Processing Continuously Moving Queries on Moving Objects: The MobiEyes Approach *

Buğra Gedik and Ling Liu

College of Computing

Georgia Institute of Technology

{bgedik,lingliu}@cc.gatech.edu

Abstract

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

Index Terms: Moving Object Databases, Spatial Queries over Moving Objects, Distributed Algorithms, Mobile Data Management.

*An extended abstract of this paper appeared in Proceedings of the International Conference on Extending Database Technology, March 2004, Crete, Greece.

1 Introduction

With the growing market of positioning technologies like GPS [gps03] and the growing popularity and availability of mobile communications, location information management has become an important problem [TP02, LPM02, PXK+02, CH02, SJLL00, PJT00, SWCD97, KGT99, AAE00]. With continued upsurge of computational capabilities in mobile devices, ranging from navigational systems in cars to hand-held devices and cell phones, mobile devices are becoming increasingly accessible. We expect that future mobile applications will require a scalable architecture that is capable of handling large and rapidly growing number of mobile objects and processing complex queries over mobile object positions.

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

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

1.1 Moving Queries on Moving Objects

A moving query on moving objects[1] (MQ for short) is a *spatial continuous moving query over locations of moving objects*. An MQ defines a spatial region bound to a specific moving object and a filter which is a boolean predicate on object properties. The result of an MQ consists of objects that are inside the area covered by the query's spatial region and satisfy the query filter.

MQs are continuous queries [LPT99] in the sense that the results of queries continuously change as

[1]The term "moving object" in this paper refers to "mobile object".

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 moving objects in real life. For instance, the query MQ_1: "Give me the number of friendly units within 5 miles radius around me during next 2 hours" can be submitted by a soldier equipped with mobile devices marching in the field, or a moving tank in a military setting. The query MQ_2: "Give me the positions of those customers who are looking for taxi and are within 5 miles (of my location at each instance of time or at an interval of every minute) during the next 20 minutes" can be posted by a taxi driver marching on the road. The focal object of MQ_1 is the solider marching in the field or a moving tank. The focal object of MQ_2 is the taxi driver on the road.

Different specializations of MQs can result in interesting and useful classes of queries on moving objects. One such specialization is the case where the target objects are still 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 moving 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". We can view the class of static spatial queries on static objects as the extreme case of moving queries on moving objects. An example of such queries is MQ_5: "Show me the list of French restaurants within 10 miles or 20 minutes of driving distance from my office building". Spatial queries of this class are not bounded with motion or temporal constraints and thus are not continuous queries. A formal definition of moving queries on moving objects (MQs) is provided in Section 2.

1.2 MobiEyes: Distributed Processing of MQs

Most of the existing approaches for processing spatial queries on moving objects are not scalable, due to their inherent assumption that location tracking and communications of mobile objects are performed and controlled by a central server. Namely, mobile objects report their position changes to the server whenever their position information changes, and the server determines which moving objects should be included in which moving queries at each instance of time or at a given time interval. For mobile applications that need to handle a large and growing number of moving objects, the centralized approaches can suffer from dramatic performance degradation in terms of server load and network bandwidth. We characterize the amount of communication required for mobile objects to deliver location updates to the server (uplink) and for central sever to disseminate important location changes to mobile objects (downlink) by the number of uplink and downlink messages consumed.

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

- *Shipping some of the MQ processing to the moving object side can be seen as a mechanism for moving computation close to places where the data is produced. Such techniques are especially beneficial when there are a large number of moving objects, generating immense number of position updates.*
- *The computational capabilities of mobile objects can be utilized in a distributed solution, which not only decreases the load on the server and the communication cost between the mobile objects and the server, but also increases the scalability of the system.*
- *The distributed solution allows us to restrict the area that covers the objects that need to be aware of a query's state (position change of the query's focal object), since it is only the objects in close proximity to the query that need to be aware of this information. The communication asymmetry*

inherent in mobile communications makes it possible to efficiently convey information regarding a query's state to appropriate set of moving objects.

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

The rest of the paper is structured as follows. In Section 2 we describe the system model, the assumptions underlying our solution, and the basic notations used in MobiEyes. In Section 3 we describe the basics of the distributed solution used in MobiEyes for efficient real-time evaluation of MQs in a mobile setup. In Section 4 we introduce several optimizations targeted to enhance the effectiveness of the MobiEyes system with respect to scalability and performance. We study the resilience of the MobiEyes system against failures of the system components in Section 5. Section 6 reports our experimental evaluation of the scalability of our solution with respect to the number of moving objects and the number of moving queries handled in a mobile system, showing the advantages of MobiEyes in terms of server load and messaging cost. We provide discussions on a selection of other important issues, such as collaboration, security, privacy, MQs with dynamic filters in Section 7. The paper concludes with a review of the related work in Section 8 and a summary and future work in Section 9.

2 System Model

This section introduces the MobiEyes system model, including the set of assumptions used in MobiEyes, the moving object model, and the moving query model. We describe the distributed algorithms for efficient processing of MQs in the next section.

2.1 System Assumptions

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

– *Moving objects are able to locate their positions*: We assume that each moving object is equipped with a technology like GPS [gps03] to locate its position. This is a reasonable assumption as GPS devices are becoming inexpensive and are used widely in cars and other hand-held devices to provide navigational support.

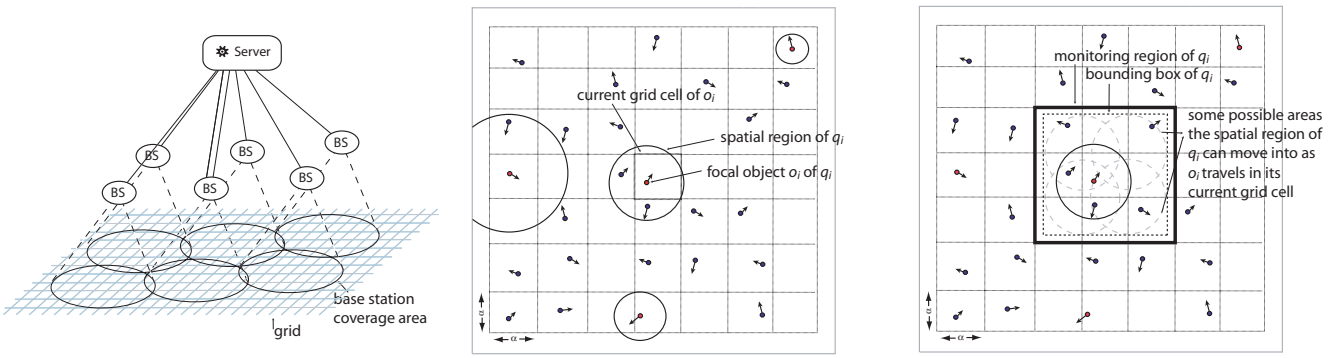
– *Moving objects have synchronized clocks*: Again this assumption can be met if the moving objects are equipped with GPS. Another solution is to make NTP [Mil91] (network time protocol) available to moving objects through base stations.

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

– *Moving objects have computational capabilities to carry out computational tasks*: This assumption represents a fast growing trend in mobile and wireless technology. The number of mobile devices equiped with computational power escalates rapidly, even simple sensors [HSW+00] today are equipped with computational capabilities.

2.2 The Moving Object Model

The current version of the MobiEyes system assumes that the geographical area of interest is covered by several base stations, which are connected to a central server. A three-tier architecture (mobile objects,



(a) Base station related concepts (b) Moving object concepts (c) MQ related concepts

Figure 1: Illustration of concepts

base stations and the server) is used in the subsequent discussions. We can easily extend the three tier to a multi-tier communication hierarchy between the mobile objects and the server. In addition, the asymmetric communication is used to establish connections from the server to the moving objects. Concretely, a base station can communicate directly with the moving objects in its coverage area through a broadcast, and the moving objects can only communicate with the base station if they are located in the coverage area of this base station.

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

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

- *Rectangle shaped region and circle shaped region* : A rectangle shaped region is defined by $Rect(lx, ly, w, h) = \{(x, y) : x \in [lx, lx + w] \ \wedge \ y \in [ly, ly + h]\}$, 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. X, Y, W and H are system level parameters to be set at the system initialization time.
- *Grid and Grid cells*: In MobiEyes, we map the universe of discourse U onto a grid G of cells, where $U = Rect(X, Y, W, H)$, each grid cell is an $\alpha \times \alpha$ square area, and α is a system parameter that defines the cell size of the grid G. Formally, a grid corresponding to the universe of discourse U can be defined as $G(U, \alpha) = \{A_{i,j} : 1 \leq i \leq M, 1 \leq j \leq N, A_{i,j} = Rect(X + i * \alpha, Y + j * \alpha, \alpha, \alpha), M = \lceil H/\alpha \rceil, N = \lceil W/\alpha \rceil\}$. $A_{i,j}$ is an $\alpha \times \alpha$ square area representing the grid cell that is located on the ith row and jth column of the grid G.
- *Position to Grid Cell Mapping*: Let $pos = (x, y)$ be the position of a moving object in the universe of discourse $U = Rect(X, Y, W, H)$. Let $A_{i,j}$ denote a cell in the grid $G(U, \alpha)$. $Pmap(pos)$ is a position to grid cell mapping, defined as $Pmap(pos) = A_{\lceil \frac{pos.x - X}{\alpha} \rceil, \lceil \frac{pos.y - Y}{\alpha} \rceil}$.
- *Current Grid Cell of an Object*: Current grid cell of a moving object is the grid cell which contains the current position of the moving object. If $o \in O$ is an object whose current position, denoted as $o.pos$, is in the Universe of Discourse U, then the current grid cell of the object is formally defined by $curr\_cell(o) = Pmap(o.pos)$.
- *Base Stations*: Let $U = Rect(X, Y, W, H)$ be the universe of discourse and B be the set of base stations overlapping with U. Assume that each base station $b \in B$ is defined by a circle region $Circle(bsx, bsy, bsr)$ 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 non-empty set of base stations. We define $Bmap(i, j) = \{b : b \in B \wedge b \cap A_{i,j} \neq \emptyset\}$. $Bmap(i, j)$ is the set of base stations that cover the grid cell $A_{i,j}$.

See Figure 1(a) and Figure 1(b) 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 moving objects that locate within the area covered by the spatial region of the query and satisfy the filter condition.

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

- *Bounding Box of a Moving Query*

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

- *Monitoring Region of a Moving Query*

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

- *Nearby Queries of an Object*

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

See Figure 1(c) for example illustrations.

3 Distributed Processing of Moving Queries

A main idea underlying the MobiEyes approach is to provide mechanisms such that each mobile object can determine by itself whether or not it should be included in the result of a moving query close by, without requiring global knowledge regarding the moving queries and the object positions. The distributed algorithms for processing moving queries in MobiEyes have three unique features. First, we define *the concept of monitoring region* for each moving query, which is used to model the part of grid area to which the spatial query region is bounded when the focal object of the query changes its position within its current grid cell. Second, we provide *concrete data structure and resource-aware distributed coordination mechanisms* for managing the communication and collaboration between the server and the moving objects, focusing on reducing the messaging cost thus the network bandwidth and power consumption at moving objects, and the server load. Our algorithms are especially effective when the number of moving objects in the geographical region considered is large and the number of MQs, relatively speaking, is small and skewed in terms of query distribution over the moving objects considered. Third, we propose a suite of *optimization techniques* for efficient processing of MQs, taking into ac-

count of the resource constraints at both the moving objects and the server. For example, we propose the lazy query propagation technique to reduce the messaging cost of the communication between the moving objects and the server. We introduce the MQ grouping techniques, which allow MobiEyes to handle hot spot queries efficiently in terms of workload control on the moving objects that are within or near by hot-spot MQs. We define the concept of safe period (inspired by the safe region technique used in [PXK+02]), which allows us to further reduce the amount of local query processing at the moving object side. In addition, similar to many centralized proposals for query processing on moving objects, we also use the dead-reckoning technique to estimate the position changes of moving objects of interest.

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

3.1 Algorithm Overview

In MobiEyes, distributed processing of moving queries consists of server side processing and mobile object side processing. We below provide a brief review of the main components and key ideas used in MobiEyes for distributed processing of MQs. A detailed technical discussion on each of these ideas is presented in the subsequent sections.

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

Monitoring Region of a MQ: To enable efficient processing at mobile object side, we introduce the monitoring region of a moving query to identify all moving objects that may get included in the query's result when the focal object of the query moves within its current cell. The main idea is to have those moving objects that reside in a moving query's monitoring region to be aware of the query and to be responsible for calculating if they should be included in the query result. Thus, the moving objects that

are not in the neighborhood of a moving query do not need to be aware of the existence of the moving query, and the query result can be efficiently maintained by the objects in the query's monitoring region in a differential manner.

Registering MQs at Moving Object Side: In MobiEyes, the task of making sure that the moving objects in a query's monitoring region are aware of the query is accomplished through server broadcasts, which are triggered by either installations of new moving queries or notifications of changes in the monitoring regions of existing moving queries when their focal objects change their current grid cells. Upon receiving a broadcast message, for each moving query in the message, the mobile objects examine their local state and determine whether they should be responsible for processing this moving query. This decision is primarily based on whether the mobile objects themselves are within the monitoring region of the query (see Section 3.3 for further detail).

Moving Object Side Processing: Once a moving query is registered at the moving object side, the moving object will be responsible for periodically tracking if it is within the spatial region of the query as this object moves, by predicting the position of the focal object of the query. Then the moving objects decide whether they should be included in the query's result or not through a simple containment check based on the query's spatial region. Changes in the containment status of moving objects with respect to moving queries are differentially relayed to the server (see Section 3.6 for further detail).

Handling Significant Position Changes: In case the position of the focal object of a moving query changes significantly (it moves out of its current grid cell or changes its velocity vector significantly), it will report to the server, and the server will relay such important position change information to the appropriate subset of moving objects through its broadcast mechanism. Technical details such as how the focal object determines significant position changes, how the server determines which subset of moving objects should be notified of position changes of a given focal object, and what information needs to be relayed will be discussed in the subsequent sections (see Section 3.4 and Section 3.5 for detailed technical discussion).

An immediate advantage of the MobiEyes distributed approach is the significant saving in terms of

server load and communication bandwidth, which result in energy-savings on the moving object side. In MobiEyes, non-focal objects do not need to report their positions or velocities to the server. Focal objects only report their positions and velocities to the server when their velocity vectors change significantly or when they move out of their current grid cells. In addition, objects notify the server in order to differentially update the query results, whenever a change is detected during their local query processing.

3.2 Data Structures

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

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

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

Reverse Query Index, RQI, is an $M \times N$ matrix whose cells are a set of query identifiers. M and N denote the number of rows and the number of columns of the Grid corresponding to the Universe

of Discourse of a MobiEyes system. $RQI(i,j)$ stores the identifiers of the queries whose monitoring regions intersect with the grid cell $A_{i,j}$. $RQI(i,j)$ represents the nearby queries of an object whose current grid cell is $A_{i,j}$, i.e. $\forall o \in O, nearby\_queries(o) = RQI(i,j)$, where $curr\_cell(o) = A_{i,j}$. Formally it is defined as follows: $RQI(i,j) = \{qid : \exists e \in SQT$ s.t. $e.qid = qid \wedge e.mon\_region \cap A_{i,j} \neq \emptyset\}$.

Grid-cell to Base-station Mapping, $Bmap$, is also an $M \times N$ matrix whose cells are a set of base stations. 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. $Bmap(i,j)$ stores the set of base stations whose coverage areas intersect with the grid cell $A_{i,j}$.

3.2.2 Moving Object-side Data Structures

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

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

3.3 Basic Operations in MobiEyes

Having described the main data structures to be used in our algorithms, in this section we describe in detail how moving queries are installed in the system, how the system handles objects that change their velocity vectors or their grid cells, and the moving query processing logic. These algorithms form the key components of the MobiEyes distributed solution. We also provide pseudo codes for these algorithms.

3.3.1 Installing Queries

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

Updating the Server State

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

There is a small complication involved in the action (3). If the focal object of the query is not present in the FOT table, and the query 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.

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 moving 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*, [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*, [Install query $(qid, pos, \overline{vel}, tm, region, filter)$])
19: **end for**

(B)**Moving Object: Received Message [Query installed]**

1: $hasMQ \leftarrow true$

(C)**Moving 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**

A simple solution is to use an additional table stored on the server side, which stores base station level [AW02, BD99, BNKS95, NL98], or higher level location information regarding moving objects. For instance, in case we store base station level location information regarding moving objects, given an object identifier it is possible to locate the base station which covers its current location. During query installation process, 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.

Installing Queries on the Moving Objects

After installing queries on the server side, the server needs to complete the installation by triggering query installation on the moving object side. This job is done by performing two tasks. First, the server sends an installation notification to the focal object with identifier oid, which upon receiving the

notification sets its $hasMQ$ variable to true. This makes sure that the moving object knows that it is now a focal object and is supposed to report velocity vector changes to the server. The second task is for the server to forward this query to all objects that reside in the query's monitoring region, so that they can install the query and monitor their position changes to determine if they become the target objects of this query. To perform this task, the server uses the mapping $Bmap$ to determine the minimal set of base stations (i.e., the smallest number of base stations) that covers the monitoring region. Then the query is sent to all objects that are covered by the base stations in this set through broadcast messages. When an object receives the broadcast message, it checks whether its current grid cell is covered by the query's monitoring region. If so, the object installs the query into its local query table LQT when the query's filter is also satisfied by the object. Otherwise the object discards the message. A sketch of the detailed algorithm is given in Algorithm 1.

3.3.2 Handling Velocity Vector Changes

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

A velocity vector change, once identified as significant, will need to be relayed to the objects that reside in the query's monitoring region through the server acting as a mediator. When the focal object of a query reports a velocity vector change, it sends its new velocity vector, its position and the timestamp at which this information was recorded, to the server. The server first updates the FOT table with the information received from the focal object. Then for each query associated with the focal object, the server communicates the newly received information to objects located in the monitoring region of the query by using minimum number of broadcasts (this can be done through the use of the grid cell to base station mapping $Bmap$). An illustration of this process is given in Figure 2, where a focal object together with its monitoring region is shown. The focal object sends its new velocity information to the

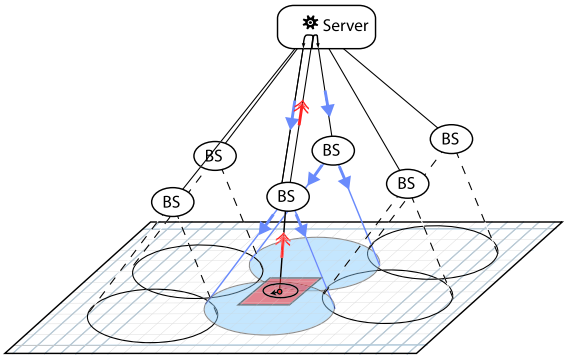


Figure 2: Conveying velocity vector changes

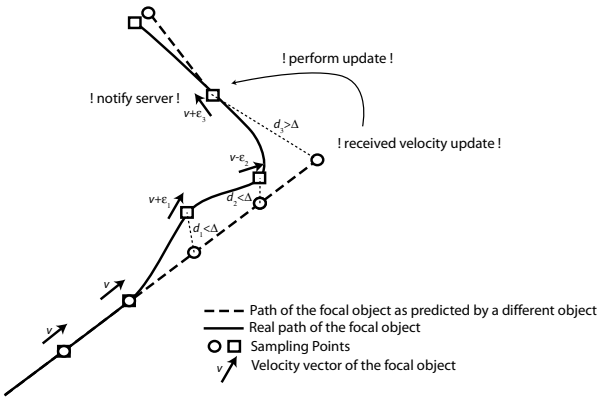


Figure 3: Dead Reckoning in MQ evaluation

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 the figure).

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

Dead Reckoning in MobiEyes

Concretely, at each time step the focal object of a query samples its current position and calculates the difference between its current position and the position that the other objects believe it to be at. In case this difference is larger than a threshold, say Δ, the new velocity vector information is relayed [2]. Figure 3 provides an illustration. In the figure, 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

[2]We do not consider the inaccuracy introduced by the motion modelling. See [WSCY99] for a discussion of motion update policies and their tradeoffs.

Algorithm 2: Moving Object Changed Velocity Vector

(A) **Moving 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: $Send(server,$ [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: $Broadcast(b,$ [Query velocity change $(e.qid, pos, \overline{vel}, tm)$])
7: **end for**
8: **end for**

(C) **Moving 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**

larger than Δ, the focal object relays its new velocity information to all objects in its monitoring region by sending an update to the server, which happens in the fifth time step in the figure. Algorithm 2 describes how velocity vector changes are handled in more detail. One way to optimize this update propagation is to let the server *batch several velocity vector updates from moving 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. This optimization is not reflected in Algorithm 2 for brevity.

3.3.3 Handling Objects that Change their Grid Cells

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

When an object changes its current grid cell, it notifies the server of this change by sending its object

Algorithm 3: Moving Object Changed its Current Grid Cell

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

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

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

$Q_{diff} \leftarrow RQI(i_c, j_c) \setminus RQI(i_p, j_p)$
for all $qid \in Q_{diff}$ **do**
  Let $e = (qid, *, *, *, *, *, *, *)$ in SQT.
  $Send(oid,$ [Install query $(e.qid, e.pos, e.\overline{vel}, e.tm, e.region, e.filter)$])
end for
Let $e = (oid, *, *, *)$ in FOT
for all $e_s = (*, oid, *, *, *, *, *, *)$ in SQT **do**
  $e_s.curr\_cell \leftarrow A_{i_c,j_c}$
  $old\_mon\_region \leftarrow e_s.mon\_region$
  Set $e_s.mon\_region$ using $e_s.region$ and $e_s.curr\_cell$
  for all $A_{i,j} \subset (old\_mon\_region \setminus e_s.mon\_region)$ **do**
    $RQI(i, j) \leftarrow RQI(i, j) \setminus \{e_s.qid\}$
  end for
  for all $A_{i,j} \subset (e_s.mon\_region \setminus old\_mon\_region)$ **do**
    $RQI(i, j) \leftarrow RQI(i, j) \cup \{e_s.qid\}$
  end for
  $combined\_area \leftarrow e_s.mon\_region \cup old\_mon\_region$
  $B' \leftarrow \bigcup_{A_{i,j} \subset combined\_area} Bmap(i, j)$
  for all $b \in B'$ **do**
    $Broadcast(b,$ [Update query $(e_s.qid, e.pos, e.\overline{vel}, e.tm, e_s.region, e_s.filter)$])
  end for
end for

(C) **MovingObject:ReceivedMessage[Update.query**$(qid, pos, \overline{vel}, tm, region, filter)$**]**

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

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

grid cell of the object to determine the set of new queries that has to be installed on this moving object. Then the server sends the set of new queries to the moving object for installation. The focal object table FOT and the server query table SQT are used to create required installation information of the queries to be installed on the object. However, if the object that changes its current grid cell is a focal object of some query, additional set of operations are performed. For each query with this object as its focal object, the server performs the following operations. It updates the query's SQT table entry by resetting the current grid cell and the monitoring region to their new values. It also updates the RQI index to reflect the change. Then the server computes the union of the query's previous monitoring region and its new monitoring region, and sends a broadcast message to all objects that reside in this combined area. This message includes information about the new state of the query. Upon receipt of this message from the server, an object performs the following operations for installing/removing a query. It checks whether its current grid cell is covered by the query's monitoring region. If not, the object removes the query from its LQT table (if the entry already exists), since the object's position is no longer covered by the query's monitoring region. Otherwise, it installs the query if the query is not already installed and the query filter is satisfied, by adding a new query entry in the LQT table. In case that the query is already installed in LQT, it updates the monitoring region of the query's entry in LQT. A sketch of the detailed algorithm is given by Algorithm 3.

3.3.4 Moving Object Query Processing Logic

A moving object periodically processes all queries registered in its LQT table. For each query, it predicts the position of the focal object of the query using the velocity, time, and position information available in the LQT entry of the query (line 3 under (A) in Algorithm 4). Then it compares its current position and the predicted position of the query's focal object to determine whether itself is covered by the query's spatial region or not. When the result is different from the last result computed in the previous time step, the object notifies the server of this change, which in turn differentially updates the query result. Algorithm 4 provides more details.

Algorithm 4: Moving Object Query Processing Logic

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

Let pos be the current position of the object and ctm be the current time.
for all e in LQT **do**
  {predict the position of the focal object}
  $fpos \leftarrow e.pos + (ctm - e.tm) * e.vel$
  if $(pos - fpos) \in e.region$ **then**
    if $e.isTarget = false$ **then**
      $Send(server,$ [Query result change $(e.qid, oid, true)$])
      $e.isTarget \leftarrow true.$
    end if
  else if $e.isTarget = true$ **then**
    $Send(server,$ [Query result change $(e.qid, oid, false)$])
    $e.isTarget \leftarrow false.$
  end if
end for

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

Let $e = (qid, *, *, *, *, *, *, *)$ in SQT.
if $isTarget = true$ **then**
  $e.result \leftarrow e.result \cup \{oid\}$
else
  $e.result \leftarrow e.result \setminus \{oid\}$
end if

4 Optimizations

We have presented the basic algorithms for distributed processing of moving queries on moving objects in MobiEyes. In this section we present three additional optimization techniques aiming at restraining the amount of local processing at the mobile object side, and further reducing the communication cost between the mobile objects and the server.

4.1 Lazy Query Propagation

The procedure presented in Section 3.3.3 for handling objects changing their current grid cells uses an eager query propagation approach. It requires each object (focal or non-focal) to contact the server and transfer information whenever it changes its current grid cell. The only reason for a non-focal object to communicate with the server is to immediately obtain the list of new queries that it needs to install in response to changing its current grid cell. We refer to this scheme as the *Eager Query Propagation* (EQP).

To reduce the amount of communication between moving objects and the server, in MobiEyes we also provide a lazy query propagation approach. Thus, the need for non-focal objects to contact the server to

obtain the list of new MQs can be eliminated. Instead of obtaining the new queries from the server and installing them immediately on the object upon a grid cell change, the moving object can wait until the server broadcasts the next velocity vector changes regarding the focal objects of these queries, to the area in which the object locates. In this case the velocity vector change notifications are expanded to include the spatial region and the filter of the queries, so that the object can install the new queries upon receiving the broadcast message on the velocity vector changes of the focal objects of the moving queries. Using lazy propagation, the moving objects upon changing their current grid cells will be unaware of the new set of queries nearby until the focal objects of these queries change their velocity vectors or move out of their current grid cells. Obviously lazy propagation works well when the gird cell size α is large and the focal objects of queries change their velocity vectors frequently. The lazy query propagation may not prevail over the eager query propagation, when: (1) the focal objects do not have significant change on their velocity vectors, (2) the grid cell size α is too small, and (3) the non-focal moving objects change their current grid cells at a much faster rate than the focal objects. In such situations, non-focal objects may end up missing some moving queries. We evaluate the *Lazy Query Propagation (LQP)* approach and study its performance advantages as well as its impact on the query result accuracy in Section 6.

4.2 Query Grouping

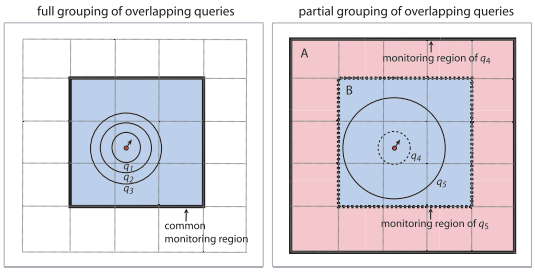


Figure 4: Grouping moving queries

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

We define a set of moving queries as *groupable MQs* if they are bounded to the same focal object.

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

Grouping MQs with Non-Matching Monitoring Regions

A clean way to handle MQs with non-matching monitoring regions is to perform grouping on the moving object side only. We illustrate this with an example. Consider an object o_j inside region B in Figure 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 combining these two queries inside its LQT

table. By this way it only needs to consider the query with smaller radius only if it finds out that its current position is inside the spatial region of the one with the larger radius.

4.3 Safe Period Optimization

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

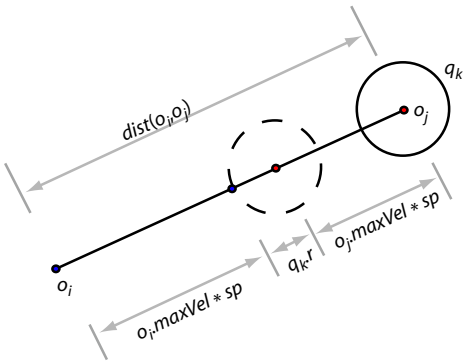


Figure 5: Safe period optimization

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 Figure 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 moving object is calculated for a query, it is safe for the object to start the periodic evaluation of this query after the safe period has passed. In order to integrate this optimization with the base algorithm, we include a *processing time* (ptm) field into the LQT table, which is initialized to 0. When a query in LQT is to be processed, ptm is checked first. In case ptm is ahead of the current time ctm, the query is skipped. Otherwise, it is processed as

usual. After processing of the query, if the object is found to be outside the area covered by the query's spatial region, the safe period sp is calculated for the query and processing time ptm of the query is set to current time plus the safe period, ctm+sp. When the query evaluation period is short, or the object speeds are low or the cell size α of the grid is large, this optimization can be very effective.

5 Reliability

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 moving 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.

5.1 Reliability of the moving objects

In MobiEyes, each moving 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 3.2). The LQT table is the most important state information maintained at the moving object. It keeps all the MQs that this moving object needs to process. In the event of failure, such as the computational unit on a moving 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 re-initialization 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 mainly depends upon whether the moving object experiencing the failure is a non-focal object or a focal object.

Failure at a non-focal moving object:

When failure happens at a moving object that is a non-focal object, we may have the moving 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 non-focal object: First, the non-focal 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 non-focal object. Second, the focal objects of some MQs in the LQT table of this non-focal 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 non-focal object by the server through broadcast, but this broadcast will not be received and processed at the non-focal object due to failure. In both cases, the location changes of the non-focal 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 non-focal object no longer contain its position; and (2) The current grid cell of the non-focal 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 non-focal object detects that the spatial region of a MQ in its LQT table no longer contains its position, the non-focal 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 non-focal object detects that the monitoring region of a 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 non-focal object.

However, due to the failure happened at the non-focal object, its LQT table can only be recovered to the new state through re-initialization with the server upon the restart of the computing unit, based on the reverse query index of the non-focal object's current grid cell. The new LQT table may not contain the MQs whose results, before the crash, included the non-focal 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 non-focal object, for two reasons: (i) The non-focal object did not send the query result changes to the server due to failure; and (ii) 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 moving 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 moving 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 re-initialization 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 moving 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 moving object:

When failure happens at a moving 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 moving 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 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 moving objects whose LQT tables contain MQs associated with the focal object. As a result, the moving 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 moving 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 non-focal 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 moving object sends query result updates to the server only when it is included into or excluded from the result of a MQ in its LQT table. With forced result updates, we additionally require that each moving object reports its state on whether it is included in or excluded from the result of a 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.

- *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.

5.2 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 [Sch90]. In the absence of failover servers, a straight forward approach to handle a server crash is to re-initialize the whole system.

The concrete analysis of the MobiEyes reliability solution through replication and forced updates is beyond the focus of this paper and will be reported separately.

6 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 moving object has to perform, by measuring on average the number of queries a moving object needs to process during each local evaluation period.

6.1 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 100,000 square miles. The number of objects we consider ranges from 1,000 to 10,000 where the number of queries range from 100 to 1,000.

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

Table 1: Simulation Parameters

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 [PXK+02].

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% 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 simulation has a time step parameter of 30 seconds. In every time step we pick a number of objects at random and set their normalized velocity vectors to a random direction, while setting their velocity to a random value between zero and their maximum velocity. All other objects are assumed to continue their motion with their unchanged velocity vectors. The number of objects that change velocity vectors during each time step is a parameter whose value ranges from 100 to 1,000.

6.2 Server Load

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

Indexing Objects. The first centralized approach to processing spatial continuous queries on moving objects is by indexing objects. In this approach a spatial index is built over object locations. We use an R*-tree [BKSS90] 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 when compared to the number of objects.

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

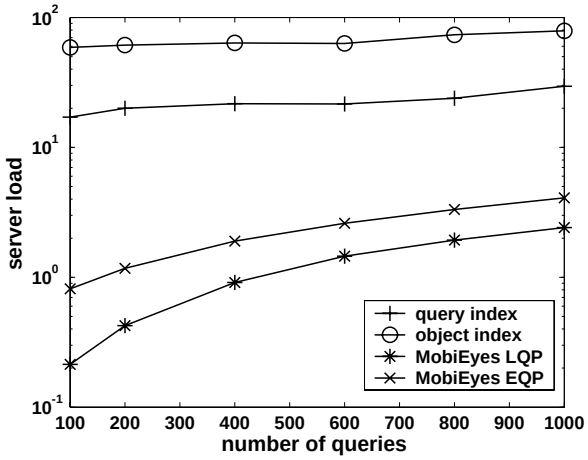


Figure 6: Impact of distributed query processing on server load

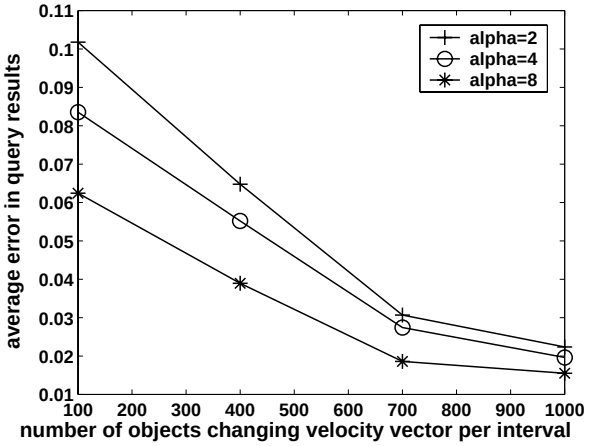


Figure 7: Error associated with lazy query propagation

side logic per time step. Figure 6 and Figure 8 depict the results obtained. Note that the y-axises, which represent the sever load, are in log-scale. The x-axis represents the number of queries considered in Figure 6, and the different settings of α parameter in Figure 8.

It is observed from Figure 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.

Figure 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 3.3.3, lazy query propagation may have some inaccuracy associated with it. Figure 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

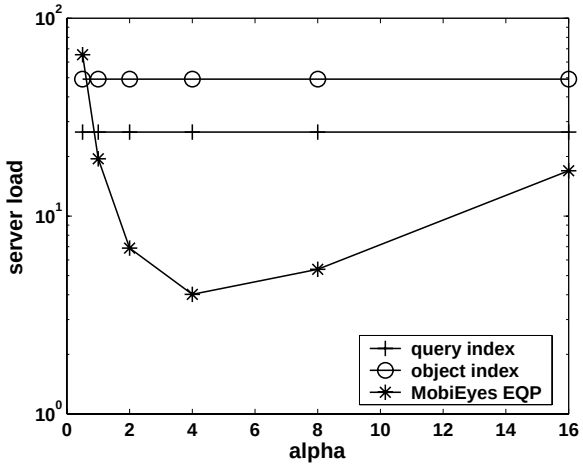


Figure 8: Effect of α on server load

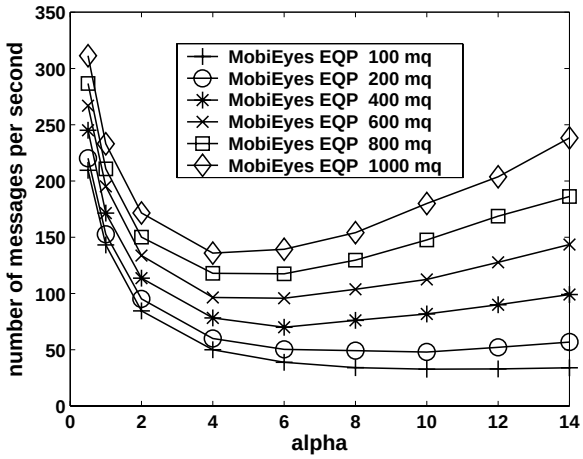


Figure 9: Effect of α on messaging cost

correct query result. Figure 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 Figure 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 Figure 7 as it shows that the error in query results increases with decreasing α.

Figure 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 further investigate selecting a good value for α in the next section.

6.3 Messaging Cost

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

One crucial concern is defining an optimal value for the parameter α, which is the length of a grid cell. The graph in Figure 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. Figure 9 shows that values in the range [4,6] are ideal for α with respect to the number of queries ranging from 100 to 1000. The optimal value of the α parameter can be derived analytically using a simple model. In this paper we omit the analytical model for space restrictions.

Figure 10 studies the effect of number of objects on the messaging cost. It plots the number of messages per second as a function of number of objects for different numbers of queries. While the number of objects is altered, the ratio of the number of objects changing their velocity vectors per time

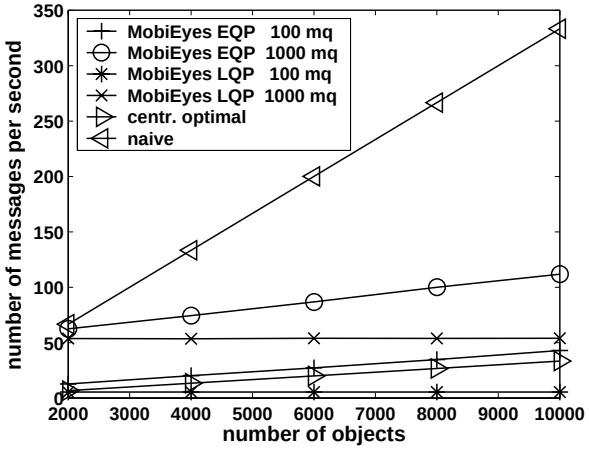


Figure 10: Effect of number of objects on messaging cost

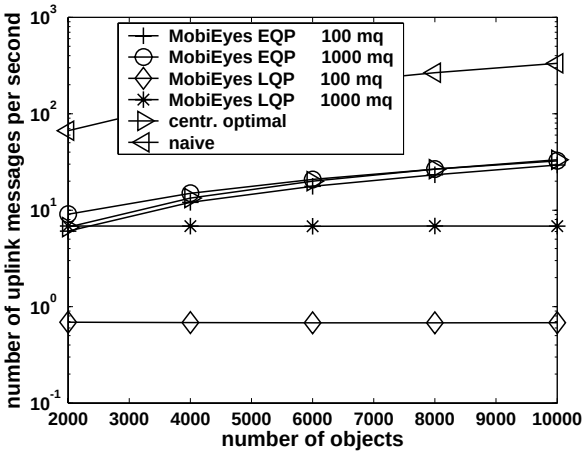


Figure 11: Effect of number of objects on uplink messaging cost

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. However, the naïve approach has a high cost when the ratio of the number of objects to the number of queries is high. In the latter case, central optimal approach provides lower messaging cost, when compared to MobiEyes with EQP, but the gap between the two stays constant as number of objects are increased. On the other hand, MobiEyes with LQP scales better than all other approaches with increasing number of objects and shows improvement over central optimal approach for smaller number of queries. Figure 11 shows the uplink component of the messaging cost. The y-axis is plotted in logarithmic scale for convenience of the comparison. Figure 11 clearly shows that MobiEyes with LQP significantly cuts down the uplink messaging requirement, which is crucial for asymmetric communication environments where uplink communication bandwidth is considerably lower than downlink communication bandwidth.

Figure 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 Figure 12 is that the messaging cost of MobiEyes with EQP scales well when compared to the central optimal approach as the gap between the two tends to decrease as the number of objects changing velocity vector

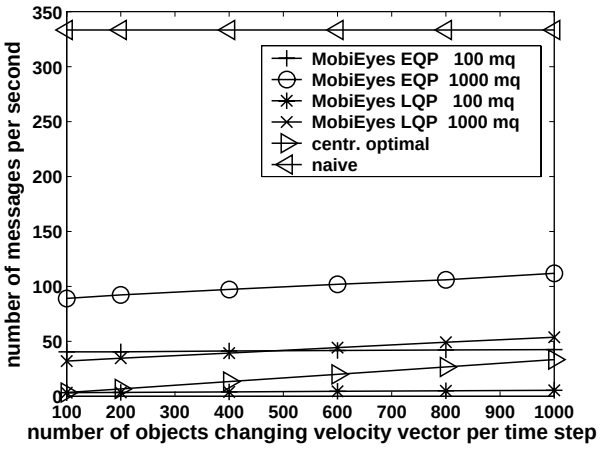


Figure 12: Effect of number of objects changing velocity vector per time step on messaging cost

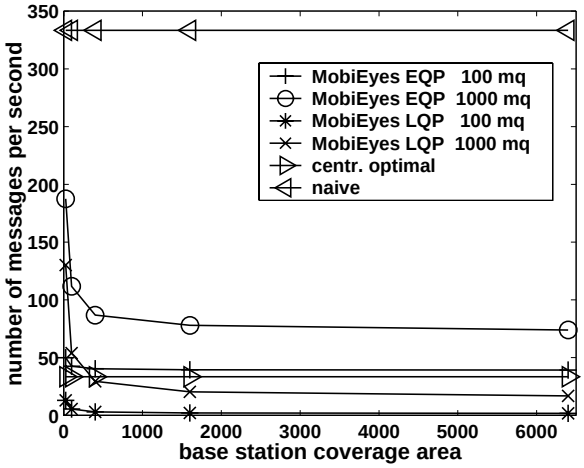


Figure 13: Effect of base station coverage area on messaging cost

per time step increases. Again MobiEyes with LQP scales better than all other approaches and shows improvement over central optimal approach for smaller number of queries.

Figure 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 Figure 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 base stations reach to a certain size, the monitoring regions associated with queries always lie in only one base station's coverage area. Although increasing base station size decreases the total number of messages sent on the wireless medium, it will increase the average number of messages received by a moving object due to the size difference between monitoring regions and base station coverage areas. In a hypothetical case where the universe of disclosure is covered by a single base station, any server broadcast will be received by any moving object. In such environments, indexing on the air [IVB94] can be used as an effective mechanism to deal with this problem. In this paper we do not consider such extreme scenarios.

6.3.1 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 Figure 8 and Figure 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.

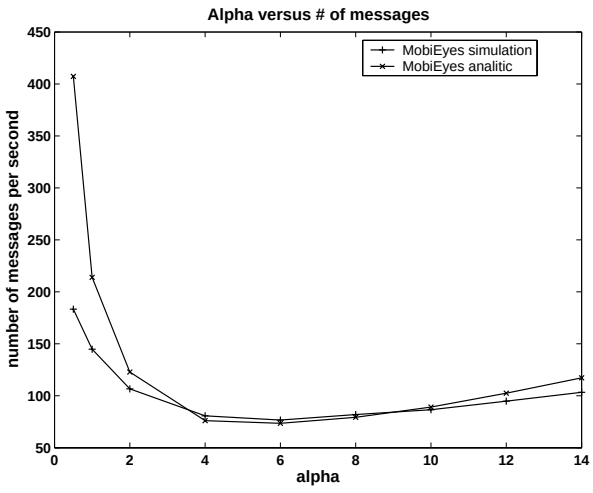


Figure 14: Comparison of the analytical messaging cost estimate with simulation results

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.

Figure 14 compares the analytical estimate on the number of messages with the results obtained from the simulation 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 num-

ber 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.

6.3.2 Per Object Power Consumption due to Communication

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.

In this section 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 [JL99], we take the power consumption of transmitter and receiver electronics as 150mW and 120mW respectively and we assume a 300mW transmit amplifier with 30% efficiency [JL99]. 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]

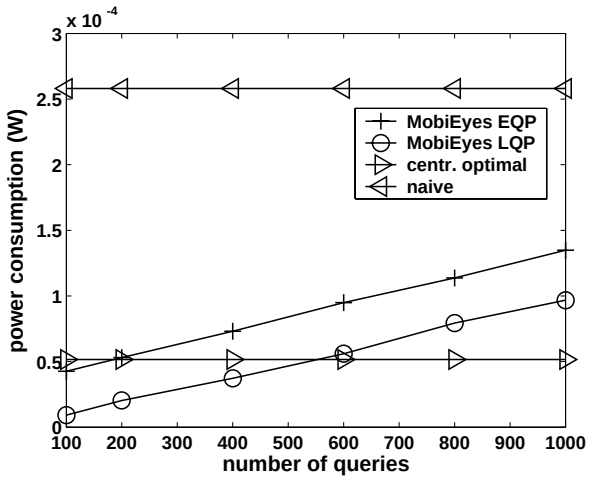


Figure 15: Effect of # of queries on per object power consumption due to communication

We simulated the MobiEyes approach using message sizes instead of message counts for messages exchanged and compared its power consumption due to communication with the naive and central optimal approaches. The graph in Figure 15 plots the per object power consumption due to communication

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

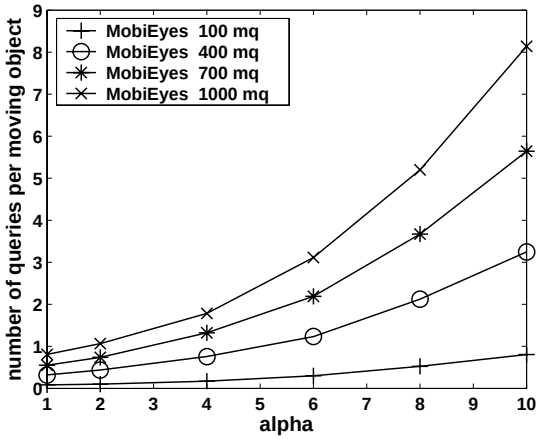


Figure 16: Effect of α on the average number of queries evaluated per step on a moving object

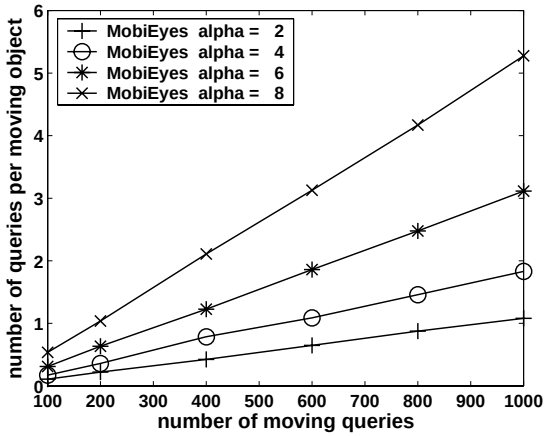


Figure 17: Effect of the total number of queries on the average number of queries evaluated per step on a moving object

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

Figure 16 and Figure 17 study the effect of α and the effect of the total number of queries on the average number of queries a moving object has to evaluate at each time step (average LQT table size). The graph in Figure 16 plots the average LQT table size as a function of α for different number of queries. The graph in Figure 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 exceeds 10 for the simulation setup. The second observation is that the average size of the LQT table increases exponentially with α where it increases linearly with the number of queries. Figure 18

studies the effect of the query radius on the number of average queries a moving object has to evaluate at each time step. The x-axis of the graph in Figure 18 represents the radius factor, whose value is used to multiply the original radius value of the queries. The y-axis represents the average LQT table size. It is observed from the figure that the larger query radius values increase the LQT table size. However this effect is only visible for radius values whose difference from each other is larger than the α. This is a direct result of the definition of the monitoring region from Section 2.

Figure 19 studies the effect of the safe period optimization on the average query processing load of a moving object. The x-axis of the graph in Figure 18 represents the α parameter, and the y-axis represents the average query processing load of a moving object. As a measure of query processing load, we took the average time spent by a moving object for processing its LQT table in the simulation. Figure 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 non-zero safe periods and decreases the cost of processing the LQT table. On the other hand, for very small values of α, like $\alpha = 1$ in Figure 19, 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.

7 Discussion

We have described the MobiEyes approach to distributed processing of moving queries, focusing on distributed data structures and distributed algorithms for efficient processing of continuously moving queries on moving objects. In this section we discuss a number of issues that are important for employing the MobiEyes solution to large-scale location management. We first discuss a set of performance optimization opportunities in MobiEyes and then describe the desired system properties such as moving object collaboration, location security, and location privacy.

Supporting MQs with Dynamic Filters

So far we considered the use of filters on static object properties. These filters are evaluated during query

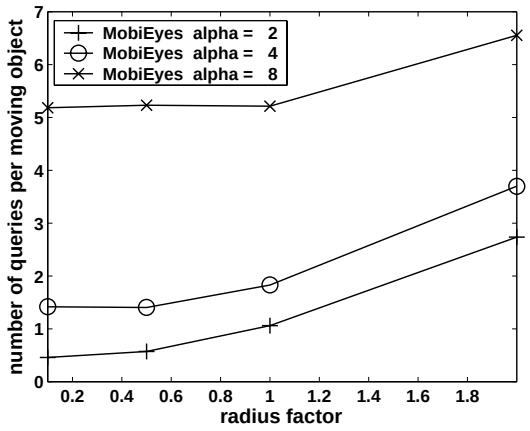


Figure 18: Effect of the query radius on the average number of queries evaluated per step on a moving object

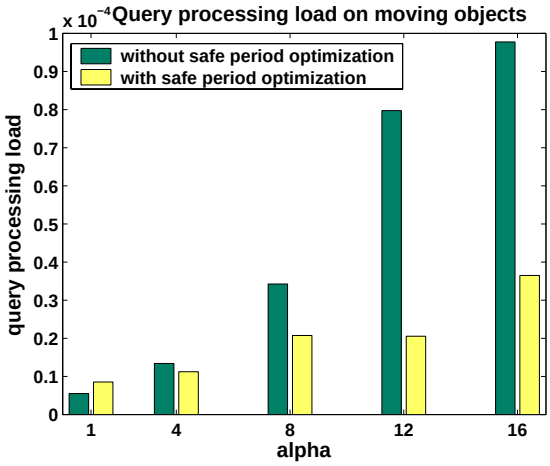


Figure 19: Effect of the safe period optimization on the average query processing load of a moving object

installation time on the moving object side. However, it may be desirable to have filters on dynamic object properties. For instance a filter like "objects that are headed towards north" is defined on a dynamic property of the moving objects, which is the movement direction in this example. Dynamic filters can be easily supported by applying a small modification to the algorithm on the moving object side. This modification includes accepting queries regardless of their filters during installation time and evaluating the filters periodically for the queries whose spatial regions cover the moving object's location.

Adding Temporal Dimension to MQs

An extension to MQs is to add a temporal dimension to predict future results of queries from current state. A simple way to support this kind of queries requires two small modifications, given that there exist a bound on maximum velocity of moving objects. First the bounding rectangles and the monitoring regions has to be expanded as if the radius of the queries are extended by the maximum velocity of an object times the temporal parameter of the query. Second, when evaluating a query in its LQT table, instead of using its current position, the moving object has to predict and use its future position according to the temporal parameter of the query.

Air Indexing in MobiEyes

In case our universe of disclosure is covered by a single broadcast medium or the base station cover-

age areas are large, moving objects will be hassled with many broadcast messages that are irrelevant to them. Air indexing [IVB94] is a useful technique in such scenarios, since it enables moving objects to selectively tune in the broadcast and read data that is relevant to them. In the context of MobiEyes, air indexing can be used to broadcast some portion of the server state regarding the moving queries in the wireless medium, so that each moving object can tune in the broadcast during appropriate times and receive the information it needs to process the moving queries. We are in the process of developing algorithms for such energy efficient spatial broadcast indexes [GSL04].

Moving Object Collaboration

MobiEyes approach requires two levels of collaboration of the moving objects in order to process moving queries in a distributed manner. The level one collaboration is the basic collaboration required by all location tracking and location information management systems. For instance, any location tracking system requires moving objects to report their positions. In addition, MobiEyes requires moving objects to perform certain amount of computation for distributed processing of MQs. We can view the requirement for moving objects to perform certain amount of computation the second level of collaboration. By performing query processing, non-focal objects report their positions less frequently. In other words, MobiEyes provides less frequent position reporting in exchange for small amount of computation. Focal objects have more responsibilities compared to non-focal objects. Given that a focal object is the initiator of the moving queries associated with it, having the query initiator to do more work is also a reasonable system behavior, because it is receiving additional service.

Security of Location Management

An interesting and important issue regarding the MobiEyes system is the security of location management. In case the software running on the moving objects is altered maliciously, strange results may occur. Several attacks are possible including false binding updates, source spoofing, highjack connections, and bombing attack (flood target by redirecting data streams). The difficult part of location security research is understanding threats and security requirements. In fact, any system that tracks position information from the moving objects is subject to the same problem. However, the MobiEyes solution

poses the additional problem of producing inconsistent results as opposed to being only subject to produce wrong results. A simple example of this is the case where a malicious moving object reports that it is included in the results of two queries whose regions do not overlap. Methods to detect this kind of or any other kind of malicious behavior and defense mechanisms to resolve the problem is an interesting topic of further study.

Location Privacy

Location privacy breaches occur if the path followed by a moving object can be determined by a non-trusted entity in the system. Private information is usually exposed by linking path information with external observations of an individual's location (like in *Observation Identification* [GG03]) or external information regarding a location-individual relationship (like in *Restricted Space Identification* [GG03]). In MobiEyes, the location server or the base stations can be regarded as trusted entities. However, a moving object cannot be assumed to be trusted from the perspective of another moving object. Since a focal object's path can be determined by a non-focal object in its monitoring region, location privacy is an issue in MobiEyes. Fortunately, there exist approaches like location k-anonymity that trade accuracy for anonymity [GL04, GG03]. Integration of location privacy into MobiEyes is a topic of future study in our research agenda.

8 Related Work

Real-time evaluation of *static* spatial queries on moving objects, at a centralized location, is a well studied topic. In [PXK+02], velocity constrained indexing and query indexing has been proposed for efficient evaluation of static continual range queries at a central location. The same problem is studied in [KPHA02], however the focus is on in-memory structures and algorithms. In [SJLL00], TPR-tree, an R-tree based indexing structure, is proposed for indexing the motion parameters of moving objects by using time parameterized rectangles and answering queries using this index. TPR* tree, an extension of TPR tree optimized for queries that look into future (predictive), is described in [TPS03]. Several other indexing structures and algorithms for handling moving object positions have been suggested in the literature [CH02, LPM02, TP02, KGT99, AAE00, BJKS02, TPS02]. There are two main points

where our work departs from this line of work.

First, most of the work done in this respect has focused on efficient indexing structures and has ignored the underlying mobile communication system and the mobile objects. To our knowledge, only the SQM system introduced in [CH02] has proposed a distributed solution for evaluation of static spatial queries on moving objects, that makes use of the computational capabilities present at the mobile objects.

Second, the concept of dynamic queries presented in [LPM02] are to some extent similar to the concept of moving queries in MobiEyes. But there are two subtle differences. First, a dynamic query is defined as a temporally ordered set of snapshot queries in [LPM02]. This is a low level definition. In contrast our definition of moving queries is at end-user level, which includes the notion of a focal object. The second subtle difference is that the work done in [LPM02] indexes the trajectories of the moving objects and describes how to efficiently evaluate dynamic queries that represent predictable or non-predictable movement of an observer. They also describe how new trajectories can be added when a dynamic query is actively running. Their assumptions are in line with their motivating scenario, which is to support rendering of objects in virtual tour-like applications. The MobiEyes solution discussed in this paper focuses on real-time evaluation of moving queries in real-world settings, where the trajectories of the moving objects are unpredictable and the queries are associated with moving objects inside the system.

Several existing research efforts on moving object databases [WSCY99, SJLL00, WXCJ98, KGT99] model moving 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 moving 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 moving object, on the moving object side, is more similar to the use of dead reckoning in on-line games and PDES [Fuj00] systems for building distributed virtual environments. There are very few attempts to use non-linear motion modeling in moving object databases [AA03]. The debate on whether linear modeling assumptions can easily carry out in practice and its possible implications are

beyond the focus of this paper. Such discussions can be found elsewhere [WSCY99].

Safe period optimization described in Section 4 is inspired by the safe region optimization introduced in [PXK+02]. 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 node. 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 moving object that belong to a restricted subset, namely the moving objects that reside within the monitoring region of an MQ.

Most of the work done in mobile communications world related with managing moving object positions is on "location management" [AW02, BD99, BNKS95, NL98]. Given the unique identifier of a mobile object, locating under which base station's coverage it is currently residing in is the problem of location management and is a fundamental service used for setting up connections between mobile objects. MobiEyes on the other hand, is a location monitoring system aiming at efficient real-time evaluation of MQs.

9 Conclusion

Significant research efforts have been dedicated to techniques for efficient processing of spatial continuous queries on moving objects in a centralized location monitoring system. Surprisingly, very few have promoted a distributed approach to real-time location monitoring. In this paper we have presented a distributed and scalable solution to processing continuously moving queries on moving 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 moving objects to distinguish continuous moving queries on moving objects from a well-studied class of static spatial queries on moving objects. Second, we design and develop a distributed algorithm for real-time evaluation of continuously moving queries on moving 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. We showed that the distributed processing of MQs significantly decreases the server load and scales well in terms of messaging cost while placing only small amount of processing burden on moving objects.

The ongoing and future work on the MobiEyes project includes performance enhancement in terms of server load, messaging cost, and per object energy consumption, and a comprehensive study of security vulnerabilities and privacy breaches related with distributed location management systems such as MobiEyes.

References

[AA03] Charu C. Aggarwal and Dakshi Agrawal. On nearest neighbor indexing of nonlinear trajectories. In *ACM PODS*, 2003.

[AAE00] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing moving points. In *ACM PODS*, 2000.

[AW02] Ian Akyildiz and Wenye Wang. A dynamic location management scheme for next generation multi-tier pcs systems. *IEEE Transactions on Wireless Communications*, 1(1):178–190, 2002.

[BD99] Amiya Bhattacharya and Sajal K. Das. Lezi-update: An information-theoretic approach to track mobile users in PCS networks. In *ACM MobiCom*, 1999.

[BJKS02] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *International Database Engineering and Applications Symposium*, 2002.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD*, 1990.

[BNKS95] Amotz Bar-Noy, Ilan Kessler, and Moshe Sidi. Mobile users: To update or not to update? *ACM Wireless Networks*, 1(2):175–185, 1995.

[CH02] Ying Cai and Kien A. Hua. An adaptive query management technique for efficient real-time monitoring of spatial regions in mobile database systems. In *IEEE IPCCC*, 2002.

[Fuj00] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley-Interscience, 2000.

[GG03] Marco Gruteser and Dirk Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *ACM/USENIX MobiSys*, 2003.

[GL04] Bugra Gedik and Ling Liu. A customizable k-anonymity model for protecting location privacy. Technical Report GIT-CERCS-04-15, CERCS, Georgia Institute of Technology, 2004.

[gps03] US Naval Observatory GPS Operations. http://tycho.usno.navy.mil/gps.html, April 2003.

[GSL04] Bugra Gedik, Aameek Singh, and Ling Liu. Energy efficient exact knn search in wireless broadcast environments. Technical Report GIT-CERCS-04-17, CERCS, Georgia Institute of Technology, 2004.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *ACM ASPLOS*, 2000.

[IVB94] Tomasz Imielinski, S. Viswanathan, and B. Badrinath. Energy efficient indexing on air. In *ACM SIGMOD*, 1994.

[JL99] J.Kucera and U. Lott. Single chip 1.9 ghz transceiver frontend mmic including Rx/Tx local oscillators and 300 mw power amplifier. *MTT Symposium Digest*, 4:1405–1408, June 1999.

[KGT99] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On indexing mobile objects. In *ACM PODS*, 1999.

[KPHA02] Dmitri V. Kalashnikov, Sunil Prabhakar, Susanne Hambrusch, and Walid Aref. Efficient evaluation of continuous range queries on moving objects. In *DEXA*, 2002.

[LPM02] Iosif Lazaridis, Kriengkrai Porkaew, and Sharad Mehrotra. Dynamic queries over mobile objects. In *EDBT*, 2002.

[LPT99] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, pages 610–628, 1999.

[Mil91] David L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications*, pages 1482–1493, 1991.

[NL98] Zohar Naor and Hanoch Levy. Minimizing the wireless cost of tracking mobile users: An adaptive threshold scheme. In *IEEE INFOCOM*, 1998.

[PJT00] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, 2000.

[PXK+02] Sunil Prabhakar, Yuni Xia, Dmitri V. Kalashnikov, Walid G. Aref, and Susanne E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[SJLL00] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *ACM SIGMOD*, 2000.

[SWCD97] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In *IEEE ICDE*, 1997.

[TP02] Yufei Tao and Dimitris Papadias. Time-parameterized queries in spatio-temporal databases. In *ACM SIGMOD*, 2002.

[TPS02] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbor search. In *VLDB*, 2002.

[TPS03] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, 2003.

[WSCY99] Ouri Wolfson, Prasad Sistla, Sam Chamberlain, and Yelena Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[WXCJ98] Ouri Wolfson, Bo Xu, Sam Chamberlain, and Liqin Jiang. Moving objects databases: Issues and solutions. In *SSDBM*, 1998.