On Reducing Communication Cost for Distributed Moving Query Monitoring Systems

Fuyu Liu Kien A. Hua Fei Xie
School of EECS
University of Central Florida
Orlando, FL, USA
{fliu, kienhua, xiefei}@cs.ucf.edu

Abstract

Moving monitoring query on moving objects is an important type of query in location based services. Existing solutions suffer from high communication cost. In this paper, we propose a distributed solution to this problem. Our approach employs two ways of communications, on-demand access and broadcast channel, to reduce communication cost. Two different indexes are proposed and evaluated using simulations. The performance results indicate that our solution achieves from 30% to 60% savings in communications when compared to MobiEyes [2].

1. Introduction

With the advance in wireless communication technology and the popularity of positioning systems, a variety of location based services have become available to the public. Among them, spatial monitoring of moving objects is fundamental to these applications. In particular, if the spatial query is anchored around a moving object (e.g., monitoring the *k* nearest neighbors of a moving object), this query is referred to as a moving query. In a more complex environment where many objects are moving, we can also have moving monitoring query over moving objects. For instance, while walking in downtown, the user can issue a query like "Find me the available TAXIs within two miles."

There are three main challenges in answering moving queries over moving objects. First, the query results must be updated constantly until the user explicitly terminates the continuous query. Second, the data points are moving constantly making location updates very expensive. Third, the region of interest (i.e., monitoring region) itself is also changing steadily adding significantly more complexity. Most existing solutions employ a centralized approach [5, 6, 7, 8], where the focus is on designing efficient data structures and algorithms. The limitations of this approach are twofold. First, the server, with finite computation and communication capability, may not be able to cope with the high location-update frequency, desired in some of the applications. Second, frequent location update is not a natural fit for energy-constrained mobile devices. It can quickly drain the mobile batteries.

Addressing the aforementioned issues is not trivial as a lower location-update rate would affect the quality of the query results. Some recent techniques have focused on distributed solutions [1, 2, 3, 4]. In a distributed environment, mobile objects monitor the query, and need to update the server only when the query result changes. This approach requires much fewer communications with the server. Nevertheless, the server needs to supply relevant information, from time to time, to support the query monitoring processes on the mobile devices. When the number of mobile objects increases or the object mobility increases, this communication cost can become quite high [1, 2, 3, 4]. A more scalable solution is desirable.

In a wireless environment, there are two ways of communications between the server and clients. One way is on-demand access, in which client and server send messages to each other whenever necessary. Another way is using broadcast, where the server periodically broadcasts data on an open wireless channel. A client can tune into the channel to retrieve the desired data.

In this paper, we leverage a hybrid of on-demand data access and periodic broadcast to design a new distributed solution for moving monitoring queries on moving objects. The server sets aside a broadcast channel to repeatedly broadcast query information. The area of interest is mapped into grid cells. When a mobile client moves from one cell into another cell, it tunes into the broadcast channel to download information on relevant queries instead of contacting the server for the information. To improve the tuning efficiency, we proposed two indexing schemes for the broadcast technique. Our simulation results indicate that the proposed solutions achieve from 30% to 60% savings in communication cost when compared to

MobiEyes [2]. The reduction in communications also significantly improves energy conservation on the mobile devices.

The remainder of this paper is organized as follows. We discuss some related work in Section 2 to make the paper self-contained. In Section 3, we present an overview of the proposed environment. We discuss the index structures in Section 4, and the query processing technique in Section 5. The simulation study is presented in Section 6. Finally, we conclude this paper in Section 7.

2. Related Work

In a wireless environment, mobile device's limited battery power is a critical concern. Because for a mobile device, sending a message consumes more energy than receiving a message, there are proposals using broadcast channel to replace the on-demand data request to save energy. In the existing proposals [10, 12, 13, 14, 15, 16], the focus is on how to design a good index to facilitate data access. Since a mobile object has the ability to switch between an active mode and a doze mode, with an effective index, it can first tune into the broadcast channel to get the predicted arrival time of the desired data, then goes back to the doze mode, and returns to the active mode to download the data when the data comes.

Two performance metrics are typically used to evaluate an index: tuning time and access latency (time). The tuning time means the total time that a mobile object needs to stay in the active mode to get data, which includes the time spent on searching index and downloading data. The access latency is referred to the total time elapsed from the moment a mobile object tuning into the broadcast channel to the moment the mobile object actually obtaining the desired data. One popular technique to reduce access latency is the (1, *m*) interleaving technique [11], as shown in Fig. 1. In this technique, a complete index is broadcast preceding every 1/*m* fraction of the full broadcast cycle. By duplicating the index for *m* times, the waiting time to reach an index can be shortened, thus access latency is reduced. Please note that this technique is orthogonal to any proposed wireless index, and thus can be applied to any index. In this paper, we also adopt this interleaving technique.

Moving monitoring query over moving objects has been studied extensively. One direction in this area is to reduce server side workload by proposing efficient server side data structure and index [5, 6, 7, 8]. Another direction is to use distributed computing to reduce both server side workload and communication cost. Gedik and Liu introduced MobiEyes [2], which is capable of answering moving range queries over moving objects. Wu et al. [1] proposed a distributed solution to answer moving *k*NN queries. Our recent works [3, 4] addressed moving range and moving *k*NN queries in street network environment. Moreover, in [9], the authors used local-area wireless network to alleviate the high communication cost problem. Trajcevski et al. [17] worked on how to aggregate query results in a distributed environment.

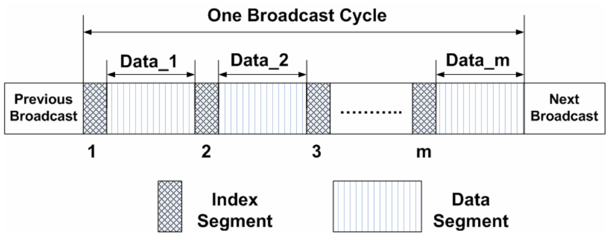


Figure 1. Layout for the Interleaving Tech.

To the best of our knowledge, our paper is the first to combine broadcast channel and distributed computing to answer moving monitoring query over moving objects.

3. System Overview

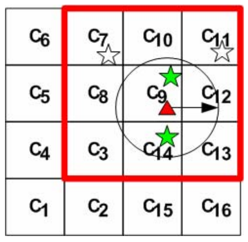
In this section, we give an overview of our system and the proposed query processing technique. The system consists of a centralized server, a number of stationary base stations and a large number of moving objects. A moving object and the server communicate with each other through base stations. We also assume that the server can broadcast data on a wireless channel accessible to all moving objects. Each moving object has a GPS-like device to determine its location and has some computing capability.

The geographical area of interest is a big rectangular area, which is mapped into grid of cells of an $\alpha \times \alpha$ square area. Every moving object has a unique object ID. When a moving object reports its location to the server, the message has the following format $<i, p_i, v_i, t_i>$, where i is the object ID, p_i is the object's position, v_i is the object's velocity, and t_i is the time when the object's position and velocity are recorded.

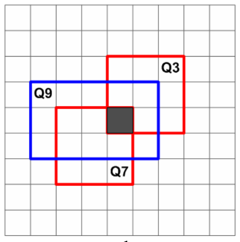
In our system, moving range queries are issued by moving objects. If a moving object has issued a moving query, we call it a *query object*; otherwise, we call it a *data object*. A moving range query is modeled as a tuple <*qid*, *oid*, *range*>, where *qid* is the query's ID, *oid* is the ID of the associated *query object*, and *range* defines the moving search area around the *query object*. A query area can be a circle (specified by a radius) or a rectangle (specified by width and length).

The result of a query is a set of identifiers of the moving objects currently residing in the query's region. An example is given in Fig. 2.a, where the terrain is divided into 16 cells, labeled according to the Hilbert Curve order as C_1, C_2, …, C_{16}. A *query object* is drawn as a triangle, and the query region is the area covered by the circle. Two *data objects* (drawn as the stars) residing in that circle are included as the query result.

One important concept is the *monitoring region* of a moving range query. Given a *query object* and its current grid cell, the query region can overlap several neighboring grid cells. The union of all grid cells this query region may overlap as the query point moves inside its current grid cell is referred to as the monitoring region for the given query. In our environment, all moving objects within the monitoring region of a query must monitor their distance to the query point and update the query result if they fall within the query region (i.e., the distance is less than the *range* field specified in the query). As an example, the nine grid cells in the upper right corner of Fig. 2.a comprise the *monitoring region* of the query. There are two *data objects* in cells C_7 and C_{11}, respectively. They are currently not part of the query result, but they need to monitor their distance to the *query object*. We note that the shape of a monitoring region is a rectangular area if the query region is a rectangle.



a



b

Figure 2. Example of (a). moving query and (b). monitoring region

If a monitoring region of a query overlaps with a given grid cell, we say that this query intersects with the cell. When an object moves around, it could exit its current cell and enter a new cell. Whenever this happens, the object needs to get a new set of queries intersecting with the new cell to continue the monitoring task. In Fig. 2.b, the monitoring regions of three queries Q_3, Q_7, and Q_9 are represented by the three rectangular areas. The shaded cell in the center intersects with all three of these queries. Any object moving into this shaded cell should monitor these three queries. Obviously, the costs associated with obtaining these relevant queries from time to time increases with the increases in the number of data objects or the increases in the object mobility. We discuss a broadcast technique to address this issue in the following section.

4. Broadcast Index Design

In existing research works [1,2,3,4], when a moving object moves to a new cell (or a new road segment), the moving object contacts the server to request a new set of queries. In this paper, we broadcast query information on a wireless channel. Instead of sending a message to the server, the moving object just tunes into the wireless channel and downloads the relevant queries. How to design an effective index becomes critical. A good index should enable short tuning time and incur little overhead on access time. We introduce two indexes in this section.

Fig. 3 shows the overall structure of the first proposed index, named as *Grid Index (GI)*. This index consists of two levels. The upper-level index is built on top of grid cells, which can be any type of index supporting a quick identification of grid cell given a geographical location, for example, a quad-tree. In our case, since the cell has a fixed pre-known size, no tree index is needed. A simple mapping function is sufficient to determine the desired cell.

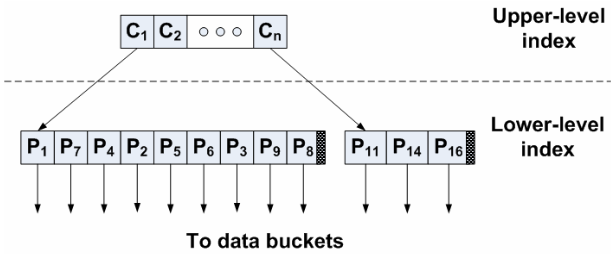


Figure 3. *Grid Index* structure

The low-level index consists of many blocks, with each block corresponding to a cell in the upper-level index. Inside each block, we store the pointers to all the queries intersecting with that corresponding cell. For example, two blocks are shown in Fig. 3. The first block stores all the pointers to the queries intersecting with the cell C_1, where P_1, P_2, etc. are the pointers to query Q_1, Q_2, etc. At the end of each block, we put a special tag to indicate the end of that block.

With the upper-level index, a moving object can map its location to the corresponding grid cell. Following the pointer in that cell, the object is directed to the lower-level index, where the object can download the pointers to the queries it should monitor. When it sees the tag for the end of a block, it stops downloading and goes to doze mode. It only returns to active mode when the interested query data buckets arrive.

One drawback of the above mentioned technique is that when a mobile object moves from one cell to another cell, it has to download a complete new set of queries from the broadcast channel that it should monitor in the new cell.

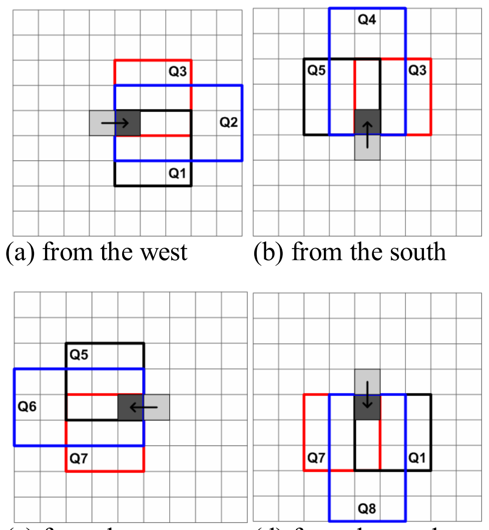
We observe that since the old cell and the new cell are adjacent, there are some queries required to be monitored by objects in both cells. When a mobile object moves from the old cell to the new cell, only a subset of the queries needs be downloaded. This inspires us to design an index that can facilitate the downloading of only the missing queries. Notice that a mobile object can move into a new cell from four directions: west, south, east, and north. We can classify the queries whose monitoring regions intersect with the new cell into different types, such that for a specific direction from which the mobile object enters the new cell only certain types of the queries need to be retrieved. An algorithm that categorizes such queries into nine types is given in Fig. 4.

Using the algorithm in Fig. 4, queries belonging to type 1, 2, or 3 should be added for monitoring when an object moves into this cell from the west side. Queries belonging to type 3, 4, or 5 should be added for monitoring when an object enters from the south side. Similarly, when an object enters from the east side, queries of type 5, 6, or 7 should be added; and queries of type 7, 8, or 1 should be added when an object enters from the north side.

```
Input: a cell c, all queries intersecting with this cell
Output: each query's type for the cell c
For each query q
  Get the monitoring region (MR) of q
  If c is a west side boundary cell of MR
    If c is the north-west corner cell of MR, q.type = 1
    Else if c is the south-west corner cell of MR,
      q.type = 3
    Else, q.type = 2
  Else if c is a south side boundary cell of MR
    If c is the south-east corner cell of MR, q.type = 5
    Else, q.type = 4
  Else if c is an east side boundary cell of MR
    If c is the north-east corner cell of MR, q.type = 7
    Else, q.type = 6
  Else if c is a north side boundary cell of MR
    q.type = 8
  Else // not a boundary cell
    q.type = 9
```

Figure 4. Algorithm to determine query type

Fig. 5 illustrates the four scenarios when an object moves into a new cell from four different directions. In the diagram, Q_1, Q_2, …, and Q_8 are examples of queries which belong to type 1, type 2, …, and type 8, respectively. As we can see, in the Fig. 5.a, when an object moves from the lightly shaded cell (the old cell) into the darkly shaded cell (the new cell), only queries with type 1, 2, and 3 need to be added. Fig. 5.b, 5.c, and 5.d are examples for the other three scenarios.



(a) from the west (b) from the south

(c) from the east (d) from the north

Figure 5. Scenarios when object changing cell

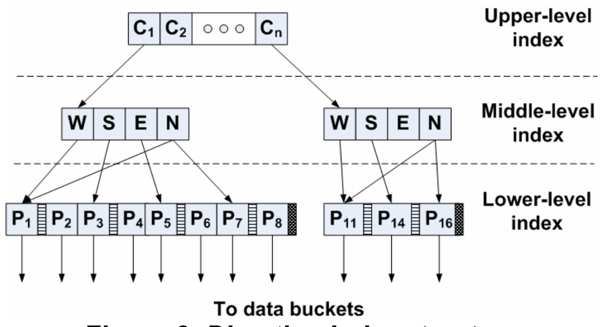


Figure 6. *Direction Index* structure

Based on the above observations, we propose a three-level index structure, called *Direction Index (DI)*. Unlike the *Grid Index*, we include one more level to represent the direction from which an object enters the new cell, as shown in Fig. 6. We use letters “W”, “S”, “E”, and “N” to represent the directions “West”, “South”, “East”, and “North”, respectively. Inside a “W” cell, there is a pointer pointing to the beginning of the list of type 1 queries in the lower-level index. It is similar for the “S” and “E” cell. For an “N” cell, it is a little more complex. Since an object entering a new cell from the north needs to get queries of type 7, 8 and 1, we provide two pointers in an “N” cell, with the first one pointing to the beginning of type 1 queries, and the second one pointing to the beginning of type 7 queries.

Besides the addition of the middle-level index, there are two differences in the lower level. The first difference is that there is no need to include pointers for type 9 queries because they are already in the object's monitoring list when it is in the old cell. This feature reduces the index size, and leads to better access time and tuning time.

The second difference is that we need to sort the pointers using the associated query types, and use a tag to mark the end of types 1, 3, 5, and 7. This feature enables an object to know when to stop downloading the index packets. As an example in Fig. 6, if an object is interested in downloading queries of types 1, 2, and 3, it first follows the pointer from the "W" cell and gets to the start of type 1 queries. It then downloads the queries until it recognizes the end tag of type 3 queries. As another example, if an object enters cell C_n from the east direction, since there is no link originated from the "E" cell, the object knows that there is no query that needs to be downloaded and it can go back to doze mode.

5. Data Structures and Query Processing

In the proposed system, query processing is distributed among moving objects and the server. After a query is initiated, the server calculates the monitoring region for that query, and notifies objects in that monitoring region to monitor this query. In this section, we first present the data structures on the server and moving objects, and then discuss how the system handles different activities.

5.1. Data Structures

There are three main data structures used by the server. The *Query Object Table* stores the list of *query objects* and their parameters including velocity, position, and the time stamp when the velocity and position are recorded. The *Server Query Table*, keeps the list of moving queries. Each query contains the *query object*'s ID, the specified range, and the monitoring region. Query result is also saved in this table. The third table is the *Reverse Query Table*, where all queries intersecting with each cell are saved. To facilitate the *Direction Index*, query's type is also stored in this table.

Each mobile object needs to maintain a *Client Query Table* to keep track of all the queries it should monitor. In this table, we store each query's ID, velocity, position, and the time when the velocity and position were reported, and the specified range.

5.2 Installing Queries

When the server receives a new moving query (with its associated moving object ID, position, velocity, and the query's search range), the server first updates the *Query Object Table* and the *Server Query Table*, then calculates the monitoring region for this new query and saves the result into the *Server Query Table*, and finally updates the *Reverse Query Table*.

After installing the query on the server, the server forwards the query's information to all moving objects. Moving objects in the monitoring region save the message and start to monitor their distance to the *query object*. For objects outside of the query's monitoring region, they can simply ignore that message.

5.3. Handling Objects Changing Query Result

For all queries in its monitoring list, an object needs to periodically predict the current positions of the *query objects* using the saved velocity, time, and position information in the *Client Query Table*. It then calculates its distance to the *query object* to determine if it is in the query's range. If the result is different from the previous result computed in the last time unit, the object sends a message to request the server to either adding it to the query result or removing it from the query result. After receiving the message from the moving object, the server locates the query from the *Server Query Table*, and updates the result accordingly.

5.4. Handling *Query Objects* Changing Velocities

Query objects need to report their new velocities to the server if there are significant changes. Once the server receives a new velocity, it broadcasts a message with the updated velocity information to moving objects in that query's monitoring region. Upon receiving the message, a moving object first updates its *Client Query Table*, and then applies the updated velocity to calculate the distance under monitoring.

5.5. Handling Objects Changing Grid Cells

When an object moves from one cell to another cell, the object first needs to determine which queries should not be monitored anymore. With the saved query's information, an object computes its minimum distance to the *query object* given that the object itself is moving within the new cell. If the minimum distance is greater than the specified query range, that query is dropped from the monitoring list. Also, the moving object needs to get new queries to monitor. In existing solutions, such as MobiEyes [2], moving objects always send messages to the server to request new queries. In our system, with the help of a broadcast channel containing query information, moving objects do not need to send messages to the server anymore. Instead, they just tune into the broadcast channel and download the necessary new queries. Two different index methods are presented in Section 4. We study the effectiveness of these two methods in Section 6.

When the moving object changing cell happens to be a *query object*, it still needs to send a message to the server. In response, the server performs the following three tasks. First, it updates the tables on the server and computes a new monitoring region for the query. Second, the server broadcasts a message to all moving objects residing in the old monitoring region to stop monitoring this query. Finally, the server notifies all moving objects in the new monitoring region to monitor this query.

5.6. Communication between Server and Objects

As we have discussed in Section 4, objects need to switch to doze mode when they are waiting for query data packets to arrive. However, objects have to stay awake to receive broadcast messages from the server, as suggested in Section 5.4 and 5.5. This presents a dilemma. To solve this problem, we can make use of synchronized time between the server and moving objects [18, 19]. With synchronized time, the server sends out messages to moving objects only at pre-scheduled time slots, on the other hand, moving objects just need to wake up periodically during doze mode at these pre-scheduled time slots. If a moving object does not receive anything from the server when it wakes up, it goes back to sleep and wakes up again at the next pre-scheduled wake up time.

6. Simulation Study

We implemented a simulator in Java to evaluate the performance of the proposed system. The system in the simulation consists of a base station, a broadcast channel, and a number of moving objects. The available bandwidth is set to 100K bps. The packet size is varied from 64 bytes to 1024 bytes. In each packet, two bytes are used for the packet ID, and two bytes are allocated to a pointer. Coordinate is represented with eight bytes. The size of a query is set to fifty bytes (to hold query ID, position, velocity, time, and range). In the broadcast channel, queries are ordered using the Hilbert curve order. For each query, we first identify the cell where the query's *query object* is located in, then calculate the Hilbert curve value for that cell, and use that value for ordering.

Our simulation is set up as follows. The area of interest is a square-shaped region of 64 × 64 square miles. The whole region is divided into grid cells, where each cell is a square with an area of α × α. Moving objects are randomly generated and placed in the region. The velocities are in the range of [0, 1] mile per time unit (equivalent to [0, 60] mile per hour) with random directions, following a Zipf distribution with a deviation of 0.7. Among the moving objects, some are randomly selected as *query objects*. The query regions have circular shape with their radius randomly chosen from the set {1, 2, 3, 4, 5} miles. At each time unit, there are ten percent of moving objects changing their velocities. The threshold for changing velocity is set as 0.1 mile per time unit. We run simulation for 10 times with different seeds and compute the average as the final simulation results. Each simulation lasts for 200 time units. The simulation was run on a Pentium 4 2.6GHz desktop with 2GB memory. In the experiments, we vary different parameters to study the performance. The parameters are listed in Table 1. If not otherwise stated, the experiment takes the default values.

Table 1. Simulation parameters

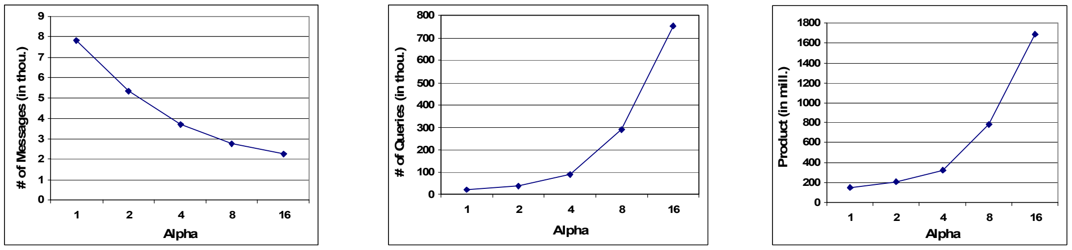
Parameter Name	Value Range (or Set)	Default
Cell Size (mile)	{1, 2, 4, 8, 16}	2
# of Objects	[2000, 10000]	10000
# of Queries	[200, 1000]	200
Packet Size (byte)	{64,128,256,512, 1024}	64

In the remainder of this section, we first determine a good cell size for the simulation study. We then compare our system with MobiEyes on communication cost. Finally, we compare the two proposed indexing schemes in terms of access time and tuning time, and discuss the effect of packet size on these two indexes.

6.1 On Selecting Cell Size (α)

One important parameter in our system is the cell size α. An optimal α value should reduce the number of messages as much as possible. To make sure that the comparison on communication cost to MobiEyes is fair (Section 6.2), we try to determine a good α value for MobiEyes.

Fig. 7.a shows that as the value of α increases, fewer messages are communicated. This can be explained as follows. When the cell size is increased, there are less messages resulting from objects changing cells. We note that when the whole terrain consists of only one cell, the number of messages drops to the minimum. However, in this situation, all objects need to monitor all queries in the system, which increases



(a) (b) (c)

Figure 7. Cell size selection. (a). # of messages vs. α, (b). # of queries monitored vs. α, (c). Product of # of messages and # of queries monitored vs. α

the computation workload on the client side considerably. To study this effect, we use the number of queries monitored by each moving object as a metric to approximate the level of computation workload. Fig. 7.b shows the average number of queries monitored by each moving object with the increases in cell size. As we can see, the curve increases sharply as expected. To find a good α value, we compute the product of the number of messages and the number of queries monitored, and then plot the curve against the cell size as shown in Fig. 7.c. This figure shows that the product is the minimum when the cell size is the smallest (equal to 1). Since the curve increases only slightly when the cell size is set to 2, we select "α = 2" as a good setting for MobiEyes. This value gives a good balance between communication cost and computation cost. We note that this α value is selected to give MobiEyes the best performance in our simulation study. It is not a universally good value.

6.2. On Communication Cost

In this section, we focus on the communication cost of the system, which is measured in terms of the number of messages communicated between the server and moving objects. Our system is compared against MobiEyes, where only the on-demand communication mechanism is used between server and moving objects. We study the sensitivity of our system on two parameters: the number of moving objects and the number of queries. The sensitivity analyses with respect to other parameters have also been studied; but we do not include the results here due to space limit.

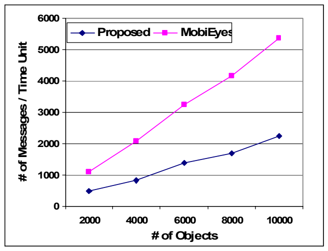
In Fig. 8.a, we vary the number of moving objects from 2000 to 10000, and measure the average number of messages per time unit. The figure shows that our system requires less than half the number of messages required by MobiEyes, which equals about 60% in savings in terms of communication cost. The reason is that there are a lot of messages due to changing-cell activities in MobiEyes. In contrast, in our system, unless the object that changes cell is a *query object*, the changing-cell activity does not entail extra message.

In Fig. 8.b, we study the effect of the number of queries on communication cost. As we can see, when the number of queries is increased from 200 to 1000, more messages are needed. This is reasonable since more queries mean more updates for query result change (Section 5.3) and *query object* velocity change (Section 5.4), which require more messages. Our method saves about 60% of messages when the number of queries is small, and still saves about 30% when the number of queries is large.

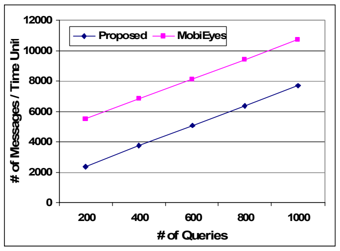
6.3. Comparison of the Proposed Indexes

In this section, we compare the performance of the *Grid Index (GI)* and *Direction Index (DI)* using access time and tuning time as metrics. For the access time, we also include a technique, called *No Index*, where no index is used. The *No Index* technique only broadcasts data packets, and clients have to download all the data packets to determine which one to keep or discard. Since the *No Index* technique does not use any index, it has the minimal access time, and thus serves as a good baseline for comparison. On the other hand, the tuning time for the *No Index* technique is prohibitively long, we do not include it when examining the tuning times for the proposed indexes.

In Fig. 9.a, we vary the number of objects to compare the two indexes and the *No Index* technique. The figure shows that the access time increases linearly when the number of objects increases. This is expected because the access time is measured as an average per time unit. When there are more moving objects, the access time for all objects adds up linearly. The figure also shows that both *GI* and *DI* need longer access time compared to the *No Index* technique, but the difference is not very big. The access times with *GI* and with *DI* are about 1.5 times and 2 times that of the *No Index* technique, respectively.

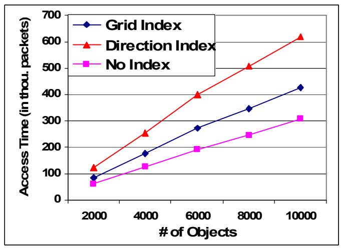


(a) Varying # of objects

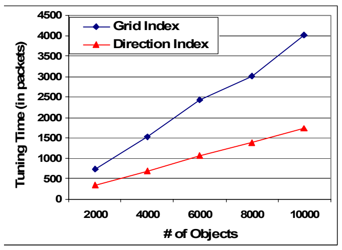


(b) Varying # of queries

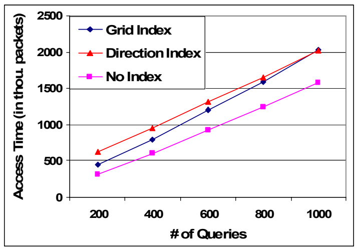
Figure 8. Comparison with MobiEyes on communication cost.



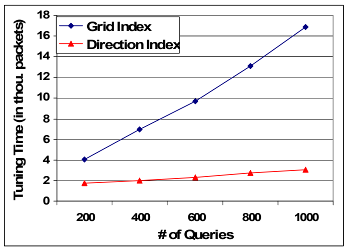
(a) Access time vs. # of objects



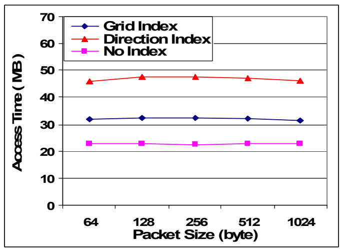
(b) Tuning time vs. # of objects



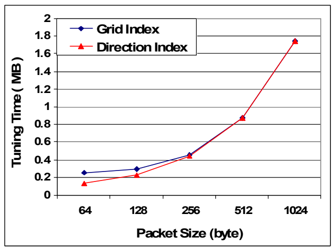
(c) Access time vs. # of queries



(d) Tuning time vs. # of queries



(e) Access time vs. packet size



(f) Tuning time vs. packet size

Figure 9. Study on access time and tuning time for the proposed indexes.

As discussed in Section 4, the lower-level index in the *DI* does not include type 9 queries, which leads to a smaller lower-level index when compared to the *GI*. However, because of the middle-level index, the total size of the *DI* is still larger that the total size of the *GI*, which explains why the *DI* needs longer access time.

We also study the tuning time of the two indexes when varying the number of objects. As shown in Fig. 9.b, the tuning times of both indexing schemes increase linearly with the increase of the number of objects, with the *GI* experiences a steeper slope. This is due to the fact that an object needs to download more queries in *GI*.

In Fig. 9.c and 9.d, we study the effect of varying the number of queries. Fig. 9.c shows the result on access time. When the number of queries is still small, *GI* is better than *DI* on access time. However, when the number of queries increases, *GI* gradually loses its advantage over *DI*. This can be explained as follows. In one broadcast cycle, there are two components to be broadcast: the index and the data, and the size of the data component is much larger than the size of the index. Although the size of the *GI* is smaller than the size of the *DI*, when the number of queries increases, the size of the data component also increases and offsets the advantage brought in by the smaller index size.

In Fig. 9.d, we can see the tuning time for the *GI* is very sensitive to the number of queries, while the tuning time for the *DI* only increases slowly with the increases in the number of queries. This is because when there are more queries, there are on average more queries to be monitored for each cell. This translates into more queries to be downloaded in *GI* when an object switches cell. But for *DI*, this problem is not very severe since only a subset of the monitoring queries for a cell need to be downloaded, which explains why the curve for *DI* only increases slowly.

For a system using broadcast channels, the size of a packet is important to the performance. We study the effect of packet size by varying the packet size from 64 bytes to 1024 bytes. The results are presented in Fig. 9.e and 9.f. In Fig. 9.e, it shows that the access times for three indexes only change slightly when the packet size varies. This is due to the fact that the access time is dominated by the total length of one broadcast cycle, and the effect of packet size on the total cycle length is almost negligible.

In Fig. 9.f, it demonstrates that the tuning times for both indexes increase quickly with the increase in packet size. When the packet size is the smallest, the system has the shortest tuning time. This is expected due to the overhead associated with using larger packets over smaller packets. One interesting finding is that the *DI* requires shorter tuning time than the *GI* when the packet size is small; but the tuning times for the two indexes converge when the packet size becomes larger. That is because more queries can be fit in a larger packet; and when an object downloads one packet, it downloads more queries than it actually needs. This cancels the advantage of using *DI*.

In conclusion, when there are a large number of queries and/or objects, *DI* is a better index than *GI*, because *DI* requires much shorter tuning time, and only demands slightly longer access time.

7. Conclusion

In this paper, we proposed to use periodic broadcast to reduce the processing cost of moving queries over moving objects. We designed two indexing schemes for the broadcast environment. The simulation results indicate that the proposed solutions achieve 30% to 60% savings over MobiEyes. Our future work includes extending this hybrid approach to address more complex queries.

8. References

1. W. Wu, W. Guo, K. L. Tan: Distributed Processing of Moving K-Nearest-Neighbor Query on Moving Objects. In *IEEE ICDE* 2007.
2. B. Gedik and L. Liu: MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *EDBT* 2004.
3. F. Liu, T.T. Do, K.A. Hua: Dynamic Range Query in Spatial Network Environments. In *DEXA* 2006.
4. F. Liu, K.A. Hua, T.T. Do: A P2P Technique for Continuous k-Nearest-Neighbor Query in Road Networks. In *DEXA* 2007: 264-276
5. K. Mouratidis, M. Hadjieleftheriou, D. Papadias, Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *SIGMOD* 2005.
6. K. Mouratidis, M.L. Yiu, D. Papadias, N. Mamoulis: Continuous Nearest Neighbor Monitoring in Road Networks. In *VLDB* 2006: 43-54.
7. H. Cho, C. Chung: An Efficient and Scalable Approach to CNN Queries in a Road Network. In *VLDB* 2005: 865-876.
8. X. Yu, K. Q. Pu, and N. Koudas: Monitoring k-nearest neighbor queries over moving objects. In *ICDE* 2005.
9. T.T. Do, F. Liu, K.A. Hua: When Mobile Objects' Energy Is Not So Tight: A New Perspective on Scalability Issues of Continuous Spatial Query Systems. In *DEXA* 2007.
10. K. Prabhakara, K.A. Hua, J.H Oh: Multi-Level Multi-Channel Air Cache Designs for Broadcasting in a Mobile Environment. In *ICDE* 2000: 167-176
11. T. Imielinski, S. Viswanathan, B. R. Badrinath: Data on Air: Organization and Access. IEEE *TKDE* 9(3): 353-372 (1997)
12. B. Zheng, J. Xu, W.C. Lee, D.L. Lee: Energy-Conserving Air Indexes for Nearest Neighbor Search. In *EDBT* 2004: 48-66
13. C.H. Chu, H.P. Hung and M.S. Chen: Variant Bandwidth Channel Allocation in the Data Broadcasting Environment. In *MDM* 2007.
14. W.C. Lee, B. Zheng: DSI: A Fully Distributed Spatial Index for Location-Based Wireless Broadcast Services. In *ICDCS* 2005: 349-358
15. J. Xu, B. Zheng, W.C. Lee, D.L. Lee: Energy Efficient Index for Querying Location-Dependent Data in Mobile Broadcast Environments. In *ICDE* 2003.
16. J. Cai, T. Terada, T. Hara, S. Nishio: An Adaptive Control Method in the Hybrid Wireless Broadcast Environment. In *MDM* 2007.
17. G. Trajcevski, H. Ding, P. Scheuermann and I. Cruz: BORA: Routing and Aggregation in Distributed Moving Objects Databases. In *MDM* 2007.
18. H. Kopetz, W. Ochsenreiter: Clock Synchronization in Distributed Real-Time Systems. *IEEE Trans. Computers* 36(8): 933-940 (1987).
19. B. Sterzbach: GPS-based Clock Synchronization in a Mobile, Distributed Real-Time System. *Real-Time Systems* 12(1): 63-75 (1997)