Report Number: WUCSE-2005-8

2005-03-01

A Query-Centered Perspective on Context Awareness in Mobile Ad Hoc Networks

Jamie Payton, Cheryl Simon, and Gruia-Catalin Roman

The wide-spread use of mobile computing devices has ledto an increased demand for applications that operate de-pendably in opportunistically formed networks. A promis-ing approach to supporting software development for suchdynamic settings is to rely on the context-aware computingparadigm, in which an application views the state of the sur-rounding ad hoc network as a valuable source of contextualinformation that can be used to adapt its behavior. Col-lecting context information distributed across a constantlychanging network remains a significant technical challenge.With this in mind, we propose a query-centered approach tosimplifying context interactions in mobile ad hoc networks.With our approach, an application programmer... **Read complete abstract on page 2.**

A Query-Centered Perspective on Context Awareness in Mobile Ad Hoc Networks

Jamie Payton, Cheryl Simon, and Gruia-Catalin Roman

Complete Abstract:

The wide-spread use of mobile computing devices has ledto an increased demand for applications that operate de-pendably in opportunistically formed networks. A promis-ing approach to supporting software development for suchdynamic settings is to rely on the context-aware computingparadigm, in which an application views the state of the sur-rounding ad hoc network as a valuable source of contextualinformation that can be used to adapt its behavior. Col-lecting context information distributed across a constantlychanging network remains a significant technical challenge.With this in mind, we propose a query-centered approach tosimplifying context interactions in mobile ad hoc networks.With our approach, an application programmer views thesurrounding world as a single data repository over whichdescriptive queries can be issued. Queries may be tran-sient, or may be more durable persistent queries that reactto changes in data or the network. Processing such queriesentails the creation and maintenance of a distributed over-lay data structure whose size needs to be under applicationcontrol. A high level of flexibility is achieved by judicioususage of mobile code fragments. In this paper, we presentthe design and implementation of our query service for adhoc networks.

2005-8

A Query-Centered Perspective on Context Awareness in Mobile Ad Hoc Networks

Authors: Payton, Jamie; Simon, Cheryl; Roman, Gruia-Catalin

March 1, 2005

Abstract: The wide-spread use of mobile computing devices has
led to an increased demand for applications that operate dependably in opportunistically formed networks. A promising approach to supporting software development for such dynamic settings is to rely on the context-aware computing paradigm, in which an application views the state of the surrounding ad hoc network as a valuable source of contextual information that can be used to adapt its behavior. Collecting context information distributed across a constantly changing network remains a significant technical challenge. With this in mind, we propose a query-centered approach to simplifying context interactions in mobile ad hoc networks. With our approach, an application programmer views the surrounding world as a single data repository over which descriptive queries can be issued. Queries may be transient, or may be more durable persistent queries that react to changes in data or the network. Processing such queries entails the creation and maintenance of a distributed overlay data structure whose size needs to be under application control. A high level of flexibility is achieved by judicious usage of mobile code fragments. In this paper, we present the design and implementation of our query service for ad hoc networks.

A Query-Centered Perspective on Context Awareness in Mobile Ad Hoc Networks

Jamie Payton, Cheryl Simon, and Gruia-Catalin Roman
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}@wustl.edu

ABSTRACT

The wide-spread use of mobile computing devices has led to an increased demand for applications that operate dependably in opportunistically formed networks. A promising approach to supporting software development for such dynamic settings is to rely on the context-aware computing paradigm, in which an application views the state of the surrounding ad hoc network as a valuable source of contextual information that can be used to adapt its behavior. Collecting context information distributed across a constantly changing network remains a significant technical challenge. With this in mind, we propose a query-centered approach to simplifying context interactions in mobile ad hoc networks. With our approach, an application programmer views the surrounding world as a single data repository over which descriptive queries can be issued. Queries may be transient, or may be more durable persistent queries that react to changes in data or the network. Processing such queries entails the creation and maintenance of a distributed overlay data structure whose size needs to be under application control. A high level of flexibility is achieved by judicious usage of mobile code fragments. In this paper, we present the design and implementation of our query service for ad hoc networks.

1. INTRODUCTION

As the trend in the availability and affordability of portable computing devices continues, we can expect heightened demand for software designed for use in dynamic mobile environments. The increasing popularity of ubiquitous computing drives the need for applications developed for ad hoc networks in particular. In these settings, network connections are formed opportunistically by devices within wireless communication range, without any assistance from a wired infrastructure. Such environments are characterized by their open and highly dynamic nature, resulting in highly unpredictable and transient interactions among resource-constrained devices.

In recent years, researchers have concluded that the key to developing rich applications for limited platforms in inherently uncertain settings is to rely on the resources offered by other hosts in the vicinity. As such, they have embraced the context-aware computing paradigm in which applications adapt their behavior according to changes sensed in their operational environments. For example, context-aware office applications such as Active Badge [8] and PARCTab [22], use an employee's location (provided to sensors by the employee's badge) to automatically direct communications (phone calls, faxes, etc.) to the correct office. Other typical context-aware applications include more sophisticated context-aware office spaces (e.g., GAIA [19]), tour guides which adapt displays according to a tourist's location and interests (e.g., Cyberguide [1] and GUIDE [5]), and context-aware note tools which attach environmental information such as time and temperature to observational notes (e.g., FieldNote [20]).

Though popular consensus seems to point to context-awareness as a programming solution for ad hoc networks, a key question remains unanswered: how can we best assist context-aware application programmers to master the difficult tasks of gathering and maintaining a diverse collection of context information originating from multiple sources distributed across the constantly changing ad hoc network? Our approach relies on the notion that context can be abstracted as a virtual data repository that reflects the continuously changing state of an application's environment. A programmer must simply query the virtual repository to gain access to available context information. This "repository" of information is actually a reflection of the collection of data items provided by applications residing on hosts within the ad hoc network. Providing such an abstraction simplifies the development task by hiding the complex distributed interactions required to obtain context distributed across the network, allowing the developer to interact with context as if it were local. Additionally, interacting with a database abstraction allows the programmer to use a uniform API to collect context from a heterogeneous set of mobile hosts and applications.

Relying on a database-like abstraction seems to be a natural choice for simplifying context interactions. In fact, ideas

have been previously explored in the development of middleware systems for ad hoc networks that rely on a Linda-like tuple space abstraction for agent coordination (e.g., LIME [14], MARS [4], EgoSpaces [11], etc). While tuple spaces are useful for managing underlying coordination needs in mobile environments, these systems are limited to the sharing policy imposed by the tuple space model and to basic tuple space operations for context interactions. In another vein of work, researchers have developed query processing systems that utilize database-like queries to obtain information from sensor networks (e.g., TinyDB [13] and Cougar [23]) and P2P environments (e.g., AmbientDB [3] and PeerDB [15]). We believe a query processing approach can provide an application developer with a rich set of operations that can simplify context interactions. As such, we introduce a query-centric approach to supporting context interactions. In contrast to the aforementioned query systems, our approach addresses the technical challenges that arise due to the intrinsic nature of ad hoc networks and targets solutions to that environment.

Our query service addresses the continuous nature of data in ad hoc networks by providing a special query construct which allows for reactive query processing and application notification. This construct, the persistent query, has semantics which are comparable to a subscription in the publish/subscribe paradigm. Such a construct is needed to support the development of applications which require notification when data previously returned as a query result has become invalid, perhaps due to the disconnection of host on which the application providing the data resides or due to the application updating its data value. This construct also supports the delivery of newly available data to the query initiator.

Our query service also addresses the need of the application programmer to limit the time, communication, and processing costs incurred by issuing queries over the ad hoc network. We allow the developer to control the scope of the query, the query propagation scheme, and the reply processing scheme in order to tailor the execution costs of the query to the needs of the application. One design option for addressing issues related to query control is to hard-code all possible customization configurations into the query service. Queries would be parameterized with a set of customization options that the programmer chooses from a fixed set. The query service simply uses the provided parameters in order to determine how to process the query. However, installing a query service which includes a comprehensive set of query options for execution may be impractical in ad hoc networks; the environment is composed of a variety of devices, many of which have limited resources. Furthermore, adhering to such a design implies advance knowledge of all possible specializations that any context-aware application might ever require. For these reasons, we rely on the use of mobile code fragments that can be installed over the network to encapsulate an application's tailored context scope definition, query propagation scheme, and reply processing scheme. Because the mobile code fragments are loaded and installed across the network at runtime as a query is issued, this approach has potential to greatly reduce the size of the code installed on each machine. Moreover, it results in a flexible, extensible, and expressive query service, since mobile code elements can be interchanged and new mobile code elements which implement arbitrary query control schemes can easily be included in the query service.

In this paper, we present the design and implementation of our query service for mobile ad hoc networks. Our goal is to promote rapid software development by simplifying the manner in which an application interacts with its context, while empowering the developer with the ability to control context interactions at a fine-grained level in a flexible manner. By relying on mobile code to encapsulate customizable elements of query execution, we are able to provide a query service that is flexible, modular, and extensible. Such adherence to a design principle of generality allows us to support a wide range of existing context-aware applications as well as those of the future.

The paper is organized as follows. In Section 2, we provide a more detailed description of the problem that we are trying to solve. An overview of our approach is presented in Section 3. Section 4 details the design and implementation of our query service for mobile ad hoc networks, and describes how to use the system through an example. Section 5 presents a comparison of related work along with a discussion of the benefits and limitations of our approach. A discussion of the approach employed in this paper is presented in Section 6, and concluding remarks appear in Section 7.

2. PROBLEM DEFINITION

Applications for ad hoc networks increasingly rely on surrounding information to perform their assigned tasks. We believe that treating the network as a single virtual data repository and utilizing a query interface on that abstration can be useful to application programmers collecting context information across the network. Our goal is to provide a query service that supports context-aware programming in mobile ad hoc networks. In this section, we define in more detail the problem which we are trying to solve. We begin by presenting the basic computational model. We then motivate the need for fine-grained control of query processing in the ad hoc network setting, and introduce our approach to supporting user-controlled queries in ad hoc networks.

We consider systems in which applications execute on physically mobile hosts. A closed set of bidirectionally connected hosts form an ad hoc network. Applications selectively provide access to their local data items for public use, and can keep some data items private. For clarity, we use the term "context item" to refer to a data item that the application makes publicly available. The term "data item" is a general term that we use to describe any piece of information that an application produces or manipulates. Data items may range from a sensor application's simple temperature reading (an integer value) to a remote printer service provider's connection proxy (a piece of executable code). By default, an application's *context* is defined to be all context items contributed by applications residing on reachable hosts that comprise the ad hoc network. Notice that this means the application's context reaches across the entire network, and is not limited to an application's local data or context items within one hop. In our approach, the developer of a context-aware application uses query constructs (e.g., SELECT, MIN, MAX,

SUM, AVG, INSERT, DELETE) to query an abstraction of the application's context, a virtual data repository, to gain access to context information. Though the developer is presented only with the data repository abstraction, behind the scenes, a query is issued over the ad hoc network and distributed context information is collected from reachable hosts and presented to the application.

As an example of an application that would benefit from such context interactions, consider a software program used for monitoring crops in the field. The field of crops is equipped with small wireless sensors which can sense a number of properties, such as temperature, moisture, humidity, and chemical balance. Roving users collect information by querying the collection of information provided by the sensor field and other applications to aid in their tasks. For instance, a small mobile robot responsible for spraying pesticides constantly moves through the field, evaluating when it is appropriate to release the chemicals. The robot can only release chemicals if the plants need them and if no humans are in the vicinity. Thus, the robot queries the collection of sensors to determine the chemical level of the crops and queries the profiles of connected mobile devices to determine if people are in the field. An environmental scientist may use his PDA to take notes on soil samples collected in the crop field. To make the notes complete, the scientist queries the sensor network field for the temperature, moisture, and chemical readings across the field. He also queries the robots for information about chemicals released in the field, e.g., the quantity, location, and time that chemicals were released. Harvesters may roam through the field, taking inventory and noting the status of the crops on their PDAs. Farmers may later collect this information to adjust irrigation or fertilization schedules.

While these and many other applications are aided by querying the ad hoc network to gain access to the wealth of distributed context information, utilizing a naive query processing approach can be problematic in such environments. An application that queries a virtual data repository is actually querying the entire ad hoc network for context information. Obviously, such an operation is expensive in terms of time, communication, and processing costs. Furthermore, once the query responses have all been returned to the application, much of the information may be discarded due to the need for locality, temporal, or other constraints imposed by the application. Moreover, the collection of information available in the ad hoc network is constantly changing, due to the mobility of hosts and dynamicity of applications. After querying the network, applications often need to know that the returned information is no longer valid or that new information is available. In the crop monitoring example, the robot queries the network to determine if there are people around before emitting potentially noxious chemicals. For safety purposes, if the robot begins spraying the crops and a human is around, the robot should stop until the human has left the area. Thus, it is important that the robot constantly know if humans are in the vicinity. With a naive query service, the robot would need to constantly query the network in order to determine if the area is clear.

The above issues must be addressed in order to develop a practical query-based solution for simplifying context interactions that is suitable for use in ad hoc networks. In response to this need, we propose a query service for ad hoc networks which puts control over query processing in the hands of the application developer. In our query service, we provide the application programmer with the ability to limit the costs of querying the collection of information. We do so by providing control over the scope of the query, the methods used to propagate the query, and the methods used to process the query and send replies. Because the ad hoc network can include resource constrained devices, we rely on the use of mobile code fragments, which are loaded and installed at runtime across the portion of that network that is network within the scope of the query, to capture these tailorable query schemes. In addition to providing control over the costs of the query through the use of mobile code, we address the issue of changing information in the ad hoc network by allowing the developer to register persistent queries, which emulate the semantics of publish/subscribe service. Registering these persistent queries on the context eliminates the need for the application to constantly reissue the same query. In the following section, we further explain persistent queries and each element of our approach to customizing query execution.

3. A FLEXIBLE APPROACH TO QUERY-BASED CONTEXT INTERACTIONS

In this section, we present our approach to tailorable query processing in ad hoc networks. We begin by giving a brief overview of how a query is executed over the ad hoc network by our query service. We then give the specifics of data representation and storage. Next, we describe in more detail how we provide programmers with control over the scope of queries, the propagation of queries over the scope, and the processing performed in the network. Finally, we discuss how to provide long-lived query constructs that are responsive to environmental changes.

An application issues a query that is packaged with its associated mobile code specializations that dictate the details of its execution. This query will be evaluated in a distributed fashion by our query service. For clarity, we henceforth refer to the application issuing the query as the *reference application*, and the host on which the application resides as the *reference host*. Once the query is issued, the mobile code elements are extracted and stored along with a unique query identifier. The query construct is evaluated on the host-level data repository, and the mobile code capturing the context definition is evaluated to determine the subset of connected neighbors that belong to the issued query's context. We call this set of neighbors the *potential context children*. Next, the query service executes the mobile propagation scheme over the potential context children to determine a set of neighbors that are referred to as the *actual context children*. The query service disseminates the query and its mobile code specializations to the actual context children. Finally, the mobile reply processing scheme for the query is executed, utilizing the local result from the host-level data repository and, in some cases, replies from the application's set of actual context children. As the query is propagated throughout the network, the query is evaluated at each network hop using the mobile code fragments packaged with the query.

3.1 Data Representation and Storage

Ad hoc networks are often comprised of a collection of heterogeneous devices, each of which may provide runtime support for a variety of applications. These applications, in turn, can provide many different kinds of data as context. In order to be useful in such heterogeneous settings, our query service must have a uniform representation of context. We choose to represent each application's context items as tuples. A tuple is a set of unordered fields, where each field is a triple of (*name*, *type*, *value*). Using tuples allows us to capture a wide range of context types, as well as to incorporate metadata about the context item using the name and type fields.

We store an application's context items, or tuples, in a tuple space associated with the host on which the application resides. Figure 1 illustrates the query service's use of tuples and the tuple space. As indicated in the figure, access to the interface of this tuple space is limited to components of the query service; applications contribute context items and access context items only through the use of queries through the query service's application programmer interface (API). Choosing this storage option allows the query service to take advantage of content-based retrieval operations which use a provided pattern, or template, to describe the desired tuple(s) to be returned. A template is similar to a tuple except that wildcards can be used in the *name* and *type* elements of a tuple field, and a tuple field's value is replaced with a constraint on the field's value. Only tuples which *match* a provided template are returned as a result. A tuple matches a template, if for every field in the template, there exists a field in the tuple with the same name and type and a value that satifies the the template's constraint. Tuple space operations used by our query service include non-blocking (or probing) variants of Linda's `rd` and `in` operations, the `inp` and `rdp`. Both operations check the tuple space for a tuple matching the provided pattern. If one or more matching tuples exist, one is selected non-deterministically and is returned. In the case of the `inp` operation, the returned tuple is also removed from the tuplespace. If no matching tuple exists, a null result is returned. Probing operations which return all matching tuples (`rdgp`, `ingp`) are also provided for use by the query service. The query service also utilizes the `out`(t) tuple space operation, which places the tuple t into the tuple space, to support insertion queries.

Another benefit of using tuple spaces is that we can take advantage of reactive constructs that have previously defined in tuple space models [14, 6, 11]. A reaction is simply an association between a pattern describing a tuple and a callback function. The semantics of these reactive constructs dictate that the appearance of a tuple in the tuple space that satisfies the reactive pattern immediately triggers the execution of the associated callback function. The insertion of the matching tuple in the tuple space and the execution of triggered callback functions occurs in a single atomic step. Such reactive constructs are extremely useful in the query service to address issues associated with reporting changing data to the application, as discussed in subsection 3.5.

3.2 Controlling the Scope of the Context

As mentioned previously, an application programmer should be able to control the cost associated with query processing by limiting the scope of the query. Essentially, we want to allow a programmer to define a context tailored to the application's particular needs. The application programer can specify a context through the use of context policies. These context policies are similar to those introduced in [11]; each policy is comprised of constraints imposed on properties of the physical network as well as on hosts and applications available within the network.

Using the network constraints provided by a programmer as part of a context policy, a distributed protocol constructs an network overlay data structure that corresponds to the desired context defined by these constraints. In the design of our query service presented in this paper, we rely on a network overlay data structure that is commonly used in ad hoc networks for routing: a spanning tree. We leverage off of the ideas proposed in [7], which constructs and maintains a spanning tree in an ad hoc network according to specified constraints on network properties. We utilize this spanning tree to further limit the context using host and agent constraints.

To construct the overlay data structure, an application programmer must provide two pieces of mobile code as part of the network constraints: a `Metric` class which encapsulates a metric defined over properties of links and hosts in the ad hoc network that will be used to calculate a logical distance of a network path, and a `Cost` class which encapsulates a bound on the metric that restricts the inclusion of hosts belonging to the context to those that are within an allowable distance. The metric must be increasing to ensure that a bounded overlay can be formed. Mobile code is used to capture the metric and bound that comprise a context policy's network constraint because it offers the developer great flexibility in defining the context. With this approach, arbitrary network constraints can be defined over properties of the ad hoc network to build the overlay data structure.

Host and application constraints that comprise the remainder of the context policy are then imposed on the resulting overlay data structure to form a more restricted context. Each host and application in the network provide a profile describing their characterizing traits. For example, a host's profile may include a unique host identifier, current location, platform type, etc. An application's profile may include an application identifier, lifetime, type, etc. Constraints are evaluated over the profile using a constraint function. Because we want to allow a number of constraint functions to be expressed, constraint functions provided as part of the constraints are presented to the query service as mobile code fragments. In essence, queries are issued over the application's spanning tree, and are processed only by applications meeting the specified application constraints running on hosts meeting the specified host constraints.

Figure 2 illustrates a context definition and the resulting spanning tree for the crop monitoring application presented in the previous section. For simplicity, we assume only one application per host in the figure, and depict each host (and hence each application) as a circle, and each network link as a line. The doubly ringed circle represents the application of interest: the application running on the robot that is responsible for spraying pesticide in the crop field. The

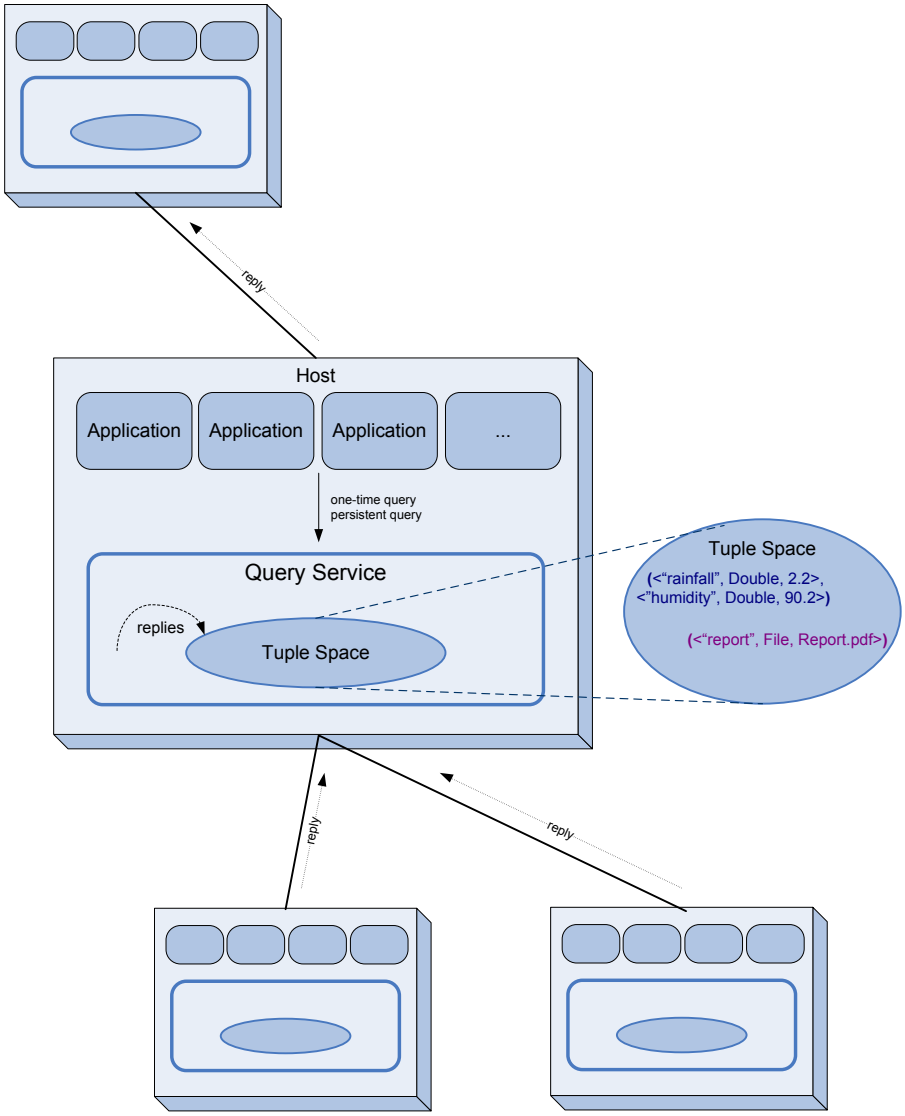

Figure 1: Each host in the ad hoc network is equipped with our query service, which uses tuples for data representation and tuple spaces for storage.

robot needs to spray pesticide on plants within a prescribed area that are in need of pest control care, and should do so only when humans are not in the vicinity. The robot needs to query chemical monitoring applications running on sensors and tracking applications running on tracking station platforms within 20 meters.

To construct such a context for the robot, the application developer provides a context policy consisting of network, host, and application contraints. To provide the network constraint, the application developer defines a metric that adds the previous physical distance and hop count to evaluate the current distance and hop count, and specifies a bound of (d, c), where d is the distance and c is the desired hop count value (in this case, $d = 20$ and $c = 3$). (The metric uses both hop count and physical distance because the evaluation of additive physical distance alone is not increasing in many circumstances and cannot ensure bounded construction of the network overlay.) As illustrated in Figure 2a, at some point in time (time t), the context defined by the network constraint only is a, b, c, where a, b, and c are hosts that meet the context specification provided by the reference application (the doubly ringed circle). The constructed spanning tree network overlay is depicted us-

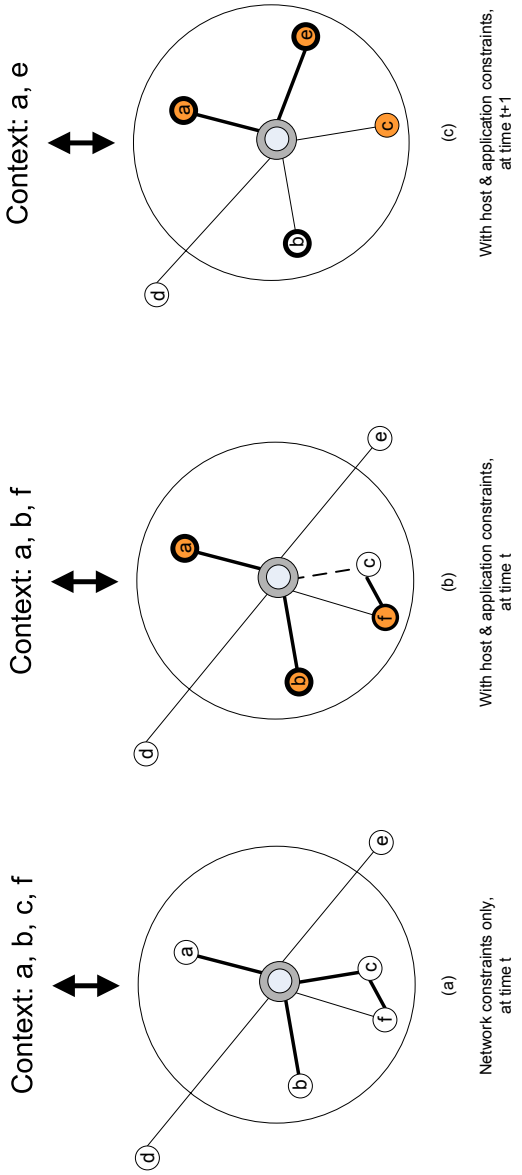

Figure 2: An application's tailored context is captured using a network overlay data structure.

ing bold lines. Figure 2b shows the network overlay and the resulting context when additional host and application constraints are applied. Satisfied host constraints are depicted by heavily outlined circles and satisfied application contraints by shaded circles. Notice the dashed line between the reference host and **c**. This illustrates the fact that **c** is technically in the network overlay formed strictly by the network constraints but does not satisfy the host and application constraints. Thus, **c** can be considered to serve as a virtual node in the spanning tree, used only to queries and reponses to and from **f**. As shown in Figure 2c, the network overlay changes over time to reflect changes in the enviornment.

3.3 Controlling Query Propagation

Different kinds of queries may have different kinds of needs. For example, an application that wants to determine the existence of a particular data item in the context issues an EXISTS query construct over the context. Typically, a query service would propagate the query over the entire context, triggering a response only when the query reaches a leaf node in the overlay network data structure, and collecting answers at the reference host from all applications in the context before returning the result to the reference application. Since this particular query does not necessarily need all data items available in the context that satisfy the query, it is possible to reduce the communication costs by

controlling the propagation of the query over the context.

One option that has potential to reduce communication costs but requires an extended execution time is to propagate the query using controlled flooding. With controlled flooding, a predicate provided as part of the query propagator is evaluated at each host. The predicate likely incorporates knowledge of the local query result. If the predicate evaluates as true, the query is not propagated further, and the reply process is initiated in order to return a result to the query initiator. Otherwise, the query is propagated to the next hop neighbors that are in the context associated with the query. Other query propagation schemes that may prove useful include random subtree and random path. In random subtree propagation, the query propagator selects some subset of next hop neighbors that are in the context and propagates the query to those neighbors, effectively limiting propagation to a pruned version of the overlay data structure. Similarly, in random path propagation, the query propagator selects a single context neighbor and propagates the query to that host, resulting in a query that travels along a single path in the network.

Rather than permanently associate a propagation method with a particular query construct, we allow the developer to consider the tradeoffs associated with each particular propagation method and specify the desired propagation method each time that a query is issued. We achieve generality and flexibility by requiring the use of a mobile code fragment to capture the desired propagation scheme. When the query programmer uses our query service API to issue a one-time or persistent query, this mobile code element is provided as a parameter. The query service receives the query packaged with the propagation scheme and extracts the mobile query propagation code. The query operation is evaluated locally, and then the query propagation code is evaluated. Evaluation of the query propagation code should result in either propagating the query further over the context or initiating the query reply process.

3.4 Controlling Query Processing and Replies

Naive evaluation of queries such as MIN, MAX, SUM, AVG, and COUNT requires the collection of query replies from all applications in the network at the reference host. The collected replies are evaluated according to the query construct, and a single result is returned to the application. This method of executing the query is cost-intensive in terms of communication. With the exception of leaf nodes, each node in the network overlay data structure must act as a router to deliver its context children's query replies. On many small devices, communication requires much more power consumption than computation. The reference application may need to minimize the resources consumed on remote platforms in order to prolong the availability of context information. To reduce communication costs, query replies are often processed in the network [12, 23, 9] such that nodes in the network aggregate query responses and communicate only the aggregate response as the query reply.

A number of algorithms exist for performing in-network processing of data. Our query service offers the developer the option of choosing an in-network processing scheme to best suit the needs of the query and the application. When an application programmer uses our query service API to issue a one-time or persistent query, she specifies an in-network processing algorithm as a parameter. Again, to satisfy our need to support arbitrarily defined processing algorithms, the query processing algorithm is encapsulated within a mobile code fragment. As the query is propagated, this mobile code fragment is extracted from the packaged query and installed on hosts determined to be in the context. The query operation is evaluated locally, the query propagation code mentioned in the previous section is evaluated, and the query processing code is evaluated to control the sending of query responses.

3.5 Managing Changing Data in Dynamic Settings

An issue of particular interest when designing a query service for ad hoc networks is the manner in which each query operation is serviced in such a dynamic setting with a changing collection of distributed data. Many applications require prolonged use of information, and should be informed of changes in the context that can affect the answer to a query. To address this need, an application can simply issue a query over the context each time that data is actually needed, constructing the overlay data structure to encapsulate the context in an on-demand fashion each time that the query is executed. We refer to such queries as one-time queries. It may, however, be more practical to maintain the overlay data structure over time and notify the application of changes in the context that impact previous query results. For these reasons, our query service provides persistent queries that are registered on a maintained context.

Persistent queries emulate the semantics of publish/subscribe services. However, they are a better choice for a query service designed for an ad hoc network enviornment because the focus is on data state rather than events. Relying on data state removes the requirement of knowing in advance the kinds of events that are relevant in an open environment. The concept of persistent queries was first introduced in [7], which introduced a protocol that supports reactive evaluation of long-lived queries. We extend their approach to provide a reference application with updates that detail changes in the network that impact the result of the application's persistent query. Such changes include the addition of new data, the removal of data previously reported as a query reply to the reference application, and the disconnection of hosts running applications which previously provided a response to the reference application's query. An application developer that registers a persistent query must be able to process replies generated by the query service as notification of such changes. Therefore, the application is required to specify the set of notifications that are of interest. It does so by implementing a listener for each kind of change. A listener for a particular kind of change processes query replies generated as a result of that change.

New data items. Addressing the issue of notifying the reference application of the addition of new context items that meet a persistent query's data specification is relatively simple. When the persistent query is issued, the query service simply registers a reaction on the tuple space using the data template provided as part of the query as the reactive pattern. The reaction's call back function simply initiates reply

processing in order to deliver the new results to the query initiator. The replies are encapsulated as a `NewDataReply` and an application processes these replies by implementing the `NewDataReplyListener` interface.

Removed data items. Dealing with deletions of context items previously reported as replies is slightly more complex since reactions cannot be triggered upon the removal of tuples from the tuple space. To address this issue, we instead register a reaction using an *anti-tuple* as the reactive pattern to signal the deletion of a reported context item. The anti-tuple is simply a copy of the tuple of interest with a value set in a special system tuple field that indicates status as an anti-tuple. When a piece of data is removed from the tuple space, a corresponding anti-tuple is inserted into the tuple space. The set of reactions are checked to see if any are triggered by the new tuple. The triggered reactions fire, and the associated call back functions are executed. As before, the reaction's callback function initiates reply processing, this time to notify the query initiator that a previously delivered context item is no longer available. Replies generated to notify applications of changes in reported data for a persistent query are of type `DeletedDataReply` and are handled by a `DeletedDataReplyListener`.

Though at first the use of anti-tuples may seem prohibitive in terms of space, the semantics of reactions dictate that insertion of the anti-tuple and the execution of callback functions associated with reactions registered on the anti-tuple appear to occur in a single atomic step, which allows the anti-tuple to be immediately removed from the tuple space.

Unavailable data items. With persistent queries, the overlay data structure that encapsulates the context associated with a persistent query is constantly updated in reponse to changes in the environment that impact the context definition. When such changes cause an application to no longer be a part of a query's context, the data that the application has previously reported to the query initiator is no longer valid. Because each host in the context associated with the query acts as a router to relay a query response the disconnection of a host can result in an entire portion of the context becoming unavailable. This data also becomes invalid to the application. The question is, how can a member of the context notify a query initiator that a context child and its descendant's data is no longer available?

Again, we rely on the use of anti-tuples and reactions to report change. However, we have indicated that a host's tuple space contains only context items contributed by applications residing on that host, which does not make it possible for a host to detect the disappearance of data provided as a reponse by context children. Therefore, we allow a host's tuple space to also include context items provided by the host's context children which have been reported as query responses. To be more specific, each time that a reply to a persistent query is relayed to an application's context parent in the overlay data structure, the query service stores the reply tuple in the host-level tuple space before processing and propagating the reply back to the query initiator. If the reply is associated with a persistent query, the query service also registers a reaction on the tuple space that incorporates an *anti-tuple* as the reactive pattern, and a call-back function. When the elimination of a host or application within the context is detected, an anti-tuple indicating such is placed in the tuple space. This triggers the reaction associated with the anti-tuple, which is designed to notify the query initiator that the data associated with the host or application specified by the anti-tuple is no longer available. Replies generated for persistent queries that indicate the unavailablility of previously reported data are of type `DataUnavailableReply` and are handled by a `DataUnavailableReplyListener`.

One issue that we have not yet addressed is the unavailability of data previously offered as query responses by a departed context child's descendants. To address this issue, we incorporate an additional special system tuple space field in each tuple that gives the path in the context overlay data structure from the query initiator to the query responder. Each time that a query initiator receives a notice that a particular context child's data is unavailable, the query initiator can use the paths of data items previously received in order to determine their validity.

4. DESIGN AND IMPLEMENTATION OF A FLEXIBLE QUERY SERVICE

We begin this section by presenting the API that a context-aware programmer uses to construct and issue queries over a tailored context. We then discuss the architecture of the query service presented in this paper and review how the query service executes a query.

4.1 Using the Query Service API

The query programmer must define a context, a propagation scheme, and a query processing scheme when issuing a query. The query service utilizes these pieces of mobile code to execute the query and deliver the desired results to the issuing application. Each of these query components is discussed in detail in the following subsections. Throughout, we use examples to illustrate the use of each concept.

4.1.1 Defining the Context

Before issuing a query, the user must define a context for the query to be issued over. This requires the programmer to provide network, host, and application constraints. The network constraints are used to define the network overlay data structure that encapsulates the context, while the host and application constraints are used to further restrict the context.

Network Constraints. In providing a mechanism to impose network constraints on the ad hoc network, we build upon the approach presented in [7] to specify and construct a context. The context is constructed using a spanning tree. This requires defining a `Metric` and a `Cost` that are used to construct the tree. The `Cost` class is used to define a property that contributes to the cost of a path in the overlay data structure. The `Metric` details how to utilize the cost evaluated at the previous hop and the cost of a link weight to determine a new cost.

To define a context, a programmer must extend the `Cost` and `Metric` classes shown in Figure 3. Defining a `Cost` subclass simply requires the programmer to define a method which

```
public abstract class Cost {
    int compareTo(Cost cost)
}

public abstract class Metric {
    private String[] monitorNames;
    public void setMonitorNames(String[] names);
    public abstract Cost wFunction(HostID otherHost);
    public abstract Cost costFunction(Cost currentD,
                                      Cost weight);
}
```

Figure 3: The `Cost` and `Metric` interfaces

compares the `Cost` object to another `Cost` object. Defining a `Metric` subclass is a bit more complex, requiring the application programmer to provide the names of environmental monitors it will use to evaluate the metric. At each host, the query manager component of the query service uses a `MonitorRegistry` provided by a supporting `monitor` package to provide the metric with access to local (on the same host) or remote (on a reachable host) monitors with the specified names. The `Metric` abstract base class also requires an extending class to implement a weight function and cost function. The `wFunction` method determines the weight of the link between the evaluating host and a neighboring host. The `costFunction` takes the cost of the path to the current host and uses the weight calculated by the weight function to determine the cost associated with including a neighbor in the context.

```
public class HopCountMetric extends Metric{
  public HopCountMetric(){
  }
  public Cost wFunction(HostID otherHost){
    //calculate the weight on the link
    HopCost weight = 1;
    return weight;
  }
  public Cost costFunction(Cost currentD,
                           Cost weight){
    HopCost newCost = currentD + weight;
    return newCost;
  }
}
```

Figure 4: An example `HopCountMetric` Class

To illustrate the use of network constraints, consider an application that wants to limit its context to a particular number of hops, h. The metric to capture this simple context definition is shown in Figure 4. We have omitted the definition of the `HopCost` class, which adheres to the `Cost` interface by storing an integer and implementing the `compareTo` method.

Constraints on Hosts. Defining constraints on the kinds of hosts that can participate in a context is relatively straightforward. Each host in the ad hoc network provides a host profile containing properties that describe the host, e.g., its unique id, disk space, platform, etc. This profile is captured as a tuple using the `HostProfile` class that extends a tuple class provided by a supporting tuple space package. Therefore, to impose host constraints, a programmer simply provides a template that describes required host properties. To provide a template, the programmer uses the `HostConstraints` class (a subclass of a template class provided by the tuple space package) to indicate which tuple fields in a host profile are of interest and to provide a constraint function that determines if the host profile field meets the needs of the application. The query service uses pattern matching of a host profile tuple against a host constraint template and the provided constraint function to determine satisfaction of constraints. Since we need to support arbitrarily defined constraint functions, these are defined as mobile code elements.

```
HostConstraint hc = new HostConstraint();
hc.addConstraint(new EConstraint(``platform'',
     new EquivalencyConstraintFunction(``PDA'')));
```

Figure 5: An example host constraint

An example use of host constraints is shown in Figure 5. The host constraint dictates that only hosts that identify themselves as PDAs in their host profiles are included in the context.

Constraints on Applications. Constraints that dictate what kind of applications may participate in the context are defined much like host constraints. Each application provides an application profile that includes application properties such as application id, application type, user, etc. The application profile is captured as a tuple, and application constraints are captured as a template. As before, pattern matching and a piece of mobile code implementing a constraint function is used by the system to determine constraint satisfaction.

```
ApplicationConstraint ac = new ApplicationConstraint();
ac.addConstraint(new EConstraint(``access code'',
     new EquivalencyConstraintFunction(key)));
```

Figure 6: An example application constraint

An example application constraint is shown in Figure 6. The constraint shown dictates that the application must have an access code that matches that defined in the variable *key*.

We provide a collection of commonly used network constraints in our infrastructure for use by the developer. The developer can use inheritance to extend the collection of metric and bound classes used to construct the overlay data structure. Likewise, we provide a collection of commonly used constraint functions that can be included in the definition of a host or application constraint.

At this point, the programmer has provided everything needed to define a context. The programmer can now use the query service API to obtain a static reference to the `QueryMananger` running on the local host. Once the manager is obtained, the programmer registers a context definition with the manager using the `createContext` method, providing network, host, and application constraints as parameters. A `ContextID` is returned to the application to identify the registered context. The application supplies this context id when issuing a query over the context. However, before issuing a query,

```
public abstract class QueryPropagator {
    public abstract Vector limitPropagation(Vector
                                        potentialChildren);
}
```

Figure 7: The `QueryPropagator` interface

the programmer must first define how the query's execution is to be controlled.

4.1.2 Defining the Query Propagation Scheme

To define a query propagation scheme, a programmer must extend the `QueryPropagator` abstract base class shown in Figure 7. To do so, a programmer must define a `limitPropagation` method which utilizes a set of potential context children and imposes the propagation constraints on this set to determine a new set of context children, which is returned to the caller. If the context children set is empty, the query is not to be propagated any further, and the propagator intiates the sending of replies by calling the `sendReply` method on the query manager.

We plan to include a number of query propagators that incorporate standard query propagation schemes with our query service. The developer can simply choose to use one of the provided mobile code fragments. However, the set of query propagators can easily be extended to incorporate mechanisms tailored to a particular application.

```
public class ControlledPropagator extends QueryPropagator{
  QueryID qID;

  public ControlledPropagator(QueryID qID){
    this.qID = qID;
  }

  public Vector limitPropagation(Vector contextChildren) {
    QueryManager mgr = QueryManager.getManager();
    Reply localResult = mgr.getLocalQueryResult(qID);
    if (localResult.getResultTuple() != null){
     mgr.sendReply(Reply);
     return (new Vector());
     }
    else
     return contextChildren;
  }
}
```

Figure 8: An example `ControlledPropagator` Class

To illustrate the development of propagators, a controlled flooding propagation scheme is shown in Figure 10. The code shows that when a local result is found that satifies the query, the propagator initiates the reply process and suspends propagation by returning an empty vector of context children to the query manager. Otherwise, the set of context children remains unchanged and is returned to the query manager to use in continuing propagation of the query.

4.1.3 Defining the Query Processing Scheme

To define a query processing scheme, a programmer must extend the `QueryProcessor` abstract base class shown in Figure 9. To do so, a programmer must define a `processReply` method that determines how received replies are processed. Because a query processor is often charged with the task of performing aggregation on replies received from its context children, the query processor may need to implement a `ChildReplyListener` interface. The processReply method is responsible for calling the `sendReply` method on the query manager to send the result to the next upstream hop in the overlay data structure.

```
public abstract class QueryProcessor {
    public abstract void processReply(Reply r)
}
```

Figure 9: The `QueryProcessor` interface

As with query propagators, we plan to provide a number of standard in-network reply processing schemes as mobile code fragments that are available as part of the query service. It is possible to extend the set of available query processors by constructing new mobile code fragments.

```
public class AggregateProcessor extends QueryProcessor
  implements ChildReplyListener{
  QueryID qID;
  Reply currentResult;
  Hashtable receivedReplies = new Hashtable();
  QueryManager mgr = QueryManager.getManager();

  public AggregateProcessor(QueryID qID){
    this.qID = qID;
  }

  public void processReply(Reply r) {
    Vector children = mgr.getChildren(r.getQueryID());
    while receivedReplies.size() != children.size()) {
     poll for child replies ...
    }

    for(i=0; i<children.size(); i++) {
     HostID hID = children.elementAt(i);
     Reply childReply = receivedReplies.get(hID);
     currentResult = aggregate(childReply);
     }

  mgr.sendReply(currentResult);
  }

  public void childReplyReceived(ChildReplyEvent cre){
    Reply r = cre.getReply();
    HostID sender = cre.getReply.getSender();
    receivedReplies.add(r, sender);
  }

  public Reply aggregate(Reply reply) {
    Reply r = ...aggregate currentResult with reply...;
  return r;
  }
}
```

Figure 10: An example `AggregateProcessor` Class

4.1.4 Issuing a Query

At this point, the programmer is ready to issue a query over a defined context using the query manager. The programmer can choose to use the API of the query manager to issue a one-time query via the `sendQuery` method, or can register a persistent query with the `registerQuery` method. In ei-

ther case, the programmer must provide a ContextID that identifies a previously registered context, an `ETemplate` that describes the desired data, the specified query option to be performed, the `QueryPropagator`, and the `QueryProcessor`.

The table in Figure 11 lists a sample set of the kinds of query operations that we expect a programmer to need when constructing context-aware applications. Notice that, in this first attempt at providing a query service for ad hoc networks, we assume a trusted enviornment. As such, a DELETE operation that we provide would allow any application to delete another application's provided context items. In an untrusted environment, access controls should be provided to determine if an application is allowed to permanently delete another application's data or if the application should instead remove the data only from consideration as part of its context. Also of note is that we expect that applications in this setting are cooperative and will provide data items for use as context via an INSERT operation.

4.2 Query Service Implementation Details

The architecture of our query service implementation is presented in Figure 12. As shown in the figure, we rely on the use of additional packages to support the operation of the query service. These packages are used to deliver messages in the ad hoc network (the message passing component), discover network neighbors (the network discovery component), and monitor environmental properties used to define the network constraints portion of the context definition (the monitor component). We assume the existence of the physical ad hoc network and a message passing mechanism; we utilize external network discovery and monitor packages.

When the query service receives a request to issue a one-time or persistent query via the `sendQuery` or `registerQuery` methods, the query manager component of the query service constructs a query to be issued over the specified context. Each query consists of a metric and bound evaluated over network properties, as well as the current cost associated with the evaluation of the metric; the host constraints, agent constraints, and their constraint functions; the query propagator; the query processor; the identifier of the query initiator; the path that the query has traversed; and the unique identifer of the query. The query manager component stores information about the query and begins to process it as described below.

First, the query manager processes the network constraints portion of the context policy in order to begin constructing the overlay data structure. To do so, the query manager uses the set of current neighbors and applies the metric to evaluate the cost of the path to each neighbor. The query manager will only include neighbors whose path cost satisfies the bound as part of the context.

To evaluate the metric, the query service uses two support packages: a network discovery package and an environmental monitoring package (shown in Figure 12). The network discovery package is used to determine the set of current neighbors. At each host, the query manager's network discovery server periodically beacons the surrounding hosts to discover the current set of neighbors. Discovery can be parameterized with policies that govern when to add or remove a neighboring host from the set of neighbors. The query manager keeps a consistent list of neighbors by implementing the `DiscoveryListener` interface of the monitor package in order to listen for events signalled by the discovery server to indicate the addition and/or removal of neighbors. The environmental monitoring package is used by the query service to gain access to monitors on local and remote hosts. The query service needs to access these monitors in order to evaluate the metric over its host and its neighboring hosts. The metric provides a list of monitor names that impact its evaluation. The query manager uses a monitor registry provided by the monitor package to access local and neighbor's monitors, and uses the unified monitor interface to query each monitor for its data value. As monitor values change, the query manager is notified and the metric is re-evaluated to determine if new neighbors are eligible to be considered as part of the context.

Once a set of neighbors have been determined to be candidates for belonging to the context, the set of neighbors is passed to the query propagator. The query propagator applies its query propagation scheme to determine a subset of the given context neighbors that are eligible for propagation and returns this set, which defines the context children. If the returned set is empty, the propagation process stops. The query manager also executes the query processor, which will initiate the process of sending replies back to the query initiator via the query manager.

When the query manager receives a propagated query, it stores the information and processes the context definition, query propagator, and query processor as described above. When the query manager receives a request to send a reply, it uses the query id associated with a reply to find the appropriate return path over the context.

5. RELATED WORK

In this section, we review related work and give a comparison to our own approach. In particular, we examine mobile databases, query processing systems, and methods for constructing spanning trees in ad hoc networks.

5.1 Mobile Databases

Over the years, researchers have explored the deployment of database systems in mobile, dynamic environments [21, 18, 2]. Mobile database systems must consider issues related to replicating data across the network such that it is readily available, ensuring the consistency of replicas across the network, and recovering from frequently terminated transactions due to disconnection. In contrast, we are not actually creating a database in our approach to providing queries over the ad hoc network. Instead, our goal is to simply hide the details of managing access to remote context information. There is no need for us to replicate data; either data is available as context to any application or it is not. Moreover, many of these database systems are targeted for nomadic network environments in which disconnection from the wired network is the exception. As such, many of the proposed solutions for data management are not applicable in mobile ad hoc networks.

5.2 Query Processing Systems

Operation Name	Definition
GET	Retrives the data matching the specified data pattern, if it exists. If more than one match exists, one is selected non-deterministically and returned
EXISTS	Returns true if data matching the specified data pattern exists and false otherwise
MIN	Returns the data item that is the "minimum" among all data itemsthat match the specified data pattern. To determine the minimum, an ordering must exist over the specified data type.
MAX	Returns the data item that is the "maximum" among all data itemsthat match the specified data pattern. To determine the maximum, an ordering must exist over the specified data type.
AVG	Returns the data item that is the "average" among all data itemsthat match the specified data pattern. To determine the average, a method of quantification must exist over the specified data type.
SUM	Returns the data item that is the "sum" among all data itemsthat match the specified data pattern. To determine the sum, a method of quantification must exist over the specified data type.
INSERT	Makes the specified data item available to others in the network. The query service places the specified data item in the host-level tuple space.
DELETE	Removes the specified data item available from consideration in the query service. The query service removes the specified data item from the host-level tuple space.

Figure 11: Operations supported by our query service

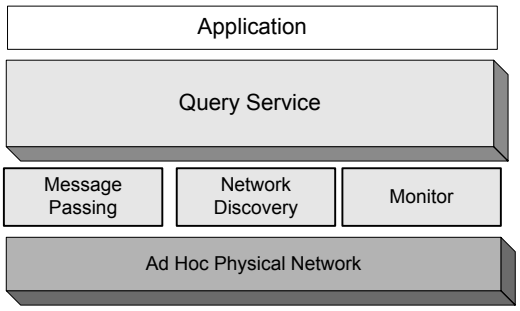

Figure 12: Query Service Architecture

Recent research has resulted in the development of data management systems for P2P network environments. For example, AmbientDB [3] provides a relational database management model that uses Chord, a Distributed Hash Table (DHT) system, to manage data sharing across a potentially large network. Users can query the network using high level database-like queries that are optimized by AmbientDB. Another distributed data system for P2P environments is PeerDB, which utilizes local SQL database table to store sharable information. Applications can issue queries over a logically federated database which includes all sharable information in the network. Because the database tables are augmented with metadata, queries can be issued in a content-based manner and no merging of schemas is needed. Similarly to our work, PeerDB utilizes mobile code to propagate a query and return its results.

Our work differs from these sytems in a number of ways. First, while PeerDB provides the ability to limit queries to nodes which have recently provided query results, it is not possible in either system to select a tailored context based on arbitrary properties of entities withint the ad hoc network. Second, neither system supports persistent queries, although this is mentioned as future work on AmbientDB. Third, unlike these systems, we do not consider data management issues such as replication and data caching. Such services are often overkill for what is needed by an application that simply needs to obtain context information from the network.

Distributed query processors have also been developed for sensor networks, e.g., TinyDB [13], TAG [12], and Cougar [23]. Since communication is very expensive in sensor network environments, many query processors for this setting use aggregation to compute intermediate results to reduce energy consumption. In addition, some systems offer a long-lived query construct that proactively deliver query results to the issuing application at the specified sampling rate. While the concepts of simplifying the task for the end user, optimizing query processing, and eliminating the need to repeatedly query the network are similar to our goals, these systems address issues specific to the expenses of sensor networks, such as dealing with lost packets, sensor failures, and inaccuracy of readings. Additionally, protocols for processing queries are designed at a low-level, using using knowledge of the sampling and sleep cycles of sensors to minimize energy consumption. To our knowledge, none of these systems consider rapid changes in topology, which frequently occurs in mobile ad hoc networks.

5.3 Spanning Tree Construction and Maintenance

In this first attempt at implementing a query service for use in ad hoc networks, we utilized a spanning tree to capture

the context. A number of ad hoc routing protocols have been developed which construct and maintain spanning trees [16, 17, 10]. As mentioned previously in the paper, our approach to the construction and maintenance of the spanning tree supporting the execution of queries is derived from the network abstractions protocol [7]. There are key differences, however, in the model of expected interaction between applications. The implementation of network abstractions required that an application explicitly reply to a query. In our implementation, we place the power of initiating replies in the query itself, using mobile code elements that are delivered as part of a query. In addition, while network abstractions supports the registration of persistent queries, the protocol is primarily designed to adjust the spanning tree in response to changes in the environment. Unlike our work, the network abstractions protocol does not report to the application when data is deleted or becomes unavailable.

6. DISCUSSION

A full implementation of the query service is underway. The query service and all required support packages are written in Java. Once the query service implementation is complete, it will be necessary to evaluate its utility and practicality. We plan to evaluate utility by using the query service to facilitate the implementation of a number of context-aware applications. The practicality of the query service can be determined by measuring different properties, such as the time required to distribute a query across the context and the communication overhead. Measuring such properties will likely require an implementation in an ad hoc network simulator such as ns-2.

In this paper, our implementation relied on the use of a particular network overlay data structure to capture the context: the spanning tree. The spanning tree is defined by an increasing metric over properties of the network links and hosts in the ad hoc network. However, defining the context in this way may rule out potential useful contexts. 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. We are interested particularly in providing a query service which provides a general method of constructing contexts such that a variety of network overlay data structures can be utilized. We hope to explore the possibility of a general approach to building structures in the network, focusing on the use of pairwise relationships defined on hosts in the network to support overlay construction and maintenance.

To this point, an issue of importance in designing query processing systems for mobile enviornments has largely been overlooked: providing a range of guarantees concerning query results that are useful to an application in a dynamic environment. Previous work on guaranteeing operations over ad hoc networks have mostly focused on providing the traditional weak and strong transactional semantics common in distributed systems for the new dynamic and mobile environment. However, we believe that there is a range of semantics in between weak and strong that would be useful in dynamic settings. For instance, an application used by a teacher that uses queries to collect exams may allow new additions to the collection of exams, but will not tolerate removal of exams. To our knowledge, no work exists to support describing and implementing a collection of such intermediate guarantees. We would like to examine this problem formally and accordingly include a range of options for evaluating queries in our query service.

7. CONCLUSIONS

Our goal is to simplify the development of context-aware applications by aiding the application developer in managing access to a dynamic collection of distributed context information. Towards that goal, we have introduced a query service for ad hoc networks. Though we simplify context interactions by allowing the programmer to query the ad hoc network as if it were a local data repository, we still provide the developer with the ability to control the execution of the query to satisfy the application's needs. We achieve generality, flexibility, extensibility, and expressiveness by relying on the use of mobile code to encapsulate customizable elements of the query service.

8. 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] D. Barbara. Mobile computing and databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):101–117, 1999.

[3] P. Boncz and C. Treijtel. Ambientdb: Relational query processing in a P2P network. In *Proceedings of the Workshop on Databases, Information Systems and Peer-to-Peer Computing 2003 (co-located with VLDB'03*, volume 2788 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

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

[5] 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 *Proc. of MobiCom*, pages 20–31. ACM Press, 2000.

[6] C.L. Fok, G.-C. Roman, and G. Hackmann. A lightweight coordination middleware for mobile computing. In *COORDINATION 2004*, volume 2949 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

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

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

[9] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th International Conference on Mobile Computing and Networking (MobiCom'00)*, August 2000.

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

[11] C. Julien and G.-C. Roman. Egocentric context-aware programming in ad hoc mobile environments. In *Proc. of 10th Int'l Symposium on the Foundations of Software Engineering*, pages 21–30, Nov. 2002.

[12] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *ACM Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2002.

[13] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD International conference on Management of Data*, pages 491–502. ACM Press, 2003.

[14] A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proc. of the 21st Int'l Conf. on Distributed Systems*, pages 524–533, April 2001.

[15] W. Ng, B. Ooi, K.-L. Tan, and A. Zhou. Peerdb: A P2P-based system for distributed data sharing. In *Proceedings of the 19th International Conference on Distributed Data Sharing*, 2003.

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

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

[18] E. Pitoura and B. Bhargava. Maintaining consistency of data in mobile distributed environments. In *International Conference on Distributed Computing Systems*, pages 404–413, 1995.

[19] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, pages 74–83, Oct.-Dec. 2002.

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

[21] A.-P. Sistla, O. Wolfson, and Y. Huang. Minimization of communication cost through caching in mobile environments. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):378–390, 1998.

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

[23] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, September 2002.