Context-Sensitive Data Structures Supporting Software Development in Ad Hoc Mobile Settings

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

Abstract

Context-aware computing, an emerging paradigm in which applications sense and adapt their behavior to changes in their operational environment, is key to developing dependable software for use in the often unpredictable settings of ad hoc networks. However, designing an application which gathers, maintains, and adapts to context can be a difficult undertaking, even for a seasoned programmer. Our goal is to simplify the programming task by hiding such issues from the programmer, allowing one to quickly and reliably produce a context-aware application for use in ad hoc networks. With this goal in mind, we introduce a novel abstraction called context-sensitive data structures (CSDS). The programmer interacts with the CSDS through a familiar programming interface, without direct knowledge of the context gathering and maintenance tasks that occur behind the scenes. In this paper, we define a model of context-sensitive data structures and present protocols which enable the programmer to construct and maintain a CSDS as a distributed structure over a mobile ad hoc network in a state of flux.

1. Introduction

In recent years, communication technology has begun to reflect the dynamic nature of our society, with computing devices becoming increasingly portable and untethered. The widespread use of mobile computing devices brings about an increased demand for software designed with mobility in mind. In fact, we can expect the number of applications designed for use in ad hoc networks to experience rapid growth. In such networks, connections are formed opportunistically between devices within wireless communication range. Applications for this environment are likely to come into routine usage in situations such as disaster recovery in which rescue workers require the ability to find and treat victims, construction site supervision in which a foreman gathers information around a site to gauge the progress of the project, and so on. These and other applications over ad hoc networks operate in open and highly dynamic environments, making it difficult for the programmer to produce reliable and dependable software.

Context-aware computing has been advocated as a solution to dealing with the programming complexity associated with such development efforts. Context-awareness refers to the ability of an application to adapt its behavior in response to environmental changes. Typically, in context-aware systems, sensors gather information about the environment, and the collected information is delivered to the application. Context-aware office applications such as Active Badge [4] and PARCTab [9], as well as tour guide applications such as Cyberguide [1] and GUIDE [2], for example, rely on events generated by a location sensor to trigger the update of a person's display to reflect the current physical location. Similarly, FieldNote [7] uses sensors to collect information about the physical environment such as time and weather, and the information delivered to the application is attached to notes taken by researchers in the field. Constructing such applications is a daunting task, requiring the developer to consider the interaction between the application and a number of possibly heterogeneous sensors in order to gather and deliver context information.

Several frameworks and infrastructures have been devised to promote efficient and reliable development of context-aware applications by masking the complexity

of interacting with heterogeneous sensors. The Context Toolkit [8], for example, relies on context widgets to gather context and to deliver it to the application, effectively separating context sensing and delivery from the rest of the application. In the Context Fabric [5], a service infrastructure approach is employed, in which context sensors are made available across the network to provide access to context information. While these context-aware support systems offer mechanisms to simplify the interaction with sensors, the programmer is still required to know the source of data to access and to operate on that data. In a mobile ad hoc network, the open and dynamic nature of the environment makes it somewhat unreasonable to assume that the programmer knows in advance the identities of various data sources; applications for use in such scenarios require a highly decoupled method of data access.

With this in mind, we propose the concept of context-sensitive data structures as the basis for a new programming methodology. A context-sensitive data structure is determined by and provides access to data available in the context; it is encapsulated as an abstract data type (ADT), which is represented by a class in a programming language such as Java or C++. Like all classes, it provides the programmer with an application programmer interface (API) to access and manipulate data. The collection of data items operated on by an instantiation of such a class changes as the context, i.e., the content of the ad hoc network, changes. The distributed data items are accessed using the API of the local class instantiation.

A simple example of the context-sensitive data structures concept can be found in the LIME [6] model, which utilizes a transiently shared tuple space data structure to support coordination among applications in a mobile ad hoc environment. However, the middleware implementation of the model is strongly tied to the tuple space abstraction. We seek to provide programmers with the ability to use context-sensitive versions of stacks, queues, trees, and other data structures to suit the particular needs of the application.

The resulting design methodology provides the designer with the flexibility to use familiar and proven programming tools for context-aware application development. It also allows one to reuse them across applications, as building context-sensitive data structures from scratch is an expensive undertaking. Our goal is to provide a general model and infrastructure to support the development of context-sensitive data structures. Developing a flexible, general infrastructure will require a great deal of study; to begin, we investigate the feasibility of providing such a model. As a first step, we examine a particular data structure, the priority queue, and devise protocols needed to support its context-sensitive operation in a mobile ad hoc network, with the intent of extracting lessons about the kinds of protocols needed in an infrastructure that supports the development of context-sensitive data structures.

The remainder of this paper is organized as follows. Section 2 summarizes the computational model and the notion of context assumed throughout this paper, and further explains our notion of an infrastructure to support development of context-sensitive data structures. A motivating example of a context-sensitive data structure and its use in developing a context-aware application is introduced in Section 3. Section 4 addresses the key elements required in an infrastructure for supporting the development of context-sensitive versions of traditional data structures. Discussions concerning the continuing development of the CSDS model and conclusions appear in Section 5.

2. Context-Sensitive Data Structures Explained

In this section, we discuss the use of context in ad hoc networks. We begin by defining the computational model. We then define context and discuss how we can limit the context over which an application operates. Finally, we discuss a design methodology for capturing evolving contextual information within a specified scope and delivering it to the application level in the guise of a locally accessible and appropriately encapsulated data structure.

In our work, we consider systems in which logically mobile agents (units of modularity and execution) execute on physically mobile hosts (simple containers for agents). Throughout this paper, we use the terms "agent" and "application" interchangeably, as agents are the components making up the application. Agent communication and migration can occur whenever the participating hosts are within communication range. A closed set of bidirectionally connected hosts form an ad hoc network.

An agent's context is defined to be any property that a connected host in the ad hoc network can sense or any data an agent in this network offers within the same application or across applications. With this definition it is evident that as the number of participants in the ad hoc network grows large, an agent's context becomes overwhelming. Consider, for instance, a driver on the highway. She may wish to act upon traffic on the highway by asking vehicles ahead for information. Though the driver only needs information for the city in which she is driving, on a very busy highway, the context may

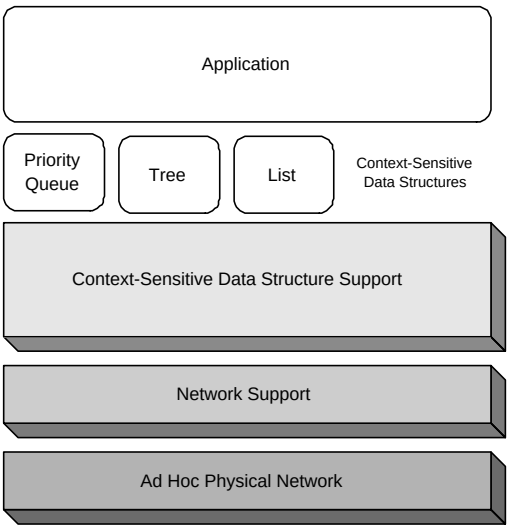

Figure 1. Proposed infrastructure architecture for providing CSDS support

span hundreds of miles. Bandwidth would be wasted by spreading the query to reachable cars outside of the city limits. Not only would the driver receive information that is not pertinent to her application, the volume of responses could render the application unusable. For reasons such as these, we will always place bounds on the application context.

Restricting the context solves only part of the problem; we must also provide decoupled access to the data available within the context in a manner that is natural to the programmer. We believe that context-sensitive data structures are an appropriate abstraction for accessing and operating on the data made available in the selected context. A context-sensitive data structure's content is determined by the state of the environment and the specification of context supplied by the application. The content of the data structure is defined as all data items on agents running on hosts in the subnet of the ad hoc network forming the application context, as well as properties sensed by the same set of hosts. The task of managing access to the data spread across the ad hoc network is hidden from the application programmer, and access to the data elements is gained only through operations defined on the ADT. Operations performed on the context-sensitive data structure can effect a change in the context of others, as can the movement of an agent that may cause it to join or leave someone else's context. As these changes in the state of the environment occur, the content of the context-sensitive data structure is changed appropriately in response. An application developer using a context-sensitive data structure can operate on the dynamically changing set of data elements that are distributed throughout the context as if the data were stored in a local, persistent data structure.

We envision the gradual development of a library of context-sensitive data structures for use by context-aware application programmers. In most cases, the application programmer should not have to implement the context-sensitive data structure; he should simply choose among the already available context-sensitive data structure implementations. The application programmer uses the API of the selected context-sensitive data structure to interact with context data as if it were local. This style of interaction is outlined in Figure 1. The figure shows an application being developed on top of one or more context-sensitive data structures.

Since many data structures share common operations, we envision providing a specialized infrastructure that supports the development of context-sensitive data structures, as shown in Figure 1. The protocols included in the infrastructure would be used to restrict the scope of the context to a subnet defined by a context specification tailored to the application. Operations of a context-sensitive data structure access and manipulate only those data elements residing on agents executing on hosts within the subnet. In addition to scoping the context, the protocols provided by the infrastructure must assist in accessing and manipulating data in the restricted context to support the implementation of context-sensitive versions of traditional data structure operations, e.g., search, modify, and iterate.

In the remainder of this paper, we investigate the software engineering potential for context-sensitive data structures. First, we offer a concrete example of a context-aware application that can benefit from the use of a particular context-sensitive data structure, the priority queue. We then explore the protocols we must provide in the infrastructure to support the development of context-sensitive data structures for use in such applications. In doing so, we seek to demonstrate the feasibility of applying the context-sensitive data structures concept and associated design methodology. Along the way, we illustrate the kind of capabilities we need to include in a more complete infrastructure as it is being developed.

3. Programming with Context-Sensitive Data Structures

The impetus behind the introduction of the context-sensitive data structure design methodology is to reduce development costs in terms of effort and er-

rors, and to make context-aware application development accessible to even novice programmers. Context-sensitive data structures provide a decoupled method of accessing and operating on data in the ad hoc network, one that is simple and natural to the programmer, using the same interface as in static settings. Moreover, their dynamically changing content is managed transparently, eliminating the potential for programming errors incurred by interacting with a complex and dynamic ad hoc network

To illustrate the utility of context-sensitive data structures and the associated design methodology, consider a disaster recovery scenario in which triage is employed to effectively treat the wounded. Victims are quickly examined to evaluate the seriousness of their injuries and are tagged with devices that emit (via wireless radio or infrared) information about their assigned injury classification, ranging from injuries that need immediate attention to those for which treatment can be postponed. Rescue teams are assigned areas in which they must arrange transport for those with the most severe injuries first, and provide as much on-site treatment as possible for these victims until transport is available. A volunteer is selected by the rescue team member to treat the most seriously wounded victim until transport arrives. A volunteer's assignment may change as the status of injured victims within the context changes. After a rescue crew member arranges on-site treatment for victim, he must arrange for the victim's transport to a hospital. As a victim is transported, they are removed from the context of the application. As new victims are discovered and their injuries evaluated, they are added to the application's context. Figure 2 illustrates this application.

A simple application for rescue team support could be constructed around the notion of a context-sensitive priority queue. Within the context-sensitive priority queue, the content of the data structure is defined by a context specification that restricts the context to a manageable area of the disaster site. The data associated with the priority queue reflects an ordering over the injured within that restricted area such that the most seriously injured victim is at the head of the queue. The context, and hence the content of the context-sensitive priority queue, is updated independently of the application's operation on the queue.

We envision our priority queue as having two operations: `getNext()` and `removeNext()`. The `getNext()` operation in our application is used to obtain access to an injury description for the victim in the context that has the highest priority injury. The injury description includes a unique injury identifier, the injury priority, and the geographical location of the injured person. The `removeNext()` operation in the disaster recovery application is used to obtain access to the injury description of the victim with the highest injury priority and to remove the injury description from the priority queue.

In the disaster recovery application, the context-sensitive priority queue data structure is populated with all of the victims in the context ordered according to injury priority. The rescue crew member uses the application to get the head of the priority queue, dispatching a volunteer to tend to the victim until transport can be arranged. Because we consider that crew members may have been assigned overlapping contexts and that the transport vehicles available to one crew member may not be available to another, the injury description obtained to dispatch treatment should still should be made available. For this reason, the dispatch function of the application is implemented using the `getNext()` operation previously described. Once treatment has been dispatched to the most severely wounded victim, the crew member uses the application on his PDA to determine if any of his transportation resources are available. If so, the application assigns the available transportation resources to the most severely injured victim in the context. Because the victim has been assigned on-site treatment and scheduled for evacuation, the victim should be removed from consideration by the rescue crew teams. Therefore, the transport scheduling function of the application should be implemented using the `removeNext()` operation previously discussed.

Building the application described from scratch can be a significant undertaking. The application developer must include functions to sense the set of neighboring hosts, to send messages to agents on reachable hosts, and to formulate and issue queries to obtain data. Query responses must be processed and placed into a traditional, static priority queue. Each time an operation is requested, the application must query the hosts in the network to ensure operation over a set of data most closely reflecting the current state of the context.

The amount of data processed is reduced, explicit maintenance by the application programmer is removed, and application development is simplified when using the context-sensitive data structures programming methodology. Figure 3 shows sample code for an implementation of the disaster recovery application using the context-sensitive priority queue. This version of the application simply defines a context, instantiates the context-sensitive priority queue, and performs processing on the priority queue using the operations made available by the API, e.g., `getNext()` and

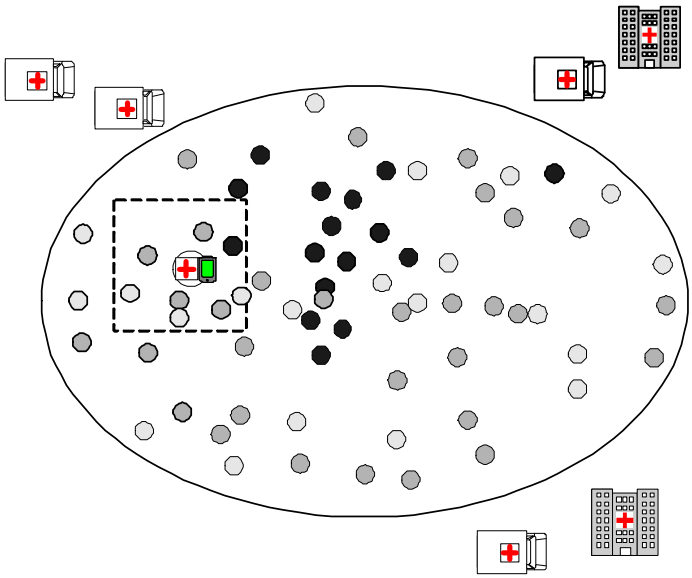

Figure 2. Disaster Recovery Scenario. The disaster site lies within the labeled oval. Rescue members are assigned to areas of the site. A particular rescue crew member (noted by the encircled cross) and his assigned area (the dashed box) are shown. The crew member uses a PDA that runs the treatment and transport application to assign available ambulances to transport the most seriously wounded victims in his area to a hospital nearby. Victims are represented by the shaded circles, with seriousness of injury reflected by darker shading.

`removeNext()`. The data structure does not have to be explicitly reconfigured by the application each time a victim is transported. Instead, an untreated victim in the context with the highest injury priority can be identified simply by using the `getNext()` operations.

This example is suggestive of the programming productivity gains one could achieve with context-sensitive data structures. In the next section, we explore what is needed to support the implementation of context-sensitive data structures like the priority queue used in the disaster recovery application.

4. Infrastructure Support for Context-Sensitive Data Structures

To support development of context-sensitive lists, trees, stacks, queues, and other data structures, we must consider what is necessary to support the implementation of their operations. First, since an application's context includes properties sensed by any connected host in the ad hoc network, the amount of data available within the context can be prohibitively large. To provide context-sensitive versions of data structures that can be used in development of reasonably efficient context-aware applications, it is necessary to consider how to limit the volume of data available to the application. One way is to limit the reach of the context itself. Second, some operations require complex queries over the data available in the context to find an element with a particular property. Asking programmers to develop protocols for querying the ad hoc network belies our philosophy of simplifying development of context-aware software. In this section, we begin to address the issues of placing a bound on the context and providing protocol support for implementing the operations of a context-sensitive data structure. Several variants of such a protocol is presented, and an example implementation of the context-sensitive priority queue's `getNext()` operation is given.

Supporting Data Queries over a Restricted Context. To restrict the amount of data included in a context-sensitive data structure, we can limit an agent's context to a subnet of the ad hoc network using the network abstractions protocol [3]. This protocol uses an agent's specification of context to build a minimum spanning tree that covers the context. The spanning tree is used to route data queries issued by the agent to all hosts within the context. It is this kind of flexible support for limiting the context that we wish

```
public class DisasterRecovery {
   PriorityQueue pq;
   Context context;
   Metric distance = predefined distance metric...;
   Cost bound = one block bound...;
   public DisasterRecovery() {
      context = new Context(distance, bound);
      PriorityQueue pq =
         new PriorityQueue(context);
   }
   public void main(String args[]) {
      while(victimsUntreated()) {
         if(volunteersAvailable()) {
            new TreatmentThread treat =
               new TreatmentThread(pq);
            treat.start();
         }
         if(transportAvailable()) {
            new TransportThread transport
               new TransportThread(pq);
            schedule.start();
         }
      }
   }

   class TreatmentThread extends Thread {
      the start method calls the run method...
      public void run() {
         dispatch(getVolunteer(),
            (pq.getNext()).id);
      }
   }

   class TransportThread extends Thread {
      the start method calls the run method...
      public void run() {
         assign(getTransport(),
            (pq.removeNext()).id);
      }
   }
}
```

Figure 3. A Context-Sensitive Data Structure Approach to the Disaster Recovery Application

to include in our infrastructure as a foundation for developing context-sensitive data structures. We envision the network abstractions protocol as the basis for providing support to developers for issuing operations on context-sensitive data structures to search for a single value.

The network abstractions protocol calculates the cost of network paths using a cost function and a bound. The cost function evaluates the cost of a path from the reference host using logical weights defined on the links of the path. Each logical link weight combines quantifiable properties of the physical link and of the hosts connected by the link. The application can specify how these properties contribute to the weight. A simple weight can be defined to assist in counting hops along a path. All links are assigned a weight of one; no host properties are used in the definition of this weight. If, when applied to these weights, the cost of a path to a host falls within the specified bound, the host is part of the context.

The construction of a minimum spanning tree is performed on-demand and in a distributed fashion. Throughout the discussion of the network abstractions protocol in this section, we refer to the agent defining the context as the reference agent and its host as the reference host. When the reference agent issues a data query, it uses the cost function to evaluate whether each of its neighbors qualifies as a member of the context by applying its cost function to the weight of the link between itself and the neighbor. If the result is within the context's bound, the reference agent sends the context definition and data query to the qualifying neighbor.

An agent receiving a context query stores the context definition. If the cost of the path is shorter than any previously stored for this context definition, then the agent names the sender of the query as its parent. The process of evaluating non-parent neighbors is repeated to determine how the query should be further distributed. This iterative distribution of the query continues along a path of connected hosts until the cost function evaluation of neighboring hosts exceeds the bound.

At any point after receiving the context query, an agent can respond to it through its parent. Any agent in the context receiving a response to a data query checks to see if the reply is intended for itself. If so, the agent processes the response. If not, the reply is forwarded to the issuer via the agent's shortest cost path.

We can use the data queries and response mechanism of the protocol as described to provide support for one possible implementation of an operation

that searches the context for a single value. Consider, for example, the `getNext()` operation on the context-sensitive priority queue. An implementation of this operation can be achieved using the network abstractions protocol to send a data query requesting a particular data element from each host in the context. The result is a collection of data, such that there is a data element originating from each host in the context. In the implementation of the `getNext()` operation, all data elements returned by the network abstractions protocol are sorted and ordered according to their priority fields. The data element with the highest priority field will be returned by the `getNext()` operation. Such an implementation can prove expensive, however, if the `getNext()` operation is repeatedly used, since the protocol must be executed and data elements must be sorted each time that the operation is issued.

Using the network abstractions protocol as described above, we can support *transient operations*. In these types of operations, a data query is issued once and is routed using a minimum spanning tree constructed on-demand for routing the query, the replies are routed to the issuing agent using the spanning tree, and the spanning tree for the query is cleaned up. In addition to transient queries, network abstractions supports the use of *persistent operations*. With a persistent query, the data query is registered on a host in the context and is evaluated each time that the context changes. To support the routing of replies to persistent queries, the spanning tree is maintained. The ability to use persistent queries would be useful in developing operations that are frequently used in dynamic networks that require a higher level of consistency. The network abstractions protocol supporting persistent queries is summarized below.

Supporting Persistent Queries. To support persistent queries, the spanning tree must be maintained in the presence of changes in the network that effect the link weights. Network abstractions assumes that hosts can detect weight changes on the links connecting them to other hosts.

The first scenario to consider is when a node detects a link weight change on a link that leads to a parent. The weight change causes the path cost to the initiator to increase or decrease. If the weight change caused the path cost to increase, then the host searches stored information about the same context to see if a shorter path to the initiating host now exists through another host. In either case, the neighbors of the node receiving the weight change notification will have to be notified of the change in path cost.

If the link on which the weight changed leads to a non-parent host. If the weight change results in a shorter path using the changed link through a non-parent node, the shortest path and parent must change.

In addition, the set of hosts belonging to the context may change. Hosts moving into the context receive the data query, and hosts moving out of the context remove it. In either case, the link weight changes effect their addition or removal from the maintained spanning tree. Persistent data queries use the tree maintained by network abstractions to deliver replies to the reference host.

One possible implementation of the `getNext()` priority queue operation, which requires the return of the largest data value present in the context, can be achieved using persistent data queries in the network abstractions protocol. The priority queue developer uses network abstractions to register persistent data queries on all hosts in the context. The result returned to the developer is a collection of data response in which each host in the context contributes a data element. As changes occur in the environment (e.g., hosts move out of the context, hosts are added to the context, or data values change), the initiator of the query is sent an updated response so that it will have a more consistent view of the state of the context. Such an implementation would often be useful in a situation where the `getNext()` operation is frequently used, since it eliminates additional overhead incurred by repeatedly issuing a transient data query, building the spanning tree, and sorting the resulting collection of data. This eliminates some of the overhead incurred with a transient query, since the spanning tree for routing does not have to be built each time that an application calls the `getNext()` operation. However, the implementation is still somewhat expensive, since all of the returned data elements must be sorted each time the `getNext()` operation is issued by the application.

When the network abstractions protocol is used for transient or persistent queries in the manner described above, the issuing agent receives responses from all nodes in the context. In an implementation of a search operation on a context-sensitive data structure, all of the results returned from the network abstractions protocol must be processed to identify the element that must be returned. Aggregating the responses from the hosts in the network before they reach the issuing agent would eliminate the additional processing required in the implementation of a search operation on a context-sensitive data structure. With this in mind, we introduce a modification of the network abstractions algorithm below which aggregates responses to data queries.

Aggregating Responses. A simple modification to the network abstractions protocol can provide aggrega-

tion of responses to data queries. The basic idea is to modify the protocol such that a node in the spanning tree must wait on the responses of all of its children. Each parent must invoke the execution of a piece of code implementing an aggregation function on the results returned by all of its children. The result of this execution is issued as the parent's response to the data query. The modification to the original network abstractions protocol required to support this behavior is described in more detail below.

As before, the construction of the minimum spanning tree is performed on-demand when an agent issues a data query. The reference agent uses a cost function to determine which neighbors receive queries. In this new version of the network abstractions protocol, any neighbor whose path cost is within the bound is added to a list of the reference agent's children. The context query is then constructed; the components of the context query include a path cost function, the bound on the cost, the initiator of the query, the sender of the query, the cost of the path to the receiving agent, and a unique context identifier. In addition, a piece of code implementing a function for aggregating data is included. The aggregation function merges several data items into a single piece of data. An example is an aggregation function that operates over several data items, and returns the data element having the maximum value.

As in the original network abstractions protocol, the constructed context query is bundled with the data query and sent to the appropriate neighbors. The agent must wait until it receives a response from its children before performing any other action. A neighbor receiving this new version of the context query stores the all of the components of the query, including the aggregation function. As before, the path cost sent in the query is used to determine if a path to the reference host can be obtained through the sender that is shorter than the path currently stored. If so, then the recipient names the sender of the query as its parent. In addition, this modified version of the network abstractions protocol requires the recipient to be removed from the sender's stored list of children. The recipient must return a message to the sender indicating that the neighbor should not be included in the sender's list of children, to keep the parent-child relationship consistent among participating parties. This prevents an agent from waiting forever on a data response from a node it considers a child, when the "child" has named another as its parent. As in the original protocol, the process of disseminating the context-query to non-parent neighbors continues until the cost of the path to a neighbor lies outside of the bound.

Agents on hosts that are included at the edge of the context know that they are the last to receive the context query because they evaluate the cost to the next neighbor and elect not to disseminate the query further. These agents are the first to evaluate the data query by using the piece of code implementing an aggregation function received in the context query. The result is passed to the parent as a data query response. Upon receiving responses from all children, a parent applies the aggregate function to merge the data into a single element. This aggregate value is passed to the host's parent, which also will eventually aggregate responses received from its children. This process continues until the reference agent receives responses from all of its children.

The modified network abstractions protocol for aggregating data results presented above would be extremely useful in limiting the amount of data processed by a developer implementing the context-sensitive priority queue's `getNext()` operation. The developer simply chooses a "maximum" aggregation function from a library of aggregation functions, and issues a data query using the modified network abstractions protocol. The returned result is the data element in the context having the maximum value. However, this solution is not the best choice for applications that frequently issue the `getNext()` operation, since a significant amount of overhead is incurred by repeatedly executing the protocol.

Supporting Persistent Queries with Aggregation. To support persistent queries in the modified version of network abstractions, we must store aggregate values at each host in the context. As before, the spanning tree must be maintained. In addition, the aggregate values now stored at each node in the tree must also be maintained. This maintenance occurs in a highly dynamic setting in which changes in the network topology and changes in the data values present on each host can occur at any moment. In our approach to maintaining the spanning tree and the aggregate values, we make certain assumptions. First, we assume that hosts can detect weight changes on the links connecting them to other hosts. Second, we assume that the data queries registered on a host are reactive and actions are triggered when a change in the data value stored at that host occurs. Third, we assume that only one change in context occurs at a time.

We first consider how to manage aggregates in a situation where a node's value changes. In such a case, the persistent data query registered on the node on which the value changed should trigger a response. The appropriate response is to evaluate the aggregation code. A simple aggregation function may perform reaggrega-

tion for its entire subtree each time the node's value is changed, always resulting in a notification to the node's parent. However, the aggregation code can be more cleverly written to eliminate unnecessary reaggregation and notification by comparing the newly generated value against the stored aggregate value. A node receiving a notification of a value change treats the notification as a response to a data query, and must always act upon it by invoking its own aggregation function.

In addition to value changes, we must consider the effects of a link weight change on the spanning tree and on the stored aggregates. First, we must consider the case in which a link upon which a link weight change was detected leads to the parent. If the weight change caused the path cost to increase, then the agent searches information about agents stored from previously received queries on the same context to see if a shorter path to the initiating agent now exists through another host. If a shorter path is found, then the agent selects a new parent. The new parent must be notified of the existence of a new child. Also, the former parent will be notified that a child has been removed. The neighbors of the node receiving the weight change notification will have to be notified of the change in path cost. Second, we must consider weight changes on links that lead to non-parent nodes. If the weight change results in a shorter path using the changed link through a non-parent node, the shortest path must change. Again, the agent changes its parent and notifies its former and new parents of the change. Neighbors will have to be notified of the change in path cost.

A node that experiences a link weight change essentially sends its new parent a data query reply. The new parent reacts to the reply by applying its aggregate function to the newly received data value. A standard aggregation function would likely request aggregates from its other children. Because the children store up-to-date aggregate values, they can immediately return those values. As intimated previously, a more intelligent aggregate function could eliminate the need for unnecessary requests for reaggregation of a subtree. This can be implemented by comparing the child's data value to the aggregate stored. If the result of the comparison reveals that the child's value should be the new aggregate, then it is stored as the host's aggregate value and a new response is sent to the host's parent. When a host has a child removed, its aggregation function is invoked. Aggregation must be performed again in its subtree because the removal of the child may have a caused a disconnection between the host and the owner of its stored aggregate.

The modified network abstractions protocol for managing persistent queries presented above is the best choice so far for implementing the context-sensitive priority queue's `getNext()` operation. Using the modified protocol in conjunction with a "maximum" aggregation function, the protocol returns the single maximum value. Because the aggregates and the spanning tree are maintained, the value returned to the application that calls the `getNext()` operation is guaranteed to always be the available maximum value in the network.

The network abstractions protocol supporting aggregation can be used to support context-sensitive operations that require querying a restricted subnet of the ad hoc network for a single value. This leads us to the next step, which is developing protocols that support other classes of operations. For instance, protocols are needed to support iteration over a data structure's data elements. Variants of the iteration protocol may be required to perform breadth-first search, depth-first search, etc. over the data items in the context. In addition, protocols are required that support safe modification and removal of data in the context.

5. Discussion

In order for the infrastructure to be usable and complete, the development of additional protocols must be considered. At the lowest level, a library of aggregation support functions should be provided to be used with the modified network abstractions protocol presented in this paper. The combination of such support functions and the modified protocol support the development of context-sensitive data structure operations that require the return of a single element having a particular property.

While the network abstractions protocol provides the ability to define a number of useful contexts, it is not possible to define all contexts that an application desires. Consider, for instance, a city employee who wants to monitor water meters distributed throughout the city. The context for his application could be defined as "all meters until a meter outside the city limits is reached". Another application might require a context based on temporal properties. For instance, an application that uses temperature data in the surrounding area to adapt its operation may only want to act upon data readings that are relatively fresh. To our knowledge, no protocols exist to define these kinds of contexts. We would like to develop protocols that allow specification of these and other contexts and to include them in our infrastructure.

As mentioned previously, another important issue to consider is access control. To date, this work assumes that every agent in the ad hoc network makes its data

available to all other agents. In reality, this is not the case. Agents should have the ability to protect all or part of their data, with different levels of protection. Access control mechanisms should be included in the infrastructure to ensure that only authorized agents are allowed to access, manipulate, or even delete another agent's data items.

Finally, we must implement and evaluate this infrastructure. The infrastructure should be flexible, allowing the context-sensitive data structure developer to use only needed components. The CSDS programmer should find the infrastructure components helpful and not burdensome to use; thus, the components of the infrastructure should have minimal programming interfaces. The operations of context-sensitive data structures that are developed using the infrastructure should perform in a reasonably efficient manner. Moreover, the concept of context-sensitive data structures as a design methodology for context-aware application development should be put to the test through the development of several applications that use context-sensitive data structures.

6. Conclusions

In this paper, we present a novel abstraction called the context-sensitive data structure designed to simplify the development of context-aware applications. A context-sensitive data structure encapsulates data items distributed among agents within a restricted portion of the ad hoc network, and provides the programmer with access to the collection of data elements as if they were local through a well-defined API. The content of the context-sensitive data structure is fluid; as the context changes, the context-sensitive data structure is reorganized to reflect the change in the state of the environment. To support the use of context-sensitive data structures as a design methodology, we propose an infrastructure that encapsulates protocols for restricting the context, for accessing data elements in the context, and for modifying data elements in the context. We envision this infrastructure as providing the context-sensitive data structure developer with a set of essential tools that can be used to develop a range of context-sensitive data structures.

Acknowledgements

This research was supported in part by the Office of Naval Research under ONR MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

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

[2] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou. Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In *Proceedings of MobiCom*, pages 20–31. ACM Press, 2000.

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

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

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

[6] A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Systems*, pages 524–533. IEEE Computer Society Press, April 2001.

[7] N. Ryan, J. Pascoe, and D. Morse. FieldNote: A handheld information system for the field. In *1st International Workshop on TeloGeoProcessing*, pages 156–163, 1999.

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

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