MobiEyes: A Distributed Location Monitoring Service Using Moving Location Queries

Buğra Gedik, *Member*, *IEEE*, and Ling Liu, *Senior Member*, *IEEE*

Abstract—With the growing popularity and availability of mobile communications, our ability to stay connected while on the move is becoming a reality instead of science fiction as it was just a decade ago. An important research challenge for modern location-based services is the scalable processing of location monitoring requests on a large collection of mobile objects. The centralized architecture, though studied extensively in literature, would create intolerable performance problems as the number of mobile objects grows significantly. This paper presents a distributed architecture and a suite of optimization techniques for scalable processing of continuously moving location queries. Moving location queries can be viewed as standing location tracking requests that continuously monitor the locations of mobile objects of interest and return a subset of mobile objects when certain conditions are met. We describe the design of MobiEyes, a distributed real time location monitoring system in a mobile environment. The main idea behind the MobiEyes' distributed architecture is to promote a careful partition of a real time location monitoring task into an optimal coordination of server-side processing and client-side processing. Such a partition allows evaluating moving location queries with a high degree of precision using a small number of location updates, thus providing highly scalable location monitoring services. A set of optimization techniques are used to limit the amount of computation to be handled by the mobile objects and enhance the overall performance and system utilization of MobiEyes. Important metrics to validate the proposed architecture and optimizations include messaging cost, server load, and amount of computation at individual mobile objects. We evaluate the scalability of the MobiEyes location monitoring approach using a simulation model based on a mobile setup. Our experimental results show that MobiEyes can lead to significant savings in terms of server load and messaging cost when compared to solutions relying on central processing of location information.

Index Terms—Spatial queries over mobile objects, distributed algorithms, mobile data management.

1 Introduction

With the growing availability of mobile communications and the rapid drop in prices for basic mobility enabling equipments like GPS devices [1], smart cell phones, and handhelds, we are entering a world where people, computers, vehicles, and other mobile objects are interconnected, and traditional wired networks are being replaced by their wireless counterparts, which facilitates our ability to stay connected while on the move.

There are two representative types of emerging location-based services: location-aware content delivery and location-sensitive resource management. The former uses location data to tailor the information delivered to the mobile users in order to increase the quality of service and the degree of personalization. Examples include delivering accurate driving directions, instant coupons to customers nearby or approaching a store, or answering location-based queries for nearest resource information like local restaurants, hospitals, gas stations, or police cars within 5 miles upon a car accident. The latter uses location data combined with route schedules and resource management plans to direct service personnel or transportation systems, optimize personnel utilization, handle emergency requests, and reschedule in response to external conditions like traffic and weather. Examples include systems for fleet management, mobile workforce management, and transportation management. Scalable location query processing is an enabling technology for all these applications [34], [33].

An important research challenge for location information management and future mobile computing applications is a scalable architecture that is capable of handling large and rapidly growing number of mobile objects and processing complex queries over mobile object positions. Significant research efforts have been dedicated to techniques for efficient processing of spatial and temporal continuous queries on mobile objects in a centralized location monitoring system [30], [19], [25], [11], [27], [24], [29], [18], [2]. However, as several researchers have pointed out [8], [14], [33], the centralized location monitoring architecture can create intolerable performance problems as the number of mobile objects grows rapidly. We observe two inherent assumptions that limit the scalability of many existing centralized approaches. First, most of the existing location management systems assume that mobile objects are only responsible for reporting their location information periodically, and the server (or a hierarchy of servers) is fully responsible for detecting interesting location changes, determining which mobile objects should be included in which moving queries at each instance of time or at a given time interval, and return those location updates that match certain thresholds to the users. For mobile applications that need to handle a large number of mobile objects, these centralized approaches can suffer from dramatic performance degradation in terms of server load and network bandwidth. The second drawback of the existing centralized architectures is the assumption that all mobile objects in a given universe of discourse and their location updates are relevant for answering moving queries posted to the system.

- *B. Gedik is with the IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532. E-mail: bgedik@us.ibm.com.*
- *L. Liu is with the College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30339. E-mail: lingliu@cc.gatech.edu.*

Manuscript received 8 Dec. 2004; revised 31 May 2005; accepted 18 Sept. 2005; published online 16 Aug. 2006.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-0329-1204.

In reality, only a small subset of mobile objects is relevant to a given moving location query at any given instance of time. Thus, the amount of processing on the location updates of those mobile objects that do not contribute to the answers of the moving location queries is unnecessary and wasted, and it may cause significant performance degradation when the number of mobile objects is large and growing dynamically.

Keeping these problems in mind, we design and develop MobiEyes, a distributed real-time location monitoring service for moving location queries over a large and growing number of mobile objects. A moving location query can be viewed as a standing location tracking request that continuously monitors the locations of mobile objects of interest and return a subset of mobile objects that satisfy certain conditions. In this paper, we focus on the distributed architecture and a suite of optimization techniques for scalable processing of moving location queries on mobile objects.

A promising approach to tackle the scalability problem of centralized location monitoring architectures is to ship certain amount of the moving query processing down to a set of "nearby" mobile objects and to have the server mainly act as a mediator between these mobile objects. *The distribution of some moving query processing effort from the server to a selective set of mobile objects can be seen as a mechanism for moving computation close to the places where the location data of interest is produced*. Such techniques are especially beneficial when there are a large number of continuously mobile objects, generating immense number of position updates, but only a small subset of the location updates are of interest to each location query. By utilizing the computational capabilities available at the mobile objects, we can reduce the load on the server, filter irrelevant location updates, and increase the overall utilization and the scalability of the system. This computation partitioning approach taken by MobiEyes for evaluating moving queries is further motivated by the rapid and continued upsurge of computational capabilities in mobile devices, ranging from GPS-based navigational systems in cars to hand-held devices and smart cell phones.

In order to control the amount of computations to be handled by the mobile objects that are nearby the spatial regions of active location queries, and to enhance the overall performance and system utilization of MobiEyes, we develop a set of optimization techniques, such as, *Moving Query Grouping*, *Lazy Query Propagation*, and *Safe Query Periods*. We use Query Grouping to constrict the amount of computation to be performed by the mobile objects and to minimize the number of messages sent on the wireless medium in situations where there are large groups of moving queries with identical focal objects. We use Lazy Query Propagation to allow trade-offs between query precision and network bandwidth cost as well as energy consumption on the mobile objects. We use Safe Periods to decrease the query-processing load on mobile objects due to periodic reevaluation of moving queries. Important metrics to validate the proposed architecture and optimizations include messaging cost, server load, and amount of computation at individual mobile objects. We present an analytical model for estimating the messaging cost of our solution, which guides us to find the optimal settings of certain system-wide parameters. We evaluate the scalability of the MobiEyes distributed location monitoring approach using a simulation model based on a mobile setup. The experimental results show that the MobiEyes approach can lead to significant savings in terms of server load and messaging cost, when compared to solutions relying on fully centralized processing of location information at the server(s).

2 System Model

The MobiEyes system model includes the set of underlying assumptions used in the design of MobiEyes, the mobile object model, and the moving query model. Before we get into a formal description of the system model, we first informally describe the concept of moving queries on mobile objects.

A moving location query on mobile objects (MQ for short) is a *spatial continuous moving query over locations of mobile objects*, and we also call a moving location query a moving query (MQ) for reference convenience. An MQ defines a spatial region bound to a specific mobile 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 [20] 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 on mobile 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 an interval of every minute) during the next 20 minutes" can be posted by a taxi driver moving 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. Different specializations of MQs can result in interesting and useful classes of queries on mobile objects. One such specialization is the case where the target objects are static objects in the query region, which leads to *moving queries on static objects*. An example of such a query is MQ_3: "Give me the locations and names of the gas stations offering gasoline for less than $1.2 per gallon within 10 miles, during next half an hour" posted by a driver of a moving car, where the focal object of the query is the car on the move and the target objects are buildings within 10 miles with respect to the location of the car on the move. Another interesting specialization is the case where the queries are posed with static focal objects or without focal objects. In this case, an MQ becomes a *static spatial continuous query on mobile objects*. An example query is MQ_4: "Give me the list of AAA vehicles that are currently on service call in downtown Atlanta (or 5 miles from my office location), during the next hour."

2.1 System Assumptions

The design of the MobiEyes system is based on the following assumptions. All these assumptions are 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 mobile objects: 1) *Mobile objects are able to locate their positions and are able to determine their velocity vector*, e.g.,

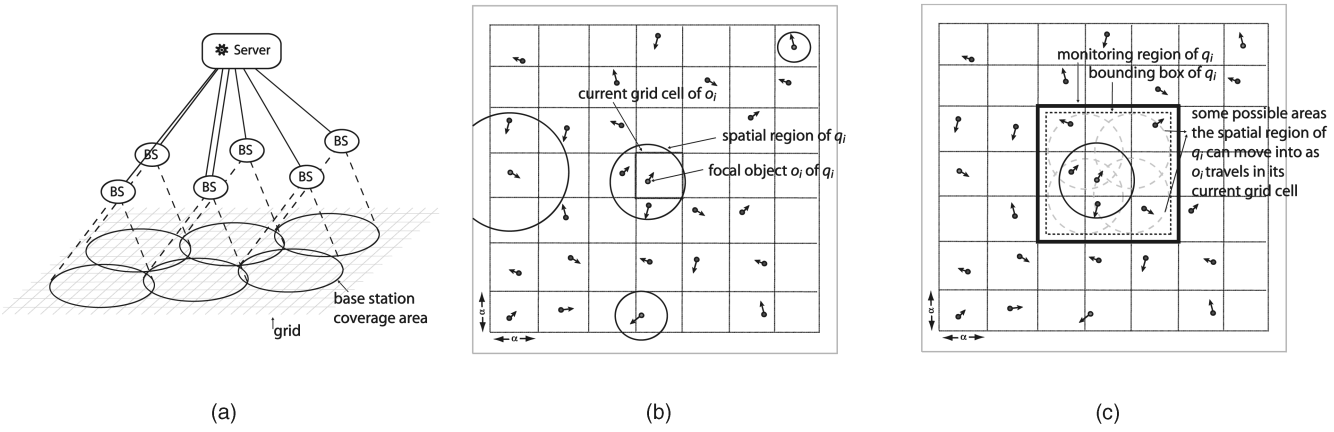

Fig. 1. Illustration of concepts. (a) Base station related concepts. (b) Mobile object concepts. (c) MQ related concepts.

using GPS [1]. 2) *Mobile objects have synchronized clocks*, e.g., using GPS or NTP [21]. 3) *Mobile objects have computational capabilities to carry out computational tasks*.

In addition, we assume that the geographical area of interest is covered by several base stations, which are connected to the service provider's server farm (simply called the server in the rest of the paper). We assume that all location service requests are served through a three-tier architecture, that consists of mobile objects, base stations, and the server. Broadcast is used to establish connections from the server to the mobile objects through the base stations. The mobile objects can only communicate with the base station if they are located in the coverage area of the base station. Base stations can communicate with the server through wired networking. Fig. 1a shows this architecture.

2.2 The Mobile Object Model

Let O be the set of mobile objects. Formally, we can describe a mobile 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 $vely$ in the y-dimension. $\{props\}$ is a set of properties about the mobile object o, including spatial, temporal, and object-specific properties (even 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

$$Rect(lx, ly, w, h) = \{(x, y) : x \in [lx, lx + w] \ \wedge \ y \in [ly, ly + h]\},$$

where lx and ly are the x-coordinate and the y-coordinate of the lower left corner of the rectangle, w is the width and h is the height of the rectangle. A circle shaped region is defined by $Circle(cx, cy, r) = \{(x, y) : (x - cx)^2 + (y - cx)^2 \leq r^2\}$, where cx is the x-coordinate and cy is the y-coordinate of the circle's center, and r is the radius of the circle.

Universe of Discourse (UoD): We refer to the geographical area of interest as the universe of discourse, which is defined by $U = Rect(X, Y, W, H)$, where X is the x-coordinate and Y is the y-coordinate of the lower left corner of the rectangle shaped region corresponding to the universe of discourse. W is the width and H is the height of the universe of discourse.

Grid and Grid cells: In MobiEyes, we map the universe of discourse $U = Rect(X, Y, W, H)$ onto a grid G of cells. 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

$$\begin{aligned} G(U, \alpha) = \{A_{i,j} : 1 \leq i \leq M, 1 \leq j \leq N,\\ A_{i,j} = Rect(X + i*\alpha, Y + j*\alpha, \alpha, \alpha),\\ M = \lceil H/\alpha \rceil, N = \lceil W/\alpha \rceil\}. \end{aligned}$$

$A_{i,j}$ 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 mobile 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 mobile object is the grid cell which contains the current position of the mobile 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 $Circle(bsx, bsy, bsr)$ with (bsx, bsy) being the center of the circle and bsr being the radius of the circle. We say that the set B of base stations covers the universe of discourse U, i.e., $\bigcup_{b \in B} b \supseteq U$.

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 nonempty 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 $A_{i,j}$.

See Fig. 1a and Fig. 1b for example illustrations.

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. Example properties include characteristics of the target objects, or specific spatial regions. A moving query "give me the list of AAA vehicles on highway 85 north and within 20 miles of my current location" can be posed by a driver of a car on the road. This query has used the filter "AAA vehicles on highway 85 north" to define the target objects of interest. For presentation convenience, in the rest of the paper we consider the result of an MQ as the set of object identifiers of the mobile objects that locate within the area covered by the spatial region of the query and satisfy the filter condition.

Formal definition of basic notations regarding MQs is given below (see Fig. 1c for illustrations): *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 a 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 mobile object o, we refer to all MQs whose monitoring regions intersect with the current grid cell of the mobile 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.

MobiEyes handles the dynamics of mobile object side query installation at the granularity of grid cells, which entails that the bounding boxes should be extended to cover an integral number of grid cells. Such extended bounding boxes are called monitoring regions. This is the main intuition behind defining monitoring regions.

We make two important observations. First, at any given time we have a set O of mobile objects and a set Q of moving queries in the MobiEyes system. When the size of the set O is relatively large compared to the size of Q, it is more likely that there are mobile objects in O, which do not have any nearby moving queries in Q. The larger the size of set O, the higher the number of mobile objects whose current grid cells do not intersect with the monitoring region of any MQs currently installed in the system. Second, each moving query in MobiEyes is associated with a stop condition specifying when the moving query will be terminated. This warrants that every moving query will be terminated at some point in time. We can view a moving query MQ as a continual query [20] of the form $\langle query, trigger, stop\rangle$. The $query$ component is defined in terms of the *focal object*, the query *region*, and the *filter*. The $trigger$ condition is the location update reporting interval of the focal object, and the $stop$ condition is the termination time of the MQ. We say an MQ is active if its stop condition is not yet met. In the rest of the paper, we model an MQ in terms of its $query$ component.

3 Distributed Architecture

Motivated by the fact that mobile users are typically interested in other mobile objects nearby regardless of the total network size, the main idea underlying the MobiEyes' distributed architecture is to have each mobile object determine by itself whether or not it should be included in the result of a location query nearby, without requiring global knowledge regarding the moving location queries and the object positions of interest. An immediate advantage of the MobiEyes approach is the significant saving in terms of server load and communication bandwidth. Concretely, in MobiEyes, mobile objects that are not focal objects of any moving location queries do not need to report their position or velocity changes to the location server if they do not have any nearby queries. The focal objects only report to the location server when their velocity vectors change above certain threshold or when they move out of their current grid cells.

A main challenge in the design of the MobiEyes distributed architecture is two fold. First, we need to develop algorithms to carefully partition the processing cost of MQs into the mediation processing at the server side and the local processing at the mobile object side. Second, we need methods to identify the most appropriate subset of mobile objects that can contribute to the answer of a given MQ in a given period of time.

We first introduce *the concept of monitoring region* for each moving query to model the part of the grid area to which the spatial query region is confined when the focal object of the query changes its position within its current grid cell. Second, we design the *concrete data structures* and the *distributed coordination mechanisms* that are location query aware for managing the communication and collaboration between the server and the chosen subset of mobile objects, focusing on reducing the server load as well as the messaging cost and, thus, the network bandwidth and power consumption at mobile objects. This architecture design is especially effective when the number of mobile objects in the geographical region considered is large and the number of MQs is relatively small and skewed in terms of query distribution over the mobile objects considered. We also develop a set of *optimizations* for efficient processing of MQs. We employ the lazy query propagation technique to reduce the messaging cost of the communication between the mobile objects and the server. We introduce the MQ grouping techniques to allow MobiEyes

to handle hot spot queries efficiently. We utilize the concept of safe period to allow further reduction on the amount of local query processing at the mobile object side. We use the dead-reckoning technique, popular in many centralized proposals for processing spatial queries on mobile objects, to predict the position changes of mobile objects of interest.

In the subsequent sections, we first describe the main data structures used in MobiEyes and then provide a detailed discussion on server side processing and mobile object side processing. We defer the discussion on the set of optimization techniques to Section 4.

3.1 Data Structures

3.1.1 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 mobile 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 mobile 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 $curr\_cell(o) = A_{i,j}$. Formally, it is defined as follows:

$$RQI(i, j) = \{qid : \exists e \in SQT \text{ s.t. } e.qid = qid \ \wedge \ e.mon\_region \cap A_{i,j} \neq \emptyset\}.$$

3.1.2 Mobile Object Side Data Structures

Each mobile 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 mobile 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 mobile object o.

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

3.2 Server Side Processing

The location server side processing consists of two main tasks. First, it handles moving location query installation requests from end-users of the location service application, including mobile users. Second, it performs the mediation of location query processing at the server side and mobile object side, including receiving and responding to 1) the significant changes in velocity vector information of the mobile objects that are focal objects of some location queries and 2) the grid cell change events resulting from movement of focal or nonfocal objects out of their current grid cells. We leave the details of the second task to Section 4.

To enable efficient processing at mobile object side, we introduce the *monitoring region of a moving location query* to identify all mobile 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 mobile 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 mobile 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 in a differential manner.

3.2.1 Installing MQs at the Location Server

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

When the server receives a new MQ 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.

There is a small complication involved in the installation of new MQs (see action 3). If the focal object of the new MQ is not present in the FOT table and the MQ is not posted by the focal object itself, then the server needs to first locate the focal object of the query in order to obtain its current position and velocity vector information. A simple solution is to use an additional table stored on the location server side, which stores base station level [4], [9], [5], [23], or higher level location information regarding mobile objects. For instance, in case we store base station level location information regarding mobile objects, given an object identifier it is possible to locate the base station which covers its current location. During query installation, this will enable us to communicate with the focal object to receive its position and velocity vector information. The granularity of the information maintained about object positions characterizes the tradeoff between the cost of locating an object (in which base station's coverage area it resides in) and the cost of maintaining this information.

After installing queries on the server side, the server needs to complete the installation by triggering query installation on the mobile 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 mobile 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 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. The detailed procedures for location query installation are given in Algorithm 1.

Algorithm 1: New Moving Query Posted to Server
(A) **Server: Received Query** $(oid, region, filter)$
1: Let $e = (oid, *, *, *)$ in FOT.
2: **if** $e \neq \emptyset$ **then**
3: $pos \leftarrow e.pos$
4: **else**
5: Locate the position, pos, and velocity vector, $\overline{vel}$, of the mobile object with identifier oid together with the time, tm, this information was recorded. In case the query is received from the object with identifier oid itself, then this information is also assumed to be received with the query. Otherwise request this information from the object.
6: Insert the entry $(oid, pos, \overline{vel}, tm)$ into FOT.
7: **end if**
8: Assign a unique identifier, qid, to the query.
9: $curr\_cell \leftarrow Pmap(pos)$
10: Set mon_region using $curr\_cell$ and $region$.
11: Insert the entry $(qid, oid, region, curr\_cell, mon\_region, filter, \emptyset)$ into SQT.
12: **for all** $A_{i,j} \subset mon\_region$ **do**
13: $RQI(i, j) \leftarrow RQI(i, j) \cup \{qid\}$
14: **end for**
15: $Send(oid, [\text{Query installed}])$
16: $B' \leftarrow \bigcup_{A_{i,j} \in mon\_region} Bmap(i, j)$
17: **for all** $b \in B'$ **do**
18: $Broadcast(b, [\text{Install query } (qid, pos, \overline{vel}, tm, region, filter)])$
19: **end for**
(B) **Mobile Object: Received Message [Query installed]**
1: $hasMQ \leftarrow true$
(C) **Mobile Object: Received Message [Install query** $(qid, pos, \overline{vel}, tm, region, filter)$**]**
1: Use $Pmap(pos)$ and $region$ to calculate the mon_region.
2: **if** current position is in mon_region and $filter(this) = true$ **then**
3: Insert entry $(qid, pos, \overline{vel}, tm, region, mon\_region, false)$ into LQT.
4: **end if**

3.3 Mobile Object Side Processing

In MobiEyes, a mobile user can issue a location query or choose to enter a sleep mode at anytime. Whenever mobile objects are awake and active, we assume that they are willing to participate in the corporative processing of nearby location queries. The task of making sure that the mobile objects in the monitoring region of a moving location query are aware of this MQ, is accomplished through server broadcasts, which are triggered by 1) server side installations of new moving location queries, 2) significant velocity vector changes of the focal objects, or 3) current grid cell changes of focal or nonfocal objects.

3.3.1 Installing MQs at the Mobile Object Side

The mobile object side processing for location query installation is performed as follows: Upon receiving a broadcast message, a mobile object examines each MQ in the broadcast message using its local state and determines whether this MQ is nearby and whether it should be registered locally. The decision is primarily based on whether the mobile object itself is within the monitoring region of the MQ and whether the query's filter is also satisfied by the mobile object. If the answer to both of these questions is yes, the mobile object registers the query into its local query table LQT. Otherwise,

the object discards the MQ. The details of these procedures are given in Algorithm 1 (C).

3.3.2 Query Processing at the Mobile Objects

A mobile object periodically processes all location queries registered in its LQT table. For each locally registered MQ, the mobile object predicts the position of the focal object of the MQ using the velocity, time, and position information available in the LQT entry of the MQ (line 3 under (A) in Algorithm 2). Then, the object compares its current position and the predicted position of the focal object of this MQ through a simple containment check based on the spatial region of this MQ, and determines whether itself is covered by the query's spatial region or not. If the result is not different than the last result computed in the previous time step, no reporting to the server is performed. Otherwise, we have one of two situations: The mobile object has just moved into the spatial region of the MQ or it has just moved out of the spatial region of this MQ. In both cases, the change is relayed to the location server. The server, in turn, differentially updates the query result to keep the answer to this MQ up to date. Algorithm 2 describes mobile object side query processing in detail.

Algorithm 2: Mobile Object Query Processing Logic

(A) **Mobile Object: Periodical Query Processing**

1: Let pos be the current position of the object and ctm be the current time.
2: **for all** e in LQT **do**
3: {predict the position of the focal object}
4: $fpos \leftarrow e.pos + (ctm - e.tm) * e.\overline{vel}$
5: **if** $(pos - fpos) \in e.region$ **then**
6: **if** $e.isTarget = false$ **then**
7: $Send(server, [\text{Query result change } (e.qid, oid, true)])$
8: $e.isTarget \leftarrow true$.
9: **end if**
10: **else if** $e.isTarget = true$ **then**
11: $Send(server, [\text{Query result change } (e.qid, oid, false)])$
12: $e.isTarget \leftarrow false$.
13: **end if**
14: **end for**

(B) **Server: Received message [Query result change** $(qid, oid, isTarget)$**]**

1: Let $e = (qid, *, *, *, *, *, *, *)$ in SQT.
2: **if** $isTarget = true$ **then**
3: $e.result \leftarrow e.result \cup \{oid\}$
4: **else**
5: $e.result \leftarrow e.result \setminus \{oid\}$
6: **end if**

4 Optimizations: Efficient and Reliable Processing of MQs

4.1 Efficient Processing of MQs

We have presented the basic algorithms for distributed processing of moving location queries. In this section, we describe four optimization mechanisms used in MobiEyes to efficiently handle system dynamics with the aim of minimizing the server load and the amount of local processing at the mobile object side, and reducing the communication cost between the mobile objects and the location server.

4.1.1 Handling Mobility of Queries and Objects

Once a moving location query (MQ) is installed in the MobiEyes system, the focal object of the MQ needs to report to the server only when there is a significant change to its location information. We consider two types of changes to be significant: 1) when a focal object moves out of its current grid cell, or 2) when the focal object of a MQ changes its velocity vector beyond a predefined threshold. On the other hand, a nonfocal object may[1] need to report to the server only when it changes its current grid cell. We describe the mechanisms for handling velocity vector changes first and then discuss the mechanisms for handling objects that change their current grid cells.

Handling Velocity Vector Changes with Dead Reckoning. The velocity vector of a mobile object will almost always change at each time step in a real world setup, although the change might be insignificant compared to the previous time step. One way to control the amount of location updates from mobile objects to the location server is to notify the server of the new velocity vector information of the focal object of a MQ and have the server to relay such update to the objects located in the monitoring region of the MQ, 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.

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 predicted using the last velocity vector information it reported to the location server. In case this difference is larger than a threshold, say Δ, the new velocity vector information is relayed to the location server.[2] Fig. 3 provides an illustration. The path of a focal object is depicted with a solid line, where its path predicted by objects in its monitoring region (based on its last velocity information relayed to the objects) is depicted with a dashed line. At each time step, the focal object first samples its position, which is depicted by small squares in the figure. Then, it calculates the position that other objects predict it to be at, which is depicted with small circles in the figure. In case the distance between these two positions is larger than Δ, the focal object notifies the server with its new velocity vector and the server relays the new velocity information of the focal object to all objects in its monitoring region through broadcast (see the fifth time step in Fig. 3).

Concretely, when the focal object of a MQ reports a significant 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 an optimized broadcast schedule, which generates the minimum number of broadcasts using the grid cell to base station mapping $Bmap$. An illustration of this process is given in Fig. 2, where a focal object together

1. It depends on whether Eager Query Processing (EQP) or Lazy Query Processing (LQP) is used

2. We do not consider inaccuracies due to motion modeling. See [35] for a discussion of motion update policies and trade-offs.

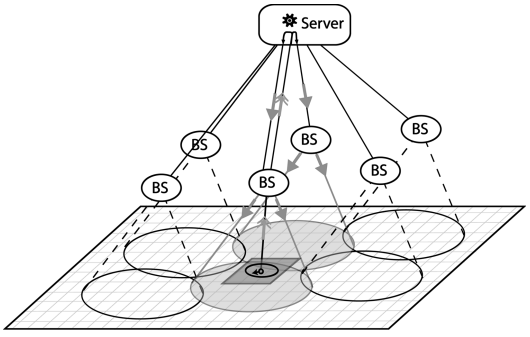

Fig. 2. Conveying velocity vector changes.

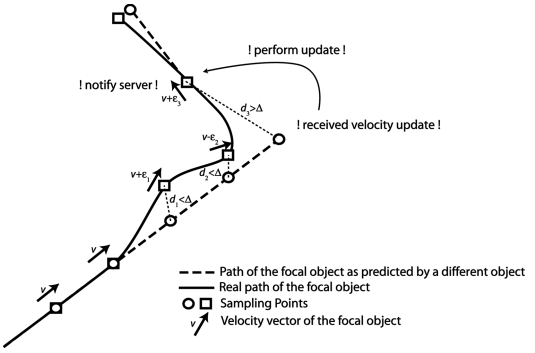

Fig. 3. Dead Reckoning in MQ evaluation.

with its monitoring region is shown. The focal object sends its new velocity information to the server (shown with double headed arrows in the figure), which in turn broadcasts this information using two base stations that cover the monitoring region of the query (shown with single headed arrows in Fig. 2). Algorithm 3 describes how velocity vector changes are handled in detail.

Algorithm 3: Mobile Object Changed Velocity Vector

(A) **Mobile Object: Periodic Velocity Change Processing**

1: **if** $hasMQ = true$ **then**
2: Let $\overline{pvel}$ be the last velocity and $ppos$ be the last position information relayed to the server where ptm is the time this information was relayed.
3: Record the new position, pos, new velocity vector, $\overline{vel}$, and the current time, tm.
4: $opos \leftarrow ppos + (tm - ptm) * \overline{pvel}$ {perform dead-reckoning}
5: **if** $|opos - pos| > \Delta$ **then**
6: $Send(server, [\text{New velocity data } (oid, pos, \overline{vel}, tm)])$
7: **end if**
8: **end if**

(B) **Server: Received Message [New velocity data** $(oid, pos, \overline{vel}, tm)$**]**

1: Let $e = (oid, *, *, *)$ in FOT.
2: $e \leftarrow (oid, pos, \overline{vel}, tm)$ {update the entry e}
3: **for all** $e = (*, oid, *, *, *, *, *)$ in SQT **do**
4: $B' \leftarrow \bigcup_{A_{i,j} \subset e.mon\_region} Bmap(i, j)$
5: **for all** $b \in B'$ **do**
6: $Broadcast(b, [\text{Query velocity change } (e.qid, pos, \overline{vel}, tm)])$
7: **end for**
8: **end for**

(C) **Mobile Object: Received Message[Query velocity change** $(qid, pos, \overline{vel}, tm)$**]**

1: Let $e = (qid, *, *, *, *, *, *)$ in LQT.
2: **if** $e \neq \emptyset$ **then**
3: $e.pos \leftarrow pos$; $e.\overline{vel} \leftarrow \overline{vel}$; $e.tm \leftarrow tm$
4: **end if**

One way to optimize the broadcast frequency of propagating velocity updates to mobile objects of interest is to let the server *batch several velocity vector updates from mobile objects and broadcast them during agreed upon time intervals* so that the mobile objects can activate their radio only during scheduled intervals, thus leading to considerable saving in terms of power consumption.

Handling Objects that Change their Grid Cells: Eager or Lazy Query Propagation. Based on the grid structure and the monitoring region of MQs in MobiEyes, as long as all mobile object move within their current cells, the list of nearby MQs responsible by the mobile objects will remain the same. However, when a mobile object changes its current grid cell, such location update may cause a change to the set of moving location queries this object is responsible for monitoring. In case the object that has changed its current grid cell is a focal object, the location update may also cause a change to the set of mobile objects responsible for monitoring the MQs bounded to this focal object.

With *Eager Query Propagation* (EQP), when an object changes its current grid cell, it immediately 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, if the object is a nonfocal object, the server only needs to find what new queries should be installed on this object and then perform the query installation on this mobile object.

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 that has to be installed on this mobile object. Then, the server sends the set of new queries to the mobile object for installation. The focal object table FOT and the server query table SQT are used to create the required installation information for the queries to be installed on the object. However, if the object that changes its current grid cell is also a focal object of some query, then for each query with this object as its focal object, the server performs the following operations:

1. It updates the query's SQT table entry by resetting the current grid cell and the monitoring region to their new values.
2. It also updates the RQI index to reflect the change.
3. Then, the server computes the union of the query's previous monitoring region and its new monitoring region, and
4. 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, a mobile object performs the following operations for installing or removing a moving location 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. The detailed procedure is given by Algorithm 4.

Algorithm 4: Mobile Object Changed its Current Grid Cell

(A) **Mobile Object: Changed Current Grid Cell**

1: Let pos be the new position of the mobile object.
2: Remove all entries e in LQT satisfying $pos \notin e.mon\_region$.
3: Let A_{i_p,j_p} be the previous and A_{i_c,j_c} be the current grid cell in which pos lies.
4: $Send(server, [\text{Object changed grid cell } (oid, (i_p, j_p), (i_c, j_c))])$

(B) **Server: Received Message [Object changed grid cell** $(oid, (i_p, j_p), (i_c, j_c))$**]**

1: $Q_{diff} \leftarrow RQI(i_c, j_c) \setminus RQI(i_p, j_p)$
2: **for all** $qid \in Q_{diff}$ **do**
3: Let $e = (qid, *, *, *, *, *, *, *)$ in SQT.
4: $Send(oid, [\text{Install query}(e.qid, e.pos, e.\overline{vel}, e.tm, e.region, e.filter)])$
5: **end for**
6: Let $e = (oid, *, *, *)$ in FOT
7: **for all** $e_s = (*, oid, *, *, *, *, *, *)$ in SQT **do**
8: $e_s.curr\_cell \leftarrow A_{i_c,j_c}$
9: $old\_mon\_region \leftarrow e_s.mon\_region$
10: Set $e_s.mon\_region$ using $e_s.region$ and $e_s.curr\_cell$
11: **for all** $A_{i,j} \subset (old\_mon\_region \setminus e_s.mon\_region)$ **do**
12: $RQI(i, j) \leftarrow RQI(i, j) \setminus \{e_s.qid\}$
13: **end for**
14: **for all** $A_{i,j} \subset (e_s.mon\_region \setminus old\_mon\_region)$ **do**
15: $RQI(i, j) \leftarrow RQI(i, j) \cup \{e_s.qid\}$
16: **end for**
17: $combined\_area \leftarrow e_s.mon\_region \cup old\_mon\_region$
18: $B' \leftarrow \bigcup_{A_{i,j} \subset combined\_area} Bmap(i, j)$
19: **for all** $b \in B'$ **do**
20: $Broadcast(b, [\text{Update query } (e_s.qid, e.pos, e.\overline{vel}, e.tm, e_s.region, e_s.filter)])$
21: **end for**
22: **end for**

(C) **Mobile Object: Received Message [Update.query** $(qid, pos, \overline{vel}, tm, region, filter)$**]**

1: Use pos and $region$ to calculate the mon_region.
2: Let $e = (qid, *, *, *, *, *, *)$ in LQT.
3: **if** current position is in mon_region **then**
4: **if** $e \neq \emptyset$ **then**
5: $e.mon\_region \leftarrow mon\_region$
6: **else if** $filter(this) = true$ then
7: Insert $(qid, pos, \overline{vel}, tm, region, mon\_region, false)$ into LQT.
8: **end if**
9: **else**
10: Remove entry e (if exists) from LQT.
11: **end if**

When using an eager query propagation scheme, we require each mobile object (focal or nonfocal) that changes its current grid cell to report this change to its location server immediately. The only reason for a nonfocal object to communicate with the server is to immediately obtain the list of new location queries that it needs to register in response to the change of its current grid cell. One way to reduce the amount of communication between mobile objects and the location server is to use a lazy query propagation scheme. This allows us to eliminate the need for nonfocal objects to contact the server to obtain the list of new MQs. Concretely, instead of obtaining the new location queries from the server and installing them immediately on the object upon a change of grid cell, the mobile object can wait until the location server broadcasts the next velocity vector changes regarding the focal objects of the MQs 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 moving location 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 MQs. Using lazy propagation, a mobile object upon changing its current grid cell will be unaware of the new set of location queries nearby until the focal objects of these location 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 change their velocity vectors frequently. The lazy query propagation may not prevail over the eager query propagation, when: 1) most of the focal objects do not change their velocity vectors to cause inaccurate predictions beyond the specified threshold frequently, 2) the grid cell size α is too small, and 3) the nonfocal objects change their current grid cells at a much faster rate than the focal objects. In such situations, nonfocal objects may end up not being included in some of nearby moving location queries. We experimentally evaluate the *Lazy Query Propagation* (LQP) approach and study its performance advantages as well as its impact on query result accuracy in Section 5.

4.1.2 Location Query Grouping

In MobiEyes, a mobile user can pose many different queries and a query can be posed multiple times by different users. Thus, many moving location queries may share the same focal object. Effective optimizations can be applied to handle multiple location queries bound to the same mobile object. These optimizations help decreasing both the computational load on the mobile 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 location queries as *groupable MQs* if they are bounded to the same focal object. We refer to those groupable MQs that have the same monitoring region as *MQs with matching monitoring regions*, and refer to the groupable MQs that have different monitoring regions as *MQs with nonmatching monitoring regions* (see Fig. 4). Based on these different sharing 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. We illustrate this with an example. Consider three MQs: $q_1 = (qid_1, oid_i, r_1, filter_1)$,

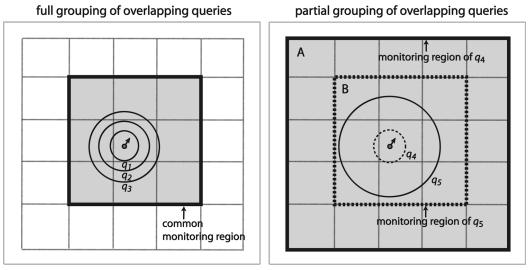

Fig. 4. Minimizing duplicate processing by grouping moving location queries.

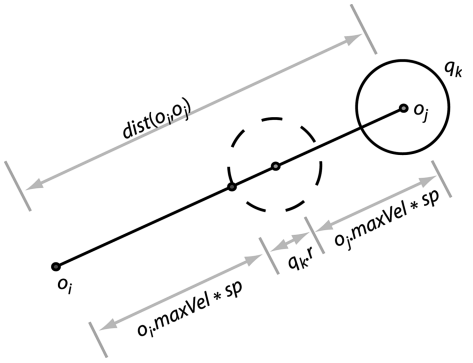

Fig. 5. Safe period optimization.

$q_2 = (qid_2, oid_i, r_2, filter_2)$, 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:

$$q_3 = (qid_3, oid_i, (r_1, r_2, r_3), (filter_1, filter_2, filter_3)).$$

To facilitate the local processing of groupable MQs, we introduce the concept of *query bitmap*, which is a bitmap containing one bit for each location query in a query group, each bit can be set to 1 or 0 indicating whether the corresponding query should include the mobile object in its result or not. When a mobile object is processing a set of groupable MQs with matching monitoring regions, it first checks if its current position is inside the spatial region of a location query with a larger radius. Only when its current position is inside the spatial region of a location query with a larger radius, it needs to consider the location queries with smaller radiuses. When a mobile 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 report. With the query bitmap the location server can easily determine whether the reporting mobile object should be included in the query results of which groupable MQs.

Grouping MQs with Nonmatching Monitoring Regions. For groupable MQs with nonmatching monitoring regions, we propose to perform the location query grouping at the mobile object side only. We illustrate this claim with an example. Consider an object o_j inside region B in Fig. 4. Since there is no global server side grouping performed for queries q_4 and q_5, o_j has both of them installed in its LQT table. o_j can save some processing by linking these two queries inside its LQT table. 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.

4.1.3 Reducing Local Processing Cost with Safe Period Optimization

In MobiEyes, each mobile 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 mobile 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, as shown in Fig. 5. 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 mobile 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.

4.2 Reliable Processing of MQs

Another important issue in distributed processing of moving queries is the level of reliability guarantee that the system can provide. In MobiEyes, the reliability of the system is defined by the reliability of the mobile objects and the reliability of the server (including the reliability of the communication between mobile objects and the server) with respect to distributed processing of moving queries.

4.2.1 Reliability of the Mobile Objects

In MobiEyes, each mobile object whose current grid cell intersects with the monitoring region of a moving query will maintain an entry associated with that query in its local MQ table (LQT) (recall Section 3.1). The LQT table is the most important state information maintained at the mobile object. It keeps all the MQs that this mobile object needs to

process. In the event of failure, such as the computational unit on a mobile object crashes, the LQT state is either lost (when the previous state information was not stored persistently and made available) or less current (when the recovery can only restart the system at the previous checkpoint of the state). One way to recover this state is to obtain the new state through reinitialization with the server upon restart. However, several problems (such as stale LQT state, incorrect query results) may occur when the mobile object continues to move in the presence of crashes on its computing unit or the focal objects of some MQs in its LQT table continue to move during the failure period. The degree of damages caused by such problems depends upon whether the mobile object experiencing the failure is a nonfocal or focal object.

Failure at a Nonfocal Mobile Object. When failure happens at a mobile object that is a nonfocal object, we may have the mobile object incorrectly included in some of the query results for an arbitrarily long time. This is primarily caused by the following two possible errors due to the failure at the nonfocal object: First, the nonfocal object may continue to move in the presence of crashes on its computing unit. Such location changes will not be reported to the server due to the failure at the nonfocal object. Second, the focal objects of some MQs in the LQT table of this nonfocal object may continue to move during the failure period and report their locations to the server whenever a significant change occurs. The location updates of these focal objects will be disseminated to this nonfocal object by the server through broadcast, but this broadcast will not be received and processed at the nonfocal object due to failure. In both cases, the location changes of the nonfocal object or the location changes of the focal objects in its LQT table, during the period of failure, may cause following events to happen: 1) The spatial regions of some MQs in the LQT table of the nonfocal object no longer contain its position and 2) the current grid cell of the nonfocal object no longer intersects with the monitoring regions of some MQs listed in its LQT table before the crash.

In an ordinary situation, when the nonfocal object detects that the spatial region of an MQ in its LQT table no longer contains its position, the nonfocal object will be removed from the result set of this MQ and this change will be reported to the server through a query result change update. Similarly, when the nonfocal object detects that the monitoring region of an MQ in its LQT table does not intersect with its current grid cell anymore, it will remove the MQ from its LQT table. Furthermore, the reverse query index maintained at the server (which corresponds to the entries in LQT table) will no longer list this MQ in the moving query list corresponding to the current grid cell of the nonfocal object.

However, due to the failure happened at the nonfocal object, its LQT table can only be recovered to the new state through reinitialization with the server upon the restart of the computing unit, based on the reverse query index of the nonfocal object's current grid cell. The new LQT table may not contain the MQs whose results, before the crash, included the nonfocal object (due to events 1) and 2) described above). As a consequence, the results of these MQs at the server side will falsely include this nonfocal object, for two reasons: 1) The nonfocal object did not send the query result changes to the server due to failure. 2) It could not send these changes to the server upon the restart, as its new LQT table does not include these MQs anymore.

In short, it is the mobile object's responsibility to report to the server when it changes its state with regard to being included in or excluded from a query's result. When the mobile object experiences failure such as crashes of the computing unit, it loses not only its LQT table, but also the local processing and reporting capability. Since the recovered LQT table through reinitialization with the server upon restart may not include some of the MQs whose query results should have been updated during the failure period to exclude the mobile object from their query result sets, the failure leads to incorrect query results maintained at the server side for an arbitrary long time.

Failure at a Focal Mobile Object. When failure happens at a mobile object that is a focal object, in addition to the problems stated above, a new problem arises: We may have stale moving query entries, corresponding to the focal object experiencing failure, residing in the LQT tables of other mobile objects. This is because location changes of the focal object that has experienced failure will be lost during the failure period. In an ordinary situation, such location changes may result in MQs associated with the focal object to be removed from the LQT tables of other objects when the current cells of these objects do not intersect with the new monitoring region of the MQs associated with the focal object.

However, due to the failure that happened at the focal object, such monitoring region changes are lost at the focal object and in turn LQT state updates are not performed at the mobile objects whose LQT tables contain MQs associated with the focal object. As a result, the mobile objects that were residing in the monitoring region just before the focal object crashed, but are not residing in the monitoring region upon the restart of the focal object, may still have an entry in their LQT tables corresponding to the failed focal object, which should have been removed in an ordinary situation. Furthermore, these entries may contain stale information about focal objects and may result in sending wrong query result updates. In the worst case, these entries may stay in the LQT tables for an arbitrarily long time.

The above-mentioned problems can be aggravated when the computational unit on a mobile object fails and *stops* for an arbitrarily long time. When this happens, we need efficient mechanisms such that queries corresponding to such failed and stopped focal objects and query result entries corresponding to such failed and stopped focal or nonfocal objects can be detected in time and removed from the system.

In MobiEyes, we introduce a simple and yet effective mechanism, called *forced updates*, to solve the problems described above.

Error Handling through Forced Updates. MobiEyes provides two kinds of forced updates to ensure the reliability of the system against failures:

Forced Result Updates: In the basic model of MobiEyes without reliability guarantee, each mobile object sends query result updates to the server only when it is included into or excluded from the result of an MQ in its LQT table. With forced result updates, we additionally require that each mobile object reports its state on whether it is included in or excluded from the result of an MQ in its LQT table to the server periodically (every T_r time unit), regardless of whether or not a change has occurred in the inclusion status of the object with respect to the results of queries in LQT.

TABLE 1
Simulation Parameters

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	1,000-100,000	10,000
nmq	Number of moving queries	100-1,000	1,000
nmo	Number of objects changing velocity vector per time step	100-10,000	1,000
$area$	Area of consideration	300 x 300 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	

Forced Velocity Vector Updates: Similarly, in the basic model of MobiEyes, each focal object sends location updates to the server whenever its velocity vector has changed significantly. With forced velocity vector updates, we additionally require that each focal object reports its location update to the server periodically (every T_r time unit) even if no significant change occurred in the velocity vector.

With the forced result updates, a query result entry is considered invalid if it is not updated during the last $c * T_r$ time units. Similarly, with forced velocity vector updates, a moving query entry in a LQT table is considered invalid if the velocity vector field of the entry is not updated during the last $c * T_r$ time units. The time interval parameter T_r adjusts the tradeoff between performance and accuracy. With smaller values of T_r errors are resolved faster (increased accuracy) but more messages need to be exchanged between mobile objects and the server. Accuracy is obtained with the price of performance. On the other hand, with larger values of T_r, we require less messages to be exchanged between mobile objects and the server (increased performance), but errors are resolved slower (decreased accuracy). The parameter c is used to control the tolerance to delays and errors in the communication.

4.2.2 Reliability of the Server

The distributed approach taken by MobiEyes significantly decreases the load on the server side, making a server crash less probable. However, a crash on the server side is a serious issue for the system. Handling a server crash requires more than recovering the server state because the state maintained persistently on the server side may not reflect the most recent updates at the server before the crash occurred. Thus, the server state upon recovery may contain stale information. Since the algorithms for updating the server state during the normal operation of the system are mostly incremental, an effective way to handle a server failure is to use a failover solution through replicated servers [28]. In the absence of failover servers, a straight forward approach to handle a server crash is to reinitialize the whole system.

5 Experiments

In this section, we describe three sets of simulation-based experiments, which are used to evaluate our solution. 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. We also give an analytical estimate of the messaging cost and compare it with the results from simulations. The third set of experiments investigates the amount of computation a mobile object has to perform, by measuring on average the number of queries a mobile 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 presented in the rest of the paper, the parameters take their default values if not specified otherwise. The area of interest considered in the simulation is a square shaped region of 300×300 square miles. The number of objects we consider ranges from 1,000 to 100,000 where the number of queries range from 100 to 1,000. These numbers can be scaled up without effecting the conclusions we draw from our experiments, as long as the object density is kept constant. The ratio of these parameters to one another closely follows the values used in [25].

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. This means that 75 percent of the objects satisfy the filter of a given query.

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 default object speeds are set high in our experimental setup in order to stress test the algorithms. We also experiment with lower object speeds (that are more favorable against MobiEyes) by scaling the values in the default object speed list by a velocity factor smaller than 1. 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 10,000.

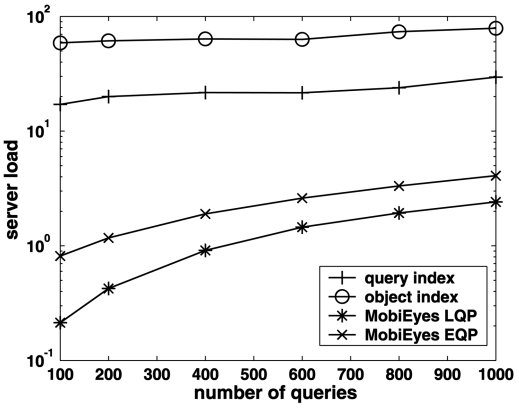

Fig. 6. Impact of distributed query processing on server load.

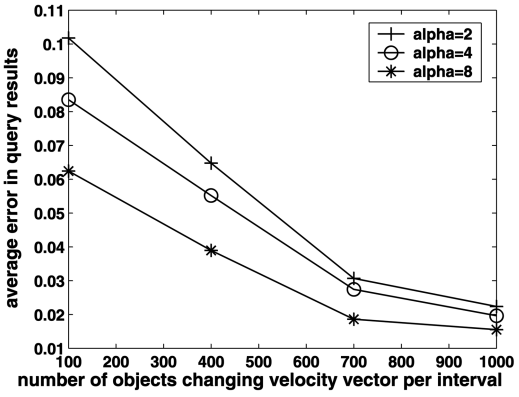

Fig. 7. Error associated with lazy query propagation.

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 mobile 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 mobile objects is by indexing objects. In this approach, a spatial index is built over object locations. We use an R^*-tree [6] 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. A better way to evaluate MQs is to use an index on queries instead of objects, as the number of queries is expected to be smaller than the number of objects.

Indexing Queries: The second centralized approach to processing spatial continuous queries on mobile objects is by indexing queries. In this approach a spatial index, again an R^*-tree indeed, is built on 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. Fig. 6 and Fig. 8 depict the results obtained. Note that the y-axes, which represent the sever load, are in log-scale. The x-axis represents the number of queries considered in Fig. 6, and the different settings of α parameter in Fig. 8.

It is observed from Fig. 6 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.

Fig. 6 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 4, lazy query propagation may have some inaccuracy associated with it. Fig. 7 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. Fig. 7 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 Fig. 7 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 Fig. 7 as it shows that the error in query results increases with decreasing α.

Fig. 8 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 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

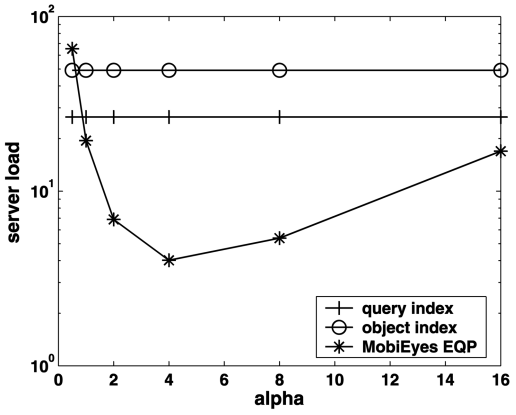

Fig. 8. Effect of α on server load.

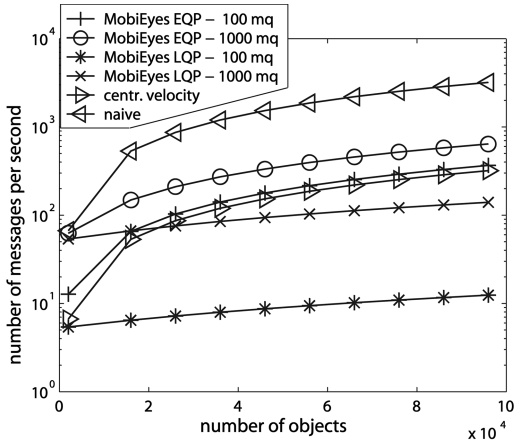

Fig. 10. Effect of number of objects on messaging cost.

further investigate the problem of selecting a good value for α, through the use of an analytical model, 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 mobile 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 mobile 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 velocity* approach. 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 Fig. 9 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. Fig. 9 shows that values in the range [4,6] are ideal for α with respect to the number of queries ranging from 100 to 1,000. The optimal value of the α parameter can be derived analytically using a simple model (see Section 5.3.1).

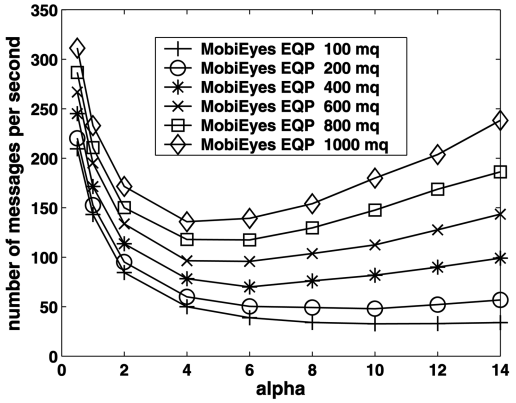

Fig. 9. Effect of α on messaging cost.

Fig. 10 studies the effect of number of objects on the messaging cost. It plots the number of messages per second (in logarithmic scale) 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 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, except the central velocity approach which provides lower messaging cost. When both the number of queries and the number of objects are small, again all approaches come close to one another, this time except the naïve approach which incurs higher messaging cost. However, all approaches other than MobiEyes with LQP, have a high messaging cost when the ratio of the number of objects to the number of queries is high. On the other hand, MobiEyes with EQP shows similar scalability with the central velocity approach. MobiEyes with LQP scales better than all other approaches, with increasing number of objects. This is because it does not require nonfocal objects to report their positions to the server.

Fig. 11 shows the messaging cost as a function of velocity factor for different numbers of objects. Note that the default object speeds are set high in our experimental setup in order to stress test the algorithms. In this experiment, velocity factor is used to scale down the object speeds. Fig. 11 shows that the relatively high messaging cost of MobiEyes with EQP is mainly due to high object speeds, where the impact of object speeds on messaging cost is similar for MobiEyes with LQP although on a smaller scale. The main reason for MobiEyes with EQP to perform better with lower speeds is the decreased number of cell crossings

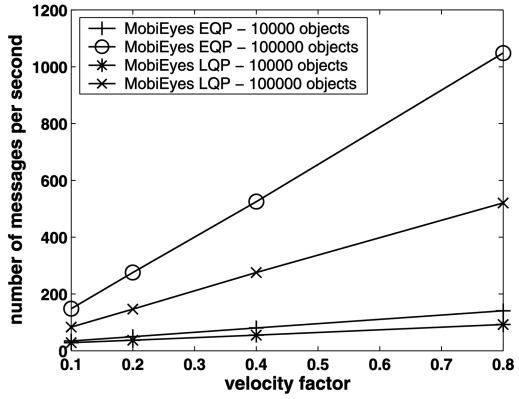

Fig. 11. Effect of objects speeds on messaging cost.

for nonfocal objects, which results in less communication with the server. We also observe from Fig. 11 that the improvement in messaging cost with decreasing object speeds is more prominent for large number of objects.

Fig. 12 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 Fig. 12 is that the messaging cost of MobiEyes with LQP scales well when compared to the central velocity approach, since the gap between the two increases as the number of objects changing velocity vector per time step increases. Note that the central velocity approach degrades to the naïve approach when all objects change their velocity vectors at each time step. MobiEyes with EQP performs better than the central velocity approach only for small number of queries and large number of objects changing velocity vector per time step.

Fig. 13 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 Fig. 13 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

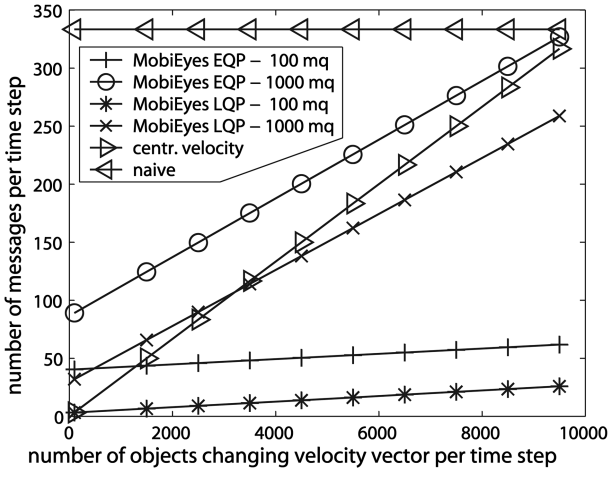

Fig. 12. Effect of number of objects changing velocity vector per time step on messaging cost.

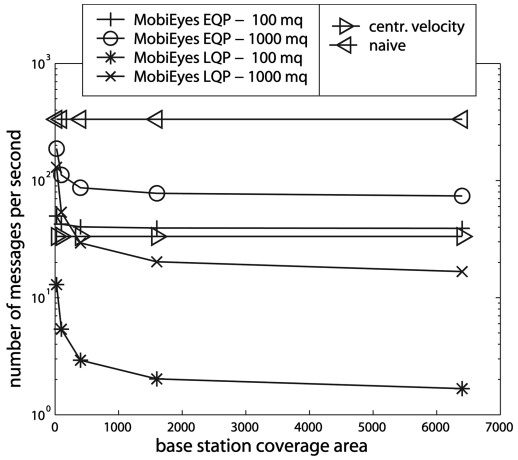

Fig. 13. Effect of base station coverage area on messaging cost.

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 mobile 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 mobile object. In such environments, indexing on the air [16] can be used as an effective mechanism to deal with this problem. We do not consider extreme scenarios like satellite broadcast and focus on cellular networks.

In MobiEyes, a large base station coverage area gives less effective results only when the total number of mobile objects in the region of coverage is large because there will be large number of focal objects in the region of coverage and many of the broadcasted updates originating from these focal objects will be discarded by a large number of mobile objects due to the mismatch between the monitoring region sizes and the base station coverage area sizes (the latter being much larger). However, large base station coverage areas make very poor use of the frequency spectrum and are not suitable for hot spot regions. Such large coverage areas are more common in sparsely populated regions where the total number of mobile nodes is relatively small. This nature of base station coverage area sizes in cellular networks is very favorable for the MobiEyes system. For recent 3G cellular network technologies such as CDMA200 1xEV-DO [26], base station coverage areas in urban settlements are small, in general, not larger than 3 miles radius.

We have evaluated the scalability of the MobiEyes in terms of the total number of messages exchanged in the system and the reduction of the server load of MobiEyes approach. Now, we study the per object power consumption due to communication between mobile objects and the server. 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 [17], we take the power consumption of transmitter and receiver electronics as 150mW and 120mW, respectively, and we assume a 300mW transmit amplifier with 30 percent

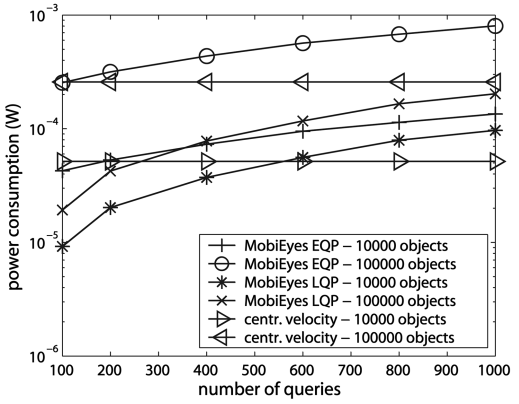

Fig. 14. Effect of the number of queries on per object power consumption due to communication.

efficiency [17]. We consider 14kbps uplink and 28kbps downlink bandwidth (typical for current GPRS technology). Note that sending data is more power consuming than receiving data.[3] 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 velocity approaches. The graph in Fig. 14 plots the per object power consumption due to communication (on logarithmic y-axis) as a function of number of queries for different numbers of mobile objects. Since the naive approach requires every object to send its new position to the server, its per object power consumption is the worst, thus it is not included in the comparison. In MobiEyes, however, a nonfocal object does not send its position or velocity vector to the server, but it receives query updates from the server. Since the cost of receiving data in terms of energy consumption is lower than transmitting, MobiEyes is expected to be effective in terms of per object power consumption. It is observed from Fig. 14 that, except for the case when the number of objects is small and the number of queries is large, the central velocity approach is outperformed by MobiEyes with LQP. MobiEyes with LQP performs especially well when the number of objects is large. On the other hand, MobiEyes with EQP performs worse than the central velocity approach, with the exception of cases where the number of queries is small. 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. This is mainly due to the difference between the size of a broadcast area and the monitoring region of a query.

5.3.1 Analytical Messaging Cost Estimate

In this section, we give an analytical estimate of the messaging cost of our solution, which can also be used to set the optimal value of the cell size parameter α of the grid G corresponding to the universe of discourse (recall Fig. 8 and Fig. 9 that the effect of α on server load is analogous). The cost estimation is based on the EQP approach and its extension to LQP is straightforward. Let $mcost(T)$ be the average number of messages exchanged (messaging cost) during a given time period of T seconds per one object. Let $avgspd$ be the average moving speed of an object and let $avgr$ be the average query radius. The messaging cost can be divided into two components, namely, the cost due to an object changing its current grid cell (denoted as $cell\_change\_cost$) and the cost due to a focal object changing its velocity vector (denoted as vel_change_cost). Let nmq, nmo, ts be defined as shown in Table 1. Then, we can estimate $mcost(T)$ as follows:

$$mcost(T) = \frac{T}{\frac{\alpha}{avgspd}} * cell\_change\_cost + \frac{T}{ts} * \frac{nmo}{no} * \frac{nmq}{no} * vel\_change\_cost.$$

The expression preceding $cell\_change\_cost$ is a crude estimate of the number of times a given object changes its current grid cell during the time interval T. The expression preceding vel_change_cost is the probability that a given object is a focal object of some query times the number of times a given object changes its velocity vector during the time interval T.

The vel_change_cost is composed of a message being sent from a focal object to the server plus the number of broadcasts performed to convey this velocity change to a set of objects that reside in the monitoring region of the given focal object, which can be estimated as:

$$vel\_change\_cost = 1 + \left(1 + \frac{mrslen}{alen}\right)^2.$$

Here, $mrslen = \alpha * (1 + 2 * \lceil avgr/\alpha \rceil)$ is the average length of the side of a monitoring region. $(1 + \frac{mrslen}{alen})^2$ is an estimate of the number of broadcast areas required to cover a monitoring region, where $alen$ is the average base station coverage area side length (listed in Table 1).

The $cell\_change\_cost$ is composed of a message being sent from the object to the server plus the cost of sending new queries of interest to the object ($new\_queries\_cost$) plus the cost of relaying the monitoring region change to a set of other objects ($region\_update\_cost$) in case the object of interest is a focal object of some query. This can be estimated as follows:

$$cell\_change\_cost = 1 + new\_queries\_cost + \frac{nmq}{no} * region\_update\_cost,$$

$$new\_queries\_cost = nmq * \frac{\alpha * mrslen}{area},$$

$$region\_update\_cost = \left(1 + \frac{mrslen}{alen}\right) * \left(1 + \frac{\alpha + mrslen}{alen}\right).$$

Let A and A' denote any two adjacent cells, assuming an object moves from cell A to cell A'. In the $new\_queries\_cost$ formula, $\alpha * mrslen$ is the size of the region that covers the possible current grid cells of focal objects whose monitoring regions contain the new cell A', but not the old cell A. Multiplying this value with $nmq/area$ gives an estimate on the number of queries whose monitoring regions intersect the new cell, but not the old cell.

The $region\_update\_cost$ formula estimates the number of broadcast areas required to cover the region which is the union of old and new monitoring regions of a query. $mrslen$ gives the length of one side of this region, where $\alpha + mrslen$ gives the other.

Fig. 15 compares the analytical estimate on the number of messages with the results obtained from the simulation

3. In this setting, transmitting costs $\sim 80\mu jules/bit$ and receiving costs $\sim 5\mu jules/bit$.

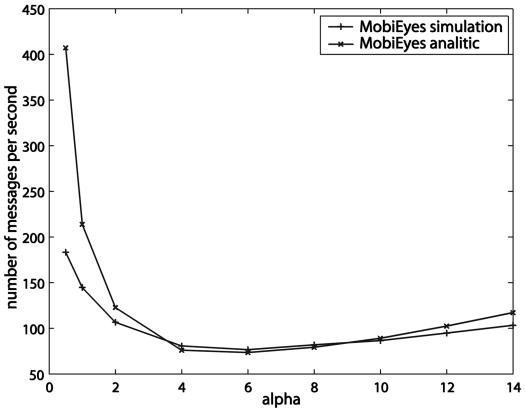

Fig. 15. Comparison of analytical messaging cost with simulation results.

for different values of α. The y-axis represents the number of messages exchanged per time step, where x-axis represents different α values. It is clear that the estimate is quite accurate for most of the values of α. When α is small, the crude estimate of the number of times an object changes its current grid cell results in overestimating the messaging cost. This problem alleviates as α increases. The small underestimate when we have larger α values is due to the fact that our analytical estimate does not include the cost of notifications sent from an object to the server when the object is added into or removed from the result of a query.

5.4 Amount of Computation on Mobile Object Side

In this section, we study the amount of computation placed on the mobile object side by the MobiEyes approach to processing of MQs. One measure of this is the number of queries a mobile object has to evaluate at each time step, which is the size of the LQT table (Recall Section 3.1).

Fig. 16 and Fig. 17 study the effect of α and the effect of the total number of queries on the average number of queries a mobile object has to evaluate at each time step (average LQT table size). The graph in Fig. 16 plots the average LQT table size as a function of α for different number of queries. The graph in Fig. 17 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 exceed 10 for the simulation setup. The second observation is that the

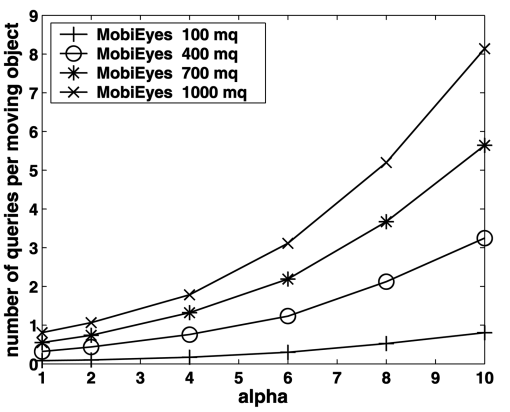

Fig. 16. Effect of α on the average number of queries evaluated per step on a mobile object.

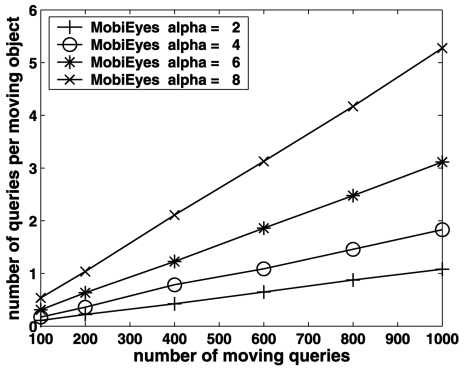

Fig. 17. Effect of the total number of queries on the average number of queries evaluated per step on a mobile object.

average size of the LQT table increases exponentially with α where it increases linearly with the number of queries.

Fig. 18 studies the effect of the safe period optimization on the average query processing load of a mobile object. The x-axis of the graph in Fig. 18 represents the α parameter, and the y-axis represents the average query processing load of a mobile object. As a measure of query processing load, we took the average time spent by a mobile object for processing its LQT table in the simulation. Fig. 18 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 nonzero 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 Fig. 18, 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

Real-time evaluation of *static* spatial queries on mobile objects, at a centralized location, is a well-studied topic. Most of the work done so far has focused on efficient indexing structures [25], [27], [32] for efficient evaluation of

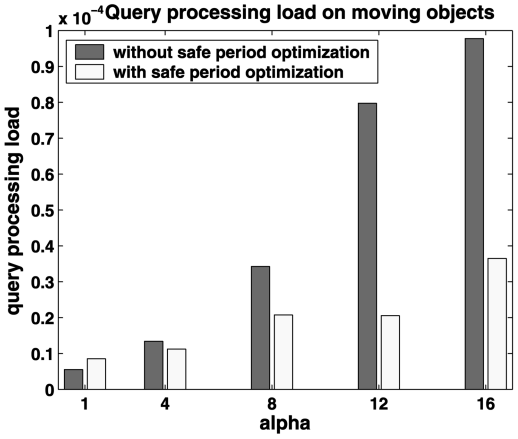

Fig. 18. Effect of the safe period optimization on the average query processing load of a mobile object.

static continual range queries at a central location, or indexing schemes and algorithms for handling mobile object positions [11], [19], [30], [18], [2], [7], [31]. Very few have considered the benefits of a careful tradeoff between computation and communication. To our knowledge, only the SQM system introduced in [11], [10] has proposed a distributed solution for evaluation of static spatial queries on mobile objects, that makes use of the computational capabilities present at the mobile objects.

Several existing research efforts on mobile object databases [35], [27], [36], [18] model mobile object trajectories as piecewise linear functions of time, and process these less frequently changing functions instead of more frequently changing object positions. This is commonly viewed as an important strategy for efficiently processing queries on mobile object positions. The design of the MobiEyes system also uses such a linear model to ease analytical derivations and engineering issues of the system. Concretely, in MobiEyes the use of velocities for predicting the positions of objects that are of interest to a mobile object, on the mobile object side, is more similar to the use of dead reckoning in online games and PDES [12] systems for building distributed virtual environments. There are very few attempts to use nonlinear motion modeling in mobile object databases [3]. A discussion on whether linear modeling assumptions can easily carry out in practice and its possible implications can be found in [35].

Safe period optimization described in Section 4 is inspired by the safe region optimization introduced in [25]. However, there are some major differences: A safe region is calculated for an object considering all queries, so that the object is guaranteed to stay outside of all query regions as long as it resides in the safe region. Also, safe regions are introduced for static range query evaluation at a centralized server. In contrast, a safe period is calculated for an object considering a single query, so that the object is guaranteed to reside outside the query region during the safe period. Compared to safe region, the safe period optimization is a more focused local optimization technique, which is computed at each mobile object that belong to a restricted subset, namely, the mobile objects that reside within the monitoring region of an MQ.

The work presented in [15] deals with the problem of monitoring changes in sensor readings and detecting mobile phenomena in sensor networks. Since the sensed phenomena may move in the environment, the set of nodes that detect this phenomena also change or "move." The queries employed in [15] are similar to queries used in our work, in the sense that the nodes in the result set may change continuously. However, we must emphasize that in MobiEyes, queries (as well as target objects in the query results) can have the property of being mobile. This is closely related with the concept of focal objects, which is missing in [15].

Recently there has been works that explicitly support efficient evaluation of moving queries over moving objects. The most notable are SINA [22] and MAI [13].

SINA [22] is a spatial query evaluation engine that makes use of the "incremental query processing" concept. The periodic reevaluations are achieved through a three phase process that refreshes the previous query results based on the current position changes using positive and negative updates, as opposed to recomputing all of the possibly invalidated results from scratch. The three phases of the query reevaluation are hashing, invalidation, and joining. The hashing phase uses a grid for indexing purposes, which is a server side data structure. This is very different than the use of grid and grid cells in our system. MobiEyes utilizes grid cells to facilitate the partitioning of moving query evaluation into a distributed coordination of server side processing and mobile object side processing. The concept of monitoring region relies on grid cells to define the set of nodes that should register a query as a nearby query and process it locally. The dynamics of the system, such as updating the set of nearby queries registered at mobile nodes, are handled with the help of the grid.

MAI [13] is a motion adaptive indexing scheme for time and IO efficient evaluation of moving queries over moving objects. MAI uses the concept of motion-sensitive bounding boxes to model moving objects and moving queries. These bounding boxes automatically adapt their sizes to the dynamic motion behaviors of individual objects. Instead of indexing frequently changing object positions, MAI indexes less frequently changing object and query motion-sensitive bounding boxes, where updates to the bounding boxes are needed only when objects and queries move across the boundaries of their boxes. MAI also uses predictive query results to optimistically precalculate query results. Motion-sensitive bounding boxes are used to incrementally update the predictive query results.

Both SINA and MAI are centralized processing based approaches, and require to receive position updates (raw positions in SINA and velocity vector changes in MAI) from all mobile nodes. This means that communication cost does not change significantly with the specific type of centralized processing engine employed. Second, since MobiEyes distributes the job of processing queries to mobile nodes of the system, the scalability in terms of server load will be drastically better than any centralized processing-based solution.

7 Conclusion

Significant research efforts have been dedicated to techniques for efficient processing of spatial queries on mobile objects in a centralized location monitoring system. Surprisingly, very few have promoted a distributed approach to real-time location monitoring. In this paper, we have presented a distributed and scalable solution to processing moving location queries on mobile objects and described the design of MobiEyes, a distributed real-time location monitoring system in a mobile environment. This paper has three unique contributions. First, we introduce the concept of moving queries on mobile objects to distinguish moving queries on mobile objects from a well-studied class of static spatial queries on mobile objects. Second, we design and develop a distributed algorithm for real-time evaluation of continuously moving queries on mobile objects, which 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. Third, we design several optimization techniques like lazy query propagation, query grouping and safe periods, to reduce the local processing load on the mobile object side and the messaging cost of the MobiEyes system. We demonstrated the effectiveness of our approach through a set of simulation-based experiments.

ACKNOWLEDGMENTS

This research was partially supported by grants from the US National Science Foundation (NSF) CSR, NSF ITR, US Department of Energy (DOE), SciDAC, IBM SUR, and HP equipment, as well as an IBM Faculty Award. Dr. Gedik's work was completed while he was with the Georgia Institute of Technology.

REFERENCES

[1] US Naval Observatory GPS Operations, http://tycho.usno.navy.mil/gps.html, Apr. 2003.
[2] P.K. Agarwal, L. Arge, and J. Erickson, "Indexing Moving Points," *Proc. ACM Symp. Principles of Database Systems (PODS)*, 2000.
[3] C.C. Aggarwal and D. Agrawal, "On Nearest Neighbor Indexing of Nonlinear Trajectories," *Proc. ACM Symp. Principles of Database Systems (PODS)*, 2003.
[4] I. Akyildiz and W. Wang, "A Dynamic Location Management Scheme for Next Generation Multi-Tier Pcs Systems," *IEEE Trans. Wireless Comm.*, vol. 1, no. 1, pp. 178-190, 2002.
[5] A. Bar-Noy, I. Kessler, and M. Sidi, "Mobile Users: To Update or Not to Update," *ACM Wireless Networks*, vol. 1, no. 2, pp. 175-185, 1995.
[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD Conf.*, 1990.
[7] R. Benetis, C.S. Jensen, G. Karciauskas, and S. Saltenis, "Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects," *Proc. Int'l Database Eng. and Applications Symp.*, 2002.
[8] F. Bennett, D. Clarke, J. Evans, A. Hopper, A. Jones, and D. Leask, "Piconet: Embedded Mobile Networking," *IEEE Personal Comm.*, vol. 4, no. 5, pp. 8-15, 1997.
[9] A. Bhattacharya and S.K. Das, "Lezi-Update: An Information-Theoretic Approach to Track Mobile Users in PCS Networks," *Proc. ACM MobiCom Conf.*, 1999.
[10] Y. Cai, K. Hua, and G. Cao, "Processing Range-Monitoring Queries on Heterogeneous Mobile Objects," *Proc. IEEE Conf. Mobile Data Management*, 2004.
[11] Y. Cai and K.A. Hua, "An Adaptive Query Management Technique for Efficient Real-Time Monitoring of Spatial Regions In Mobile Database Systems," *Proc. IEEE Int'l Performance Computing and Comm. Conf.*, 2002.
[12] R.M. Fujimoto, *Parallel and Distributed Simulation Systems.* Wiley-Interscience, 2000.
[13] B. Gedik, K.-L. Wu, P.S. Yu, and L. Liu, "Motion Adaptive Indexing for Moving Continual Queries over Moving Objects," *Proc. ACM Conf. Information and Knowledge Management*, 2004.
[14] Z.J. Has, "Panel Report on Ad Hoc Networks," *Mobile Computing and Comm. Rev.*, vol. 2, no. 1, 1998.
[15] Q. Huang, C. Lu, and G.C. Roman, "Spatiotemporal Multicast in Sensor Networks," *Proc. ACM MobiSys Conf.*, 2003.
[16] T. Imielinski, S. Viswanathan, and B. Badrinath, "Energy Efficient Indexing on Air," *Proc. ACM SIGMOD Conf.*, 1994.
[17] J. Kucera and U. Lott, "Single Chip 1.9 GHz Transceiver Frontend MMIC Including Rx/Tx Local Oscillatorrs and 300 mw Power Amplifier," *MTT Symp. Digest*, vol. 4, pp. 1405-1408, June 1999.
[18] G. Kollios, D. Gunopulos, and V.J. Tsotras, "On Indexing Mobile Objects," *Proc. ACM Symp. Principles of Database Systems*, 1999.
[19] I. Lazaridis, K. Porkaew, and S. Mehrotra, "Dynamic Queries over Mobile Objects," *Proc. Int'l Conf. Extending Database Technology (EDBT)*, 2002.
[20] L. Liu, C. Pu, and W. Tang, "Continual Queries for Internet Scale Event-Driven Information Delivery," *IEEE Trans. Knowledge and Data Eng.*, pp. 610-628, 1999.
[21] D.L. Mills, "Internet Time Synchronization: The Network Time Protocol," *IEEE Trans. Comm.*, pp. 1482-1493, 1991.
[22] M.F. Mokbel, X. Xiong, and W.G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases," *Proc. ACM SIGMOD Conf.*, 2004.
[23] Z. Naor and H. Levy, "Minimizing the Wireless Cost of Tracking Mobile Users: An Adaptive Threshold Scheme," *Proc. IEEE INFOCOM Conf.*, 1998.
[24] D. Pfoser, C.S. Jensen, and Y. Theodoridis, "Novel Approaches in Query Processing for Moving Object Trajectories," *Proc. Conf. Very Large Databases*, 2000.
[25] 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 Trans. Computers*, vol. 51, no. 10, pp. 1124-1140, Oct. 2002.
[26] QualComm, "Wireless Access Solutions Using 1xEV-DO," http://www.qualcomm.com/technology/1xev-do/webpapers/wp_wirelessaccess.pdf, 2005.
[27] S. Saltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez, "Indexing the Positions of Continuously Moving Objects," *Proc. ACM SIGMOD Conf.*, 2000.
[28] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299-319, 1990.
[29] A.P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao, "Modeling and Querying Moving Objects," *Proc. IEEE Int'l Conf. Data Eng.*, 1997.
[30] Y. Tao and D. Papadias, "Time-Parameterized Queries in Spatio-Temporal Databases," *Proc. ACM SIGMOD Conf.*, 2002.
[31] Y. Tao, D. Papadias, and Q. Shen, "Continuous Nearest Neighbor Search," *Proc. Conf. Very Large Databases*, 2002.
[32] Y. Tao, D. Papadias, and J. Sun, "The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries," *Proc. Conf. Very Large Databases*, 2003.
[33] O. Wolfson, "The Opportunities and Challenges of Location Information Management," *Proc. Intersections of Geospatial Information and Information Technology Workshop*, 2001.
[34] O. Wolfson, "Moving Objects Information Management: The Database Challenge," *Proc. Conf. Next Generation Information Technologies and Systems (NGITS)*, 2002.
[35] O. Wolfson, P. Sistla, S. Chamberlain, and Y. Yesha, "Updating and Querying Databases that Track Mobile Units," *Distributed and Parallel Databases*, vol. 7, no. 3, pp. 257-387, 1999.
[36] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, "Moving Objects Databases: Issues and Solutions," *Statistical and Scientific Database Management*, 1998.

Bugra Gedik received the BS degree from the Computer Science Department of Bilkent University, Turkey, and the PhD degree from the College of Computing at the Georgia Institute of Technology. He is currently a research staff member at the IBM T.J. Watson Research Center. His research interests are in data intensive distributed computing systems, including peer-to-peer computing, mobile and sensor-based data management, and data stream processing. He is a member of the IEEE.

Ling Liu is an associate professor in the College of Computing at the Georgia Institute of Technology. There, she directs the research programs in the Distributed Data Intensive Systems Lab (DiSL), examining research issues and technical challenges in building large scale distributed computing systems that can grow without limits. Dr. Liu and the DiSL research group have been working on various aspects of distributed data intensive systems, ranging from decentralized overlay networks, exemplified by peer to peer computing, and data grid computing to mobile computing systems and location based services, sensor network computing, and enterprise computing systems. She has published more than 150 international journal and conference articles. Her research group has produced a number of software systems that are either open source or directly accessible online, among which the most popular ones are WebCQ and XWRAPElite. Most of her current research projects are sponsored by the US National Science Foundation, DoE, DARPA, IBM, and HP. She is on the editorial board of several international journals, such as *IEEE Transactions on Knowledge and Data Engineering*, *International Journal of Very Large Database Systems* (VLDBJ), and the *International Journal of Web Services Research*. She has chaired a number of conferences as a PC chair, a vice PC chair, or a general chair, including the IEEE International Conference on Data Engineering (ICDE 2004, ICDE 2006, ICDE 2007), the IEEE International Conference on Distributed Computing (ICDCS 2006), and the IEEE International Conference on Web Services (ICWS 2004). She is a senior member of the IEEE.