MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System

Buğra Gedik and Ling Liu

College of Computing, Georgia Institute of Technology
{bgedik,lingliu}@cc.gatech.edu

Abstract. Location monitoring is an important issue for real time management of mobile object positions. Significant research efforts have been dedicated to techniques for efficient processing of spatial continuous queries on moving objects in a centralized location monitoring system. Surprisingly, very few have promoted a distributed approach to real-time location monitoring. In this paper we present a distributed and scalable solution to processing continuously moving queries on moving objects and describe the design of MobiEyes, a distributed real-time location monitoring system in a mobile environment. Mobieyes utilizes the computational power at mobile objects, leading to significant savings in terms of server load and messaging cost when compared to solutions relying on central processing of location information at the server. We introduce a set of optimization techniques, such as *Lazy Query Propagation*, *Query Grouping*, and *Safe Periods*, to constrict the amount of computations handled by the moving objects and to enhance the performance and system utilization of Mobieyes. We also provide a simulation model in a mobile setup to study the scalability of the MobiEyes distributed location monitoring approach with regard to server load, messaging cost, and amount of computation required on the mobile objects.

1 Introduction

With the growing market of positioning technologies like GPS [1] and the growing popularity and availability of mobile communications, location information management has become an important problem [17, 10, 14, 5, 15, 13, 16, 9, 2] in mobile computing systems. With continued upsurge of computational capabilities in mobile devices, ranging from navigational systems in cars to hand-held devices and cell phones, mobile devices are becoming increasingly accessible. We expect that future mobile applications will require a scalable architecture that is capable of handling large and rapidly growing number of mobile objects and processing complex queries over mobile object positions.

Location monitoring is an important issue for real time querying and management of mobile object positions. Significant research efforts have been dedicated to techniques for efficient processing of spatial continuous queries on moving objects in a centralized location monitoring system. Surprisingly, very few have promoted a distributed approach to real-time location monitoring over a large and growing number of mobile objects.

In this paper we present a distributed approach to real-time location monitoring over a large and growing number of mobile objects. Concretely, we describe the design of MobiEyes, a distributed real-time location monitoring system for processing *moving queries over moving objects* in a mobile environment. Before we describe the motivation of the MobiEyes system and the main contributions of the paper, we first give a brief overview of the concept of moving queries.

1.1 Moving Queries over Moving Objects

A moving query over moving objects (MQ for short) is a *spatial continuous moving query over locations of moving objects*. An MQ defines a spatial region bound to a specific moving object and a filter which is a boolean predicate on object properties. The result of an MQ consists of objects that are inside the area covered by the query's spatial region and satisfy the query filter.

MQs are continuous queries [11] in the sense that the results of queries continuously change as time progresses. We refer to the object to which an MQ is bounded, the *focal object* of that query. The set of objects that are subject to be included in a query's result are called *target objects* of the MQ. Note that the spatial region of an MQ also moves as the focal object of the MQ moves. There are many examples of moving queries over moving objects in real life. For instance, the query MQ_1: "Give me the number of friendly units within 5 miles radius around me during next 2 hours" can be submitted by a soldier equipped with mobile devices marching in the field, or a moving tank in a military setting. The query MQ_2: "Give me the positions of those customers who are looking for taxi and are within 5 miles (of my location at each instance of time or at an interval of every minute) during the next 20 minutes" can be posted by a taxi driver marching on the road. The focal object of MQ_1 is the solider marching in the field or a moving tank. The focal object of MQ_2 is the taxi driver on the road.

1.2 MobiEyes: Distributed Processing of MQs

Most of the existing approaches for processing spatial queries on moving objects are not scalable, due to their inherent assumption that location monitoring and communications of mobile objects are controlled by a central server. Namely, mobile objects report their position changes to the server whenever their position information changes, and the server determines which moving objects should be included in which moving queries at each instance of time or at a given time interval. For mobile applications that need to handle a large and growing number of moving objects, the centralized approaches can suffer from dramatic performance degradation in terms of server load and network bandwidth.

In this paper we present MobiEyes, a distributed solution for processing MQs in a mobile setup. Our solution ships some part of the query processing down to the moving objects, and the server mainly acts as a mediator between moving objects. This significantly reduces the load on the server side and also results in savings on the communication between moving objects and the server.

This paper has three unique contributions. First, we present a careful design of the distributed solution to real-time evaluation of continuously moving queries over moving objects. One of the main design principles is to develop efficient mechanisms that utilize the computational power at mobile objects, leading to significant savings in terms of server load and messaging cost when compared to solutions relying on central processing of location information at the server. Second, we develop a number of optimization techniques, including *Lazy Query Propagation*, and *Query Grouping*. We use the Query Grouping techniques to constrict the amount of computation to be performed by the moving objects in situations where a moving object is moving in an area that has many queries. We use Lazy Query Propagation to allow trade offs between query precision and network bandwidth cost and energy consumption on the moving objects. Third, but not least, we provide a simulation model in a mobile setup to study the scalability of the MobiEyes distributed location monitoring ap-

proach with regard to server load, messaging cost, and amount of computation required on the mobile objects.

2 System Model

2.1 System Assumptions

We below summarize four underlying assumptions used in the design of the MobiEyes system. All these assumptions are either widely agreed upon by many or have been seen as common practice in most existing mobile systems in the context of monitoring and tracking of moving objects.

− *Moving objects are able to locate their positions*: We assume that each moving object is equipped with a technology like GPS [1] to locate its position. This is a reasonable assumption as GPS devices are becoming inexpensive and are used widely in cars and other hand-held devices to provide navigational support.

− *Moving objects have synchronized clocks*: Again this assumption can be met if the moving objects are equipped with GPS. Another solution is to make NTP [12] (network time protocol) available to moving objects through base stations.

− *Moving objects are able to determine their velocity vector*: This assumption is easily met when the moving object is able to determine its location and has an internal timer.

− *Moving objects have computational capabilities to carry out computational tasks*: This assumption represents a fast growing trend in mobile and wireless technology. The number of mobile devices equiped with computational power escalates rapidly, even simple sensors [6] today are equipped with computational capabilities.

2.2 The Moving Object Model

MobiEyes system assumes that the geographical area of interest is covered by several base stations, which are connected to a central server. A three-tier architecture (mobile objects, base stations and the server) is used in the subsequent discussions. We can easily extend the three tier to a multi-tier communication hierarchy between the mobile objects and the server. In addition, the asymmetric communication is used to establish connections from the server to the moving objects. Concretely, a base station can communicate directly with the moving objects in its coverage area through a broadcast, and the moving objects can only communicate with the base station if they are located in the coverage area of this base station.

Let O be the set of moving objects. Formally we can describe a moving object $o \in O$ by a quadruple: $\langle oid, pos, \overline{vel}, \{props\}\rangle$. oid is the unique object identifier. pos is the current position of the object o. $\overline{vel} = (velx, vely)$ is the current velocity vector of the object, where $velx$ is its velocity in the x-dimension and $vely$ is its velocity in the y-dimension. $\{props\}$ is a set of properties about the moving object o, including spatial, temporal, or object-specific properties, such as color or manufacture of a mobile unit (or even the application specific attributes registered on the mobile unit by the user).

The basic notations used in the subsequent sections of the paper are formally defined below:

− *Rectangle shaped region and circle shaped region* : A rectangle shaped region is defined by $Rect(lx, ly, w, h) = \{(x, y) : x \in [lx, lx + w] \ \wedge \ y \in [ly, ly + h]\}$ and a circle shaped region is defined by $Circle(cx, cy, r) = \{(x, y) : (x - cx)^2 + (y - cx)^2 \leq r^2\}$.

− *Universe of Discourse (UoD)*: We refer to the geographical area of interest as the universe of discourse, which is defined by $U = Rect(X, Y, W, H)$. X, Y, W and H are system level

parameters to be set at the system initialization time.

– *Grid and Grid cells*: In MobiEyes, we map the universe of discourse, $U = Rect(X, Y, W, H)$, onto a grid G of cells, where each grid cell is an $\alpha \times \alpha$ square area, and α is a system parameter that defines the cell size of the grid G. Formally, a grid corresponding to the universe of discourse U can be defined as $G(U, \alpha) = \{A_{i,j} : 1 \leq i \leq M, 1 \leq j \leq N, A_{i,j} = Rect(X+i*\alpha, Y+j*\alpha, \alpha, \alpha), M = \lceil H/\alpha \rceil, N = \lceil W/\alpha \rceil\}$. $A_{i,j}$ is an $\alpha \times \alpha$ square area representing the grid cell that is located on the ith row and jth column of the grid G.

– *Position to Grid Cell Mapping*: Let $pos = (x, y)$ be the position of a moving object in the universe of discourse $U = Rect(X, Y, W, H)$. Let $A_{i,j}$ denote a cell in the grid $G(U, \alpha)$. $Pmap(pos)$ is a position to grid cell mapping, defined as $Pmap(pos) = A_{\lceil \frac{pos.x-X}{\alpha} \rceil, \lceil \frac{pos.y-Y}{\alpha} \rceil}$.

– *Current Grid Cell of an Object*: Current grid cell of a moving object is the grid cell which contains the current position of the moving object. If $o \in O$ is an object whose current position, denoted as $o.pos$, is in the Universe of Discourse U, then the current grid cell of the object is formally defined by $curr\_cell(o) = Pmap(o.pos)$.

– *Base Stations*: Let $U = Rect(X, Y, W, H)$ be the universe of discourse and B be the set of base stations overlapping with U. Assume that each base station $b \in B$ is defined by a circle region $Circle(bsx, bsy, bsr)$. We say that the set B of base stations covers the universe of discourse U, i.e. $\left(\bigcup_{b \in B} b\right) \supseteq U$.

– *Grid Cell to Base Station Mapping*: Let $Bmap : \mathbb{N} \times \mathbb{N} \rightarrow 2^B$ define a mapping, which maps a grid cell index to a non-empty set of base stations. We define $Bmap(i, j) = \{b : b \in B \ \wedge \ b \cap A_{i,j} \neq \emptyset\}$. $Bmap(i, j)$ is the set of base stations that cover the grid cell $A_{i,j}$.

2.3 Moving Query Model

Let Q be the set of moving queries. Formally we can describe a moving query $q \in Q$ by a quadruple: $\langle qid, oid, region, filter \rangle$. qid is the unique query identifier. oid is the object identifier of the focal object of the query. $region$ defines the shape of the spatial query region bound to the focal object of the query. $region$ can be described by a closed shape description such as a rectangle, or a circle, or any other closed shape description which has a computationally cheap point containment check. This closed shape description also specifies a binding point, through which it is bound to the focal object of the query. Without loss of generality we use a circle, with its center serving as the binding point to represent the shape of the region of a moving query in the rest of the paper. $filter$ is a Boolean predicate defined over the properties $\{props\}$ of the target objects of a moving query q. For presentation convenience, in the rest of the paper we consider the result of an MQ as the set of object identifiers of the moving objects that locate within the area covered by the spatial region of the query and satisfy the filter condition.

A formal definition of basic notations regarding MQs is given below.

– *Bounding Box of a Moving Query*: Let $q \in Q$ be a query with focal object $fo \in O$ and spatial region $region$, let rc denote the current grid cell of fo, i.e. $rc = curr\_cell(fo)$. Let lx and ly denote the x-coordinate and the y-coordinate of the lower left corner point of the current grid cell rc. The *Bounding Box* of a query q is a rectangle shaped region, which covers all possible areas that the spatial region of the query q may move into when the focal object fo of the query travels within its current grid cell. For circle shaped spatial query region with radius r, the bounding box can be formally defined as $bound\_box(q) = Rect(rc.lx - r, rc.ly - r, \alpha + 2r, \alpha + 2r)$.

– *Monitoring Region of a Moving Query*: The grid region defined by the union of all grid

cells that intersect with the bounding box of a query forms the monitoring region of the query. It is formally defined as, $mon\_region(q) = \bigcup_{(i,j)\in S} A_{i,j}$, where $S = \{(i,j) : A_{i,j} \cap bound\_box(q) \neq \emptyset\}$. The monitoring region of a moving query covers all the objects that are subject to be included in the result of the moving query when the focal object stays in its current grid cell.

– *Nearby Queries of an Object*: Given a moving object o, we refer to all MQs whose monitoring regions intersect with the current grid cell of the moving object o the *nearby queries* of the object o, i.e. $nearby\_queries(o) = \{q : mon\_region(q) \cap curr\_cell(o) \neq \emptyset \wedge q \in Q\}$. Every mobile object is either a target object of or is of potential interest to its nearby MQs.

3 Distributed Processing of Moving Queries

In this section we give an overview of our distributed approach to efficient processing of MQs, and then focus on the main building blocks of our solution and the important algorithms used. A comparison of our work with the related research in this area is provided in Section 6.

3.1 Algorithm Overview

In MobiEyes, distributed processing of moving queries consists of server side processing and mobile object side processing. The main idea is to provide mechanisms such that each mobile object can determine by itself whether or not it should be included in the result of a moving query close by, without requiring global knowledge regarding the moving queries and the object positions. A brief review is given below on the main components and key ideas used in MobiEyes for distributed processing of MQs. We will provide detailed technical discussion on each of these ideas in the subsequent sections.

Server Side Processing: The server side processing can be characterized as mediation between moving objects. It performs two main tasks. First, it keeps track of the significant position changes of all focal objects, namely the change in velocity vector and the position change that causes the focal object to move out of its current grid cell. Second, it broadcasts the significant position changes of the focal objects and the addition or deletion of the moving queries to the appropriate subset of moving objects in the system.

Monitoring Region of a MQ: To enable efficient processing at mobile object side, we introduce the monitoring region of a moving query to identify all moving objects that may get included in the query's result when the focal object of the query moves within its current cell. The main idea is to have those moving objects that reside in a moving query's monitoring region to be aware of the query and to be responsible for calculating if they should be included in the query result. Thus, the moving objects that are not in the neighborhood of a moving query do not need to be aware of the existence of the moving query, and the query result can be efficiently maintained by the objects in the query's monitoring region.

Registering MQs at Moving Object Side: In MobiEyes, the task of making sure that the moving objects in a query's monitoring region are aware of the query is accomplished through server broadcasts, which are triggered by either installations of new moving queries or notifications of changes in the monitoring regions of existing moving queries when their focal objects change their current grid cells. Upon receiving a broadcast message, for each MQ in the message, the mobile objects examine their local state and determine whether they should be responsible for processing this moving query. This decision is based on whether the mobile objects themselves are within the monitoring region of the query.

Moving Object Side Processing: Once a moving query is registered at the moving object side, the moving object will be responsible for periodically tracking if it is within the spatial

region of the query, by predicting the position of the focal object of the query. Changes in the containment status of the moving object with respect to moving queries are differentially relayed to the server.

Handling Significant Position Changes: In case the position of the focal object of a moving query changes significantly (it moves out of its current grid cell or changes its velocity vector significantly), it will report to the server, and the server will relay such position change information to the appropriate subset of moving objects through broadcasts.

3.2 Data Structures

In this section we describe the design of the data structures used on the server side and on the moving object side, in order to support distributed processing of MQs.

Server-side Data Structures

The server side stores four types of data structures: the focal object table FOT, the server side moving query table SQT, the reverse query index matrix RQI, and the static grid cell to base station mapping $Bmap$.

Focal Object Table, $FOT = (\underline{oid}, pos, \overline{vel}, tm)$, is used to store information about moving objects that are the focal objects of MQs. The table is indexed on the oid attribute, which is the unique object identifier. tm is the time at which the position, pos, and the velocity vector, $\overline{vel}$, of the focal object with identifier oid were recorded on the moving object side. When the focal object reports to the server its position and velocity change, it also includes this timestamp in the report.

Server-side Moving Query Table, $SQT = (\underline{qid}, oid, region, curr\_cell, mon\_region, filter, \{result\})$, is used to store information about all spatial queries hosted by the system. The table is indexed on the qid attribute, which represents the query identifier. oid is the identifier of the focal object of the query. $region$ is the query's spatial region. $curr\_cell$ is the grid cell in which the focal object of the query locates. mon_region is the monitoring region of the query. $\{result\}$ is the set of object identifiers representing the set of target objects of the query. These objects are located within the query's spatial region and satisfy the query filter.

Reverse Query Index, RQI, is an $M \times N$ matrix whose cells are a set of query identifiers. M and N denote the number of rows and the number of columns of the Grid corresponding to the Universe of Discourse of a MobiEyes system. $RQI(i, j)$ stores the identifiers of the queries whose monitoring regions intersect with the grid cell $A_{i,j}$. $RQI(i, j)$ represents the nearby queries of an object whose current grid cell is $A_{i,j}$, i.e. $\forall o \in O$, $nearby\_queries(o) = RQI(i, j)$, where $curr\_cell(o) = A_{i,j}$.

Moving Object-side Data Structures

Each moving object o stores a local query table LQT and a Boolean variable $hasMQ$.

Local Query Table, $LQT = (qid, pos, \overline{vel}, tm, region, mon\_region, isTarget)$ is used to store information about moving queries whose monitoring regions intersect with the current grid cell in which the moving object o currently locates in. qid is the unique query identifier assigned at the time when the query is installed at the server. pos is the last known position, and $\overline{vel}$ is the last known velocity vector of the focal object of the query. tm is the time at which the position and the velocity vector of the focal object was recorded (by the focal object of the query itself, not by the object on which LQT resides). $isTarget$ is a Boolean variable describing whether the object was found to be inside the query's spatial region at the last evaluation of this query by the moving object o. The Boolean variable

$hasMQ$ provides a flag showing whether the moving object o storing the LQT is a focal object of some query or not.

3.3 Installing Queries

Installation of a moving query into the MobiEyes system consists of two phases. First, the MQ is installed at the server side and the server state is updated to reflect the installation of the query. Second, the query is installed at the set of moving objects that are located inside the monitoring region of the query.

Updating the Server State
When the server receives a moving query, assuming it is in the form $(oid, region, filter)$, it performs the following installation actions. (1) It first checks whether the focal object with identifier oid is already contained in the FOT table. (2) If the focal object of the query already exists, it means that either someone else has installed the same query earlier or there exist multiple queries with different filters but the same focal object. Since the FOT table already contains velocity and position information regarding the focal object of this query, the installation simply creates a new entry for this new MQ and adds this entry to the sever-side query table SQT and then modifies the RQI entry that corresponds to the current grid cell of the focal object to include this new MQ in the reverse query index (detailed in step (4)). At this point the query is installed on the server side. (3) However, if the focal object of the query is not present in the FOT table, then the server-side installation manager needs to contact the focal object of this new query and request the position and velocity information. Then the server can directly insert the entry $(oid, pos, \overline{vel}, tm)$ into FOT, where tm is the timestamp when the object with identifier oid has recorded its pos and $\overline{vel}$ information. (4) The server then assigns a unique identifier qid to the query and calculates the current grid cell ($curr\_cell$) of the focal object and the monitoring region (mon_region) of the query. A new moving query entry $(qid, oid, region, curr\_cell, mon\_region, filter)$ will be created and added into the SQT table. The server also updates the RQI index by adding this query with identifier qid to $RQI(i, j)$ if $A_{i,j} \cap mon\_region(qid) \neq \emptyset$. At this point the query is installed on the server side.

Installing Queries on the Moving Objects
After installing queries on the server side, the server needs to complete the installation by triggering query installation on the moving object side. This job is done by performing two tasks. First, the server sends an installation notification to the focal object with identifier oid, which upon receiving the notification sets its $hasMQ$ variable to true. This makes sure that the moving object knows that it is now a focal object and is supposed to report velocity vector changes to the server. The second task is for the server to forward this query to all objects that reside in the query's monitoring region, so that they can install the query and monitor their position changes to determine if they become the target objects of this query. To perform this task, the server uses the mapping $Bmap$ to determine the minimal set of base stations (i.e., the smallest number of base stations) that covers the monitoring region. Then the query is sent to all objects that are covered by the base stations in this set through broadcast messages. When an object receives the broadcast message, it checks whether its current grid cell is covered by the query's monitoring region. If so, the object installs the query into its local query table LQT when the query's filter is also satisfied by the object. Otherwise the object discards the message.

3.4 Handling Velocity Vector Changes

Once a query is installed in the MobiEyes system, the focal object of the query needs to report to the server any significant change to its location information, including significant velocity changes or changes that move the focal object out of its current grid cell. We describe the mechanisms for handling velocity changes in this section and the mechanisms for handling objects that change their current grid cells in the next section.

A velocity vector change, once identified as significant, will need to be relayed to the objects that reside in the query's monitoring region through the server acting as a mediator. When the focal object of a query reports a velocity vector change, it sends its new velocity vector, its position and the timestamp at which this information was recorded, to the server. The server first updates the FOT table with the information received from the focal object. Then for each query associated with the focal object, the server communicates the newly received information to objects located in the monitoring region of the query by using minimum number of broadcasts (this can be done through the use of the grid cell to base station mapping $Bmap$).

A subtle point is that, the velocity vector of the focal object will almost always change at each time step in a real world setup, although the change might be insignificant. One way to handle this is to convey the new velocity vector information to the objects located in the monitoring region of the query, only if the change in the velocity vector is significant. In MobiEyes, we use a variation of dead reckoning to decide what constitutes a (significant) velocity vector change.

Dead Reckoning in MobiEyes

Concretely, at each time step the focal object of a query samples its current position and calculates the difference between its current position and the position that the other objects believe it to be at (based on the last velocity vector information relayed). In case this difference is larger than a threshold, say Δ, the new velocity vector information is relayed [1].

3.5 Handling Objects that Change their Grid Cells

In a mobile system the fact that a moving object changes its current grid cell has an impact on the set of queries the object is responsible for monitoring. In case the object which has changed its current grid cell is a focal object, the change also has an impact on the set of objects which has to monitor the queries bounded to this focal object. In this section we describe how the MobiEyes system can effectively adapt to such changes and the mechanisms used for handling such changes.

When an object changes its current grid cell, it notifies the server of this change by sending its object identifier, its previous grid cell and its new current grid cell to the server. The object also removes those queries whose monitoring regions no longer cover its new current grid cell from its local query table LQT. Upon receipt of the notification, the server performs two sets of operations depending on whether the object is a focal object of some query or not. If the object is a non-focal object, the only thing that the server needs to do is to find what new queries should be installed on this object and then perform the query installation on this moving object. This step is performed because the new current grid cell that the object has moved into may intersect with the monitoring regions of a different set of queries than its previous set. The server uses the reverse query index RQI together with the previous and the new current grid cell of the object to determine the set of new queries

[1] We do not consider the inaccuracy introduced by the motion modeling.

that has to be installed on this moving object. Then the server sends the set of new queries to the moving object for installation. The focal object table FOT and the server query table SQT are used to create required installation information of the queries to be installed on the object. However, if the object that changes its current grid cell is a focal object of some query, additional set of operations are performed. For each query with this object as its focal object, the server performs the following operations. It updates the query's SQT table entry by resetting the current grid cell and the monitoring region to their new values. It also updates the RQI index to reflect the change. Then the server computes the union of the query's previous monitoring region and its new monitoring region, and sends a broadcast message to all objects that reside in this combined area. This message includes information about the new state of the query. Upon receipt of this message from the server, an object performs the following operations for installing/removing a query. It checks whether its current grid cell is covered by the query's monitoring region. If not, the object removes the query from its LQT table (if the entry already exists), since the object's position is no longer covered by the query's monitoring region. Otherwise, it installs the query if the query is not already installed and the query filter is satisfied, by adding a new query entry in the LQT table. In case that the query is already installed in LQT, it updates the monitoring region of the query's entry in LQT.

Optimization: Lazy Query Propagation

The procedure we presented above uses an eager query propagation approach for handling objects changing their current grid cells. It requires each object (focal or non-focal) to contact the server and transfer information whenever it changes its current grid cell. The only reason for a non-focal object to communicate with the server is to immediately obtain the list of new queries that it needs to install in response to changing its current grid cell. We refer to this scheme as the *Eager Query Propagation (EQP)*.

To reduce the amount of communication between moving objects and the server, in MobiEyes we also provide a lazy query propagation approach. Thus, the need for non-focal objects to contact the server to obtain the list of new MQs can be eliminated. Instead of obtaining the new queries from the server and installing them immediately on the object upon a grid cell change, the moving object can wait until the server broadcasts the next velocity vector changes regarding the focal objects of these queries, to the area in which the object locates. In this case the velocity vector change notifications are expanded to include the spatial region and the filter of the queries, so that the object can install the new queries upon receiving the broadcast message on the velocity vector changes of the focal objects of the moving queries. Using lazy propagation, the moving objects upon changing their current grid cells will be unaware of the new set of queries nearby until the focal objects of these queries change their velocity vectors or move out of their current grid cells. Obviously lazy propagation works well when the gird cell size α is large and the focal objects of queries change their velocity vectors frequently. The lazy query propagation may not prevail over the eager query propagation, when: (1) the focal objects do not have significant change on their velocity vectors, (2) the grid cell size α is too small, and (3) the non-focal moving objects change their current grid cells at a much faster rate than the focal objects. In such situations, non-focal objects may end up missing some moving queries. We evaluate the *Lazy Query Propagation (LQP)* approach and study its performance advantages as well as its impact on the query result accuracy in Section 5.

3.6 Moving Object Query Processing Logic

A moving object periodically processes all queries registered in its LQT table. For each query, it predicts the position of the focal object of the query using the velocity, time, and position information available in the LQT entry of the query. Then it compares its current position and the predicted position of the query's focal object to determine whether itself is covered by the query's spatial region or not. When the result is different from the last result computed in the previous time step, the object notifies the server of this change, which in turn differentially updates the query result.

4 Optimizations

In this section we present two additional optimization techniques aiming at controlling the amount of local processing at the mobile object side and further reducing the communication between the mobile objects and the server.

4.1 Query Grouping

It is widely recognized that a mobile user can pose many different queries and a query can be posed multiple times by different users. Thus, in a mobile system many moving queries may share the same focal object. Effective optimizations can be applied to handle multiple queries bound to the same moving object. These optimizations help decreasing both the computational load on the moving objects and the messaging cost of the MobiEyes approach, in situations where the query distribution over focal objects is skewed.

We define a set of moving queries as *groupable MQs* if they are bounded to the same focal object. In addition to being associated with the same focal object, some groupable queries may have the same monitoring region. We refer to MQs that have the same monitoring region as *MQs with matching monitoring regions*, where we refer to MQs that have different monitoring regions as *MQs with non-matching monitoring regions*. Based on these different patterns, different grouping techniques can be applied to groupable MQs.

Grouping MQs with Matching Monitoring Regions
MQs with matching monitoring regions can be grouped most efficiently to reduce the communication and processing costs of such queries. In MobiEyes, we introduce the concept of *query bitmap*, which is a bitmap containing one bit for each query in a query qroup, each bit can be set to 1 or 0 indicating whether the corresponding query should include the moving object in its result or not. We illustrate this with an example. Consider three MQs: $q_1 = (qid_1, oid_i, r_1, filter_1)$, $q_2 = (qid_2, oid_i, r_2, filter_2)$, and $q_3 = (qid_3, oid_i, r_3, filter_3)$ that share the same monitoring region. Note that these queries share their focal object, which is the object with identifier oid_i. Instead of shipping three separate queries to the mobile objects, the server can combine these queries into a single query as follows: $q_3 = (qid_3, oid_i, (r_1, r_2, r_3), (filter_1, filter_2, filter_3))$. With this grouping at hand, when a moving object is processing a set of groupable MQs with matching monitoring regions, it needs to consider queries with smaller radiuses only if it finds out that its current position is inside the spatial region of a query with a larger radius. When a moving object reports to the server whether it is included in the results of queries that form the grouped query or not, it will attach the query bitmap to the notification. For each query, its query bitmap bit is set to 1 only if the moving object stays inside the spatial region of the query and the filter of the query is satisfied. With the query bitmap, the server is able to infer information about individual query results with respect to the reporting object.

Parameter	Description	Value range	Default value
ts	Time step	30 seconds	
α	Grid cell side length	0.5-16 miles	5 miles
no	Number of objects	1,000-10,000	10,000
nmq	Number of moving queries	100-1,000	1,000
nmo	Number of objects changing velocity vector per time step	100-1,000	1,000
$area$	Area of consideration	100,000 square miles	
$alen$	Base station side length	5-80 miles	10 miles
$qradius$	Query radius	{3, 2, 1, 4, 5} miles	
$qselect$	Query selectivity	0.75	
$mospeed$	Max. object speed	{100, 50, 150, 200, 250} miles/hour	

Table 1: Simulation Parameters

Grouping MQs with Non-Matching Monitoring Regions
A clean way to handle MQs with non-matching monitoring regions is to perform grouping on the moving object side only. We illustrate this with an example. Consider an object o_j that has two groupable MQs with non-matching monitoring regions, q_4 and q_5, installed in its LQT table. Since there is no global server side grouping performed for these queries, o_j can save some processing only by combining these two queries inside its LQT table. By this way it only needs to consider the query with smaller radius only if it finds out that its current position is inside the spatial region of the one with the larger radius.

4.2 Safe Period Optimization

In MobiEyes, each moving object that resides in the monitoring region of a query needs to evaluate the queries registered in its local query table LQT periodically. For each query the candidate object needs to determine if it should be included in the answer of the query. The interval for such periodic evaluation can be set either by the server or by the mobile object itself. A safe-period optimization can be applied to reduce the computation load on the mobile object side, which computes a safe period for each object in the monitoring region of a query, if an upper bound ($maxVel$) exists on the maximum velocities of the moving objects.

The safe periods for queries are calculated by an object o as follows: For each query q in its LQT table, the object o calculates a worst case lower bound on the amount of time that has to pass for it to locate inside the area covered by the query q's spatial region. We call this time, the *safe period*(sp) of the object o with respect to the query q, denoted as $sp(o, q)$. The safe period can be formally defined as follows. Let o_i be the object that has the query q_k with focal object o_j in its LQT table, and let $dist(o_i, o_j)$ denote the distance between these two objects, and let $q_k.region$ denote the circle shaped region with radius r. In the worst case, the two objects approach to each other with their maximum velocities in the direction of the shortest path between them. Then $sp(o_i, q_k) = \frac{dist(o_i,o_j)-r}{o_i.maxVel+o_j.maxVel}$.

Once the safe period sp of a moving object is calculated for a query, it is safe for the object to start the periodic evaluation of this query after the safe period has passed. In order to integrate this optimization with the base algorithm, we include a *processing time* (ptm) field into the LQT table, which is initialized to 0. When a query in LQT is to be processed, ptm is checked first. In case ptm is ahead of the current time ctm, the query is skipped. Otherwise, it is processed as usual. After processing of the query, if the object is found to be outside the area covered by the query's spatial region, the safe period sp is calculated for the query and processing time ptm of the query is set to current time plus the safe period, ctm+sp. When the query evaluation period is short, or the object speeds are low or the cell size α of the grid is large, this optimization can be very effective.

5 Experiments

In this section we describe three sets of simulation based experiments.The first set of experiments illustrates the scalability of the MobiEyes approach with respect to server load. The second set of experiments focuses on the messaging cost and studies the effects of several parameters on the messaging cost. The third set of experiments investigates the amount of computation a moving object has to perform, by measuring on average the number of queries a moving object needs to process during each local evaluation period.

5.1 Simulation Setup

We list the set of parameters used in the simulation in Table 1. In all of the experiments, the parameters take their default values if not specified otherwise. The area of interest is a square shaped region of 100,000 square miles. The number of objects we consider ranges from 1,000 to 10,000 where the number of queries range from 100 to 1,000.

We randomly select focal objects of the queries using a uniform distribution. The spatial region of a query is taken as a circular region whose radius is a random variable following a normal distribution. For a given query, the mean of the query radius is selected from the list $\{3, 2, 1, 4, 5\}$(miles) following a zipf distribution with parameter 0.8 and the std. deviation of the query radius is taken as 1/5th of its mean. The selectivity of the queries is taken as 0.75.

We model the movement of the objects as follows. We assign a maximum velocity to each object from the list $\{100, 50, 150, 200, 250\}$(miles/hour), using a zipf distribution with parameter 0.8. The simulation has a time step parameter of 30 seconds. In every time step we pick a number of objects at random and set their normalized velocity vectors to a random direction, while setting their velocity to a random value between zero and their maximum velocity. All other objects are assumed to continue their motion with their unchanged velocity vectors. The number of objects that change velocity vectors during each time step is a parameter whose value ranges from 100 to 1,000.

5.2 Server Load

In this section we compare our MobiEyes distributed query processing approach with two popular central query processing approaches, with regard to server load. The two centralized approaches we consider are indexing objects and indexing queries. Both are based on a central server on which the object locations are explicitly manipulated by the server logic as they arrive, for the purpose of answering queries. We can either assume that the objects are reporting their positions periodically or we can assume that periodically object locations are extracted from velocity vector and time information associated with moving objects, on the server side. We first describe these two approaches and later compare them with the distributed MobiEyes distributed approach with regard to server load.

Indexing Objects. The first centralized approach to processing spatial continuous queries on moving objects is by indexing objects. In this approach a spatial index is built over object locations. We use an R*-tree [3] for this purpose. As new object positions are received, the spatial index (the R*-tree) on object locations is updated with the new information. Periodically all queries are evaluated against the object index and the new results of the queries are determined. This is a straightforward approach and it is costly due to the frequent updates required on the spatial index over object locations.

Indexing Queries. The second centralized approach to processing spatial continuous queries on moving objects is by indexing queries. In this approach a spatial index, again an

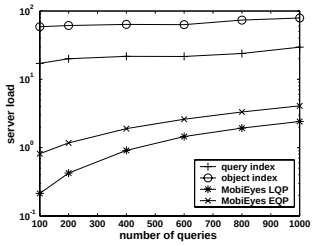

Fig. 1: Impact of distributed query processing on server load

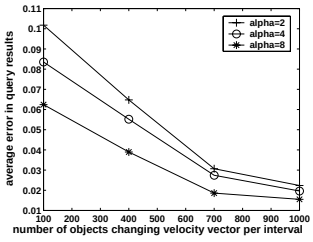

Fig. 2: Error associated with lazy query propagation

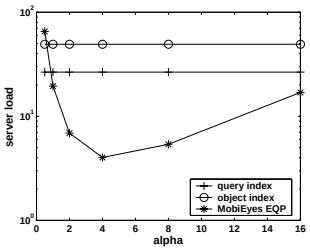

Fig. 3: Effect of α on server load

R*-tree indeed, is built over moving queries. As the new positions of the focal objects of the queries are received, the spatial index is updated. This approach has the advantage of being able to perform differential evaluation of query results. When a new object position is received, it is run through the query index to determine to which queries this object actually contributes. Then the object is added to the results of these queries, and is removed from the results of other queries that have included it as a target object before. We have implemented both the *object index* and the *query index* approaches for centralized processing of MQs. As a measure of server load, we took the time spent by the simulation for executing the server side logic per time step. Figure 1 and Figure 3 depict the results obtained. Note that the y-axises, which represent the sever load, are in log-scale. The x-axis represents the number of queries considered in Figure 1, and the different settings of α parameter in Figure 3.

It is observed from Figure 1 that the MobiEyes approach provides up to two orders of magnitude improvement on server load. In contrast, the object index approach has an almost constant cost, which slightly increases with the number of queries. This is due to the fact that the main cost of this approach is to update the spatial index when object positions change. Although the query index approach clearly outperforms the object index approach for small number of queries, its performance worsens as the number of queries increase. This is due to the fact that the main cost of this approach is to update the spatial index when focal objects of the queries change their positions. Our distributed approach also shows an increase in server load as the number of queries increase, but it preserves the relative gain against the query index.

Figure 1 also shows the improvement in server load using lazy query propagation (LQP) compared to the default eager query propagation (EQP). However as described in Section 3.5, lazy query propagation may have some inaccuracy associated with it. Figure 2 studies this inaccuracy and the parameters that influence it. For a given query, we define the *error* in the query result at a given time, as the number of missing object identifiers in the result (compared to the correct result) divided by the size of the correct query result. Figure 2 plots the average error in the query results when lazy query propagation is used as a function of number of objects changing velocity vectors per time step for different values of α. Frequent velocity vector changes are expected to increase the accuracy of the query results. This is observed from Figure 2 as it shows that the error in query results decreases with increasing number of objects changing velocity vectors per time step. Frequent grid cell crossings are expected to decrease the accuracy of the query results. This is observed from Figure 2 as it shows that the error in query results increases with decreasing α.

Figure 3 shows that the performance of the MobiEyes approach in terms of server load worsens for too small and too large values of the α parameter. However it still outperforms the

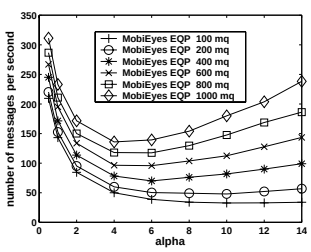

Fig. 4: Effect of α on messaging cost

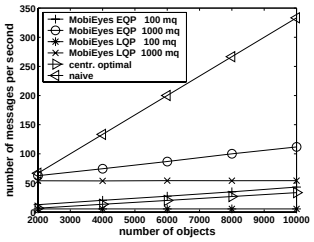

Fig. 5: Effect of # of objects on messaging cost

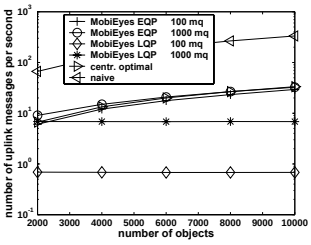

Fig. 6: Effect of # of objs. on uplink messaging cost

object index and query index approaches. For small values of α, the frequent grid cell changes increase the server load. On the other hand, for large values of α, the large monitoring areas increase the server's job of mediating between focal objects and the objects that are lying in the monitoring regions of the focal objects' queries. Several factors may affect the selection of an appropriate α value. We further investigate selecting a good value for α in the next section.

5.3 Messaging Cost

In this section we discuss the effects of several parameters on the messaging cost of our solution. In most of the experiments presented in this section, we report the total number of messages sent on the wireless medium per second. The number of messages reported includes two types of messages. The first type of messages are the ones that are sent from a moving object to the server (uplink messages), and the second type of messages are the ones broadcasted by a base station to a certain area or sent to a moving object as a one-to-one message from the server (downlink messages). We evaluate and compare our results using two different scenarios. In the first scenario each object reports its position directly to the server at each time step, if its position has changed. We name this as the *naïve* approach. In the second scenario each object reports its velocity vector at each time step, if the velocity vector has changed (significantly) since the last time. We name this as the *central optimal* approach. As the name suggests, this is the minimum amount of information required for a centralized approach to evaluate queries unless there is an assumption about object trajectories. Both of the scenarios assume a central processing scheme.

One crucial concern is defining an optimal value for the parameter α, which is the length of a grid cell. The graph in Figure 4 plots the number of messages per second as a function of α for different number of queries. As seen from the figure, both too small and too large values of α have a negative effect on the messaging cost. For smaller values of α this is because objects change their current grid cell quite frequently. For larger values of α this is mainly because the monitoring regions of the queries become larger. As a result, more broadcasts are needed to notify objects in a larger area, of the changes related to focal objects of the queries they are subject to be considered against. Figure 4 shows that values in the range [4,6] are ideal for α with respect to the number of queries ranging from 100 to 1000. The optimal value of the α parameter can be derived analytically using a simple model. In this paper we omit the analytical model for space restrictions.

Figure 5 studies the effect of number of objects on the messaging cost. It plots the number of messages per second as a function of number of objects for different numbers of queries. While the number of objects is altered, the ratio of the number of objects changing their velocity vectors per time step to the total number of objects is kept constant and equal to its

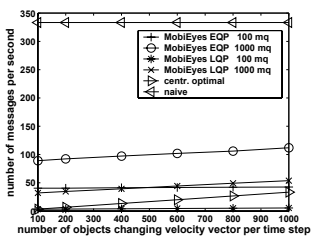

Fig. 7: Effect of number of objects changing velocity vector per time step on messaging cost

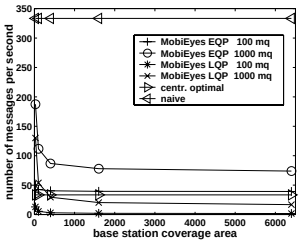

Fig. 8: Effect of base station coverage area on messaging cost

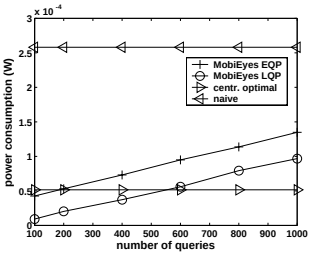

Fig. 9: Effect of # of queries on per object power consumption due to communication

default value as obtained from Table 1. It is observed that, when the number of queries is large and the number of objects is small, all approaches come close to one another. However, the naïve approach has a high cost when the ratio of the number of objects to the number of queries is high. In the latter case, central optimal approach provides lower messaging cost, when compared to MobiEyes with EQP, but the gap between the two stays constant as number of objects are increased. On the other hand, MobiEyes with LQP scales better than all other approaches with increasing number of objects and shows improvement over central optimal approach for smaller number of queries. Figure 6 shows the uplink component of the messaging cost. The y-axis is plotted in logarithmic scale for convenience of the comparison. Figure 6 clearly shows that MobiEyes with LQP significantly cuts down the uplink messaging requirement, which is crucial for asymmetric communication environments where uplink communication bandwidth is considerably lower than downlink communication bandwidth.

Figure 7 studies the effect of number of objects changing velocity vector per time step on the messaging cost. It plots the number of messages per second as a function of the number of objects changing velocity vector per time step for different numbers of queries. An important observation from Figure 7 is that the messaging cost of MobiEyes with EQP scales well when compared to the central optimal approach as the gap between the two tends to decrease as the number of objects changing velocity vector per time step increases. Again MobiEyes with LQP scales better than all other approaches and shows improvement over central optimal approach for smaller number of queries.

Figure 8 studies the effect of base station coverage area on the messaging cost. It plots the number of messages per second as a function of the base station coverage area for different numbers of queries. It is observed from Figure 8 that increasing the base station coverage decreases the messaging cost up to some point after which the effect disappears. The reason for this is that, after the coverage areas of the base stations reach to a certain size, the monitoring regions associated with queries always lie in only one base station's coverage area. Although increasing base station size decreases the total number of messages sent on the wireless medium, it will increase the average number of messages received by a moving object due to the size difference between monitoring regions and base station coverage areas. In a hypothetical case where the universe of disclosure is covered by a single base station, any server broadcast will be received by any moving object. In such environments, indexing on the air [7] can be used as an effective mechanism to deal with this problem. In this paper we do not consider such extreme scenarios.

Per Object Power Consumption due to Communication

So far we have considered the scalability of the MobiEyes in terms of the total number of

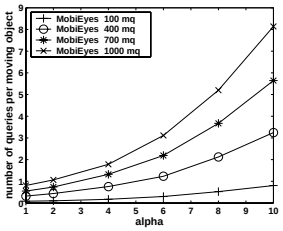

Fig. 10: Effect of α on the average number of queries evaluated per step on a moving object

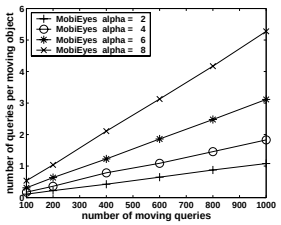

Fig. 11: Effect of the total # of queries on the avg. # of queries evaluated per step on a moving object

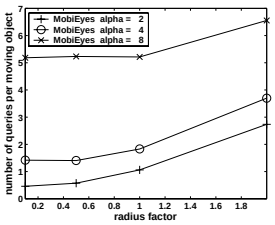

Fig. 12: Effect of the query radius on the average number of queries evaluated per step on a moving object

messages exchanged in the system. However one crucial measure is the per object power consumption due to communication. We measure the average communication related to power consumption using a simple radio model where the transmission path consists of transmitter electronics and transmit amplifier where the receiver path consists of receiver electronics. Considering a GSM/GPRS device, we take the power consumption of transmitter and receiver electronics as 150mW and 120mW respectively and we assume a 300mW transmit amplifier with 30% efficiency [8]. We consider 14kbps uplink and 28kbps downlink bandwidth (typical for current GPRS technology). Note that sending data is more power consuming than receiving data. [2]

We simulated the MobiEyes approach using message sizes instead of message counts for messages exchanged and compared its power consumption due to communication with the naive and central optimal approaches. The graph in Figure 9 plots the per object power consumption due to communication as a function of number of queries. Since the naive approach require every object to send its new position to the server, its per object power consumption is the worst. In MobiEyes, however, a non-focal object does not send its position or velocity vector to the server, but it receives query updates from the server. Although the cost of receiving data in terms of consumed energy is lower than transmitting, given a fixed number of objects, for larger number of queries the central optimal approach outperforms MobiEyes in terms of power consumption due to communication. An important factor that increases the per object power consumption in MobiEyes is the fact that an object also receives updates regarding queries that are irrelevant mainly due to the difference between the size of a broadcast area and the monitoring region of a query.

5.4 Computation on the Moving Object Side

In this section we study the amount of computation placed on the moving object side by the MobiEyes approach for processing MQs. One measure of this is the number of queries a moving object has to evaluate at each time step, which is the size of the LQT (Recall Section 3.2).

Figure 10 and Figure 11 study the effect of α and the effect of the total number of queries on the average number of queries a moving object has to evaluate at each time step (average LQT table size). The graph in Figure 10 plots the average LQT table size as a function of α for different number of queries. The graph in Figure 11 plots the same measure, but this time as a function of number of queries for different values of α. The first observation from these two figures is that the size of the LQT table does not exceeds 10 for the simulation setup.

[2] In this setting transmitting costs $\sim 80\mu jules/bit$ and receiving costs $\sim 5\mu jules/bit$

The second observation is that the average size of the LQT table increases exponentially with α where it increases linearly with the number of queries.

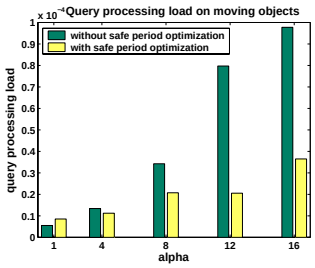

Fig. 13: Effect of the safe period optimization on the average query processing load of a moving object

Figure 12 studies the effect of the query radius on the number of average queries a moving object has to evaluate at each time step. The x-axis of the graph in Figure 12 represents the radius factor, whose value is used to multiply the original radius value of the queries. The y-axis represents the average LQT table size. It is observed from the figure that the larger query radius values increase the LQT table size. However this effect is only visible for radius values whose difference from each other is larger than the α. This is a direct result of the definition of the monitoring region from Section 2.

Figure 13 studies the effect of the safe period optimization on the average query processing load of a moving object. The x-axis of the graph in Figure 12 represents the α parameter, and the y-axis represents the average query processing load of a moving object. As a measure of query processing load, we took the average time spent by a moving object for processing its LQT table in the simulation. Figure 12 shows that for large values of α, the safe period optimization is very effective. This is because, as α gets larger, monitoring regions get larger, which increases the average distance between the focal object of a query and the objects in its monitoring region. This results in non-zero safe periods and decreases the cost of processing the LQT table. On the other hand, for very small values of α, like $\alpha = 1$ in Figure 13, the safe period optimization incurs a small overhead. This is because the safe period is almost always less than the query evaluation period for very small α values and as a result the extra processing done for safe period calculations does not pay off.

6 Related Work

Evaluation of static spatial queries on moving objects, at a centralized location, is a well studied topic. In [14], Velocity Constrained Indexing and Query Indexing are proposed for efficient evaluation of this kind of queries at a central location. Several other indexing structures and algorithms for handling moving object positions are suggested in the literature [17, 15, 9, 2, 4, 18]. There are two main points where our work departs from this line of work.

First, most of the work done in this respect has focused on efficient indexing structures and has ignored the underlying mobile communication system and the mobile objects. To our knowledge, only the SQM system introduced in [5] has proposed a distributed solution for evaluation of static spatial queries on moving objects, that makes use of the computational capabilities present at the mobile objects.

Second, the concept of dynamic queries presented in [10] are to some extent similar to the concept of moving queries in MobiEyes. But there are two subtle differences. First, a dynamic query is defined as a temporally ordered set of snapshot queries in [10]. This is a low level definition. In contrast, our definition of moving queries is at end-user level, which includes the notion of a focal object. Second, the work done in [10] indexes the trajectories of the moving objects and describes how to efficiently evaluate dynamic queries that represent predictable or non-predictable movement of an observer. They also describe how new trajectories can be added when a dynamic query is actively running. Their assumptions are in line with their motivating scenario, which is to support rendering of objects in virtual tour-like

applications. The MobiEyes solution discussed in this paper focuses on real-time evaluation of moving queries in real-world settings, where the trajectories of the moving objects are unpredictable and the queries are associated with moving objects inside the system.

7 Conclusion

We have described MobiEyes, a distributed scheme for processing moving queries on moving objects in a mobile setup. We demonstrated the effectiveness of our approach through a set of simulation based experiments. We showed that the distributed processing of MQs significantly decreases the server load and scales well in terms of messaging cost while placing only small amount of processing burden on moving objects.

References

[1] US Naval Observatory (USNO) GPS Operations. http://tycho.usno.navy.mil/gps.html, April 2003.

[2] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *PODS*, 2000.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[4] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *International Database Engineering and Applications Symposium*, 2002.

[5] Y. Cai and K. A. Hua. An adaptive query management technique for efficient real-time monitoring of spatial regions in mobile database systems. In *IEEE IPCCC*, 2002.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *ASPLOS*, 2000.

[7] T. Imielinski, S. Viswanathan, and B. Badrinath. Energy efficient indexing on air. In *SIGMOD*, 1994.

[8] J.Kucera and U. Lott. Single chip 1.9 ghz transceiver frontend mmic including Rx/Tx local oscillators and 300 mw power amplifier. *MTT Symposium Digest*, 4:1405–1408, June 1999.

[9] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *PODS*, 1999.

[10] I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic queries over mobile objects. In *EDBT*, 2002.

[11] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, pages 610–628, 1999.

[12] D. L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications*, pages 1482–1493, 1991.

[13] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, 2000.

[14] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[15] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, 2000.

[16] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *ICDE*, 1997.

[17] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD*, 2002.

[18] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, 2002.