Report Number: WUCSE-2006-49

2006-01-01

A Query-Centric Approach to Supporting the Development of Context-Aware Applications for Mobile Ad Hoc Networks, Doctoral Dissertation, August 2006

Jamie Payton

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. This dissertation presents a query-centered approach to simplifying context interactions in mobile ad hoc networks. Using... **Read complete abstract on page 2.**

A Query-Centric Approach to Supporting the Development of Context-Aware Applications for Mobile Ad Hoc Networks, Doctoral Dissertation, August 2006

Jamie Payton

Complete 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. This dissertation presents a query-centered approach to simplifying context interactions in mobile ad hoc networks. Using such an approach, an application programmer views the surrounding world asa single data repository over which descriptive queries can be issued. Distributed context information appears to be locally available, effectively hiding the complex networking tasks required to acquire context in an open and dynamic setting. This dissertation identifies the research issues associated with developing a query-centric approach and discusses solutions to providing query-centric support to application developers. To promote rapid and dependable software development, a query-centric middleware is provided to the application programmer. These solutions provide the means to reason about the correctness of an application's design and potentially to reduce programmer effort and error.

A Query-Centric Approach to Supporting the Development of Context-Aware Applications for Mobile Ad Hoc Networks, Doctoral Dissertation, August 2006

Authors: Jamie Payton

Corresponding Author: payton@uncc.edu

Web Page: http://www.cs.uncc.edu/~payton

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. This dissertation presents a query-centered approach to simplifying context interactions in mobile ad hoc networks. Using such an approach, an application programmer views the surrounding world asa single data repository over which descriptive queries can be issued. Distributed context information appears to be locally available, effectively hiding the complex networking tasks required to acquire context in an open and dynamic setting. This dissertation identifies the research issues associated with developing a query-centric approach and discusses solutions to providing query-centric support to application developers. To promote rapid and dependable software development, a query-centric middleware is provided to the application programmer. These solutions provide the means to reason about the correctness of an application's design and potentially to reduce programmer effort and error.

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

A QUERY-CENTRIC APPROACH TO SUPPORTING THE DEVELOPMENT OF CONTEXT-AWARE APPLICATIONS FOR MOBILE AD HOC NETWORKS

by

Jamie Jenelle Payton, M.S. Computer Science, B.S. Computer Science

Prepared under the direction of Professor Gruia-Catalin Roman

---

A dissertation presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF SCIENCE

August 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

A QUERY-CENTRIC APPROACH TO SUPPORTING THE DEVELOPMENT OF CONTEXT-AWARE APPLICATIONS FOR MOBILE AD HOC NETWORKS

by

Jamie Jenelle Payton

ADVISOR: Professor Gruia-Catalin Roman

August 2006

Saint Louis, Missouri

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. This dissertation presents a query-centered approach to simplifying context interactions in mobile ad hoc networks. Using such an approach, an application programmer views the surrounding world as a single data repository over which descriptive queries can be issued. Distributed context information appears to be locally available, effectively hiding the complex networking tasks required to acquire context in an open and dynamic setting. This dissertation identifies the research issues associated with developing a query-centric approach and discusses solutions to providing query-centric support to application developers. To promote rapid and dependable software development, a query-centric middleware is provided to the application programmer. These solutions provide the means to reason about the correctness of an application's design and potentially to reduce programmer effort and error.

Contents

List of Tables

List of Figures

Acknowledgments

I would like to first thank those that made it possible for me to pursue graduate study. I would like to thank my advisor, Catalin Roman, for the opportunity to study under his direction and for the guidance he has offered in achieving success in an academic career. I would also like to thank my first mentor, Rose Gamble, who opened my eyes to the possibility of a research career and provided me with invaluable research opportunities as an undergraduate.

I would also like to acknowledge others that contributed to the direction of my research. Thank you to the members of my dissertation committee, Chris Gill, Chenyang Lu, Tao Ju, and Guy Genin, for their insightful comments and suggestions. Thanks also to members of the Mobilab for their feedback on ideas, papers, and talks. Thanks to Drew Frank, who helped evaluate the QueryME middleware by implementing simulations of context acquisition protocols. I would like to especially thank Christine Julien for her contributions to our joint work on Context UNITY, for her work on the Network Abstractions protocol which inspired my context acquisition protocol work, and for interesting discussions about query consistency guarantees.

Thank you to the many others who have impacted my life. I would like to thank Sara Wilemon, Tammy Alverson, Leigh Flagg, Matt Hampton, and Richard Souvenir, who are wonderful friends. Finally, I am especially grateful to my parents, James R. and Patricia Payton, who have always given me their encouragement, support, and love.

Jamie Jenelle Payton

Washington University in Saint Louis
August 2006

Chapter 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. Applications for such settings are likely to come into routine usage in situations such as disaster recovery in which rescue workers must find and treat victims, construction supervision in which a foreman gathers information around a site to gauge progress, military scenarios in which soldiers can use information collected from roving robots and autonomous aircraft to navigate safely through a dangerous battlefield, etc.

In recent years, researchers have concluded that the key to developing rich applications for limited platforms in inherently uncertain settings is to rely opportunistically on the resources offered by other hosts in the vicinity. As such, software engineers have been encouraged to embrace the context-aware computing paradigm in which applications adapt their behavior according to changes sensed in their operational environments. Using a context-aware design, a disaster recovery scenario application senses the presence of victims in the surrounding area and uses the information to arrange on-site treatment and transport to a nearby hospital for the most seriously injured. As the application's operational environment changes (e.g., victims are treated and transported), the application responds by accordingly adjusting the treatment and transport plan. Though context-awareness is touted as an appropriate design strategy for applications deployed in dynamic environments, constructing context-aware software for ad hoc networks is a complex task. Programmers developing these adaptive applications must gather and maintain a diverse collection of context information that originates from multiple sources and is distributed across an open and constantly changing network.

The goal of this research is to encourage rapid and dependable development of context-aware applications by domain programmers, who may have expertise in the application domain but are novices in managing the complex network interactions required to interact with a dynamic and distributed context. A desirable approach is to separate the complex networking tasks required to facilitate context interactions from application programming tasks and capture them within a middleware framework that provides the domain programmer with a familiar and simple, yet expressive, application programming interface (API). While frameworks and infrastructures have been devised to simplify context-aware application development, the solutions are often not appropriate for use in mobile ad hoc networks (e.g., the Context Toolkit [33] and the Context Fabric [15]) or are tied to a particular abstraction that may limit application design (e.g., LIME [26], MARS [7], Limone [11], EgoSpaces [19]).

This dissertation introduces a new abstraction to simplify context-aware application design that allows programmers to view the surrounding ad hoc network as a database. The collection of contextual information available to a particular application can be abstracted as a global virtual data repository that reflects the continuously changing state of the application's environment. A programmer must simply query the virtual repository to gain access to available context information distributed across the ad hoc network. This approach allows developers to interact with distributed context information as if it were local, effectively hiding the complex networking tasks required to acquire context in the open and dynamic setting of an ad hoc network.

To support the use of a query-centric approach to context-aware application development, this dissertation introduces a database-like abstraction embodied within a middleware system. A number of issues arise when adapting a query-centric model to the mobile ad hoc network environment. First, changes that occur within the environment may impact the collection of context information that is available to a

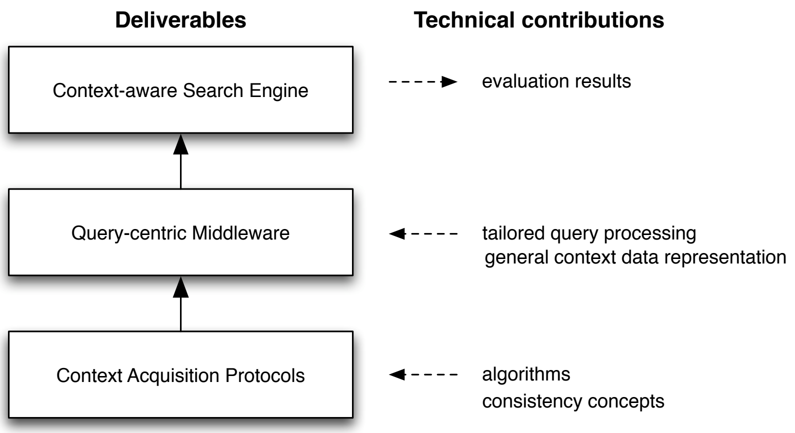

Deliverables
Technical contributions
Context-aware Search Engine
evaluation results
Query-centric Middleware
tailored query processing
general context data representation
Context Acquisition Protocols
algorithms
consistency concepts

Figure 1.1: Dissertation Contributions

user. This means that as a user is collecting context information, new results may become available as a result of environmental changes or old results may be rendered unusable. The typical query model must be extended to allow the query-centric middleware system to notify an application when changes in the environment occur that may impact the availability and pertinence of delivered query results. Though previous work has taken a periodic polling approach to notifying the application of such occurrences, a reactive approach can be more beneficial in helping an application developer to achieve the desired application semantics and can be more efficient than a periodic approach. Second, collecting all available context information can be expensive. Often, context-aware programs utilize only a specific type of information that originates within a small surrounding area or from applications which have particular properties. The execution of a query should occur within an application-defined scope so that an application only receives responses that originate from context providers that it deems to be desirable. Third, the network is comprised of many different kinds of devices, each serving as a platform to a variety of applications. This kind of environment results in a collection of potential context information in a wide range of data formats. The query-centric middleware should allow applications to provide and retrieve different kinds of context information that are created in various formats.

Figure 1.1 depicts the contributions of this dissertation. On the left are the software deliverables that result from this study. These include protocols for acquiring context from the ad hoc network that are reactive to changes in the environment, a query-centric middleware system that supports the use of virtually any kind of information as context data, and a context-aware search application that demonstrates the use of the query system.

The remainder of this dissertation is organized as follows. Related work is presented in Section 2. A query-centric model is introduced in Section 3, and a middleware system which provides an implementation of the query-centric model to provide simplified context interactions to the application developer is introduced in Section 4. Section 5 shows how the the query-centric middleware system can be used to develop applications through discussion of the development of a context-aware search application for mobile ad hoc networks. Section 6 presents protocols that are incorporated into the middleware which dynamically propagate a query to an application-specified portion of the ad hoc network and adaptively delivers query results. Conclusions appear in Section 7.

Chapter 2

Related Work

This section introduces context-aware computing and discusses currently available solutions that support simplified development of context-aware applications. Because this research takes a query-centric approach to simplifying application development, mobile database systems and query processing systems for sensor networks are reviewed and compared to the approach outlined in this proposal. The section concludes with a description of semantic routing, which is a method of supporting a query-centric abstraction at the protocol level.

2.1 Context-Aware Computing

Context-aware computing, in which applications sense and adapt to changes in the surrounding environment, has been identified as an appropriate design strategy for applications that operate in highly dynamic environments. Over the years, a number of context-aware applications have been deployed. Context-aware office applications such as Active Badge [14] and PARCTab [38], 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 [30]), tour guides which adapt displays according to a tourist's location and interests (e.g., Cyberguide [1] and GUIDE [9]), and context-aware note tools which attach environmental information such as time and temperature to observational notes (e.g., FieldNote [32]).

Though these applications are designed for more static networks, the context-aware paradigm offers the ability to opportunistically take advantage of the resources available in the surrounding environment; as such, facilitating context-awareness has been of significant interest to the mobile ad hoc network research community. In many of the applications mentioned above, the use of context was limited to the kind of information that the application expects to use. Moreover, the solutions for acquiring and using context were created from scratch each time. For instance, the tour guide applications considered only how to acquire and use location information from a particular kind of device to determine what information to display to the tourist. In contrast, the approach in this dissertation will encapsulate tasks related to context acquisition and management within a support infrastructure that can be used by an application programmer to use virtually any kind of information as context for behavioral adaptation.

2.2 Support Infrastructures for Context-Awareness

Others have also attempted to encapsulate context acquisition within frameworks, toolkits, and middleware systems supporting context-aware application development. The Context Toolkit [33] was designed to simplify context interactions by separating context acquisition tasks from the application. With the Context Toolkit, applications can discover and use context widgets, software components that deliver context information. A number of reusable context widgets are provided as part of the toolkit. An infrastructure with a similar goal is the Context Fabric [15], which presents an application developer with services that are used to acquire the desired context. While these support systems simplify interactions with sensors, the programmer must still know the source of data to access and operate on it. In an ad hoc network, the open and dynamic nature of the environment makes it unreasonable to assume advance knowledge of the identities of data sources; applications for use in such scenarios require a highly decoupled method of data access.

Middleware systems based on the concept of the Linda tuplespace coordination model have been introduced to provide decoupled, context-aware communication in the presence of mobility, including LIME [26], MARS [7], TuCSoN [28], Limone [11], and

EgoSpaces [19]. In all of these systems, the contents of the tuplespace available to an application can be thought of as its context. The systems vary in their purpose and capabilities. MARS and TuCSoN were designed to support decoupled communication among mobile agents. In these systems, a tuplespace is associated with each host in the network; agents interact by migrating to a host and using the tuplespace. LIME was the first to introduce reactive and transiently shared tuplespaces for use in mobile ad hoc networks. In the LIME model, each agent is associated with a tuplespace. When two agents become able to communicate (i.e., they are co-located or located on hosts within communication range), their tuplespaces become logically federated. Therefore, both agents share a symmetric context. Limone and EgoSpaces introduced the notion of asymmetric contexts. As in LIME, each agent is associated with a tuplespace. However, these systems allow the agent to provide policies that dictate in which portion of a shared context it is interested.

While these systems are useful for managing underlying coordination needs in mobile environments, they are tied to the tuplespace abstraction. This limits an application programmer's ability to interact with context using only tuplespace operations and the semantics they offer. Consider, for instance, a disaster recovery application which needs to display information about the most seriously injured victims within the area so that those victims can be treated on-site and transported to a local hospital. Using tuplespace operations, the programmer would use a tuplespace retrieval operation to collect all injury information. The programmer must write a function that sorts the data to find the most severely injured victims. This sorting of data would be unnecessary with a database-like query that returns the set of maximumally valued items. Moreover, without a careful and preconceived design of a planned interaction scheme between the providers and users of victim information, the disaster recovery application would not be able to update the display to reflect the fact the reported victims are no longer the most seriously injured in the area (i.e., after transport) when implementing the application using tuple space operations.

2.3 Mobile Databases

The idea of using a query language to access a collection of data organized in a database originated decades ago. The continued use of databases for data management indicates the successful adoption of this particular abstraction. As computing environments have evolved, so has the notion of a database. In response to the availability of network communication, databases became distributed. As devices become increasingly mobile, the research community has responded by investigating the deployment of databases in mobile and peer-to-peer network settings.

Over the years, researchers have explored the deployment of database systems in mobile, dynamic environments [35, 29, 2]. Mobile database systems 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 a related vein of work, recent research has resulted in the development of data management systems for peer-to-peer (P2P) network environments. For example, AmbientDB [4] provides a relational database management model that uses Chord [36], 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 interesting 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. Data is augmented with metadata, allowing queries to be issued in a content-based manner and eliminating the need to merge schemas. An interesting design choice is PeerDB's use of mobile code to propagate a query and return its results.

The research presented in this dissertation is based on the belief that a database-like abstraction offers a convenient way to acquire context data distributed across an unknown and changing environment, with queries providing a descriptive and content-based method of interaction. In contrast to the work cited above, the work here does not actually create and maintain a database in our approach. Instead, an illusion of a database is created which encompasses the context data available within the mobile ad hoc network and provides the application developer with that abstraction to access context data; behind the scenes, queries will be executed in a distributed fashion over the ad hoc network. Creating and maintaining a database distributed across the ad hoc network is also overkill for the supporting context acquisition; the goal of the work is to simply hide the details of managing access to remote context information. Unlike in mobile database systems, there is no need to replicate data since the proposed system deals with context that has a spatial and temporal nature. In the view presented in this dissertation, data is either available as context to any application or it is not—there is no notion of ensuring the availability of context data across the entire network.

In addition to this divergence in philosophy, my research differs from the aforementioned systems in a number of ways. First, many mobile databases are targeted for nomadic network environments in which a mobile device temporarily disconnects from the wired network only to reconnect later, and disconnection is the exception. As such, many of the proposed solutions for data management are not applicable to mobile ad hoc networks. Second, the mobile ad hoc network can grow large and the amount of context delivered to an application can be overwhelming. To reduce performance penalties and to take advantage of the notion that applications require very specific kinds of contexts, my approach seeks to limit the execution of queries to a portion of the network deemed to be important to the user as context. While PeerDB provides the ability to limit queries to nodes which have recently provided query results, that approach is not generalizable to accommodate a wide range of tailored contexts based on arbitrary properties of entities within the ad hoc network. Finally, applications operating in ad hoc networks may require the ability to evaluate a query over a prolonged period of time as the circumstances of the surrounding environment change. For instance, a context-aware fire alarm that uses temperature data to determine when to sound an alert will need to monitor the changes in the state of the temperature constantly in order to respond quickly to rising temperatures. None of the database systems discussed above support prolonged evaluation of queries, although this is mentioned as future work on AmbientDB. A key feature of the query-centric model presented in this dissertation is that persistent queries can be evaluated reactively to report relevant changes in context to the application.

2.4 Query Processing Systems

Similar in concept to databases are distributed query processors which have been developed for sensor networks, e.g., directed diffusion [17], TinyDB [23], TAG [22], and Cougar [40]. In these systems, the entire sensor network is treated as a single table in a database. Users at designated resource-rich nodes on the fringe of the network can issue SQL-like queries over this "table" to obtain sensor readings. 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 delivers query results proactively to the issuing application (e.g., periodically at a specified sampling rate).

While the concepts of simplifying the task for the end user, optimizing query processing, and eliminating the need to query the network repeatedly are similar to the goals of this dissertation, there are reasons why these approaches are not appropriate for use in ad hoc networks. For instance, these systems require that all data providers submit information in a uniform data record structure. This requirement is unreasonable in an open, dynamic, and heterogeneous ad hoc network environment. Because of the widely varying nature of context data available in an ad hoc network, it is impractical to assume that the uniform aggregation techniques used in sensor networks can be directly applied to mobile ad hoc networks. Most importantly, to my knowledge, none of these sensor network query processing systems considers adapting to rapid changes in topology, which frequently occur in mobile ad hoc networks.

Recently, researchers have begun to consider the need to address the mobility of the user when issuing a query over a network. The MobiQuery protocol [21] aims to support mobile users that wish to collect sensor readings as they move across a sensor network. Queries are augmented with spatial constraints that dictate the area in which query results are valid as well as temporal constraints that dictate the desired delivery rate and freshness of data. The service relies on a "pre-fetch" approach in which knowledge of the user's anticipated mobility is used to prepare the appropriate sensor nodes for receiving the query.

MobiQuery is an interesting first step to incorporating spatial and temporal constraints for this environment and to using knowledge of mobility to adjust query processing proactively. One of the main differences between the Mobiquery approach and the one outlined in this dissertation is their focus on supplying fresh data by sampling and reporting results periodically. In contrast, the query model presented here will allow for reactive data reporting. This approach has two significant benefits. First, by generating a new message only when data changes, it may be possible to reduce the cost of querying for information. Second, by using a reactive approach, it may be possible to report changes in data that are of interest to the application that a periodic sampling approach would miss. Another significant difference is that the Mobiquery approach provides support at the protocol level for acquiring information, whereas the approach outlined here presents an application developer with a high-level programming abstraction whose execution details are embodied within a middleware system.

In general, the work in this dissertation is distinguished from all of the aforementioned query processing systems in several ways. First, the provision of heterogeneous data in its native format is supported by using metadata to evaluate queries. Second, in-network aggregation of various data types is supported. Rather than "hardwiring" the functionality into the query service, a more flexible and general approach is taken such that the middleware can be customized dynamically with mobile code elements that are propagated and installed across the network. Finally, the query-centric middleware is designed to deliver results to long-lived queries in a reactive manner, where reported query results are sensitive to changes in both data and network topology.

2.5 Semantic Routing

The idea of limiting a query to a relevant portion of data is also utilized in semantic routing, an approach to supporting scalable and efficient search in peer-to-peer networks. With semantic routing, network nodes are assigned to belong to groups according to the kind of data they provide, and queries are routed only to pertinent groups. Generally, groups are defined either implicitly though gossip and observation (e.g., Socialized Net [5], Neurogrid [18], and REMINDIN [37]) or explicitly by giving nodes the ability to join a group. In most of these approaches, groups are defined prior to the submission of a query.

In contrast to these semantic routing protocols, the query-centric approach presented in this dissertation can be thought of as creating a semantic group on the fly that is defined not only by the kind of data that a node in the network offers but also by its locality with respect to the query issuer. The “group” is formed implicitly by the query issuer which requires that receiving nodes meet certain properties to be considered part of the group that evaluates the query. Additionally, as mentioned previously, my approach provides a higher-level programming abstraction than what is offered by a network-level protocol such as semantic routing.

Chapter 3

A Query-Centric Perspective

When using typical programming mechanisms to collect context distributed across the mobile ad hoc network, the programmer must identify the source of the desired context and contact the provider in order to obtain it. The unpredictability of interactions and frequent disconnections in the network setting make this a difficult task for software developers. The goal of this work is to simplify context-aware software development by providing a query-centric abstraction for context acquisition that masks the details of complex network communication programming tasks required to collect and deliver context distributed across a mobile ad hoc network. The use of queries alleviates the developer from such responsibilities, instead allowing programmers to retrieve context declaratively, simply by describing the desired context information.

This chapter introduces a query-centric model for context-awareness. The approach to developing such a model is based on the following principles:

An application should not be burdened with context that is not pertinent to its purpose. Collecting all the context information available to an application can be overwhelmingly expensive in terms of communication and computation, especially in expansive ad hoc networks. This expense may be unnecessary for many context-aware applications. Often, context-aware programs utilize only a specific type of information that originates within a small surrounding area to adapt behavior. For example, a context-aware word processing program may use context information to adaptively manage constant access to the nearest available printer. Though the word processing software could collect information from the entire office building, information that is useful to the application is limited to the areas in the office building in which the user has physical access to printers. Furthermore, the context-aware word processing program requires only information concerning printers and other relevant equipment to adapt its behavior.

For these reasons, the query model introduced here allows users to limit context interactions to occur within a manageable and pertinent portion of the ad hoc network. Developers can describe the desired portion of the network by providing *context policies* that place restrictions on which network nodes can receive and respond to a query. Context policies embody constraints on properties of network links, hosts, and applications and are used by the model to make decisions about how to propagate and execute the query within the network.

An application's view of context should reflect the state of the surrounding environment. Changes in the environment may impact the collection of context information that is available to a user. This means that as a user is collecting context information, new results may become available as a result of environmental changes or old results may be rendered unusable. For instance, a driver may use an application to collect traffic information from surrounding cars on the highway. As cars enter the highway, additional context information becomes available to the traffic monitoring application; as cars exit the highway, the traffic information that they have provided to the traffic monitoring application is obsolete. It is important to ensure that context-aware applications are receiving contextual information that is pertinent to its purpose and that appropriately reflects the current state of the environment. For this reason, our query-centric model offers developers the option of submitting persistent queries, which are evaluated reactively within the network.

An application should be able to provide and obtain many different kinds of information as context. The ad hoc network is comprised of many different kinds of devices, each serving as a platform to a variety of applications. This kind of environment results in a collection of potential context information in a wide range of data formats. Requiring users to alter the way that they represent and store context information necessitates extra effort on the part of context providers and may hinder the sharing of data as a result. Instead, our query model allows programmers to provide many different kinds of context information. A baseline representation of context allows for uniform treatment of context by a support infrastructure, while the use of searchable metadata supports the evaluation of queries and the use of programmer-provided strategies supports tailored, in-network manipulation of query results.

The remainder of this chapter describes how these principles are incorporated into a new query model that manages access to context data transparently in open and dynamic mobile ad hoc network settings. The query-centric model introduced in this dissertation relies on the use of a network overlay data structure to support the execution of queries over the application-specified portion of the mobile ad hoc network. At its most basic level, the general approach to supporting query execution can be described as follows. When an application issues a query, the query is disseminated to hosts within the mobile ad hoc network. As the query is disseminated, a network overlay is constructed in a distributed fashion. At each host, the received query is evaluated against locally available context data stored in an updated context repository, and the constructed network overlay is used to route query replies back to the issuer of the query (also referred to as the *reference host* or *reference application*). In the following sections, we describe in detail how this simple network overlay approach is extended to adhere to the outlined design principles when applied in the constantly changing, diverse setting of an ad hoc network. This section concludes with a summary of the operations that can be issued using our query-centric model.

3.1 Controlling the Scope of Query Execution

As mentioned previously, one way to control the cost associated with query processing across a large ad hoc network is to limit the scope of the query's execution. To support application-controlled scoping of query execution and to ensure that an application only interacts with context that is pertinent to its purpose, our query-centric model allows the application to provide context policies (similar to those introduced in [19]) describing desirable properties of context data, applications, hosts, and network links. For instance, an application used by a member of a disaster response team may supply a context policy that specifies that context information should only be considered if it comes from injury monitoring applications (a constraint on the type of application) running on patient monitoring devices (a constraint on the type of host) that are located within 20 meters of the responder's current location (a spatial constraint on the network).

Each query is associated with a context policy that limits the scope of its execution. When a query is issued by an application, the context policy bundled with that query is used to construct a *network overlay* such that only the context contributors that satisfy the constraints detailed in the policy are included in the overlay. Query execution is limited to nodes within the overlay. The following sections describe each component of the context policy and how it contributes to the construction of the network overlay. An example is also presented that illustrates the use of context policies to construct a tailored network overlay for scoped query execution. We then describe how the model allows for additional control to be placed over the scope of the network using propagation policies. A presentation of query operations provided by the model concludes the section.

3.1.1 Network Constraints

Network constraints capture restrictions on the properties of the network links to restrict evaluation of a query to a subnet of the entire mobile ad hoc network. Some applications may require that queries be propagated over links that have particular properties. Such properties may include latency, bandwidth, throughput, delay, jitter, and so on. Network constraints may also be used to capture a spatial constraint over the area of the network in which a query should be evaluated or may place some requirements on properties of the links. Spatial constraints may be given in terms of network distance (e.g., number of hops) or physical distance (e.g., within a number of meters, city blocks, etc.).

Because network constraints are used as part of a context policy to scope the mobile ad hoc network by constructing a network overlay for query execution, the network constraints must be provided in a way that allows for the construction of a subnet of the network. The remainder of this section discusses the representation of network constraints and how they are used to construct a network overlay.

Using network constraints to construct the network overlay. The first step in constructing the network overlay is to utilize the network constraints portion of a context policy. As the query is issued, a distributed protocol is initiated which constructs an overlay that corresponds to the desired context defined by these constraints. As a starting point, we rely on a network overlay data structure that is commonly used in ad hoc networks for routing: a shortest cost path tree. We extend the ideas proposed in [12] in which a spanning tree is constructed and maintained in an ad hoc network that corresponds to constraints on network properties.

The basic premise of their idea is to treat the mobile ad hoc network as a weighted graph, representing hosts as nodes and network links as edges. Edge weights are assigned that correspond to the host and network link properties that are considered in the network constraints. An application's network constraints are provided as a metric that is used to evaluate the cost of a path in the graph and a bound on the metric that restricts the inclusion of hosts to those that lie on a path that has an allowable cost. The metric is defined as a recursive function that uses the cost of the path to the previous node and the cost of the link to a one-hop neighbor in order to calculate the cost of a path to that neighbor. To ensure that a bounded overlay can be formed, the metric is required to be strictly increasing. As discussed later, we extend this network overlay to include host and application constraints as well.

At this point, the problem of constructing the network constraints overlay has been reduced to a problem that has a known solution; we essentially use a distributed shortest path algorithm to construct the spanning tree overlay, where the cost of the path is determined by the network constraints metric with the added caveat of including a bound to halt execution of path construction. To construct the tree overlay, the query initiator uses the metric to determine the cost of including neighboring hosts in the network overlay. If the cost of the path to a neighbor is within the bound, the query initiator propagates the metric, bound, and current path cost to the neighbor along with the query. The recipient will record the information and will use the metric and the current cost of the path to determine which, if any, of its neighbors should also receive the query. This continues until the bound is reached and the query cannot be propagated further. Because the goal is to form a shortest cost path tree, a node that receives a query from another with a shorter cost path to the query issuer will be chosen as its new parent, and the updated path cost information will be propagated to its neighbors whose new path cost falls within the bound.

3.1.2 Host Constraints

Each host in the network provides a profile describing its characterizing traits. For example, a host's profile may include attributes which include its unique host identifier or platform type. The profile may also declare characteristics that detail the host's resource specifications, such as remaining battery power, processing capacity, or available storage. Profiles may even be used to provide information about the quality of a host's past contributions to the results of previously submitted queries. The profile is represented as a collection of named, typed attributes. For example, a rescue vehicle's host profile may be captured as:

$$\begin{gathered}\langle\ (\text{name} = \text{HostID, type} = \text{integer, value} = 123),\\ (\text{name} = \text{HostType, type} = \text{string, value} = \text{“firetruck”}),\\ (\text{name} = \text{InjuryTransport, type} = \text{boolean, value} = \text{true}),\\ (\text{name} = \text{Capacity, type} = \text{integer, value} = 2)\ \rangle\end{gathered}$$

Host constraints are provided as part of a context policy that is submitted along with a query. The host constraints are used to determine which of the hosts that meet the network constraints may permit the applications it supports to provide results for a query. These constraints are encoded as a constraint function that is evaluated over a host profile, i.e., as a collection of constraints on the names, types, and values of host attributes. When a constraint function includes multiple constraints over multiple host properties, the host profile must satisfy all of those constraints in order for the host to satisfy the constraint function. For example, a rescue worker at a disaster recovery site may need to perform a query to see which rescue vehicles are available to transport two victims. A host constraint specifying these host-specific needs could be represented as:

$$\begin{gathered}\langle\ (\text{name} = \text{InjuryTransport, type} = \text{boolean, value} = \text{true}),\\ (\text{name} = \text{Capacity, type} = \text{integer, value} \geq 2)\ \rangle\end{gathered}$$

3.1.3 Application Constraints

Similarly to hosts, every application provides a profile that catalogs its distinguishing characteristics. An application's profile may include an application identifier, expected lifetime, system support requirements, etc. It may also provide detailed information about the interface that it provides and the native format of its data. As with host profiles, an application profile is represented as a set of named, typed attributes.

Queries are packaged with context policies that include constraints on properties of applications in order to restrict which applications may provide context data as a result of a query operation. As before, constraints are provided as a constraint function that is evaluated over the profile as the query is propagated. A rescue worker at a disaster recovery site, who needs to query the environment to acquire information from a particular patient monitoring application she previously encountered, may provide the following application constraint:

$$\begin{gathered}\langle\ (\text{name} = \text{AppID, type} = \text{integer, value} = 456),\\ (\text{name} = \text{ApplicationType, type} = \text{string, value} = \text{``patient monitor''})\ \rangle\end{gathered}$$

Imposing host and application constraints on the network overlay. As a spanning tree is constructed using the network constraints of a context policy, host and application constraints in the context policy are imposed at each node in the network overlay. Query evaluation occurs only over context data provided by applications meeting the specified application constraints running on hosts meeting the specified host constraints. Though nodes in the spanning tree overlay that do not meet the host and application constraints cannot provide results to a query operation, the host can act as a "virtual node" in the network overlay. This simply means that the host acts only as a router of query replies without actually contributing any context data owned by applications it supports, to the result of the query.

With this approach to defining context policies, arbitrary constraints can be defined over properties of the mobile ad hoc network and used to build the overlay data structure. In the next section, we use an example application to further illustrate the use of context policies in the construction of a network overlay to support scoped query execution.

3.1.4 Application Example

To demonstrate the use of a tailored context, consider a disaster recovery scenario in which triage is employed to 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 the assigned injury classification, ranging from injuries that need immediate attention to those for which treatment can be postponed. Rescue teams are assigned areas within which they must arrange transport for the most severely injured first and provide as much on-site treatment as possible for these victims until transport is available. The rescue team members use PDAs with wireless communication capabilities to coordinate activities and to obtain and display the status of victims and emergency medical personnel. An emergency medical technician (EMT) is selected by the rescue team member to treat the most seriously wounded victim until transport arrives. An EMT's assignment may change as the status of injured victims within the context changes. After a rescue crew member arranges on-site treatment for a victim, he must arrange for the victim's transport to a hospital. As victims are transported, they are removed from the context of the application. As new victims are discovered and their injuries evaluated, they are added to the context. Figure 3.1.4 illustrates this application. In the figure, the disaster site lies within the large oval. A rescue crew member (the encircled cross) uses a PDA that runs an application to assign to the most seriously wounded victims in the designated area (the dashed box) on-site treatment and ambulance transport to a nearby hospital. Victims are shown as circles, with seriousness of injury reflected by darker shading.

Figure 3.2 illustrates a context definition and the resulting spanning tree for the disaster recovery application. 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 a PDA carried by the rescue worker who is responsible

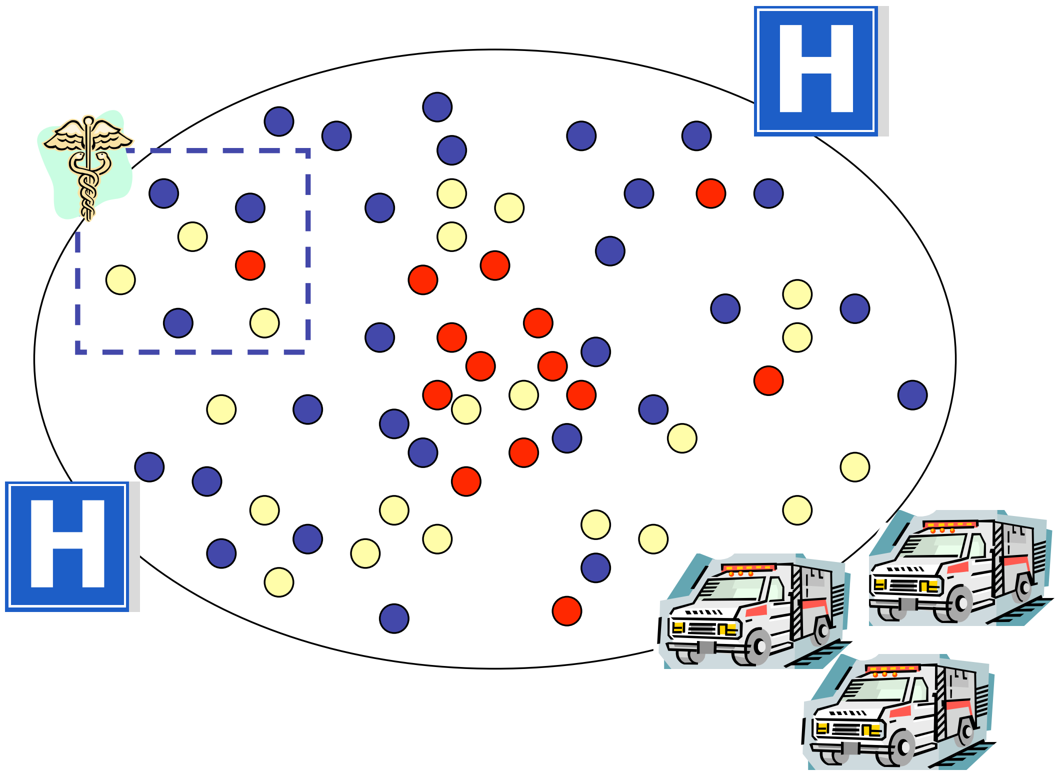


Figure 3.1: Disaster Recovery Scenario

for treating and transporting the wounded in a particular area of the disaster site. The rescue team member needs to query injury monitoring applications running on patient monitoring devices within 20 meters.

To construct such a context for the rescue worker, the application developer provides a context policy consisting of network, host, and application constraints. To provide the network constraint, the application developer may, for example, define a metric that adds the previous physical distance and hop count to evaluate the current distance and hop count, and specify 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 3.2a, at time t, the context defined by the network constraint alone is a, b, c, where a, b, and c are hosts

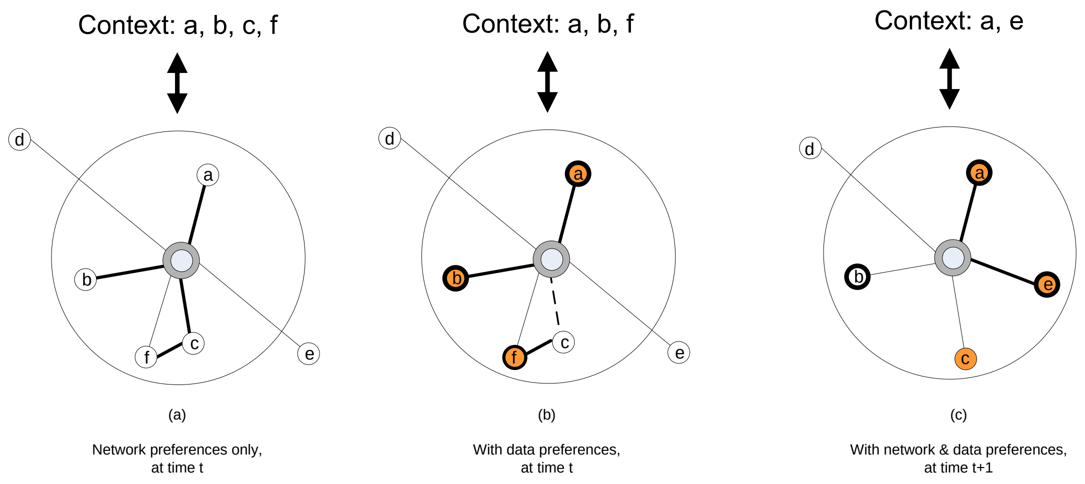

Context: a, b, c, f
a
b
c
d
e
f
(a)
Network preferences only, at time t
Context: a, b, f
(b)
With data preferences, at time t
Context: a, e
(c)
With network & data preferences, at time t+1

Figure 3.2: A Network Overlay Corresponding to a Disaster Recovery Team Member's Context Definition

that meet the context specification provided by the reference application (the doubly ringed circle). The constructed spanning tree network overlay is depicted using bold lines. Figure 3.2b 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 constraints by shaded circles. Notice the dashed line between the reference host and **c**, which 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 route queries and responses to and from **f**. As shown in Figure 3.2c, the network overlay changes over time to reflect changes in the environment that impact the satisfaction of the context policy. The next section discusses how to address such changes as well as any changes in data that may occur in the network.

As mentioned previously, applications associate a context policy with each query. The application may define several context policies to be associated with queries that acquire different kinds of information. For instance, the disaster recovery application may also issue a query to acquire information about vehicles that are available to transport a victim to a hospital. The context policy associated with that query could stipulate that only vehicle applications running on ambulances and fire trucks within a 2 block radius should provide information about their availability. The application may also reuse context policies across multiple queries, or change the context policy associated with a query over time.

The algorithm for constructing a network overlay which incorporates network, host, and application constraints will be detailed in Section 6.

3.1.5 Propagation Policies

Typically, a query is propagated to all nodes within the application-specified query scope, triggering a response only when the query reaches a leaf node in the overlay network data structure. Answers are collected at the reference host from all applications in the context before returning the result to the reference application. However, some applications may have needs which are not best served by this kind of approach to query propagation and execution. For example, an application that wants to determine the existence of a particular data item in the context may issue a query construct over the context that determines the existence of a particular piece of context data. Since this particular query does not necessarily need all data items available in the context that satisfy the query, it may be possible to reduce the communication costs by controlling the propagation of the query.

Rather than assume the use of a single propagation method for all queries, our query model allows the developer to consider the tradeoffs associated with each particular propagation method and to specify the desired propagation method each time that a query is issued. Generality, flexibility, and extensibility of this approach are achieved by requiring that a query propagation policy be supplied with each query submission that dictates how the query is to be distributed within the constructed network overlay. Evaluation of the query propagation policy at each node in the overlay should result in either propagating the query further over the context or initiating the query reply process. Essentially, a propagation policy can be captured as a function that takes a set of neighboring hosts that meet the context policy requirements to be part of the network overlay and returns a (possibly empty) set of neighbors that should actually receive the query.

For example, one option appropriate to determining the existence of a particular data item is to propagate the query using controlled flooding, which has the potential to reduce communication costs but requires an extended execution time. With a controlled flooding propagation policy, 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 to 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. At each host, the propagation policy defines the set of hosts to which it will propagate the query:

$$PropagationSet(N, h, d) \equiv \langle\ \mathbf{set}\ n : d \notin D(h) \land n \in N :: n\rangle$$

where N is the set of neighboring nodes that satisfy the context policy associated with the same query as this propagation policy, h is the host on which the propagation policy is being evaluated, d is the piece of context data requested in the query, and $D(h)$ is the local set of context data at host h.

Other query propagation policies 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, which results in a query that travels along a single path in the network. A number of other useful query propagation policies specific to the needs of applications remain to be discovered. Our query model's approach to query propagation policies is general, flexible, and extensible, allowing any imaginable propagation policy to be provided with a query.

3.2 Reflecting Environmental Change

An issue of particular interest when designing a query-centric middleware 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. It may, however, be more practical to monitor the environment and notify the application of changes that impact previously delivered query results.

To address these issues, the query-centric model provides *persistent queries* which emulate the semantics of subscriptions in publish/subscribe systems. In publish/subscribe systems, network nodes can register interest in events by submitting a subscription to the event service. Each time that an event occurs, the event service delivers event notifications to interested parties. A subscriber will continue to receive event notifications for that event until it indicates to the event service that it is no longer interested in the event, by unsubscribing for that event. In contrast to the publish/subscribe model, the focus of persistent queries is on data state rather than the occurrence of predefined events. This kind of interaction is especially suitable for mobile ad hoc networks, since it is an open environment in which hosts can quickly become connected or disconnected at any moment. When publish-subscribe systems are used in such scenarios, hosts may miss pertinent events due to the unfortunate timing of forming or dissolving a network connection. Although some systems support disconnected event delivery (e.g., SIENA [8] and JEDI [10]), most only are able to buffer the events for a pre-determined period.

In our query-centric model, an application can register a persistent query to acquire context data as it becomes available in the mobile ad hoc network. The approach used in our query-centric model to service persistent queries is inspired by the work presented in [12], which introduced a protocol to support the construction and reactive maintenance of a spanning tree overlay by imposing weights associated with network links, using a metric and bound to calculate the shortest path to each node, and reacting to weight changes that impact the cost of a path to adjust the overlay accordingly. In our model, a similar reconfiguration protocol that mends the network overlay is used in addressing the frequent disconnections that are likely to occur within the query's associated network overlay due to the mobility of hosts and to changing properties of hosts and agents.

The persistent queries in our query model handle the reconfiguration of the network overlay to propagate the query to hosts that join the overlay and to prevent hosts that are no longer part of the overlay from delivering replies. Our persistent queries also go beyond network overlay maintenance to provide an application with updates that detail several different kinds of 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 a query.

Using this extended notion of persistent queries, the application can expect to continue to receive new results to a persistent query as it becomes connected to new nodes that provide the desired context or as the context data is made available by known providers. Furthermore, the application can expect to receive notice when the data is no longer available, due to disconnection of the provider of the reported context data item, or modification or deletion of a reported context data item. The application will receive context data and notifications of changes to reported results (until it is no longer interested and deregisters the query), which are processed by context data notification handlers that are registered with each persistent query.

3.3 Accommodating Heterogeneous Context Data

Context information may be obtained from various sources in numerous data formats. To support the free exchange of information, it is important to allow context data providers to share their data without extra effort; they should be able to provide context without changing the way that they represent and store information. Programmers should be allowed to provide many different kinds of context information, and a query-centric model should take an extensible approach to accommodating the various forms of information. Users of context data should still be able to utilize the acquired context data in a meaningful way, and should be able to process the context data in accordance with its intent and purpose using the original format if desired.

To this end, our query-centric model provides a baseline representation of context within the query service to allow for uniform treatment of context data by the support infrastructure. This essentially results in treatment of context data as a “black box”; a system which implements the query-centric model would have no knowledge of the contents, format, or purpose of the context data and would only be able to deliver the context data. It would simply collect all context data items within the query scope specified by the application programmer and return them to the query issuer. To support meaningful evaluations of queries based on the *content* of context data, the query-centric model employs the use of metadata that describes each context data item. The query operates over the metadata of context data items at each node to determine whether or not the data item should be returned as a query result. Metadata is represented as a set of named, type fields. The content-based data request embodied in a query is essentially a function that evaluates constraints over metadata fields. All constraints in the data request must be satisfied for the metadata to satisfy the constraint function. When the metadata constraint function is satisfied, the context data item associated with the metadata is returned to the query issuer as a query reply.

With naive evaluation of queries, all context data items whose metadata meets the data description in the issued query are collected across the network and returned to the query issuer. The collected replies are evaluated on the local host according to the semantics of the query construct and the result is returned to the application. Some queries, such as those that ask for the minimum data value, require that all the available data be examined but that only a single result be returned to the application. Executing the query in a naive fashion can be cost-intensive in terms of communication on smaller mobile devices. To reduce communication costs, query systems often perform sophisticated processing on data items sent as query replies within the network as the reply is delivered to the query issuer [22, 40, 17], e.g., nodes in the network may aggregate query responses and communicate only the aggregate responses as the query reply.

Because the query model supports the use of several different kinds of context data formats and there are a number of algorithms available to perform in-network processing of data, the model offers the option of choosing an in-network processing scheme that best suits the needs of the query and the application. To support arbitrarily defined processing algorithms, the query model allows for tailored manipulation of returned context information in a manner specific to the data format through the use of programmer-provided query processing strategies. Such strategies are supplied with each query as it is submitted. As the query is propagated, its query operation is evaluated locally on the context data's associated metadata and the query processing policy is evaluated on the context data item to perform in-network aggregation or a more sophisticated form of query processing.

3.4 Query-Centric Operations

The query-centric model introduced here is intended to simplify the programming tasks associated with the acquisition of context across a mobile ad hoc network. The model presents a database-like abstraction to the application for acquiring context. An underlying support system that implements the model manages the processing of the query across the mobile ad hoc network and hides these details from the application programmer. To the programmer, it appears as if a query is being executed on a local data repository.

When a programmer wishes to collect data from across the network, she can choose to issue a simple query or a persistent query. The simple query constructs the network overlay used to scope its execution as the query is propagated, uses the overlay for the delivery of query replies, and then discards the overlay. To issue a simple query, an application programmer uses a **sendQuery** operation. When using the **sendQuery** operation, the programmer must provide a context policy, a query propagation policy, a query processing policy, a data constraint function that describes the desired data, and the specified query option to be performed. Query operations that are explicitly

Table 3.1: Typical Query Operations

Operation Name	Definition
GET	Retrieves the data matching the specified data pattern, if it exists. If more than one match exists, one is selected non-deterministically and returned.
GETALL	Retrieves all data matching the specified data pattern, if it exists. If more than one match exists, all are 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 items that 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 items that 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 items that 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 items that match the specified data pattern. To determine the sum, a method of quantification must exist over the specified data type.

supported by the simple query model include GET, which returns a single result per context data provider and GETALL, which returns all query results. Additional query operations can be supported through the use of the aforementioned query processing policies. A set of commonly used query operations is listed in Table 3.1.

A persistent query continues to deliver new context data results and notifications of modifications or unavailability of previously reported results. An application programmer uses the **registerQuery** operation to register a persistent query. As when using the **sendQuery** operation, the programmer must provide a context policy, a query propagation policy, a query processing policy, a data constraint function that describes the desired data, and the specified query option to be performed. In addition, the programmer must provide a policy that dictates how an application should respond to notifications of new, modified, or unavailable results. For persistent queries, the network overlay is maintained and the context data is monitored until the persistent query is deregistered. An application developer uses the **deregisterQuery** operation to terminate evaluation of the query.

Chapter 4

QueryME: A Query-Centric Middleware

This chapter introduces the QueryME middleware. QueryME (Query in an ad-hoc Mobile Environment) encapsulates the query-centric model introduced in the previous chapter, and aims to simplify the application development tasks associated with the acquisition of context distributed across a mobile ad hoc network. QueryME addresses the technical challenges that arise due to the intrinsic nature of ad hoc networks and targets solutions to supporting context-aware programming to that environment. The middleware addresses the need of the application programmer to acquire only context that is pertinent to the task at hand. The developer is provided with control over the scope of a query using context policies and query propagation schemes. The middleware also addresses the fact that data may be continuously changing in ad hoc networks by providing a special query construct which allows for reactive query processing and application notification of changes to the network and reported data. This construct, the persistent query, has semantics which are comparable to a subscription in the publish/subscribe paradigm. Finally, the middleware addresses the need to accommodate several different data types, providing a baseline representation for internal query processing by the middleware and supporting reply processing that can be tailored by the application to operate over the native format of reported context data results.

One design option for addressing such issues related to controlling query execution is to hard-code all possible customization configurations into the QueryME. Queries would be parameterized with a set of customization options that the programmer chooses from a fixed set. The middleware would simply use the provided parameters in order to determine how to process the query. The benefits of such an approach is its simplicity; such a straightforward approach to designing the middleware system can be implemented in an efficient manner. However, adhering to such a design implies advance knowledge of all possible specializations that any context-aware application might ever require. For this reason, QueryME relies 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 results in a flexible, extensible, and expressive middleware, since mobile code elements can be interchanged and new mobile code elements which implement arbitrary query control policies can easily be included. Such adherence to a design principle of generality allows the middleware to support a wide range of existing context-aware applications as well as those of the future. While the introduction of mobile code elements does introduce overhead, the overhead corresponds to the complexity of the encoded solution. We believe that most specializations, while powerful, will be relatively simple and will introduce minimal overhead.

We begin this chapter by giving an overview of how the QueryME middleware evaluates a query. We then discuss the architecture and implementation of QueryME. The chapter is concluded with a presentation of the interface that a context-aware programmer uses to construct and issue queries over a tailored context.

4.1 An Overview of Query Execution in QueryME

An application can submit a request for context data by using the query operations provided by the QueryME middleware. First, the application constructed using QueryME registers itself with the QueryME query manager on the local host, providing its application profile during registration. The application then packages a context policy (network constraints consisting of a metric and bound evaluated over network link properties, host constraints, and application constraints), a query propagation policy, and a reply processing policy with its content-based request for context data.

An additional policy for handling notifications is also bundled with the data request when a persistent query is registered. Each of these policy specializations is captured as a mobile code fragment, and is provided to the query manager as a parameter to the query operation.

When the query manager receives a request from an application to issue a one-time or persistent query, the QueryME query manager creates a query to be issued over the specified context. The query includes the mobile code specializations, the content-based data request, the identifier of the query initiator; the path along which the query has been propagated; and the unique identifier of the query. The query manager extracts the mobile code elements and installs them locally for execution. The query manager then 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 evaluate the network constraints metric, the query manager uses a network discovery package (described in Section 4.2.1) to find the set of one-hop neighbors and an environmental monitoring package (described in Section 4.2.2) to gain access to monitors on local and remote hosts 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 neighboring 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, or if existing contributors must be removed.

Once a set of neighbors has been determined to contain only 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*. The context children is the set of neighboring nodes that make up the current node's children in the network overlay spanning tree. If the returned set of context children is empty, the propagation process stops. Otherwise, the query manager disseminates the query and its mobile code specializations to the actual context children.

Next, the host constraints of the query are evaluated at the local host using the host profile owned by the query manager. If the host constraints are satisfied, then the content-based request for data is processed over a data repository that holds the context data supplied by applications residing on the host. For each local result, the application constraints of the query are applied to the application's profile. The profile is obtained from the query manager using the application identifier that is stored as part of a context data element. If the application satisfies the constraint, the local result is packaged as a query reply. If the query is a persistent query, the query is continuously evaluated over the local data repository and reports of newly available data satisfying the query or of the removal of results reported to the query issuer are also packaged as query replies.

Finally, the mobile reply processing scheme for the query is invoked to return the reply to the query issuer. When the query manager receives a request to send a reply from a mobile reply processor, it uses the query ID associated with the reply to find the appropriate return path over the defined context and sends the reply to the query manager on the host that is the next hop back along that path.

4.2 QueryME Architecture and Implementation

The architecture of our QueryME implementation is presented in Figure 4.1. As shown in the figure, we rely on the use of additional units of functionality to support the operation of the query service. For the ease of discussion, we refer to these units of functionality as components, though we do not imply that these are "software components" in the technical sense of the term. These components are used to deliver messages in the ad hoc network (the *messaging* component), discover network neighbors (the network *discovery* component), and to monitor environmental properties used to define the network constraints portion of the context definition (the *monitoring* component). We assume the existence of the physical ad hoc network and a

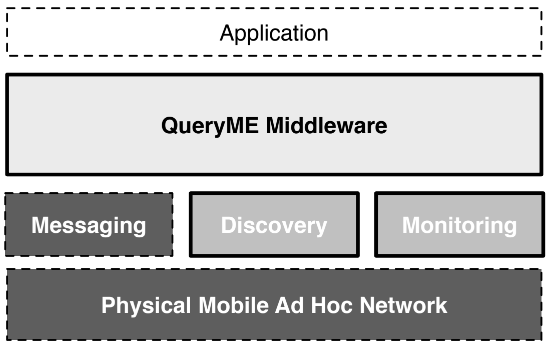


Figure 4.1: The QueryME Architecture

message passing mechanism; we utilize an external network discovery package developed in [19] and constructed a supporting environmental monitoring package which can be used independently of the QueryME middleware system. The core component of the QueryME middleware distribution, the `QueryManager`, utilizes these support packages to deliver the functionality of the QueryME operations offered to context-aware application programmers. Network discovery, environmental monitoring, and query manager components are discussed in detail in the following sections.

4.2.1 Discovering Network Neighbors

In the QueryME middleware, each query manager uses the network discovery component to discover its surrounding neighbors. By separating the network discovery behavior from query management tasks, we allow for the most appropriate network discovery mechanism to be employed with QueryME. In addition, by separating network discovery concerns from query management, we allow for reuse of the network discovery component as a building block in the development of other applications or infrastructures. By default, the QueryME middleware utilizes a network discovery package that provides an implementation of a simple discovery mechanism: one that informs a host of all one-hop neighbors, i.e., other hosts in direct communication. The details of the network discovery component used to implement this default discovery behavior are discussed below.

```
public class DiscoveryServer implements BeaconListener{
   public static DiscoveryServer getServer();
   public void initialize(int beaconPeriod, String mcastAddr,
      int mcastPort, int addPeriod, long disconnectTime,
         long removePeriod, int firstPort);
   public void start();
   public void stop();
   public void addDiscoveryListener(DiscoveryListener dl);
   public void removeDiscoveryListener(DiscoveryListener dl);
}

public interface DiscoveryListener {
   public void neighborAdded(NeighborAddedEvent nae);
   public void neighborRemoved(NeighborRemovedEvent nre);
}
```

Figure 4.2: The Network Discovery Application Programmer Interface

Our network discovery component is implemented as a Java package containing a discovery server and a discovery listener interface. At each host, a network discovery server periodically beacons the surrounding hosts to discover the current set of neighbors. The length of the beaconing period can be adjusted to suit the environment. For example, in a rapidly changing network, the discovery server may beacon more frequently so that the neighbor set managed by the server more accurately reflects the frequent connections and disconnections in the environment. Discovery can also be parameterized with policies that govern when to add or remove a neighboring host from the set of neighbors.

The interface of the network discovery component used by the QueryME middleware is shown in Figure 4.2. Essentially, an application that wishes to utilize the network discovery package initializes and starts the `DiscoveryServer` and uses the server's `addDiscoveryListener` method to register a `DiscoveryListener`. When a host is added to the set of neighbors, the discovery server calls the `neighborAdded` method for all registered `DiscoveryListener`s. Similarly, the network discovery server calls the `neighborRemoved` method for all registered `DiscoveryListener`s when a host is removed from the set of neighbors.

In the QueryME middleware, each query manager uses a network discovery server and a discovery listener to discover the set of neighboring hosts. The query manager keeps a consistent list of neighbors by implementing the `DiscoveryListener` interface of the network discovery component in order to listen for events signaled by the discovery server to indicate the addition and/or removal of one-hop neighbors. In this particular implementation, the discovery policy adds a host to the set of neighbors if a beacon is received from the host for a number of consecutive beacon periods (by default, two beacon periods). A host is no longer considered as a neighbor when it is not heard from for a number of consecutive beacon periods (by default, three beacon periods).

The use of more sophisticated discovery components may be beneficial. For example, conserving energy while discovering useful neighbor sets might be the driving design motivation. Birthday protocols [25] have been developed for static ad hoc networks where certain assumptions hold about the relationships between the devices. These networks are still quite dynamic, however, because nodes can be deployed and fail at various times, and require constant discovery. Group communication mechanisms for mobile networks [16] can extend a node's neighborhood to include nodes to which it is not directly connected. Such protocols create a list of nodes with which a group member can reliably communicate. The integration of group communication protocols allows applications to access sensing devices available throughout the group instead of restricting remote sensing to one-hop neighbors. Due to the decoupling between the network discovery component and the QueryME middleware, the most appropriate policy for discovering neighbors can be selected for use at each host.

4.2.2 Monitoring Environmental Properties

To determine the logical costs of network paths as dictated by the network constraints portion of a context policy, the QueryME middleware requires access to properties of the network links that impact the link cost as defined in the network constraints by the application. For this reason, we developed the CONSUL (CONtext Sensing User Library) package [13], a lightweight solution for monitoring the surrounding environment. CONSUL provides application developers with access to environmental information through a simplified interface. Programmers explicitly define the kind

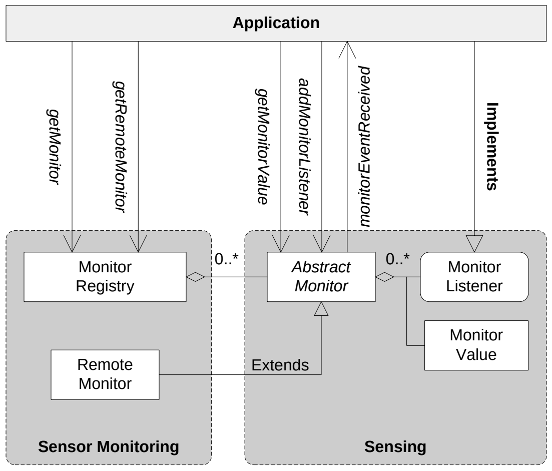


Figure 4.3: The internal class diagrams for the components of CONSUL.

of environmental information of interest and the exact location of the application providing the information. Because CONSUL is designed to operate in mobile ad hoc networks where advance knowledge of interaction partners is an unreasonable assumption, CONSUL relies on the use of the aforementioned network discovery component to support context acquisition. CONSUL can be used as a stand-alone solution for monitoring environmental properties, or as with QueryME, as a building block for creating more sophisticated systems.

Two software units contribute to providing CONSUL's environmental monitoring functionality: the sensing unit and the sensor monitoring unit. Figure 4.3 shows the internal class diagrams for these two components and how they interact with each other and with the application.

Sensing. The sensing component allows software to interface with sensing devices connected to a host. Each device has a corresponding software object (a *monitor*). In CONSUL, each monitor extends an `AbstractMonitor` base class and contains its current value in a variable (e.g., the value of a location monitor might be represented by a variable of type `Location`). An application can react to changes in monitor values by implementing the `MonitorListener` interface and registering itself with the monitor. To ensure that any listeners registered for changes are notified, the monitor should perform these changes through the `setValue()` method in the base class. Applications can also call the `getMonitorValue()` method provided by the base

```
public abstract class AbstractMonitor {
   public AbstractMonitor(String ID, IMonitorValue value);
   public AbstractMonitor(StringID);
   public AbstractMonitor();
   public String getID();
   public IMonitorValue getValue();
   public void addMonitorListener(MonitorListener ml);
   public void removeMonitorListener(MonitorListener ml);
   protected void setValue(IMonitorValue value);
}

public interface MonitorListener {
   public void monitorEventReceived(MonitorEvent me);
}

public class RemoteMonitor extends AbstractMonitor {
   public RemoteMonitor(HostID remoteHost, int localPort,
      String ID);
   public String getID();
   public IMonitorValue getValue();
   public void addMonitorListener(MonitorListener ml);
   public void removeMonitorListener(MonitorListener ml);
}

public class MonitorRegistry {
   public MonitorRegistry(int localPort);
   public void stop();
   public synchronized void addMonitor(AbstractMonitor m);
   public synchronized void removeMonitor(AbstractMonitor m);
   public synchronized void removeRemoteMonitor(AbstractMonitor m);
   public synchronized void removeRemoteMonitor(String ID,
      HostID hID);
   public synchronized AbstractMonitor getMonitor(String ID);
   public synchronized AbstractMonitor getRemoteMonitor(String ID,
       HostID hID);
   public synchronized AbstractMonitor[] getMonitors();
   public synchronized AbstractMonitor[]
      getRemoteMonitors(HostID hID);
}
```

Figure 4.4: The CONSUL Application Programmer Interface

class to obtain these values on demand. The interfaces for the `AbstractMonitor` and the `MonitorListener` classes are shown in Figure 4.4.

Figure 4.5 demonstrates an example class that extends `AbstractMonitor` to collect various information about the status of a particular link. The details related to acquiring and using packet information to calculate link properties are omitted. From CONSUL's perspective, the important pieces are how the extending class interacts with the base class.

```
public class LinkMonitor extends AbstractMonitor{
  public LinkMonitor(...){
    //call the AbstractMonitor constructor
    super("LinkProperties");
    ...
  }
  public void packetReturnEvent(PacketReturnEvent event){
    //use the packet info to calculate link properties
    ...
    double latency = ...
    double bandwidth = ...;
    double throughput = ...;
    // use info to create a LinkProperties object
    LinkProperties linkProperties =
     new LinkProperties(latency, bandwidth, throughput);
    //set local value variable, which will notify listeners
    setValue(linkProperties);
  }
}
```

Figure 4.5: The LinkMonitor Class.

To assist application developers, CONSUL includes several `MonitorValue`s for programmers to use when building monitors or constructing more complex `MonitorValue`s. These value types reside in a library to which application developers can add new types. For example, the library contains an `IntValue` that can be used for sensors whose state can be represented as a single integer value. There are also aggregate values, e.g., `DateValue`, that build on the simple value types. A list of available monitor value types is shown in Table 4.1. In addition to being available for developers to use, they also serve as examples for defining new values. Figure 4.6 shows a class that extends `ArrayValue` to supply information about the

```
public class LinkProperties extends ArrayValue {
  public LinkProperties(double latency, double bandwidth,
    double throughput) {
    super(new IMonitorValue [] {
      new DoubleValue(latency), new DoubleValue(bandwidth),
        new DoubleValue(throughput)});
  });
  public double getLatency() {
    return ((DoubleValue)getValues()[0]).getValue();
  }
  public double getBandwidth() {
    return ((DoubleValue)getValues()[1]).getValue();
  }
  public double getThroughput() {
    return ((DoubleValue)getValues()[2]).getValue();
  }
}
```

Figure 4.6: The LinkProperties Class.

latency, bandwidth, and throughput of a network link, which are represented via the standard `DoubleValue`s that are provided as part of CONSUL.

Sensor Monitoring. The sensor monitoring component maintains a registry of monitors available on the local host (*local monitors*) and on hosts found by the network discovery package (*remote monitors*). As described above, local monitors make the services available on a host accessible to applications. To gain access to local monitors, the application requests them by name (e.g., “Location”) from the registry, which returns a handle to the local monitor. The interface for the `RemoteMonitor` class is shown in Figure 4.4.

To monitor context information on remote hosts, the monitor registry creates `RemoteMonitor`s that connect to concrete monitors on remote hosts. These `RemoteMonitor`s serve as proxies to the actual monitors; when the values change on the monitor on the remote host, the `RemoteMonitor`’s value is also updated. To access remote monitors, the application provides the ID of the host (which can be retrieved from the network discovery package) and the name of the monitor to the registry’s `getRemoteMonitor()` method. This method creates a proxy, connects it to the remote monitor, and returns a handle. The application can then interact with this handle as if it were a local monitor.

Table 4.1: Monitor Value Types Provided By CONSUL

Monitor Value Name
`ArrayValue`
`BooleanValue`
`ByteArrayValue`
`ByteValue`
`CharValue`
`DateValue`
`IntValue`
`LongValue`
`DoubleValue`
`StringValue`

The QueryME middleware uses the CONSUL monitoring package to evaluate the network constraints metric that is provided with a query as part of a context policy, and thus to determine the cost of a network path, where the cost of each link is defined by the application according to some properties of the network links. The application programmer may define the cost of each network link using monitors defined over link properties. In this case, the network constraints metric provided by the application must include the monitors, both local and remote, in a stored list of monitors that are used by the metric. The QueryME middleware accesses the list of monitors in the metric and uses them to acquire the current cost of each link when calculating the cost of a path to a network node.

4.2.3 QueryME QueryManager

At each host, a QueryME `QueryManager` is used to handle shared context data and to process queries and replies. In this section, we describe key features of a query manager that support query execution over an application's defined context.

Registering Applications

Since each query manager is responsible for storing the data that applications running on the local host offer to share as context with others in the network, it keeps track of all applications that are running on the host. The `QueryMananger` does this through a registration process. The first time that an application contacts a query manager, the application is required to provide the query manager with its application profile and to register itself. The query manager keeps a registry of all application profiles for applications on the local host, indexed by the unique id of the application. As will be detailed later, these application profiles are used by the query manager to determine if the application constraints associated with a query are satisfied by the provider of context data.

Using Tuples to Represent Context Data

QueryME uses tuples as the baseline representation for context data items. A tuple is a set of unordered fields, where each field is a triple (*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. Each context data item has two fields: a metadata field and the data field. The metadata field is also a tuple whose named, typed fields describe the data object.

The query manager stores context data items provided by applications residing on the local host in a tuple space. 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 to the query manager. This option allows the query manager to take advantage of a tuplespace's 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 satisfies the the template's constraint.

We developed a custom tuplespace package for use by the query manager to support query operations. Tuplespace operations for the `Tuplespace` class developed for QueryME include non-blocking (or probing) variants of Linda's `rd` and `in` operations, the `inp` and `rdp`, similar to those of other tuplespace-based middleware systems (e.g., LIME [26] and Limone [11]. Both operations check the tuple space for a tuple matching the provided template. If one or more matching tuples exist for a `rdp` operation, 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 tuplespace also provides an `out`(t) operation, which places the tuple t into the local tuple space. The ELights package introduced in [19] is used by the `Tuplespace` as the implementation of tuples and templates.

The query manager uses the `out(t)` operation to support context data insertion operations. The inserted tuple t has two fields: one for metadata (represented as a tuple) and another for the data item to be shared (represented as a serializable object). Tuples that are inserted into a query manager's tuplespace are extended with special system fields that uniquely identify the tuple, uniquely identify the application that provided them, and that indicate their status. These fields are used internally by the QueryME middleware; they are not made available to the application through content-based retrieval operations.

The non-destructive operations (`rdp(p), rdgp(p)`) are used by the query manager to support the implementation of one-time queries while the removal operations (`inp(p), ingp(p)`) are used to support the deletion of shared context data from the local tuplespace. Both forms rely on the use of metadata to find the desired tuple. The query manager supplies a pattern p of the form:

$$\langle(\text{``}metadata'', ETuple.class, MetadataConstraintFunction(metadata))\rangle$$

where metadata is a template that corresponds to the metadata being searched for. The `MetadataConstraintFunction` constraint function is designed to operate only over the metadata field of context data item tuples in the tuplespace rather than both the metadata and data object fields. The metadata tuple of each context data item must match the metadata template supplied as a parameter to the `MetadataConstraintFunction` in order for the context data item tuple to be returned as a result.

Using Reactions to Handle Persistent Queries

Another benefit of designing the query manager to use a tuplespace for storing context data is that we can take advantage of reactive constructs that have been previously defined in tuplespace models [26, 11, 19]. The `ReactionManager` component of the query manager allows for reactions to be registered on the manager's tuplespace via the `registerReaction` method. A reaction is simply an association between a pattern (template) 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 tuplespace and the execution of triggered callback functions occurs in a single atomic step. The reaction will continue to fire each time that a tuple is found that matches the reactive pattern until the reaction is deregistered (using the `deregisterReaction` method of the `ReactionManager`). Such reactive constructs are extremely useful in the query service to address issues associated with reporting changing data to the application.

New data items. Addressing the issue of notifying the reference application of the addition of new context data items that meet a persistent query's data specification is relatively simple. When the persistent query is issued, each query manager that receives the query simply registers a reaction on the tuplespace using the data template provided as part of the query as the reactive pattern. The reaction's call back function initiates reply processing using the query's reply processor in order to deliver the new results to the query initiator. The replies are encapsulated as a data reply. An application dictates how these replies are managed by submitting a persistent reply handler at the time the query is submitted.

Removed data items. Dealing with deletions of context items previously reported as replies is slightly more complex since reactions are based upon the state of the tuplespace and cannot be triggered by the removal of tuples. To address this issue, we have extended the notion of reactions over tuplespaces to support the notification of removal of data. We do so through the introduction of *anti-tuples.* The anti-tuple is simply a copy of the tuple of interest with a value set in a special system tuple field that indicates its status as an anti-tuple. An anti-tuple is not a piece of data, but rather indicates the removal of a particular piece of data. When a piece of data is removed from the tuple space, a corresponding anti-tuple is inserted into the tuple space.Each time that an 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. 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 inserted and then immediately removed from the tuple space.

To support notification of removed data items in QueryME, the `ReactionManager` registers a reaction using an anti-tuple as the reactive pattern to signal the deletion of a reported context item. 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 identified as deleted data replies. As before, an application dictates how these kind of replies are handled by submitting a persistent reply policy with the persistent query.

Unavailable data items. With persistent queries, the overlay data structure that encapsulates the context associated with a persistent query is constantly updated in response 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.

To allow a member of the context to notify a query initiator that a context child and its descendant's data is no longer available, the query manager once again relies on the use of anti-tuples and reactions to report change. 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 query manager's tuplespace before processing and propagating the reply back to the query initiator. A system field in the tuple is used to identify the tuple as a reply to a query rather than as a locally available result. 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 unavailability of previously reported data are reported as unavailable replies and are handled according to an application's submitted persistent reply processing instructions.

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 *path* 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 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.

Using Profiles to Evaluate Constraints

The QueryME query manager has a `HostProfiles` that is used to - the host constraints of a received query. If the host profile satisfies the host constraints, the query manager evaluates the query operation over the local tuple space to get a set of potential results. If any context item tuples match the data constraints portion of a query, then the query manager examines each potential query result to determine if

```
public abstract class Application {
   private ApplicationProfile profile;
   private ApplicationID id;
   private QueryManager manager;

   public Application();
   public ApplicationProfile getApplicationProfile();
   public void addProfileField(EField field);
   public void removeProfileField(EField field);
   public ApplicationID getApplicationID();
   public QueryManager getManager();
}
```

Figure 4.7: The `Application` Abstract Class

the owner of the context data item tuple satisfies the application constraints associated with the query. The query manager does this by using the system field of a tuple to acquire the application identifier of its owner and uses this ID to look up the application profile in its list of registered applications. If an application's profile matches the application constraint associated with a query, all context data item tuples in the set of potential results that are provided by that application are eligible to be returned as a result by the query manager.

4.3 QueryME API

Any application that uses QueryME must extend the `Application` class (Figure 4.7). The `Application` class encapsulates an `ApplicationProfile` describing the application, a unique identifier, and a handle to the query manager running on the local host. If the application's local host is not currently running a query manager, the constructor results in the query manager's initialization. The application is then registered with the query manager, providing its application profile during registration. The `Application` class also provides support for accessing and manipulating the application profile, and for accessing the query manager. The query manager is used to perform all context-related operations.

```
public class ContextDataItem extends ETuple {
   public ContextDataItem(Serializable data);
   public ContextDataItem(ETuple metadata, Serializable data);
   public ContextDataItem addMetadataField(EField field)
   public ContextDataItem removeMetadataField(EField field)
   public ETuple getMetadata();
   public Serializable getData();
   public EField getMetadataField(String fieldName);
   public EField[] getMetadataFields();
   public ETemplate getMetadataTemplate();
}
```

Figure 4.8: The `ContextDataItem` interface

The QueryME middleware can be used to provide data for use as context by others or to perform a tailored, content-based search for context data within the mobile ad hoc network. The following sections describe how an application programmer can perform context provision and acquisition using the QueryME API.

4.3.1 Providing Context Data

As mentioned earlier, QueryME provides a baseline representation of context within the query service to allow for uniform treatment of context by the middleware, while providing an interface to application programmers that allows them to submit their data as context for use by others in the mobile ad hoc network. Because the QueryME middleware is implemented in Java, the only restriction is that it must be possible to convert the data into a serializable Java object. Context data providers must also supply metadata describing each object to allow the middleware service to evaluate the satisfaction of queries over what it would otherwise consider as "black box" context data objects.

To share data as context, an application must first construct a `ContextDataItem` that encapsulates the context data item's metadata and the data object. A

`ContextDataItem` is essentially a tuple with two fields: a metadata tuple and a serializable data object. The context data provider can alter the metadata description by adding or removing fields using the `addMetadataField` and `removeMetadataField` methods, respectively. The `ContextDataItem` interface is shown in Figure 4.8.

Once the context data item tuple is constructed, the user may submit the item to the query manager on the local host using the manager's `insert` method. The user must also supply its unique application identifier when submitting the context data item. This allows for the owner of the item to be identified and its application profile to be evaluated to determine whether or not the context data provider satisfies the application constraints portion of a context policy associated with a query.

The `insert` operation of the query manager places the context data tuple in the local tuple space owned by the local query manager. The tuple will be available for local evaluation of single queries on the local host. In addition, the insertion of a new tuple into the local tuplespace results in the firing of any reactions that are registered on the tuplespace for a matching data pattern. Therefore, if the newly inserted tuple meets a persistent query's context policy and data constraints, the query issuer will receive notification of the availability of a new query result.

An example of context data *insertion* is performed by an injury monitoring application in the `provideStatusReport` method shown in Figure 4.9. In this example, the injury monitoring application is submitting an injury monitoring report for an injured victim. The application describes the report as a medical document (specifically, an injury status report) and gives the victim's current injury classification.

A context data provider may wish to discontinue sharing information or may want to update the shared context data item. The application can use the query manager's `delete` method to discontinue sharing the data as context. The user must describe the context data item to be removed by creating an `ETemplate` which describes the metadata associated with the context data item that is to be removed. The programmer can acquire a template that matches the provided metadata by using the `getMetadataTemplate` method of the `ContextDataItem` object, which returns a template that corresponds to the provided metadata tuple. The application's unique identifier must also be provided at the time the context data is to be removed. The

```
public class InjuryMonitor extends Application {
   private QueryManager mgr = QueryManager.getManager();
   private int injuryClassification;
   private String patientDescription;
   private DiagnosisDocument diagnosis;
   private TreatmentDocument recommendedTreatment;
   private StatusDocument patientLog;

   ...
   application specific code for acquiring victim injury info
   ...

   supply report for use by rescue workers
   public void provideStatusReport() {
      ContextDataItem statusLog =
         new ContextDataItem(patientLog);
      statusLog.addMetadataField(new EField(‘‘dataType’’,
         String.class, ‘‘medical document’’));
      statusLog.addMetadataField(new EField(‘‘medicalDocType’’,
         String.class, ‘‘injury status document’’));
      statusLog.addMetadataField(new EField(‘‘injuryClass’’,
         Integer.class, new Integer(injuryClassification));
      mgr.insert(statusLog, getID());
   }

   remove invalid report
   public void removeStatusReport(ContextDataItem dataItem) {
      ETemplate toRemove = dataItem.getMetadataTemplate();
      mgr.delete(toRemove, getID());
   }
}
```

Figure 4.9: An Example Provider of Context Data

`delete` method will result in the removal of all context data items owned by the application (as identified by the provided unique identifier) that match the provided template and are stored in the local query manager's tuplespace. However, the application programmer must be careful to construct the template to be as specific as possible so as to remove only the single context data item that is intended to be removed. The programmer can acquire a template that matches the provided metadata by using the `getMetadataTemplate` method of the `ContextDataItem` object, which returns a template that corresponds to the provided metadata tuple.

An example of context data *deletion* is shown in Figure 4.9 as well. In this example, an injury monitoring application removes an previously provided injury status report using the `removeStatusReport` method shown. The application uses the previously constructed `ContextDataItem` to acquire the template for removal.

4.3.2 Acquiring Context Data

When submitting a query, the application programmer must provide a description of the desired data as well as the query's associated mobile code specializations that dictate how it should be processed within the mobile ad hoc network. Along with the description of desired data, the application programmer must provide a context policy, a propagation policy, a query propagation policy, and a reply processing policy when issuing a query. In addition, persistent queries require the provision of policies to handle notifications of new, deleted, and unavailable query results. The query service utilizes these pieces of mobile code to tailor the execution of each submitted query and to deliver the desired results to the issuing application. Each of these query components is discussed in detail in the following sections. Throughout, examples are used to illustrate the use of each concept. This section concludes by describing how a user submits one-time and persistent queries.

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.

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

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

Figure 4.10: The `Cost` and `Metric` Interfaces

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 [12] to specify and construct a context. The context is constructed using a spanning tree. This requires defining a `Metric` class and a `Cost` class 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` class 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 4.10. Defining a `Cost` subclass simply requires the programmer to define a method which 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 the supporting `CONSUL` environmental monitoring package to provide the metric with access to local (on the same host) or remote (on a reachable remote 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 `weightFunction` 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 weightFunction(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 4.11: 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 4.11. We have omitted the definition of the `HopCost` class, which adheres to the `Cost` interface by storing an integer and implementing the `compareTo` method.

The implementation of the hop count metric does not require the programmer to utilize any monitors. However, more sophisticated metrics may require access to properties of the network links. Consider, for example, an application that requires the end-to-end link latency be within a threshold to guarantee a minimum level of performance. An example metric which captures an additive latency metric is shown in Figure 4.12. The metric uses the `LinkMonitor` and `LinkProperties` classes introduced earlier. The `LatencyCost` class, which is omitted from the figure, holds a double value and provides a `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,

```
public class LatencyMetric extends Metric{
  public LatencyMetric(){
  }
  public Cost weightFunction(HostID otherHost){
    //calculate the weight on the link
    QueryManager mgr = QueryManager.getManager();
    LinkMonitor monitor =
     mgr.getMonitorRegistry().getMonitor(``LinkMonitor'');
    LinkProperties properties =
     (LinkProperties) monitor.getValue();
    DoubleValue latency = properties.getLatency();
    LatencyCost weight = new LatencyCost(latency);
    return weight;
  }
  public Cost costFunction(Cost currentD,
                           Cost weight){
    LatencyCost newCost = currentD + weight;
    return newCost;
  }
}
```

Figure 4.12: An Example `LatencyMetric` Class

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(``firetruck'')));
```

Figure 4.13: An Example Host Constraint

An example use of host constraints is shown in Figure 4.13. The host constraint dictates that only hosts that identify themselves as firetrucks 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(‘‘AppType’’,
    new EquivalencyConstraintFunction(‘‘patient monitoring’’)));
```

Figure 4.14: An Example Application Constraint

An example application constraint is shown in Figure 4.14. The constraint shown dictates that the application must be described as being a patient monitoring application.

We provide a collection of commonly used network constraints in our QueryME middleware for use by the application 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 for a query's execution. 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 that comprise a context policy as parameters. A `ContextID` is returned to the application to identify the registered context. The application supplies this context identifier

```
public interface QueryProcessor {
   public Vector limitPropagation(Vector
                            potentialChildren);
}
```

Figure 4.15: The `QueryPropagator` Interface

when issuing a query over the context. However, before issuing a query, the programmer must first define how the query's execution is to be controlled, dictating how queries can be propagated and how replies are processed.

Defining the Query Propagation Policy

To define a query propagation policy, a programmer must define a class which implements the `QueryPropagator` interface shown in Figure 4.15. To define the query propagation policy, a programmer must define a `limitPropagation` method. This method uses a set of potential context children provided by the QueryME middleware which satisfy the network constraints and imposes the propagation policy on this set to determine a new set of context children. This set is returned to the QueryME `QueryManager`. If the context children set calculated by the `limitPropagation` method is empty, the query is not to be propagated any further.

A number of query propagators that incorporate standard query propagation schemes are provided with the QueryME middleware. The context-aware application developer can simply choose to use one of the provided mobile code fragments. However, the set of query propagators can also be easily extended to incorporate mechanisms tailored to a particular application.

To illustrate the development of propagators, a controlled flooding propagation scheme is shown in the `limitPropagation` method in Figure 4.18. The code shows that when a local result is found that satisfies 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.

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

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

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

Figure 4.16: An Example Query Propagator

```
public interface ReplyProcessor {
    public void processReply(QueryID qID, Reply r);
}
```

Figure 4.17: The `ReplyProcessor` Interface

Defining the Reply Processing Policy

To define a reply processing policy, a programmer must implement the `processReply` method of the `ReplyProcessor` interface shown in Figure 4.17. The `processReply` method determines how replies received by the query manager on the local hosts are processed. If the reply is to a query that was not issued by the local host, the processReply method is responsible for calling the `sendReply` method on the query manager to continue to propagate the reply along the path to the query issuer.

A number of standard in-network reply processing policies 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 that implement the ReplyProcessor interface.

```
public class AggregateProcessor extends QueryProcessor{
  QueryID qID;
  Hashtable receivedReplies = new Hashtable();
  Vector children;
  QueryManager mgr = QueryManager.getManager();

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

  public void processReply(Reply r) {
    if children.contains(r.getSender()) {
      receivedReplies.add(r.getSender(), r);
    }

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

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

Figure 4.18: An example AggregateProcessor Class

Describing the Desired Data

The purpose of a query is to acquire some piece of context data according to a content-based description of the desired data. One way to think of a content-based description is as a constraint over data, and the results returned for a query must satisfy that data constraint. In QueryME, queries are actually executed over the metadata of context data items. Since metadata is represented as a tuple in QueryME, the data constraint portion of a query is captured as a template. The application designs the data constraint template to give constraints over named, typed fields in the metadata of the desired context data item. As stated previously, the system will apply the `MetadataConstraintFunction` to evaluate a submitted data constraint. The data constraint is satisfied over a context data item tuple if for every field in the data constraint template, there exists a field in the metadata tuple of the context data item tuple that has the same name and type, and a value that satisfies the constraint given for that field in the data constraint template.

Issuing a Query

To issue a one-time query, the application uses its handle to the query manager on the local host and uses its `sendQuery` method. The application supplies the `ContextID` acquired when creating a context policy, the data constraints that form the content-based request for data, the desired query operation (GET or GETALL), and the specialized processors. The query also registers a reply listener that will only be invoked when replies are returned to the reference application.

Registering a Persistent Query

To issue a persistent query, the application calls the `registerQuery` method on the local query manager. The application must supply all the information given for a one-time query, as well as a mobile code element that manages changes to reported data. An application that registers a persistent query can expect to receive notifications of newly available data results and reports generated due to the deletion or unavailability of previously delivered query results. At the time that a persistent query is registered, a programmer must provide a policy for handling updated query results. To do so, the application programmer implements the `PersistentReplyListener` interface shown in Figure 4.19 and provides the listener to the query manager when the query is registered. The `processReply` method determines how new data replies (each represented as a `DataReply`) are processed. Reports on the unavailability of data and deleted

```
public interface PersistentReplyListener {
   public void replyReceived(QueryID qID, DataReply dReply);
   public void unavailableReplyReceived(QueryID qID,
      UnavailableReply unReply);
   public void deletedReplyReceived(QueryID qID,
       DeletedReply delReply);
}
```

Figure 4.19: The `PersistentReplyListener` Interface

data are processed according to the instructions in the `processUnavailableReply` and `processDeletedReply` methods, respectively.

4.4 Chapter Summary

The QueryME middleware is a realization of the query-centric model presented in the previous chapter. The implementation uses mobile code to support tailored processing of queries within a limited scope, tuples to support the encapsulation of various context data types and content-based retrieval using metadata, and reactions to support prolonged evaluation of queries in a continuously changing environment. In this chapter, we presented the design and implementation of our QueryME middleware and described the programming interface that an application developer can use to construct context-aware applications. The QueryME approach is extensible and flexible, allowing for additional mobile code specializations for to be plugged in at the time of query execution. Though this requires the application programmer to supply several mobile code elements as part of query for tailored processing, a small library of commonly used specializations can be easily constructed and provided to the application developer to use and extend in order to simplify the process.

Chapter 5

Gander: A Context-Aware Search Engine

The World Wide Web (WWW) has proven to be a successful metaphor for accessing information distributed across the entire planet. Today, almost one billion people acquire information from the Internet [27] using web pages. In recent years, the applicability of this mode of information storage and retrieval has been extended to the wireless environment. In fact, it is now possible to surf the web using a mobile phone while on the go through a cell phone tower connected to the wired Internet. Such impressive growth in the use of the World Wide Web is a testament to its utility and importance to our modern life.

While ad hoc networks lack connectivity to the Internet, the ability to extend the WWW metaphor to such an environment can be highly beneficial. In an ad hoc network, connections are formed opportunistically between mobile devices within wireless communication range, without any assistance from a wired infrastructure. By treating the ad hoc network as if it were a WWW-like distributed information repository, a user who is part of an ad hoc network can easily take advantage of the wealth of information provided by mobile and stationary hosts encountered in the surrounding operational environment.

This chapter introduces Gander, a search engine designed to retrieve web pages distributed across devices in a mobile ad hoc network. In Gander, search is personalized according to user-supplied preferences used to limit the extent of the search and properties of the data being searched for. A weary shopper looking for a soothing place to relax with a cup of coffee may specify, for instance, the need to know about web pages posted by coffee shops within a four block area. Once preferences are specified, a user can either submit a simple immediate search for such data or can request a persistent search which generates a dynamically maintained list of web pages meeting the stated preferences, continuously updated to reflect the changing environment as the shopper makes her way along the avenue and as coffee shops open for business. With either kind of search, the user can simply click on any one of the returned web pages in the list to display it.

At heart, Gander is a context-aware application, adapting its behavior according to changes sensed in its environment. From a software engineering perspective, the development of Gander demonstrates how one can exploit a general purpose query-centric middleware to construct applications that share the need to be responsive to context changes. Gander is implemented using QueryME, the middleware introduced in this dissertation that is specifically designed to give an application access to contextual information tailored to evolving application needs. As discussed in the previous chapter, QueryME takes a query-centric approach to supporting context-aware application development. Context policies provided by Gander users are used by the middleware to restrict the scope of the query evaluation both in terms of network coverage and host participation. The middleware constructs an overlay network in accordance with the application-supplied policy (e.g., within two hops, within four blocks) which captures the application's context. The use of such policies keeps the associated context interaction costs in line with the actual needs of the application, which are likely to involve a limited and often localized spatial domain. A Gander submitted search is actually a query evaluated over the overlay network, rather than across the entire ad hoc network. A simple query immediately examines the hosts included in the overlay network and returns the results to the issuing application. Persistent queries are evaluated by maintaining the overlay network as the physical ad hoc network evolves, and by continuously sending reports to the issuing application as pertinent data becomes available (e.g., as coffee shops are approached during the walk) or as previously reported results become irrelevant (e.g., coffee shops that are now too far or in the wrong direction). The middleware shields the application programmer from the details of maintaining the network overlay, yet allows for user-customized strategies for query propagation and evaluation through the use of a protocol which exploits mobile code technology.

The remainder of this chapter is organized as follows. Gander is introduced in Section 5.1. Section 5.2 shows how the query language constructs provided by our middleware can be used to implement search functionality in the Gander application. Evaluation of the Gander application is presented in Section 5.3. The chapter is summarized in Section 5.4.

5.1 A Context-Aware Search Engine

The Internet has experienced rapid growth over the last decade. In the course of this time span, Internet search engines have become indispensable to many users in helping them find webpages that provide the information, products, or services they need. A number of popular search engines have emerged over the years. Google, Yahoo!, and MSN Search are the leading search tools, each service handling millions of searches per day [27]. The popularity of these and other search engines may be attributed to the ease of use that they offer. Instead of requiring a user to know the exact Internet address of a website that provides information or services of interest to a user, search engines allow for content-based retrieval of webpages. Though designed for use in an Internet environment, th work presented in this chapter is based on the belief that the abstraction provided by Internet search engines can benefit users operating in an environment without Internet connectivity; people whose wireless mobile devices form ad hoc networks could use search engines to find relevant information, products, and services provided by other users within the same ad hoc network.

To understand how to use search engines in the ad hoc network setting, it is necessary to understand first how search engines operate in an Internet environment. Internet search engines typically work by building a directory of webpages that are available on web servers across the Internet, and indexing the webpages according to their content. The process of building the directory is usually automated. For example, Google uses a web crawler which periodically scans the Internet for webpages and uses an indexing algorithm to build a directory. The resulting search engine service directory is stored on a collection of servers. Users perform a search by entering a query in the search engine interface, usually within a web browser. To search engine users, it appears that they are searching the web, but in reality they are searching the directory of webpages stored on the search engine service's bank of servers. When presented with search results, users are given a list of links to webpages indexed within the directory that meet the search criteria.

The characteristics of ad hoc networks make it impractical to utilize these kinds of search engines directly. An ad hoc network is comprised solely of mobile devices and does not utilize a fixed infrastructure. Reliance on a logically centralized directory of webpages is not feasible in this environment, where roving users equipped with mobile devices may move out of connectivity range of the directory servers or may never encounter a directory server at all. Moreover, rapid configuration changes that frequently occur in ad hoc networks invalidate the typical web crawler approach to maintaining a directory which is reasonably consistent with the webpages that are currently available in the environment. Finally, while an Internet search engine uses stored information from the entire network to execute every search, roving users of a search engine in an ad hoc network are often concerned only with results that are available within their current locality.

To address these issues, we have developed Gander. Gander is an application designed to allow mobile device users to search for webpages that provide useful information about businesses and services of interest in the user's surrounding area without connectivity to the Internet. Gander users act both as providers and as searchers of webpages. In a provider role, a user is giving permission to the Gander application to make a particular webpage public. When acting in a searcher role, a user is asking Gander to find webpages that meet given search criteria. This section outlines how providers and searchers use the Gander interface.

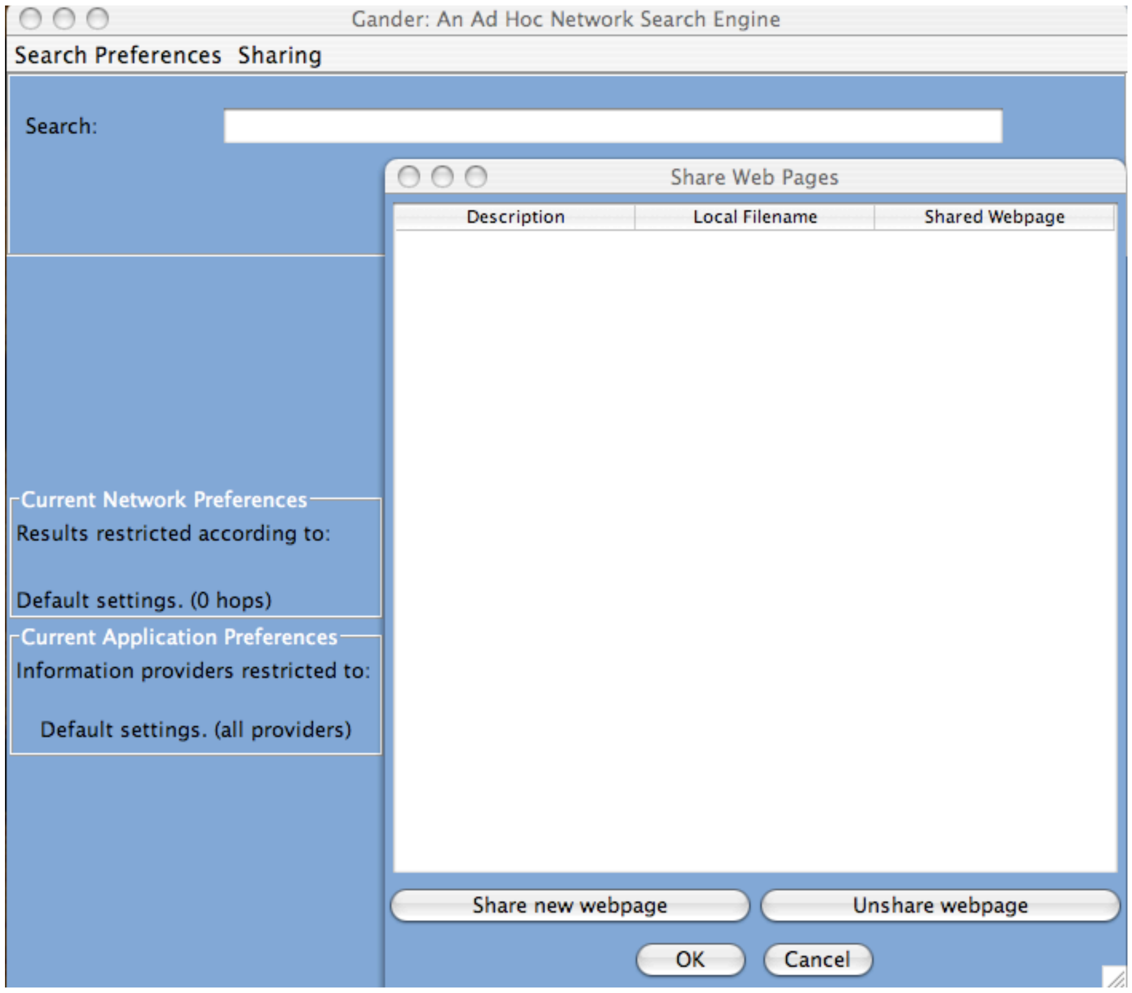
A user may provide web pages for viewing by others by allowing Gander to include the pages in distributed search operations. The process of providing webpages consists of three steps: choosing a webpage to share, providing information that may impact selection in a search, and describing the content of the webpage. The process for sharing webpages via the Gander interface is shown in Figure 5.1 and is described in detail below.

To share a webpage, a provider first selects the “Share Webpages” option from the Gander menu to begin the process. The user then selects the HTML file stored on the local file system to be shared from a dialog box. Next, since Gander allows a searcher to tailor search results to include only webpages from described providers, Gander is supplied with properties that characterize the provider of a webpage and potentially reflect the relevance or quality of search results provided by the user. These properties may be acquired by Gander directly from the provider or may be obtained through observation of prior interactions with the provider. Currently, a Gander provider supplies such information directly by selecting a descriptor from a list that identifies the kind of business or service that the user represents. For instance, the owner of a local coffee shop may chose to use Gander to share her webpage advertising the shop’s menu and another webpage advertising upcoming live entertainment events hosted at the shop; this provider describes herself as a coffee shop and restaurant. The list of webpages that the user has chosen to share with Gander is updated to reflect the addition; in the case of the coffee shop owner, the list reflects two entries: both the menu and entertainment webpages. A user can easily stop sharing the webpage at any time by simply selecting the page from the list and clicking the “Unshare” button.

The final step in the process of sharing webpages is describing the content of the webpage; this description is used by Gander to match a user’s search request to a webpage. Such content-based search has proven to be essential in searching the vast expanse of the Internet; given the flexibility such an approach offers, it is important to incorporate this concept in Gander when matching provided webpages to search queries. In Internet search engines that provide content-based search, each webpage is cataloged in a directory using metadata which describes its content. Typically, a webcrawler-based approach creates an entry in a search directory that associates the webpage with every word included in the content of the webpage (commonly used words, e.g., “the”, “was”, etc., are excluded). In a simplified view of the process, search queries submitted by users are parsed and the resulting components are used to retrieve matching webpages from the directory. This approach to creating metadata to support content-based search cannot be applied in Gander, since creating such a large search index would be infeasible in resource-poor ad hoc networks.

Several options could be incorporated into Gander to support the construction of searchable metadata descriptions of webpages. One option is manual entry of metadata, in which a provider describes a webpage using a small number of search terms when it becomes shared. At the other end of the spectrum is automated construction of metadata in which the Gander software parses the information contained in the webpage to construct a short description. In either case, metadata may be constructed by using the content of the webpage to determine a collection of descriptive terms, either by providing a list of words, selecting a category from a list, or selecting particular terms from a list. Gander is designed to offer flexibility when determining how a webpage is described as searchable metadata. Metadata construction is encapsulated within a separate component, allowing for a "plug-and-play" approach to selecting a descriptive strategy. Currently, Gander is provided with a default component which implements a manual metadata construction approach: users are required to provide strings describing the webpage as metadata upon sharing the page.

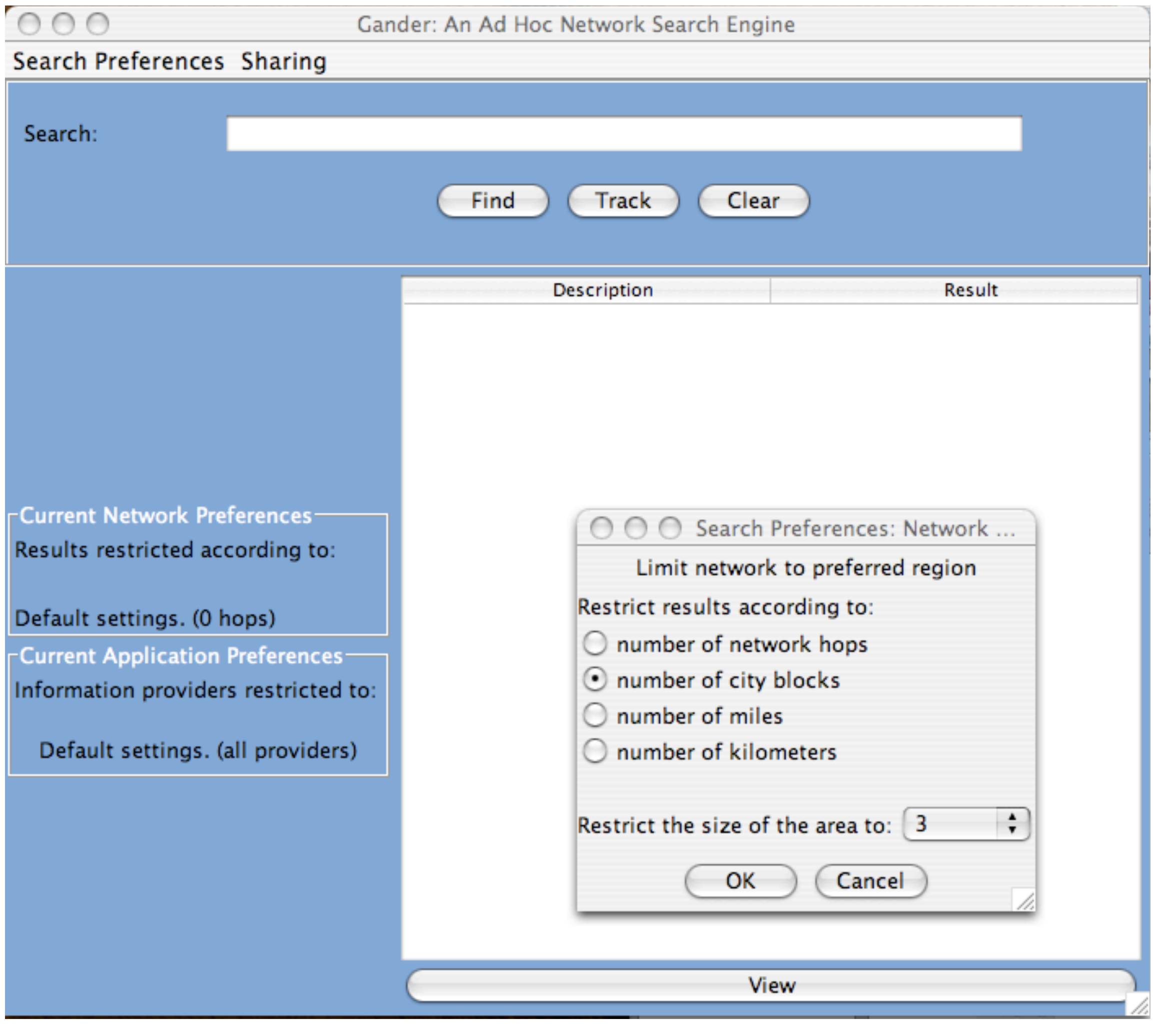
Gander users who wish to search for information in the surrounding environment may provide preferences for directed, personalized searches that are sensitive to context changes. Gander offers two kinds of preference settings: network and application preferences. The user can select network preferences to limit the scope of the search to the specified portion of the network surrounding the user. The Gander user interface allows the user to limit searches to an area surrounding the user bounded by a selected number of network hops, city blocks, miles, or kilometers. As shown in Figure 5.2, a weary shopper using Gander is tired of walking and wants to find a place to sit and have a cup of coffee within the next 3 blocks. Application preferences can be used in Gander to limit search to providers that are likely to provide data of interest. In the current implementation of Gander, the user can describe data providers of interest by selecting any number of attributes from a list. As shown in Figure 5.3, the shopper is hoping to find a cup of coffee prepared for her in a coffee shop or restaurant rather than going to a grocery store to get the supplies to prepare the coffee for herself. If our shopper (or any other user) does not set the network or application preferences before submitting a search request then the default settings will be used: the network preferences set to five network hops and application preferences set to include all public pages.



Current Network Preferences
Results restricted according to:
Default settings. (0 hops)
Current Application Preferences
Information providers restricted to:
Default settings. (all providers)

Figure 5.1: Sharing Webpages in Gander

A user can submit a search request at any time. Two kinds of searches are possible: immediate and persistent. An immediate search request terminates once results are returned, while a durable search request provides continually updated search results. To perform an immediate search, the user provides a search string and clicks the "Find" button. A persistent search is performed similarly, except the user clicks the "Track" button. A weary shopper using Gander to find a nearby cup of coffee enters the word "coffee" and clicks to start the search. The ad hoc network is searched for webpages that match the request using the search input (e.g., "coffee") and the metadata associated with webpages (e.g., "coffee", "mocha", "latte", etc.). At the conclusion of the immediate search, a list of results are displayed and remain until the



Current Network Preferences
Results restricted according to:

Default settings. (0 hops)

Current Application Preferences
Information providers restricted to:

Default settings. (all providers)

Limit network to preferred region

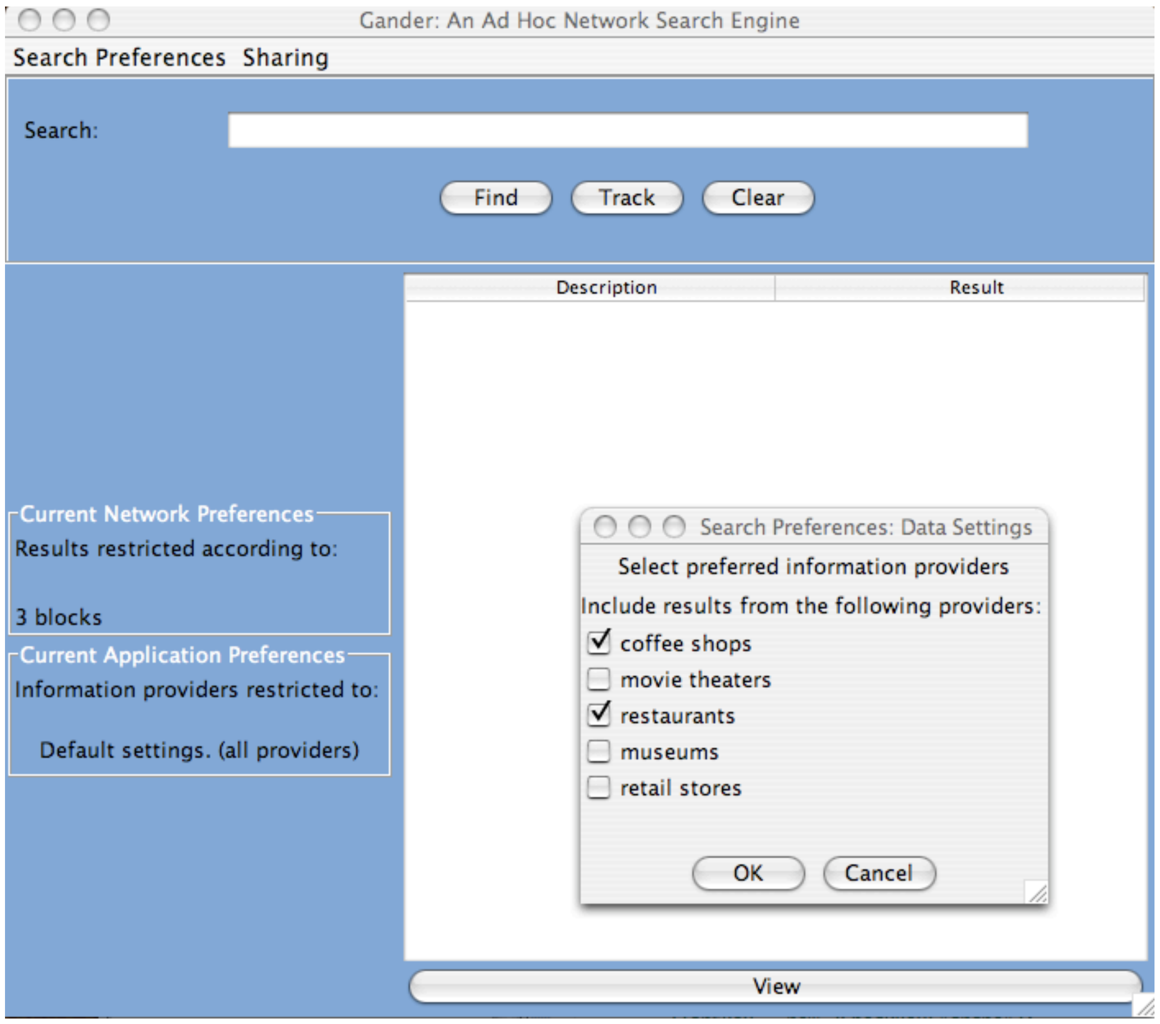
Restrict results according to:

( ) number of network hops
[✓ number of city blocks]
( ) number of miles
( ) number of kilometers

Restrict the size of the area to: 3

Figure 5.2: Network Preferences in Gander

user clicks the "Clear" button. With a persistent search, the list of results is populated and updated as the environment changes, mostly due to mobility of users which impacts the availability of results. For example, the weary shopper's search results for "coffee" at coffee shops within two blocks changes as she continues walking. The user can scroll through the list of returned results and view the associated webpage by clicking the "View" button. Figure 5.4 shows the shopper's search for a coffee given her network (3 blocks) and application (coffee shop or restaurant) preference, and the displayed search results.



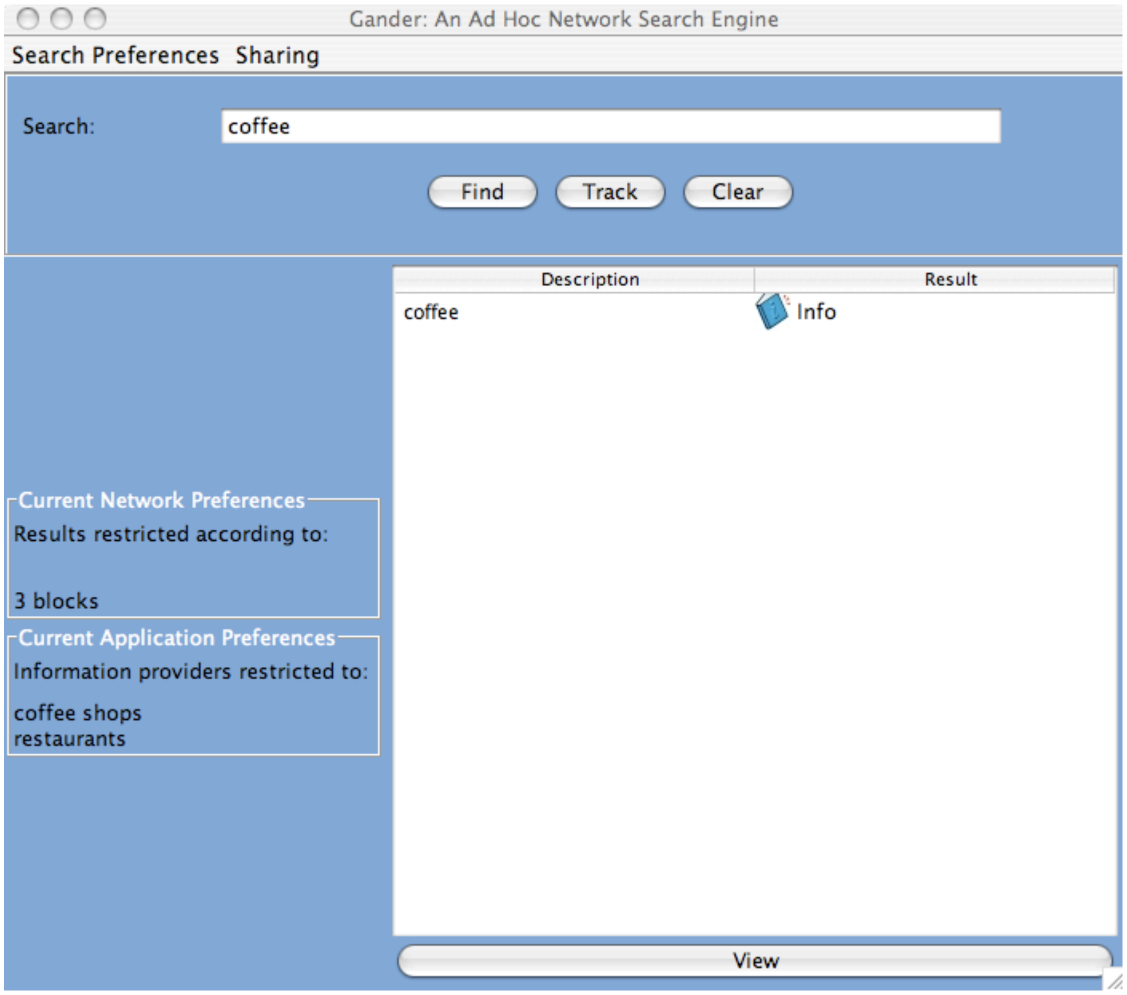
Current Network Preferences
Results restricted according to:
3 blocks

Current Application Preferences
Information providers restricted to:
Default settings. (all providers)

Search Preferences: Data Settings
Select preferred information providers
Include results from the following providers:
- [✓] coffee shops
- [ ] movie theaters
- [✓] restaurants
- [ ] museums
- [ ] retail stores

Figure 5.3: Application Preferences in Gander

Gander has been built using the QueryME middleware system presented in Section 4 that is designed to support context-aware applications. The following section demonstrates how the middleware is used to support Gander's ad hoc network search functionality.


Gander: An Ad Hoc Network Search Engine

Search Preferences Sharing

Search: coffee

Find Track Clear

Description	Result
coffee	Info

Current Network Preferences
Results restricted according to:

3 blocks

Current Application Preferences
Information providers restricted to:

coffee shops
restaurants

View

Figure 5.4: An Example Search in Gander

5.2 Using QueryME to Implement Gander

Gander is implemented in Java and uses the QueryME middleware to collect context information from the surrounding environment. Because QueryME is used by Gander for this purpose, most of the code required to implement the search engine is focused on using QueryME constructs and on providing a graphical user interface. QueryME is well-suited to support the development of Gander since searching for information can easily be implemented through the use of queries. In this section, we detail how QueryME is used in the implementation of Gander.

The `Gander` class extends the `Application` class in QueryME. The `Gander` constructor calls the constructor of its parent class, which results in the initialization of the local QueryME `QueryManager`. The `Gander` constructor also sets up the default network, host, and application constraints that are used to focus a Gander search. The default constraints are designed to give wide search coverage without overburdening the application with results. By default, the network constraints dictate that the search is limited to 10 hops. Host and application constraints require only that a host provide a unique host identifier in its profile and an application provide a unique application identifier in its host profile.

As stated in the previous section, a Gander user limits the scope of a Gander search by customizing the constraints through the user interface. Gander provides a limited number of frequently used constraints that may be useful to a mobile user in a particular scenario. The provision of constraints through the Gander user interface results in calling Gander's `updateNetworkConstraints` and `updateHostConstraints` methods. These methods store the updated constraints, which will be provided to the QueryME middleware when the user performs a simple or persistent search. These constraints comprise the context policy that will be used by QueryME to scope the execution of a query that corresponds to a Gander search request.

Gander supports the provision of webpages to be searched for and used by others in the network. When the user elects to share webpages, Gander allows the user to select the HTML file from the local filesystem and to supply a textual description of the contents and purpose of the webpage. Gander will supply this description to QueryME as the webpage's associated metadata. To do so, it converts the string into an `ELights` `ETuple`. Commas within the provided string are used as delimiters so that multi-word phrases can be captured as a metadata field. Each field in a tuple has a name, type, and value. In the metadata tuples that Gander provides, all fields are of type String and the name and values are the same: a phrase in the provided string description of the webpage. Once the metadata tuple is constructed, Gander converts the HTML file that is provided by the user into a Java `HTMLDocument` object. The `HTMLDocument` is serializable, meaning that it can be transferred over the network without loss of information. Gander then packages the metadata `ETuple` and the `HTMLDocument` together by creating a QueryME `ContextDataItem`. Gander then provides the webpage as context by using its handle to the local QueryME query manager and using the `insert` operation on the manager. Once the query manager performs the insert operation, the webpage will be available to other Gander applications in the mobile ad hoc network, as well as any other context-aware applications that are implemented using QueryME.

```
public class EquivalencyConstraintFunction extends ConstraintFunction{
  private Serializable value;

  public EquivalencyConstraintFunction(Serializable value){
    this.value = value;
  }

  public Serializable getValue() {
    return value;
  }

  //Determines whether the value of the specified field is
  //equal to the value provided to the constraint function
  //upon its construction.
  public boolean evaluate(EField field) {
    boolean toReturn = field.getValue().equals(value);
  }

}
```

Figure 5.5: The Equivalence Constraint Function used for Metadata Templates

As mentioned in the previous section, a Gander user can choose to discontinue sharing a data item by selecting an entry in the list of shared webpages and clicking on the "Unshare" button. Behind the scenes, Gander looks up the metadata associated with that particular entry and uses it to construct a corresponding metadata template. This template, in the form of an `ELights ETemplate`, is provided to Gander's local query manager in a `delete` operation. The constraint function over each field in the template is an `EquivalencyConstraintFunction` (shown in Figure 5.5, which requires that each field in a tuple must have a data value equivalent to that provided by the template. The `delete` operation will remove all `ContextDataItems` that are owned by the application and whose metadata tuple matches the template from the local tuple space. Once this deletion operation is performed, the webpage is no longer available to others in the network.

```
public SimplePropagator implements QueryPropagator{
   SimplePropagator() {
   }

   public void limitPropagation(QueryID qID, Vector children) {
      return children;
   }
}
```

Figure 5.6: The Simple Query Propagator used by Gander

```
public SimpleReplyProcessor implements ReplyProcessor{
   private QueryManager manager;

   SimpleReplyProcessor(QueryManager manager) {
      this.manager = manager;
   }

   public void processReply(QueryID qID, Reply r) {
      mananger.sendReply(r);
   }
}
```

Figure 5.7: The Simple Reply Processor used by Gander

A Gander user can either perform a simple or persistent search for webpages that have been made available in the mobile ad hoc network. When the user performs a search by entering search terms and clicking on the "search" or "track" button, Gander submits either a one-time or persistent query, respectively, to the QueryME `QueryManager`.

To submit a query, Gander must provide its mobile code specializations. If the user has provided customized network and application constraints for the search, Gander will retrieve the appropriate mobile code elements and provide them as part of the query. Gander propagates all queries in the same way: all nodes that meet the constraints should receive the query. The query propagator that implements this approach (shown in Figure 5.6) is supplied with all queries submitted by Gander. Gander also takes a standardized approach to processing webpages returned as a result of a QueryME query: replies are simply propagated back along the return path to the issuer of the query. This simple approach to query processing is demonstrated in Figure 5.7.

Gander uses the search text string to construct a data constraint for the query. Essentially, Gander is creating a metadata template to be matched over the metadata fields of `ContextDataItem`s. Again, commas are used as delimiters in the search string so that multi-word search phrases can be used. Gander uses the `EquivalencyConstraintFunction` as the constraint for each template field. This constraint function will be evaluated over each field in a metadata tuple.

Gander uses the `sendQuery` operation on the QueryManager when a simple search is requested by the user. The mobile code specializations are packaged with the data request and Gander specifies that the operation should return all query results. Because this search request is treated as a one-time query, Gander also registers a listener for query results for its query. This listener is executed by QueryME only when query results arrive at the local host of the query issuer. When the listener is notified that results have arrived, Gander takes the results and updates the display so that the user can click and view the returned webpages.

When a persistent search is requested by the user, Gander uses the `registerQuery` operation on the QueryManager. Again, the mobile code specializations are provided as parameters as well as the data request and the type of operation to be performed (i.e., GETALL, which returns all matching results). Since this search requested is a persistent query, Gander must register a `PersistentReplyListener` that dictates how data results are to be handled as well as reports of unavailable and deleted results. The persistent reply listener used by Gander will update the display to add newly available results to the list of webpages returned by the search and will remove results which have been deleted or are unavailable from the list as well.

5.3 Evaluation of Gander

Though the primary goal behind the development of Gander is to provide a novel search engine application with interesting features, a secondary goal is to use the

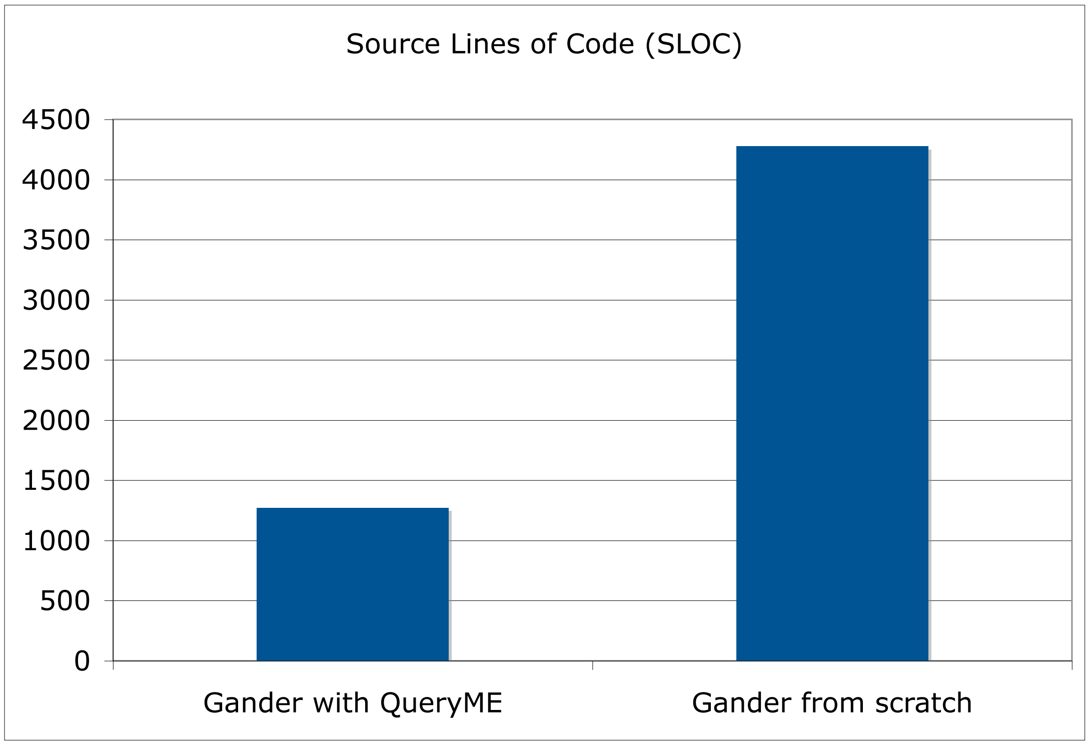


Figure 5.8: Gander Lines of Code With QueryME versus Without QueryME

Table 5.1: Comparison of Gander Development Effort

	Gander with QueryME	Gander from Scratch
Source Lines of Code	1271	4276
Development Effort Estimate (Person-Months)	3.09	11.04
Schedule Estimate (Months)	3.84	6.23
Total Estimated Cost*	$34,753	$124,231

**cost based on average annual developer salary of $56,286*

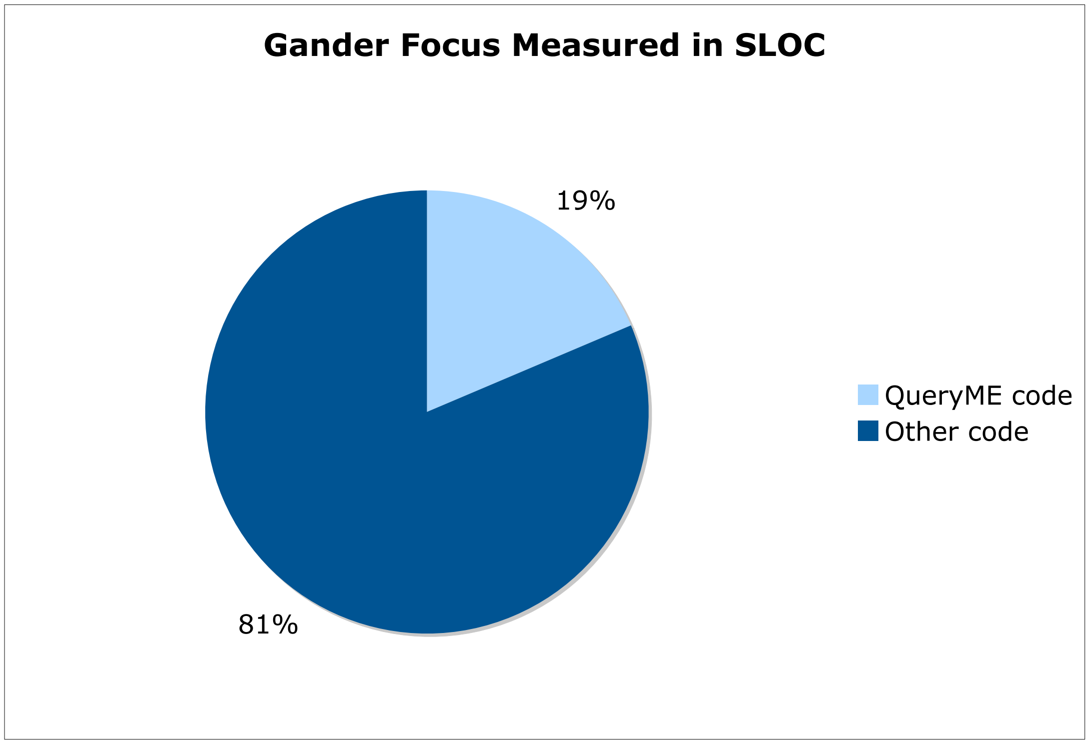


Figure 5.9: Focus of Programmer Effort for Gander When Using QueryME

application to demonstrate QueryME's potential for reducing programmer effort. We use the Gander implementation to evaluate how QueryME can benefit a context-aware application developer. We use a widely accepted line count-based software metrics tool called SLOCCount [39] to perform our evaluation of software development effort. The SLOCCount tool evaluates an application based on a count of the source lines of code (SLOC) used to implement the application. SLOCCount counts only physical lines of code, which means that blank lines and comments are omitted from the count. Using the SLOC data, SLOCCount uses a well known software cost model, the original Constructive Cost Model (COCOMO 81) [3], to estimate developer effort, project schedule, and project cost. Though a more recent version of COCOMO model exists, it requires a count of logical lines of code, or the number of statements. Counting statements versus physical lines of code is a more complex task and SLOCCount is currently unable to calculate this metric, as are many other automatic source code analysis tools. However, using physical SLOC data to compute effort using the original COCOMO model still yields results which are useful in estimating the magnitude of programmer effort.

We first used the SLOCCount tool to compare the programmer effort expended by implementing the Gander application from scratch versus using the QueryME middleware to support the context acquisition tasks in Gander. Figure 5.8 shows the physical source lines of code counted by SLOCCount for two implementations of Gander: Gander SLOC implemented with QueryME and Gander implemented without QueryME. The programmer effort is reduced from 4,276 SLOCs without QueryME to 1,271 when the QueryME middleware is used. In other words, the amount of code that an application programmer must write to develop the Gander application when using QueryME is approximately one-third of what a programmer that does not use QueryME must write in order to develop Gander. The results of applying the COCOMO model to these physical SLOC counts are shown in Table 5.1. As the table illustrates, the development costs associated with Gander can be significantly reduced by using the QueryME middleware.

We next used the SLOCCount tool to determine how much of a programmer's effort was focused on using QueryME constructs versus implementing the application. We found that implementing Gander using QueryME required very little programming effort in order to acquire webpages from others within the mobile ad hoc network

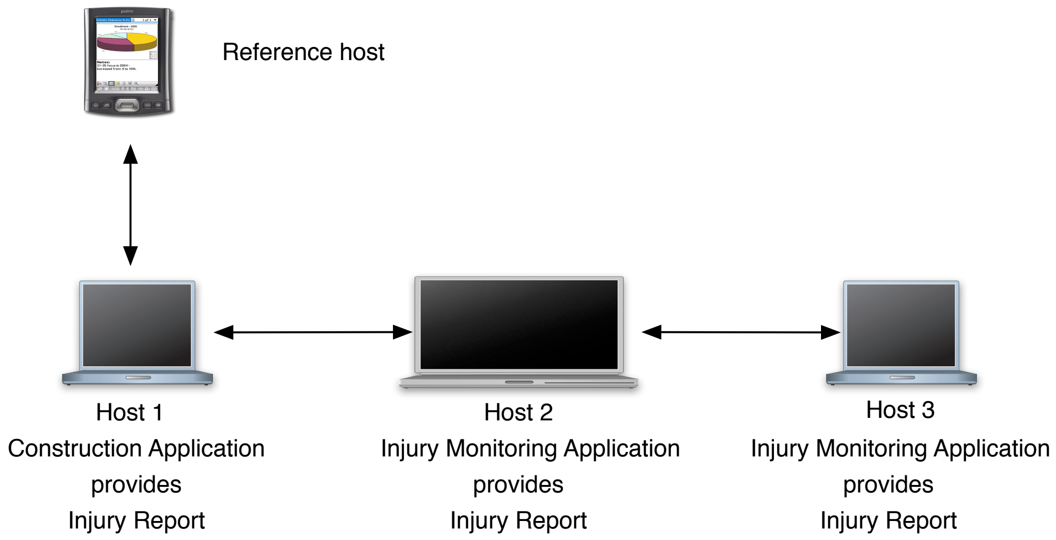


Figure 5.10: Evaluating Gander on a Small Ad Hoc Network

using QueryME constructs. In fact, as shown in Figure 5.9, less than 20 percent of the source lines of code required to implement the application were focused on context acquisition. The remaining lines of code focused on application specific tasks; most lines of code were dedicated to implementing the user interface. This calculation of programmer effort in lines of code includes all QueryME related code, including mobile code specializations. These mobile code specializations comprise 13 percent of the entire code required to develop Gander. Since the mobile code specializations used here are actually quite simple and were already made available to QueryME users, we can reasonably argue that only 6 percent of developer effort was targeted towards using the QueryME application programmer interface for context acquisition.

To perform an empirical evaluation of Gander, we deployed the application on multiple hosts equipped with wireless communication capabilities. Because forming an actual mobile ad hoc network requires a large amount of space, we limited our evaluation to occur over a small number of hosts. We arranged three laptops in a straight line configuration to form a small mobile ad hoc network. Each of these hosts is equipped with the Gander application. On two of the hosts, we use Gander to provide webpages. Though these hosts are portable laptops, we keep them stationary to simplify the experiment. We use a fourth host, a small ultra mobile pc equipped with Gander, to search the network for webpages. To simplify discussion, we refer to this host as the reference host throughout this section.

To evaluate the reliability with which the Gander finds requested webpages in a tailored context, we specified using the Gander interface that the reference host should limit the search to occur within two hops and should only acquire webpages from injury monitoring applications. All of the hosts in the network provide injury report webpages and only one of these hosts within two hops of the reference host supports an injury monitoring application. The configuration of the reference host and these providers is shown in Figure 5.10.

After issuing a simple search using Gander, we found that the application displayed only the webpage that was provided by Host 2 which runs the injury monitoring application. To evaluate Gander's persistent search, we moved the reference host in and out of communication range of Host 1. This resulted in the discovery of a new result each time the reference host was moved into communication range of Host 1 (the single result that meets the provided constraints provided by the application on Host 2) and the removal of the result from the list each time the reference host moved out of communication range of Host 1 making the result provided by Host 2 unreachable.

5.4 Chapter Summary

This chapter introduced the Gander search engine for mobile ad hoc networks. Gander supports dynamic search for webpages over an application-specified portion of the network. Searches may be simple searches which query the network and return a set of available webpages, or may be persistent searches which adaptively display the set of webpages that are currently available in the mobile ad hoc network. Gander is the first search engine of its kind; it performs adaptive context-aware searches across a mobile ad hoc network. Though interesting in its own right, the development of Gander demonstrates the use of the QueryME middleware to develop context-aware applications.

Chapter 6

A Context Acquisition Protocol

One step that is necessary to promote the adoption of the query-centric perspective by application programmers is to evaluate the QueryME middleware system. This chapter presents an evaluation of the context acquisition protocol that supports the evaluation of the QueryME constructs over the mobile ad hoc network. We begin this section with a description of the context acquisition protocol. Next, we discuss the evaluation of the protocol through simulation. We then present optimizations of the protocol that aim to reduce the overhead associated with maintaining the network overlay when issuing persistent queries. Our protocol for context acquisition is based on a protocol for building and maintaining a shortest cost tree overlay in mobile ad hoc networks, presented in [12]. We utilize the same mechanisms for constructing and maintaining a network overlay according to network constraints. Our protocol, however, includes extensions to support host and application constraints, as well as notifications of changes in data. As discussed in the previous chapter, our protocol also supports the use of application-specified query propagation strategies. These propagation are applied to the set of network entities that satisfy a query's associated context policy to further restrict the propagation of a query. For simplicity of presentation, we omit discussion of various query propagation strategies in the problem and protocol descriptions in this chapter. We focus instead on presenting a solution for context acquition that applies to the general case in which a query is propagated to all network nodes that satisfy the query's associated context policy.

6.1 Problem Description and Solution Overview

We treat the mobile ad hoc network as a graph, with the nodes of the graph representing hosts and the edges representing physical network links. Formally, let V be the set containing all nodes representing mobile hosts and let E be a set containing all communication links that are up at a particular instant in time. Clearly E changes over time, as hosts change physical location and connections are dropped or established. The combination of V and E forms a connectivity graph, $G = (V, E)$. Each connected subgraph of G represents a mobile ad hoc network.

Properties of network links such as latency, bandwidth, throughput, delay, jitter, etc. are represented as edge weights on the graph. Formally, let us assume that each edge $e_{i,j}$ in E is associated with a weight $w_{i,j}$ that captures a network link property of interest. Properties of hosts such as its unique identifier, remaining battery power, processing capacity, or available storage, are formally represented as a profile in a variable h_i that is associated with node v_i in V. Properties of an application such as its unique identifier, expected lifetime, system support requirements, provided interface, or native data format are captured in a profile represented as as a variable $a_{i,\alpha}$ that corresponds to an application α that is associated with a node v_i in V.

We are interested in issuing queries from a particular node in the network. As such, we are interested in forming a network overlay structure rooted at that reference node. Therefore, the remainder of this discussion focuses on the use of the connected subgraph of G that contains the reference node. We refer to this connected subgraph as $G' = (V', E')$ where V' is the of nodes and E' is the set of edges that form the connected subgraph in which the reference node is a member.

The network constraints provided as part of a context policy are represented as a metric recursively defined over edge weights in G'. The problem of determining how to propagate a query over an application-specified portion of the mobile ad hoc network can be solved by using the cost of the paths as determined by the metric over edge weights $w_{i,j}$ to construct a shortest path cost tree. For a path $p_{i,j}$ originating at node i and ending at node j, the path cost is denoted $c_{i,j}$. To calculate the cost of a path $p_{i,k}$, a node uses the metric over the cost of $p_{i,j}$ and $w_{j,k}$, i.e., the weight of the edge between node j and k. To ensure that construction of the tree terminates, the metric must be strictly increasing and must be bounded. The resulting subgraph of G' (call it $\bar{G}'$) represents the reference node's context that corresponds to the network constraints of its context policy.

To determine which nodes in $\bar{G}'$ can allow its applications to process the query and deliver results, we associate a boolean variable $hSat_i$ with each node v_i in $\bar{G}'$ that states whether or not the host constraints portion of a context policy is satisfied by the node's associated host profile h_i. Nodes for which the value of $hSat_i$ is true can propagate query results on behalf of the applications running on the local node. The nodes for which $hSat_i$ is false may still propagate the query and route its replies for other hosts in the context but cannot be the originating host of replies to a query. The nodes for which $hSat_i$ is true, along with the edges connecting them, form a subgraph of $\bar{G}'$. Only applications residing on nodes within this subgraph whose application profiles satisfy the application constraints portion of the context policy are eligible to contribute query results. To determine which applications operating on a node can process the query and deliver results, we associate a boolean variable $aSat_{i,\alpha}$ with each node v_i in G' that states whether or not the application constraints portion of a context policy is satisfied by the application α's profile on the node v_i. Applications for which the value of $aSat_i$ is true can submit query results to be propagated to the query issuer.

6.2 Context Acquisition Protocol

The context acquisition protocol constructs a network overlay that corresponds to an application-supplied context policy that is provided at the time a query is submitted for execution. The protocol constructs the overlay in a distributed fashion, propagating the query as hosts are determined to be eligible for inclusion in the overlay. The constructed overlay is used to deliver replies back to the issuer of the query. For persistent queries, the overlay is maintained in the face of changes to the network, hosts, and applications. We begin by describing how queries are used to construct the overlay and initiate replies. We then describe how the network overlay is maintained for persistent queries.

6.2.1 Building the Network Overlay

Each query is comprised of a unique query identifier and the unique identifier of the issuing application. The query also contains the identifier of the host that sent the query. Notice that this field is not necessarily the same as the initiator of the query; as the query is propagated throughout the network, the state of this field is updated. The query also includes the path to the current node, and the cost of the path. So that the network constraints can be evaluated at each node, the metric used to evaluate the cost of the path to neighboring hosts is included in the query along with the bound on the metric. To support the evaluation of host and application constraints at each node, these pieces of information are included in the query as well. The query also includes, of course, the application's content-based request for data.

Each host that receives a query uses the information provided as part of the query as well as some local information to determine which of its neighbors also satisfy the network constraints and thus should receive the query. The local information required to update the query and determine which neighbors are eligible to receive it includes a record of the host's unique identifier (used to update the sender and path fields of the query); the host's parent in the shortest cost tree (used to maintain the tree, as discussed in the next section); the cost of the path to the current host (to update the path cost information encapsulated in the query); and a set of connected neighbors, the weight of the link that connects the current host to each neighbor, the metric, and the bound on the metric (to determine which neighbors should receive the updated query). Since there may be multiple paths available to each of the neighbors through the current node with differing costs, the host also keeps track of all paths to neighboring nodes and their costs. Keeping track of all paths allows the protocol to ensure that a query that arrives at a host will always be sent along the shortest cost path to its neighbors. For each neighbor that satisfies the network constraints (i.e., the cost of the path to the neighbor is within the bound), the host propagates the query to the neighbor along the shortest cost path.

Once a host has evaluated the network constraints metric and propagated the query to other nodes, it evaluates the host constraints against the local host profile. If the host constraints are satisfied, the application constraints are evaluated for each application running on the local host. If an application's profile satisfies the constraints, the application's content-based data request can be matched against the application's data. At this point, a reply is generated and the host can propagate the reply to the query issuer using the return path associated with the query.

6.2.2 Maintaining the Network Overlay

We assume that hosts connected by a network link are both notified of changes on the associated edge weight. When a weight change notification arrives, each node takes action to reconfigure the network overlay. The action that the node takes is determined by what kind of weight change has occurred. The various weight changes are described below along with the response that a node should make to reconfigure the network overlay. The actions taken by network nodes to respond to changes in edge weights that correspond to network constraints are handled exactly as in [12].

Change in Weight of Edge to Parent

The weight change will causes the cost of the path to the current node to the parent to be increased or decreased, and each of these cases is handled differently.

Path cost increases. If the cost of the path through the parent is increased, then it is possible that the host has already received notice of a shorter cost path to the query issuer. The host records the new, increased path cost through the parent and stores this information in its list of possible paths to the query issuer. The host then searches its lists of alternate paths to the query issuer. If a shorter cost path exists, then the hosts sets the cost of the alternate path as its new cost, sets a new parent, calculates the cost of the paths to neighboring nodes using the query's association metric, and propagates the updated path and path costs to all neighbors within the bound. Notice that since a shorter cost path is available, the network overlay may grow to include nodes to which the path cost was previously outside the bound.

If the host does not find a shorter cost path and the new, increased path cost is still within the bound, the host propagates the new, increased path cost to neighboring nodes that should be in the context. Notice that this can result in the context shrinking, since nodes that were previously part of the context are no longer considered part of the context. In this case, the node propagates a clean-up message to nodes that are no longer considered to be part of the context.

Path cost decreases. If the cost of the path through the parent is decreased, then the shortest path just became shorter. The parent remains the same and the current cost of the path is updated. This updated path cost is sent to all neighbors whose updated path cost falls within the bound. Notice that the network overlay may grow in this case, since neighbors whose path cost were previously outside of the bound may now satisfy the network constraints.

Change in Weight of Edge to Non-parent

The weight change will either cause the cost of the path through the non-parent to the current node to be increased or decreased.

Path cost through non-parent increases. If the path cost through a non-parent increases, then the weight change did not result in a shorter cost path. The host stores the path and its cost, however, as an alternate path that may be used in the future.

Path cost through non-parent decreases. If the path cost through a non-parent decreases, then a shorter cost path to the query issuer may be available. The host compares the new, decreased path cost to its current path cost to the query issuer. If the new cost is less than the current cost, then the host sets the sender as its new parent, sets the path as its new path, and sets the cost as its new cost. The host then uses the new cost to calculate the cost of the paths to neighboring nodes using the query's association metric, and propagates the updated path and path costs to neighbors within the bound. Once again, since a shorter cost path is available, the network overlay may grow to include nodes to which the path cost was previously outside of the bound. The host sends a clean-up notification to these nodes.

6.2.3 Accounting for Host, Application, and Data Changes

If the host constraints are satisfied at a particular host, the application constraints are evaluated for each application running on the local host. If an application's profile satisfies the constraints, the application's content-based data request can be matched against the application's data. However, the profiles of hosts and applications may change over time. Each time that a host or application profile is changed, the corresponding constraint is evaluated over the updated profile. If the constraints are still satisfied, no action is taken. However, if either of the constraints is no longer satisfied, a report is generated for the query issuer. If the host constraint is not satisfied, the report indicates that data provided by applications running on that host is no longer valid. If the host constraint is satisfied but the application constraint is not, the report indicates that data provided by that application is no longer valid.

As new data satisfying the content-based data request of the query becomes available, a reply is generated and propagated to the query issuer. If data that was previously reported as a result is deleted, a reply indicating the unavailability of data is generated and propagated to the query issuer. When a host that has previously provided results to a query on behalf of applications running on it becomes disconnected from the query issuer, a report is generated by the host's parent in the network overlay to indicate that results reported by the disconnected node (and its subtree) are no longer valid.

6.3 Protocol Evaluation

We used the ns-2 network simulator [6, 24] to evaluate our protocol over a 100 node mobile ad hoc network contained within a 1000x1000m^2 area. We used transmission range to adjust the density of the mobile ad hoc network and give results for sparsely, optimally, and densely connected networks. Results for static ad hoc networks show that the optimum number of neighbors for each node is 6 [20]. More recent work in mobile ad hoc networks shows us that that there is not a global optimum number of neighbors [31]; rather, the optimum number of neighbors is dependent on the mobility of the nodes in the network. For our simulations, we vary the transmission range among 50, 150, and 250 meters to model sparsely, optimally, and densely connected networks, respectively. Results from [12] show network densities versus transmission range under the same simulations settings used here. At 50 meters, each network node on average has 1 neighbor. At 150 meters, each node has about 9 neighbors. At 250 meters, each node has about 24 neighbors. These densities correspond appropriately with the expected optimum for the mobility model and speed that we use in our simulations, a random waypoint mobility model with 0s pause time and speed in the range of 1 to 20 m/s. Each result shown in the graphs in the following sections is the average of 50 simulation trials. To simplify the simulations, we define contexts by using a network constraint over hop count.

We begin by investigating how accurately the network overlay constructed as a result of one-time query evaluation reflects the context that existed at the time the query was issued. If we use the simulator to take a snapshot at the instant we issue the query, we can see which hosts satisfy the context policy and should belong to the context. We can compare this snapshot of the environment with the set of participants in the context and quantify the result. We call this the *consistency* of the context. For each node that is present in the snapshot of the environment but does not receive the query, we consider an error in constructing a consistent network overlay.

Next, we describe how reliably the results for a one-time query are returned to the application by our context acquisition protocol. We use the simulator to take a snapshot to determine which nodes should provide a result to the query issuer and then compare this set of expected results with the received results.

We conclude by evaluating the tradeoffs between using one-time and persistent queries to acquire continuous data. We calculate the overhead required to construct the context and deliver replies for both one-time queries and persistent queries. We also show how often a one-time query must be issued to mimic the semantics of persistent queries.

6.3.1 Evaluating Context Consistency for One-time Queries

Because the context acquisition protocol relies on the use of an underlying neighbor discovery package, we created a network discovery package in the simulator that

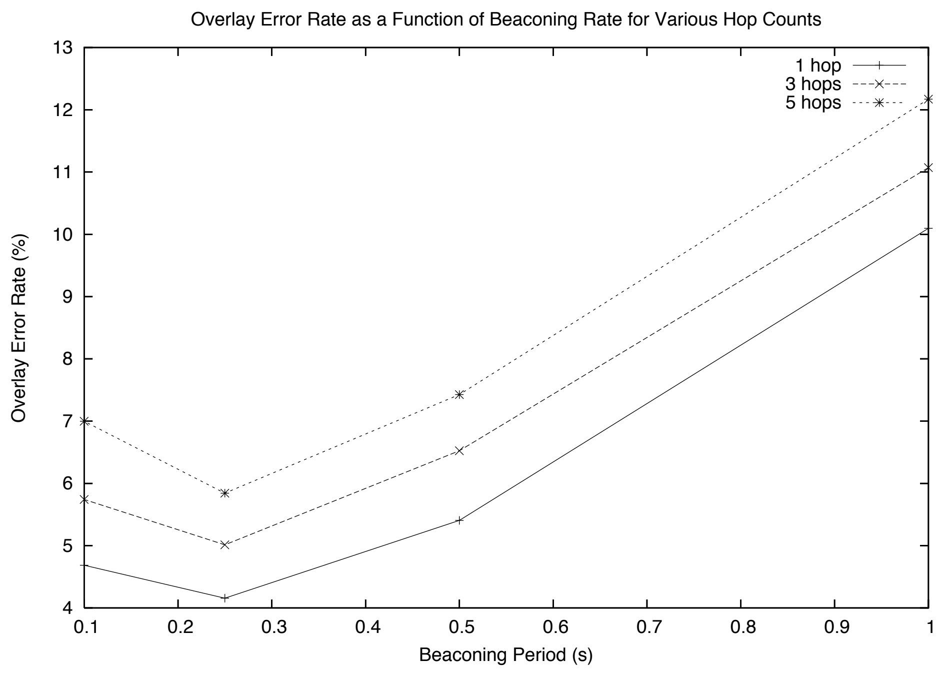


Figure 6.1: Measuring the Impact of Beaconing Rate on Consistency for One-time Queries in a Sparsely Connected Network

maintains a neighborhood for each node. The network discovery component is distributed, so that each host maintains a list of one-hop neighbors in its neighborhood. To be considered a neighbor, the network component must hear two beacons from a one-hop neighbor in consecutive beaconing periods. A neighbor is no longer part of a node's neighborhood when no beacons are received from it in three consecutive beaconing periods. This network discovery component is parameterizable, allowing for the beaconing period to be adjusted. The beaconing period impacts how well the neighborhood managed by the network discovery component reflects the state of the environment. Since the list of neighbors within the neighborhood is used by our context acquisition protocol to decide how to propagate the query, the beaconing period in turn impacts the consistency with which the context acquisition protocol defines the context as it exists at the time the query was issued.

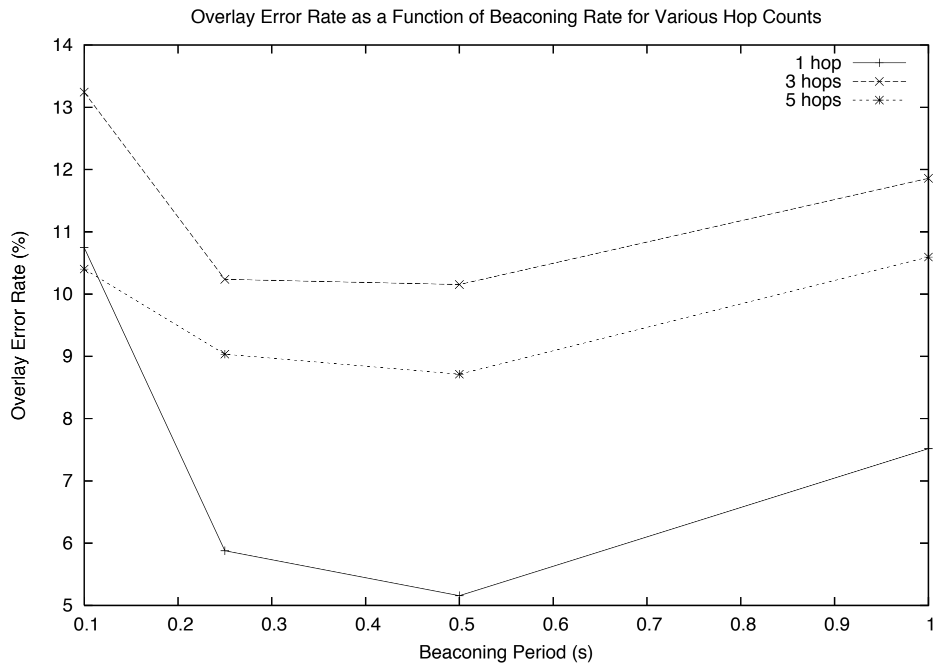


Figure 6.2: Measuring the Impact of Beaconing Rate on Consistency for One-time Queries in an Optimally Connected Network

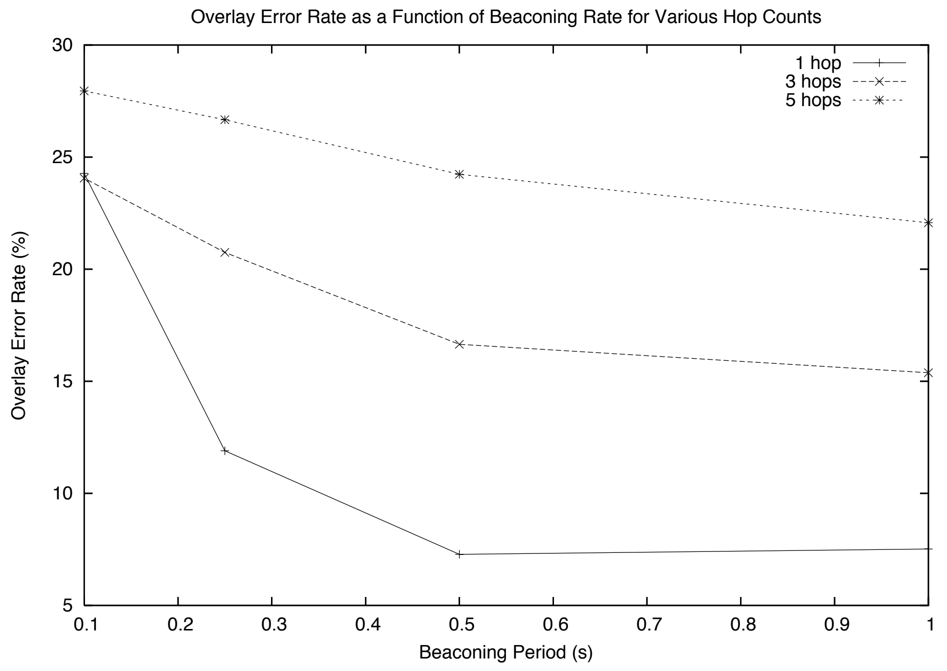


Figure 6.3: Measuring the Impact of Beaconing Rate on Consistency for One-time Queries in a Densely Connected Network

We varied the beacon rate so that one beacon is sent every 1000ms, 500 ms, 250 ms, and 100 ms. Figures 6.1, 6.2, and 6.3 illustrate how the beaconing period affects the consistency of the a constructed network overlay with respect to the state of the environment. The graphs show the results for issuing a single query across contexts of varying sizes in sparse (Figure 6.1), optimum (Figure 6.2), and dense networks (Figure 6.3).

It is of interest to first note that independent of the beacon rate, the graphs show that the accuracy of the context deteriorates as the context is defined to be larger. In larger contexts, the query must be propagated across multiple hops to form the network overlay. As the query is propagated across a larger portion of the network, it becomes more likely that hosts may move in and out of communication range during construction of the context. This decrease in consistency for larger contexts is especially apparent as the network becomes more dense. As the density of the network increases, a broad context definition results in a network overlay that contains a large portion of the nodes in the network, and these nodes all broadcast to propagate a message to one another at roughly the same time. This may result in what is referred to as a broadcast storm, which is likely the reason for the extremely high error rate for three- and five-hop contexts in an extremely dense network. These results suggest that an application that requires extremely consistent results would be better served using a small context definition. Applications that require strong guarantees should avoid using large context definitions in densely populated networks.

The graphs show that as the beacon rate is gradually increased, the consistency of the context improves for all network densities. At some point, however, increasing the beaconing rate negatively impacts the accuracy of the context. This is likely due to network congestion caused by very frequent beaconing. The 802.11 standard uses the CSMA/CA MAC layer, which means that nodes must detect that the the medium is available before they can broadcast a message. If the medium is not available, the node backs off and waits to broadcast the message. While a node waits to broadcast the message, hosts may move so that they are no longer in the context. The point at which frequent broadcasting becomes a problem is, as expected, related to the density of the network. In extremely dense networks, frequent beaconing becomes a problem more rapidly since an increased number of neighbors are simultaneously trying to send beaconing messages more frequently. As the network load is increased by issuing additional queries over the network, we can expect that the problem is compounded.

We select the highest possible beaconing rate that provides an increase in context consistency but does not create debilitating network congestion in sparse, optimum, and dense networks. Unless otherwise specified, we use a beaconing rate of 500 ms in the remainder of experiments. Though we are aware that increasing the beaconing rate increases the overhead required to construct a context, we are willing to trade overhead for consistency. It should be noted, however, that for scenarios in which reduced overhead is more desirable, the results indicate that the consistency with which the network overlay is constructed with the longest beaconing period (1000ms) is still reasonable.

6.3.2 Evaluating the Reliability of Result Delivery for One-time Queries

We are interested in the reliability with which replies are delivered to the query issuer. Query results are routed along the reverse path used to propagate the query to nodes in the context. This path information is recorded in the reply and does not change. Since hosts may move as the result is being propagated back to the issuer, the reverse path used to route the reply may not be available and replies may be dropped. We measure the number of replies to a query that are generated and determine the number that are delivered successfully. Since the reply route is static, altering the beaconing rate does not impact the success of reply delivery. Therefore, we evaluate reply success only for the previously chosen beacon period of 500 ms.

Figure 6.4 illustrates the reliability with with replies are delivered to a query issuer. The graph show the results for issuing a single query across contexts of varying sizes in sparse, optimum, and dense networks. The graph shows that the delivery success rate is somewhat erratic in a sparsely connected network in comparison to the delivery success results for optimal and densely connected networks. The delivery success rate for one-hop contexts in sparsely connected networks is also much lower than that for optimally and densely connected networks. In optimally connected and densely connected networks, results are delivered approximately 96% and 94% of the

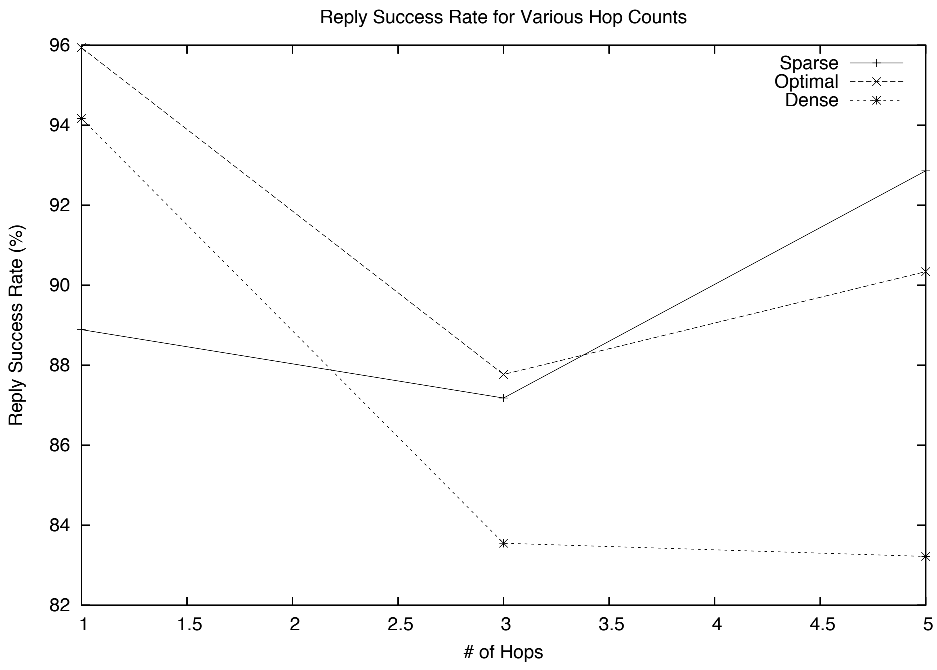


Figure 6.4: Reply Delivery Rate for One-time Queries in Various Networks

time in one-hop contexts. As we would expect, the delivery success rate drops for larger, three-hop contexts to about 88% and 84% for optimal and dense networks, respectively. For densely connected networks, the reply delivery success rate continues to drop as the context grows. Interestingly, the same is not true for sparsely or optimally connected networks, in which the delivery success rates increase to 93% and 90%, respectively. The conclusion drawn from this data is that replies are delivered with reasonable success in sparsely, optimally, and densely connected networks. To improve the success of reply delivery, the application programmer should consider the expected conditions of the network environment and choose a context of the appropriate size to deliver the desired delivery success rate.

6.3.3 Comparing One-time and Persistent Queries

It is possible to mimic the semantics of a persistent query by using a one-time query to poll for data. The accuracy with which a one-time query mimics a persistent query

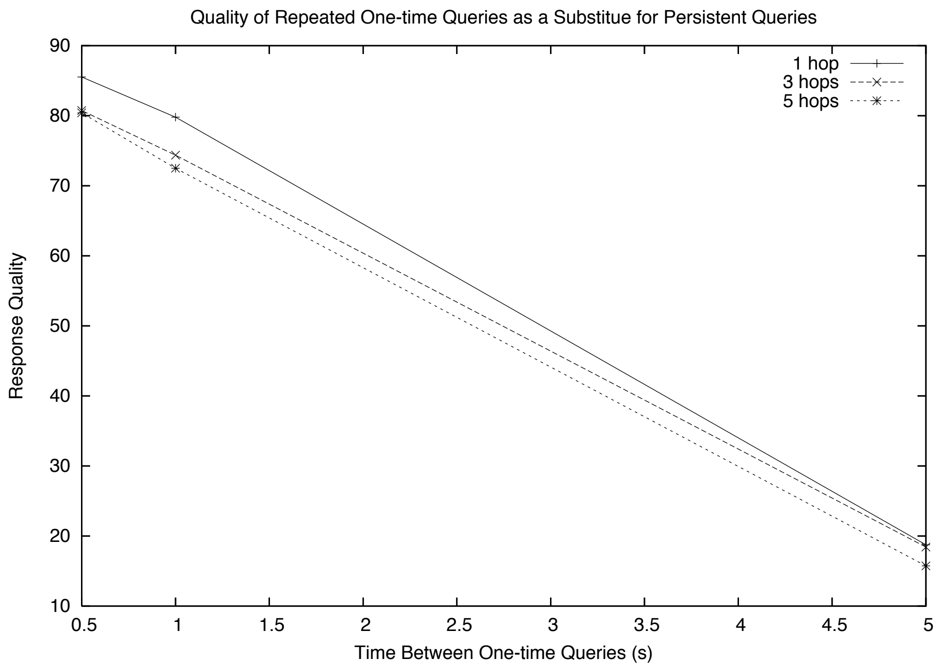


Figure 6.5: Approximating a Persistent Query by Polling with Queries in a Sparsely Connected Network

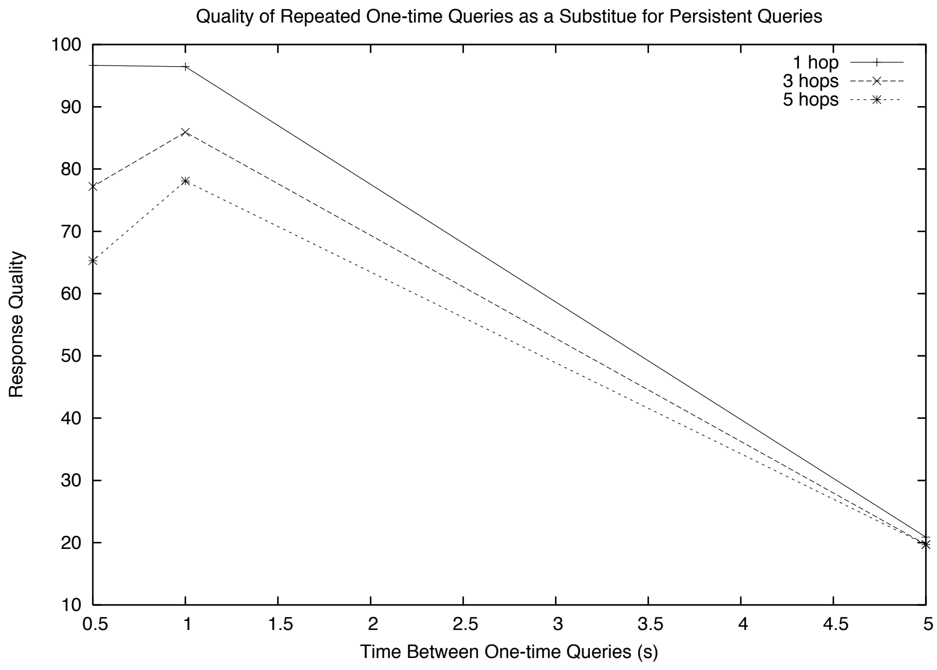


Figure 6.6: Approximating a Persistent Query by Polling with Queries in an Optimally Connected Network

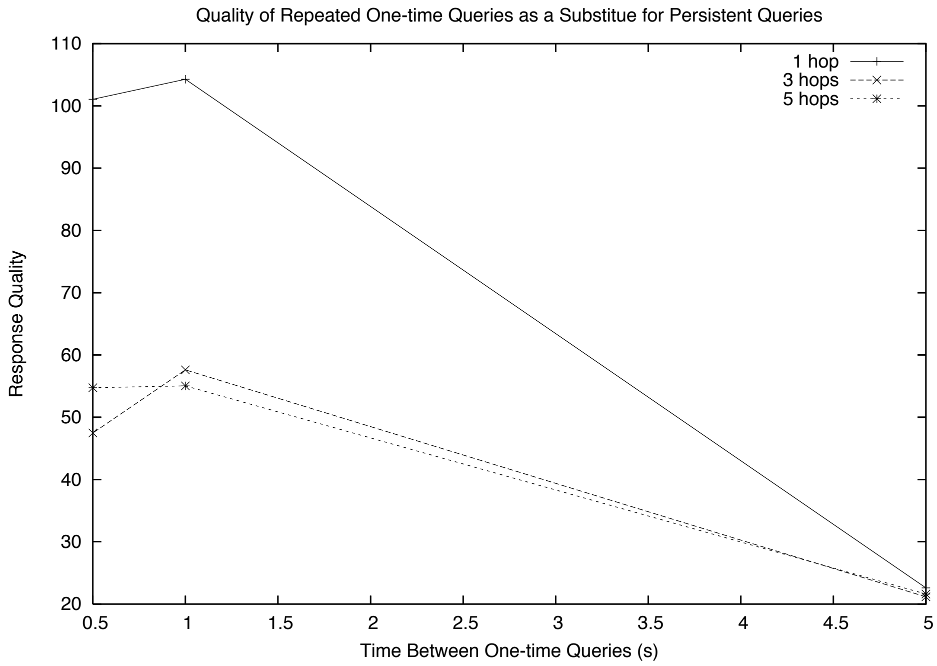


Figure 6.7: Approximating a Persistent Query by Polling with Queries in a Densely Connected Network

is dependent on the frequency with which the one-time query is issued and how often the data changes. We compare the number of query results issued by the persistent query to the number of new results obtained across multiple one-time queries. We call this the response quality. Figures 6.5, 6.6, 6.7 show the quality for queries evaluated in sparse, optimal, and dense networks. In these experiments, we randomize the rate at which data changes and increase how frequently the one-time query is issued during the evaluation of a persistent query. In our experiment, each piece of data changes at a random rate between one change per second to one change every 10 seconds. We compared a persistent query evaluated over 30 seconds to a polling approach that repeatedly issues a one-time query evaluated every 500 ms, 1 s, and 5 s within the same 30 second time span as the persistent query.

The graphs show that as the frequency with which a one-time query is issued is increased, the response quality of a query increases. However, as the network becomes more dense, issuing a query more frequently results in a decrease in the response quality, especially for larger contexts. Once again, this is likely due to network congestion.

Figure 6.8 shows the overhead associated with approximating persistent queries in an optimally connected network when using a polling approach in which a one-time query is issued every 500 ms. Again, the persistent query and the polling approach are evaluated over a 30 second time interval with each node's data changing at random rate in the range of 1 data change per second to 10 data changes per second. The number of overhead messages for the one-time query shown include those which are required to construct the context. The number of overhead messages counted for the persistent query include those required to initially construct the context as well as those required to maintain it, i.e., those required to propagate edge weight change notifications and queries. The one-time query that is shown in the graph was issued three times during the execution of the persistent query. As shown in the graph, the persistent query requires fewer messages to provide results which, as shown in the previous subsection, are more consistent with the data available in the network. In a 3 hop context, we see a savings of about 8,000 messages while at 5 hops we see a savings of about 10,000 messages when using a persistent query. These results suggest that the use of a persistent query is likely worth the cost of overhead associated

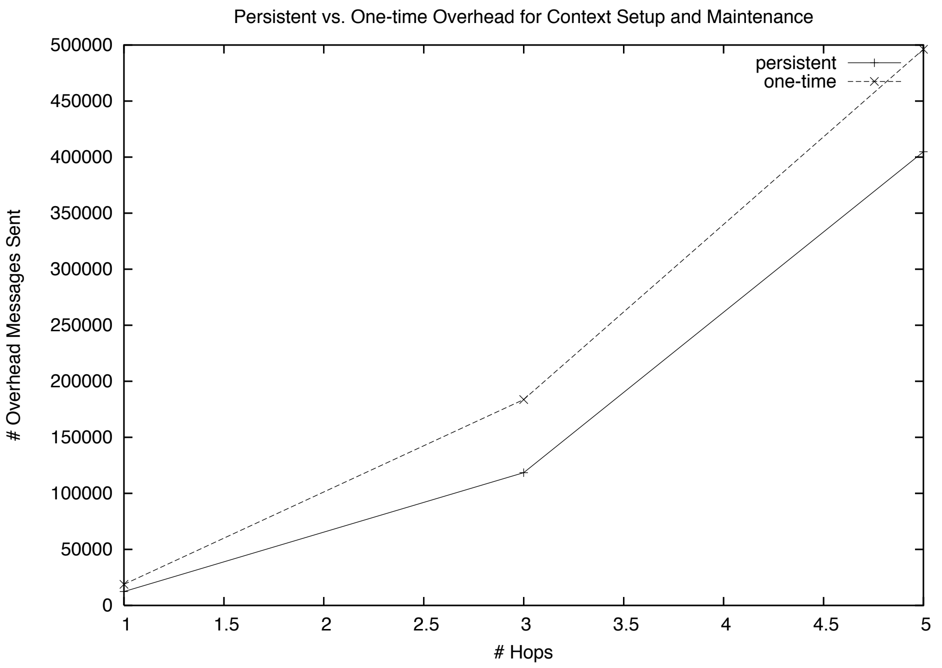


Figure 6.8: Message Overhead for Constructing a Context Using One-time and Persistent Queries in an Optimally Connected Network

with maintaining the context in order to acquire results that are consistent with continuously changing data.

6.4 Protocol Optimizations

When evaluating persistent queries over an application-specified portion of the mobile ad hoc network using our query model, it is imperative to address the frequent disconnections that are likely to occur within the network overlay due to the mobility of hosts. Such disconnection is handled through the reconfiguration protocol, discussed earlier in this chapter, which mends the network overlay. However, the execution of a reconfiguration protocol can be expensive in a highly dynamic environment and can disrupt the process of executing queries. To address this issue, we present optimizations to the reconfiguration protocol that attempt to limit the amount of reconfiguration required when changes occur in the network that impact the cost of a path.

6.4.1 Truncating Propagation of a Path Cost Change Notification

In the maintenance protocol described in the previous section, each node is required to propagate changes on edge weights that impact the cost of the path through the node. A very simple optimization trades the possibility of a larger, more inclusive context for fewer reconfigurations. In this optimization, if a node that is already part of the context receives notice of an edge weight change that results in a shorter cost path to the node, the node does not propagate the updated path cost to its neighbors. Instead, the context determined prior to the edge weight change continues to be maintained. This may result in a case in which nodes that are eligible for inclusion in the context given the updated path cost are actually not included, but the number of messages required to perform reconfiguration of the network overlay is reduced. This optimization is available as part of the QueryME implementation.

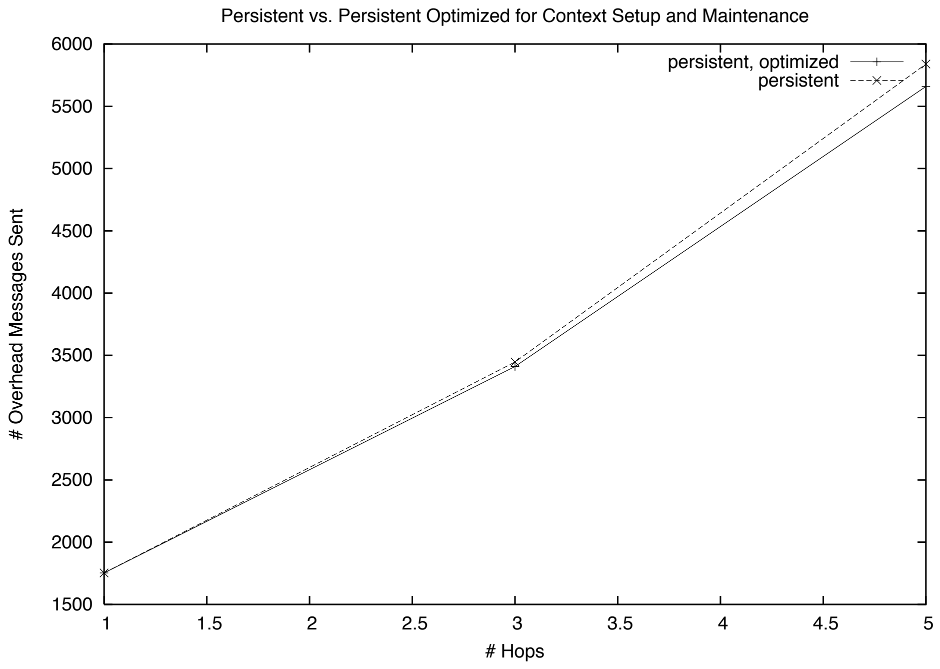


Figure 6.9: Message Overhead for Constructing a Context Using Persistent and Optimized Queries in a Sparsely Connected Network

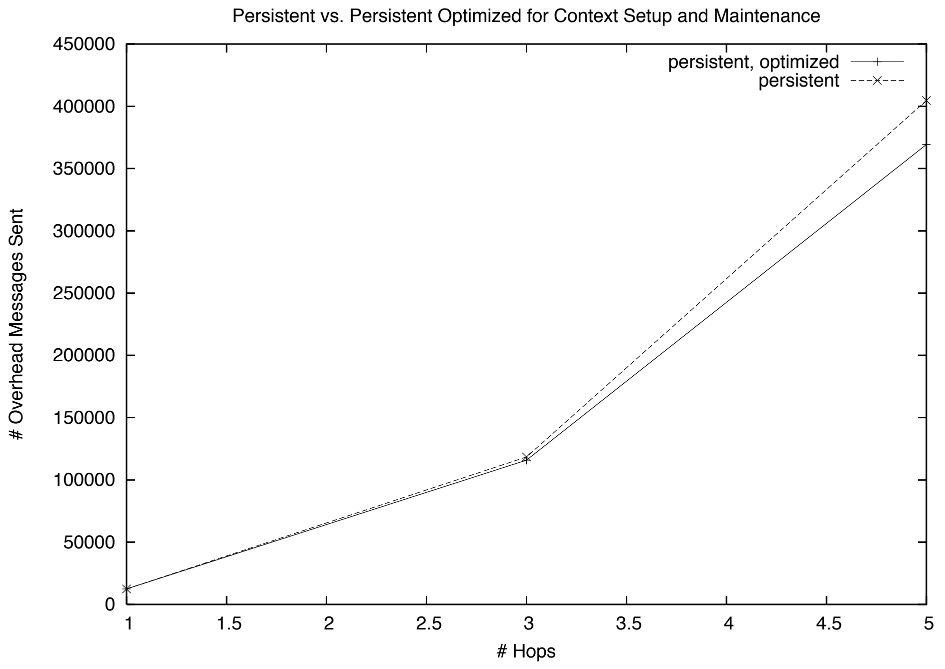


Figure 6.10: Message Overhead for Constructing a Context Using Persistent and Optimized Queries in an Optimally Connected Network

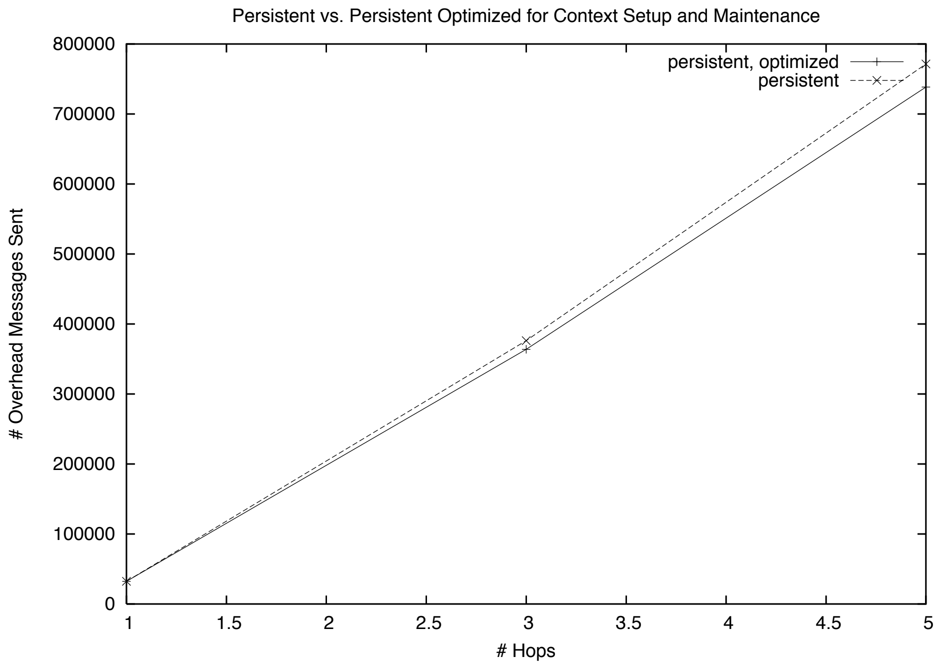


Figure 6.11: Message Overhead for Constructing a Context Using Persistent and Optimized Queries in a Densely Connected Network

We evaluated this optimization using the ns-2 simulator and the network simulation setup described in the previous section. Specifically, we compared the overhead incurred by issuing a persistent query and an optimized persistent query over a duration of 30 seconds. The results are shown for sparsely connected networks in Figure 6.9, for optimally connected networks in 6.10, and for densely connected networks in 6.11. As the graphs show, the optimized persistent query provides a small amount of savings in terms of message overhead, regardless of network density.

6.4.2 Building a More Stable Overlay

A common source of path cost change in mobile ad hoc networks is the disconnections that occur in highly dynamic mobile environments. One way that we can reduce the amount of reconfiguration is to attempt to limit the disconnections which occur within the overlay by utilizing mobility information to build a more stable network overlay.

We provide each host in the network with the ability to generate a motion profile which describes its future movements as a function of time and location. Our approach to motion profiles is based on the work in [34], which uses motion profiles to support long-lived use of mobile services. It is assumed that each host has absolute and perfect knowledge about its future motions. Each motion profile is given as a function which maps the given time to the host's location in space.

We use an epidemic algorithm to gossip motion profiles. Using the motion profiles, we attempt to construct a stable version of the network overlay in order to reduce the need to employ an expensive reconfiguration protocol. Each host is equipped with an inRange function that maps the host's motion profile, the motion profile of another host, and a time to a boolean value. Essentially, this function uses the provided time parameter to find the location for that time in each of the motion profiles and determines if those locations are within communication range of one another. We integrate this function into our neighbor discovery component so that a host is only added to the neighborhood if it satisfies the inRange function for a given period of time that corresponds to the time it takes to evaluate a query. This time is related to the time that it takes to process the query and return results over the application-specified region. By integrating the function into the neighbor discovery support package that is used by the QueryME middleware, we ensure that only neighbors which will be in range for a satisfactory period of time will be considered as part of the context.

For both of the context acquisition protocol optimizations presented in this section, there is a tradeoff between the consistency with which the application's specified context is reflected by the network overlay and the reduced overhead associated with using motion profiles in neighbor discovery. In a highly dynamic, densely populated mobile ad hoc network, a search application used by a shopper to find a cup of coffee is likely best served by delivering only a fraction of the available context and so the optimization should be employed. However, a search and rescue application operating in similar network conditions must find all of the victims at a disaster site. In this case, the optimization should not be applied. Overall, the decision of when to apply the optimization is based on the needs of the application. The application using the protocol to acquire context must determine which solution is more appropriate to support its needs in the current environment.

6.5 Chapter Summary

This section describes the context acquisition protocol used by the QueryME middleware to execute queries over the mobile ad hoc network. The mobile ad hoc network is treated as a weighted graph, allowing for the construction of the network overlay that encapsulates the network constraints portion of a context policy to be computed using a specialized distributed algorithm for constructing a shortest cost path tree. Host and application constraints are applied on top of the tree to determine which nodes can evaluate the data request of a query. The reconfiguration protocol that performs maintenance of the context for persistent queries is based upon the protocol in [12]. The evaluation of the construction and reconfiguration protocols show the utility of one-time and persistent queries in various scenarios. Optimizations of the reconfiguration protocol are presented. Evaluation of these optimizations is needed to quantify the tradeoff in the accuracy with which they can be used to deliver context with the reduction in overhead that they provide.

Chapter 7

Conclusions

This dissertation has introduced a new programming model for developing context-aware applications based on the use of a query-centric abstraction that helps to reduce domain programmer effort and errors. The development of such an abstraction allows context-aware application developers to view the surrounding and constantly changing environment as if it were a locally accessible database. Using this abstraction, a programmer can largely ignore the complex tasks associated with context interactions across a mobile ad hoc network and can instead focus on implementation details that are specific to the application domain.

The key technical contribution of this dissertation is the extension of the traditional query model to allow context-aware applications to deal with the dynamic and open nature of ad hoc networks. First, since the ad hoc network can grow large, queries are executed only over the portion of the network that the application has declared to of interest by associating context policies with a submitted query. Second, to support the evaluation of queries over the continuously changing mobile ad hoc network environment, we extend the typical query model so that an application is notified when changes in the environment occur that may impact the availability and pertinence of delivered query results. To support this feature, queries are complemented by the incorporation of a new notion of persistent query evaluation that reactively reports newly available query results as well as changes in the state of previously reported query results. Third, since the mobile ad hoc network is heterogeneous, we adapt the query model to eliminate the need for a predetermined database schema and allow applications to provide and retrieve different kinds of context information. Our query-centric model provides a flexible and extensible approach to incorporating different types of context data by using an underlying representation of context information so that it can be treated in a uniform manner and associating each piece of context data with metadata. Queries are evaluated only over the metadata and tailored result processing strategies can be provided at the time the query is issued that detail how to perform data-type specific tasks such as in-network aggregation of data.

These concepts have been incorporated into the implementation of QueryME, a query-centric middleware that can be used for general context-aware application support. To capture the query-centric model within a middleware framework that can be generally applied to develop a wide range of context-aware applications, we have developed mechanisms to support application-specific definitions of context, protocols to collect context in the face of environmental change, and infrastructure support for tailored in-network processing of query results. Tailored query processing strategies that dictate the query's context policy, propagation policy, and reply processing policy are captured as mobile code elements and are provided by the application to the query system at the time the query is issued. These mobile code elements are dynamically distributed and executed within the network as the query is evaluated. Employment of mobile code allows us to support both a large class of query evaluation strategies as well as a wide range of data.

In this dissertation, we performed an evaluation of QueryME operations through simulation of its underlying protocols in a mobile ad hoc network. The simulation results suggest that it is feasible to utilize QueryME to support an application's context acquisition tasks. The use of QueryME middleware has also been demonstrated in application development to support the implementation of Gander, a context-aware search engine application that can be used to find webpages within a logical region of interest in a mobile ad hoc network. From a software engineering perspective, the development of Gander demonstrates how one can exploit a general purpose query-centric model to construct applications that share the need to be responsive to context changes.

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] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[4] 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.

[5] N. Borch. Improving semantic routing efficiency. In *Proceedings of Second International Workshop on Hot Topics in Peer-to-Peer Systems (HOT-P2P'05)*, July 2005.

[6] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

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

[8] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 9(3):332–383, August 2001.

[9] 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.

[10] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 2001.

[11] 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.

[12] 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.

[13] G. Hackmann, C. Julien, J. Payton, and G.C. Roman. Supporting generalized context interactions. In T. Gschwind and C. Mascolo, editors, *Proceedings of the Fourth International Workshop on Software Engineering and Middleware (SEM 2004)*, volume 3437 of *Lecture Notes in Computer Science*, pages 91–106. Springer-Verlag, March 2005.

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

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

[16] Q. Huang, C. Julien, and G.C. Roman. Relying on safe deistance to achieve strong patitionable group meembership in ad hoc networks. *IEEE Transactions on Mobile Computing*, 3:192–205, 2004.

[17] 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.

[18] S. Joseph. Neurogrid: Semantically routing queries in peer-to-peer networks. In *Proceedings of the International Workshop on Peer-to-Peer Computing (associated with NETWORKING 2002)*, pages 202–214, May 2002.

[19] 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.

[20] L. Kleinrock and J. Silvester. Optimum transmission radii in packet radio networks or why six is a magic number. In *Proceedings of the IEEE National Telecommunications Conference*, pages 4.3.1–4.3.5, 1978.

[21] C. Lu, G. Xing, O. Chipara, and C.-L. Fok. A spatiotemporal query service for mobile users sensor networks. In *Proceedings of the 25th International Conference on Distributed Computing Systems*, pages 381–390, 2005.

[22] 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.

[23] 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.

[24] S. McCanne and S. Floyd. ns network simulator.

[25] M. McGlynn and S. Borbash. Birthday protocols for low energy deployment and flexible neighbor discovery in ad hoc wireless networks. In *Processings of the ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 137–145, 2001.

[26] 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.

[27] Nielsen//NetRatings. Nielsen netratings search engine ratings, July 2005.

[28] A. Omicini and F. Zambonelli. The TuCSoN coordination model for mobile information agents. In *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, June 1998.

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

[30] 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.

[31] E. Royer, P. Melliar-Smit, and L. Moser. An analysis of the optimum node density for ad hoc mobile networks. In *Proceedings of the IEEE International Conference on Communications 2001*, pages 857–861, 2001.

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

[33] 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.

[34] R. Sen, R. Handorean, G.-C. Roman, and G. Hackmann. Knowledge-driven interactions with services across ad hoc networks. Technical report, Washington University, Department of Computer Science and Engineering, St. Louis, Missouri, 2005. submitted to the International Journal on Collaborative Information Systems.

[35] 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.

[36] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[37] C. Tempich, S. Staab, and A. Wranik. Remindin': Semantic query routing in peer-to-peer networks based on social metaphors. In *Proceedings of the 13th International Conference on the World Wide Web (WWW'04)*, pages 640–649. ACM Press, 2004.

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

[39] David A. Wheeler. Sloccount (http://www.dwheeler.com/sloccount).

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

Vita

Jamie Jenelle Payton
Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045
One Brookings Drive
St. Louis, MO 63130-4899

Date of Birth	January 15, 1979
Place of Birth	Tulsa, Oklahoma

Education

B.S. in Computer Science, University of Tulsa, May 2001

M.S. in Computer Science, Washington University in St. Louis, May 2004

D.Sc. in Computer Science, Washington University in St. Louis, August 2006

Professional Experience

Mobile Computing Laboratory
Washington University in Saint Louis
Graduate Research Assistant
August 2001 - August 2005
Research focused on providing abstractions to simplify the development of context-aware applications designed for use in mobile ad hoc networks. Developed formal model of context-awareness and a query-centric middleware to aid in application design and development.

Software Engineering and Architecture Team
University of Tulsa
Undergraduate Research Assistant
Spring 1998 - July 2001

Investigated the impact of software architecture characteristics on the software integration process. Research focused on discovering which mismatches between architectural characteristics result in integration problems and the encapsulation of solutions within building blocks called integration elements.

Teaching Experience

CSE 537: Mobile Computing
Fall 2005
Co-Course Developer and Guest Lecturer
Assisted in determining course direction and developing the course syllabus, collected and organized course materials, developed and delivered a series of lectures on publish/subscribe systems for distributed and mobile environments, aided in the design of homework assignments.

CSE 501: Programming Concepts and Practice
Summer 2005
Instructor
Designed course syllabus; developed lecture materials, lab assignments, and exams; managed a teaching assistant; aided in grading assignments.

CSE 548: Concurrent Systems: Design and Verification
Fall 2003
Guest Lecturer
Presented lectures on synchronization and the dining philosophers problem, showing solutions in the UNITY notation and outlining correctness proofs.

CSE 730: Distributed System Design Research Seminar
Fall 2001, Spring 2002
Organizer
Assisted in developing the seminar theme, facilitated selection of papers for presentation, maintained course website, and participated in seminar discussions.

Publications

Journal Publications

1. G.-C. Roman, J. Payton. A Principled Exploration of Coordination Models. Theoretical Computer Science, 336, 2005, pp. 367-401.

2. G.-C. Roman, J. Payton. A Termination Detection Protocol for Use in Mobile Ad Hoc Networks. Automated Software Engineering Journal, 12(1), January 2005, pp. 81-99.

3. L. Davis, R. Gamble, J. Payton. The Impact of Component Architectures on Interoperability. The Journal of Systems and Software, 61(1), March 2002, pp. 31-45.

Conference/Workshop Publications

1. G. Hackmann, C. Julien, J. Payton, and G.-C. Roman. Supporting Generalized Context Interactions. Proceedings of the 4th International Workshop on Software Engineering and Middleware, co-located with ASE'04, Linz, Austria, September 2004, pp. 91-106.

2. C. Julien, G.-C. Roman, and J. Payton. Context-Sensitive Access Control for Open Mobile Agent Systems. Proceedings of the 3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2004), co-located with ICSE'04, Edinburgh, Scotland (UK), May 2004, pp. 42-48.

3. J. Payton, C. Julien, and G.-C. Roman. Context-Sensitive Data Structures Supporting Software Development in Ad Hoc Networks. Proceedings of the 3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2004), co-located with ICSE'04, Edinburgh, Scotland (UK), May 2004, pp. 34-41.

4. G.-C. Roman, C. Julien, and J. Payton. A Formal Treatment of Context Awareness, (invited paper), Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE'04), Lecture Notes in Computer Science 2984, Barcelona, Spain, March 2004, pp. 12-36.

5. G.-C. Roman and J. Payton. Mobile UNITY Schemas for Agent Coordination (invited paper). Proceedings of the Tenth International

Workshop on Abstract State Machines, Lecture Notes in Computer Science 2589, Springer, March 2003, pp. 126-150.

6. R. Handorean, J. Payton, C. Julien, and G.-C. Roman. Coordination Middleware Supporting Rapid Deployment of Ad Hoc Mobile Systems. First International ICDCS Workshop on Mobile Computing Middleware (MCM03), May 2003, pp. 362-368.

7. C. Julien, J. Payton, and G.-C. Roman. Reasoning About Context-Awareness in the Presence of Mobility. Proceedings of the 2nd International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA03), Electronic Notes in Computer Science 97, July 2004, pp. 259-276.

8. J. Payton, G. Jonsdottir, D. Flagg, R. Gamble. Merging Integration Solutions for Architecture and Security Mismatch. Proceedings of the International Conference on COTS-Based Software Systems (ICCBSS 2002), Lecture Notes in Computer Science 2255, February 2002, pp. 199-208.

9. L. Davis, R. Gamble, J. Payton, G. Jonsdottir, D. Underwood. A Notation for Problematic Architecture Interactions. Proceedings of the 9th International Symposium on Foundations of Software Engineering (ESEC/FSE-9), 2001, pp. 132-141.

10. J. Payton, L. Davis, D. Underwood, R. Gamble. Using XML for an Architecture Interaction Conspectus. International Workshop on Foundations of Coordination. Proceedings of XML Technologies and Software Engineering Workshop (XSE 2001), co-located with the 23rd International Conference on Software Engineering, 2001.

11. L. Davis, J. Payton, R. Gamble. How System Architectures Impede Interoperability. 2nd International Workshop On Software and Performance (WOSP 2000), 2000, pp. 145-146.

12. J. Payton, R. Gamble, S. Kimsen, L. Davis. The Opportunity for Formal Models of Integration. 2nd IEEE International Conference on Information Reuse and Integration (IRI-2000), 2000.

13. L. Davis, J. Payton, R. Gamble. Toward Identifying The Impact Of COTS Evolution On Integrated Systems. 2nd Workshop on Commercial off the Shelf Software, 2000.

14. J. Payton, R. Keshav, R. Gamble. System Development Using the Integrating Component Architectures Process. ICSE '99 Workshop on Ensuring Successful COTS Development, 1999.

Under Review
1. G.-C. Roman, C. Julien, and J. Payton. Modeling Adaptive Behaviors in Context UNITY. Technical Report WUCSE-2005-48, Washington University, Department of Computer Science and Engineering, St. Louis. 2005.

Professional Service

Reviewer
Journal of Theoretical Computer Science, Special Issue with Selected Revised Papers from Fundamental Approaches to Software Engineering (FASE), 2005.

Workshop Organizer
International Workshop on Discovery and Composition of Services in Dynamic Mobile Environments (DCSDME), co-located with the 5th IEEE International Conference on COTS-Based Software Systems (ICCBSS06), February 2006.

University Service

National Conference on Graduate Student Leadership
Technology Consultant (November 2005)
Washington University in Saint Louis

Computer Science and Engineering Graduate Student Association
President (Fall 2004- Fall 2005)
Washington University in Saint Louis

Association of Graduate Engineering Students
President (Spring 2005 present)
Washington University in Saint Louis

Preparing Future Faculty Seminar
Co-Organizer (Fall 2004 to Fall 2005)
Department of Computer Science and Engineering
Washington University in Saint Louis

Graduate Student Research Seminar
Organizer and Moderator (Fall 2003 to Fall 2005)
Department of Computer Science and Engineering
Washington University in Saint Louis

Honors

NSF Graduate Research Fellowship Honorable Mention (2001, 2002)

CRA-W Distributed Mentor Project Award (Summer 1999)

August 2006

Short Title: Queries in Mobile Ad Hoc Networks Payton, D.Sc. 2006