is reset to $0$. In the second case, $\max D$ is the same for this node as the previous node. Here, $\max D$ remains the same, $\max C$ is set to be the maximum of its old value and the current $c$ incremented by one, the distance vector to this host is stored, and $c$ is incremented by one.

Figure 4b shows the same nodes as in 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 ($\max D$) and 2 is the bound on the maximum of the number of hops for which the maximum distance did not change ($\max C$). 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 of values, $\max D$, $\max C$.

4. CONTEXT COMPUTATION

The protocol developed takes advantage of the fact that an application running on reference host, $\alpha$, does not necessarily need to know which other hosts are part of the 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 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.

4.1 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 many special conditions encountered in this new type of network. Hosts in 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 [16] and Clusterhead Gateway Switch Routing [9] mimic traditional routing protocols because they maintain consistent up-to-date information for routes to all other nodes in the network [18]. This class of algorithms is based on modifications to the classical Bellman-Ford Routing algorithm [6]. Maintaining routes for every other node in the network can become quite costly. Performance comparisons [3] 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 [16] builds on the DSDV algorithm but minimizes the routing overhead by creating routes on demand. Dynamic Source Routing (DSR) [11] requires that nodes maintain routes for source nodes of which they are aware in the system. Finally, the Temporally Ordered Routing Algorithm (TORA) [13] 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 [5]. In this scheme, routes are chosen from the source to the destination based on the 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, we would like a host to be able to specify abstractly 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 [9, 10]. 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 [2, 12]. 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 are initially registered 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 will be influenced by the unicast and multicast protocols described above. We will need to address many of the same concerns as these protocols. Like them, our solution must account for the frequent mobility of the nodes, the transient nature of the connections, and the changing properties of both the nodes and the 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 in an acquaintance list 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 that our protocol can reach a fixed point.

4.2 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.

<table>
<thead>
<tr>
<th>Query, q</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.num</td>
<td>the application sequence number of q</td>
</tr>
<tr>
<td>q.s</td>
<td>the sender of this copy of q</td>
</tr>
<tr>
<td>q.n</td>
<td>NOT necessarily the reference node</td>
</tr>
<tr>
<td>q.d</td>
<td>the distance from the reference to q.s</td>
</tr>
<tr>
<td>q.d</td>
<td>the distance from the reference to the host at which the query is arriving</td>
</tr>
<tr>
<td>q.D</td>
<td>the bound on the cost function</td>
</tr>
<tr>
<td>q.Cost</td>
<td>the cost function</td>
</tr>
</tbody>
</table>

Figure 5: The Components of a Query

Figure 5 shows the components of a query. Besides the components shown, each query also carries some data information for its corresponding application. 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. 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 this host.

The explanation of the protocol is divided into two sections: tree building and tree maintenance. After the presentation of the building of the shortest path tree, it will be easy to add the maintenance to the algorithm. 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

Figure 6 shows the state variables that a host participating in a context computation must hold. This is the information for a host, $\beta$, that is part of $\alpha$'s acquaintance list. This shows only the information needed for participation in $\alpha$'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. Two worth discussing are the sets $C$ and $I$. $C$ 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. The set \( F \) contains all of the neighbors that this host knows are in the reference host's context. This host will use this information when we discuss later how to recover the memory used to store a context specification's state on a host.

Tree Building

The application is assumed to maintain the weights on the links in the network by updating them in response to changes in the contextual information important to that application. We also assume that the weights for the links have been calculated and that each host has been notified of the weights of the links connecting to it. For each of these links, a host should know both the weight of the link and the host on the other side of the link.

Any information that a particular host requires for computation of another host’s context arrives in a query; there is no requirement for a host to keep any information about a global state. We assume that the topological changes in the network and the application’s issuance of queries are atomic with respect to each other. Also, assume that the queries are atomic with respect to each other, i.e., one query finishes completely before the application issues the next one.

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.

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 \( \infty \). Subsequent copies of the same query are disregarded unless they offer a lower cost path. As shown in the second IF block of the \( \text{QUERYARRIVES} \) action in Figure 6, 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 \( \text{PropagateQuery} \) function, described with the protocol's support functions in Figure 7. 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 result is \( c \) 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.

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 process it regardless of whether or not it arrived on the currently stored shortest path. A host does not wait for new queries 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 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 \( \text{AppProcessQuery} \) support function described in Figure 7.

![Figure 6: State Variables](image)

![Figure 7: Context Computation](image)

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 query 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 query.

The family on vacation issues both transient and persistent operations over its context of five miles. When the car needs gas, the family asks for the nearby gas stations. This is a transient operation. Because the family is vacationing, however, they would also like a list of nearby points of inter-
supported functions displayed in a list. To accomplish this, the system registers a persistent operation on the context of five miles. As the car moves across the highway, the list is updated to reflect the changing points of interest.

The protocol presented in Figure 8 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 neighbors of an included 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 when we introduce tree maintenance to the protocol. Because the protocol in Figure 8 does not maintain any tree on the tree, there is also no way for a host to recover the memory used by this context specification. We will see in the next section how adding tree maintenance allows us to clean up context specification storage and accommodate persistent queries.

**Tree Maintenance**

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.

When an application needs to leave persistent operations on other hosts in its context, that context needs to be maintained, even when no new queries are issued over it. Hosts whose cost grows as a result of a network topology change should 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 the topology changes are atomic with respect to the registration and deregistration of the persistent operations and the transmission of the results of these operations.

Because both hosts connected by the link are notified of any change, both can take measures to recalculate the shortest path trees if necessary. Figure 9 shows the same protocol presented in Figure 8. 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.

Figure 7: Support Functions

**Figure 8: Context Computation and Maintenance**

**Figure 9: Context Computation and Maintenance**
Figure 10: Context Computation and Maintenance with Clean Up Mechanism

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. As a car moves across the country, it leaves information about its specified context on every host it encounters. The car may never come back, so each host that was part of the context 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 decreased in cost, thereby pushing neighbors out of the context. To account for this, after propagating the query to all neighbors within the bound, D, the host should also send a clean up message to all neighbors not within D.

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

5. 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 some 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.

6. 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. 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 they change over time.

To ensure the atomicity of an application's operations, 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-9770399. 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.

7. REFERENCES