Distributed Processing of Moving K-Nearest-Neighbor Query on Moving Objects

Wei Wu Wenyuan Guo Kian-Lee Tan
School of Computing
National University of Singapore

Abstract

A moving k-nearest-neighbor (MKNN) query is a continuous k-nearest-neighbor (KNN) query issued by a moving object. As both the query owner and other mobile objects are moving, the influenced area (i.e., cells in the cellular networks), and query result of a MKNN query change with time. Existing processing techniques for MKNN queries are all centralized approaches which rely on the location update messages from moving objects. However, these approaches typically employ complex data structures and algorithms. Moreover, the server may not be able to cope with a high location report rate which is necessary to ensure accurate and correct answers. In this paper, we propose a distributed strategy to process MKNN queries in real-time. In our scheme, called disMKNN, the server and moving objects collaborate to maintain the KNN of a MKNN query. While the server keeps track of a MKNN query's influenced cells, moving objects within the cells monitor their own relationships (i.e., whether they are part of the KNN answers) to the query. Results of an extensive performance study show the effectiveness of disMKNN.

1. Introduction

Continuous spatial queries on moving objects are important for region monitoring and location based services. k-nearest-neighbor (KNN) query is one of the most important kinds of spatial queries. Thanks to the positioning system (e.g. GPS) and wireless communication technologies (e.g. cellular networks), moving objects can report their locations to a server, and even issue a query on the move. If the query is a continuous query and is related to the initiating object's location, this query is a moving query.

A moving k-nearest-neighbor (MKNN) query is a continuous k-nearest-neighbor query issued by a moving object. Due to the movement of both the query owner and other moving objects, the influenced area (i.e., cells in the cellular networks) and query result of a MKNN query change with time. Existing processing techniques for MKNN queries are all centralized approaches which rely on the location update messages from moving objects [10, 9, 8, 14, 13, 6, 5, 7, 4]. These approaches typically employ complex data structures and algorithms. Moreover, the rate of location update has great impact on system load and query result accuracy. If the location report rate is high, the server will receive large amount of messages and may not be able to process them fast enough; however, if the location report rate is low, the query result may be wrong, because during the time interval between an object's two consecutive location updates, the server does not have accurate location information about the object.

In this paper, we propose a distributed MKNN processing scheme, called disMKNN, where the server and mobile objects cooperate to process MKNN queries in real-time. disMKNN is motivated by two key observations. First, a MKNN query influences only one or several adjacent cells (in the cellular networks) that are changing over time. Second, the objects in the influenced cells can determine their relationships with respect to the query owner if they know the positions and velocity vectors of the query owner and the kth result object of the query. (We assume every object knows its location and velocity.) In other words, given these information, an object can calculate its real-time distance to the query owner, and the real-time distance between the query owner and the query's kth result object. Comparing the two distances, it knows whether it is changing the query result. Thus, in disMKNN, the server's role is to find out and update the cells that are influenced by each query and keeps the objects in these cells informed about the positions and velocity changes of the query owner and the kth result object. On the other hand, the moving objects in each influenced cell report to the server when they find that they are changing the query result.

disMKNN has several advantages. First, the processing is distributed between the server and the moving objects. In fact, the server does not need to employ complex data structures and algorithms. In addition, moving objects no longer need to report their locations to the server periodically. Second, only objects in the influenced cells of a query need to participate in its processing.

1-4244-0803-2/07/$20.00 ©2007 IEEE

Distributed processing of spatial queries on moving objects has been studied in the literature [3, 2, 11]. However, these works all deal with continuous range queries, and none of the proposed techniques applies to moving KNN queries. MobiEyes [3] processes a moving range query on moving objects by letting the objects that are near to the query's monitoring region calculate their distances to the query owner. MQM [2] partitions the workspace into rectangular sub-domains. Depending on a moving object's capacity, the server assigns a number of sub-domains as its resident domain, and the object is made aware of the range queries intersecting its resident domain. When the object moves out of its resident domain, it requests another resident domain from the server. In [11], the entire service area is partitioned into a set of service zones and the method for dynamically allocating the service zones to a number of servers is discussed.

To the best of our knowledge, this is the first reported work investigating the distributed processing solution for MKNN queries. The rest of the paper is organized as follows. Section 2 describes the overview of the system. Detailed distributed MKNN scheme is given in Section 3. In section 4 we report the experimental results, and finally, we conclude in section 5.

2. System Overview

Figure 1 depicts the system architecture. The system uses the cellular network infrastructure for communication, and consists of a centralized server, a number of stationary base stations and a large number of moving objects. The server mediates the distributed processing of the MKNN queries in the system. Base stations are connected to the server. Each base station covers a region (i.e., a cell in the cellular networks) which is determined by its transmission range. The cells not only provide a way of wireless communication between moving objects and server, but also are natural boundaries that divide the moving objects into subsets. Base stations do not particpate in any processing; they just relay messages between moving objects within their cells and the server. We assume that the server knows each cell's covered area, that is, the location and transmission range of each cell's base station.

In the system, every object registers with a base station in the cellular networks. Every object has an unique object ID, and each query has a unique query ID. When an object issues a continuous KNN query, it tells the server the query ID and the value of k. To simplify the discussion, we assume that an object submits at most one continuous KNN query to the system, thus we can use q_i as the ID of the query issued from object o_i.

When a moving object reports its location to the server, its location is represented as (i, p_i, v_i, ts_i), where i is the object's ID, ts_i is the time point when these information is recorded, p_i is its location at time point ts_i, and v_i is its velocity at time point ts_i.

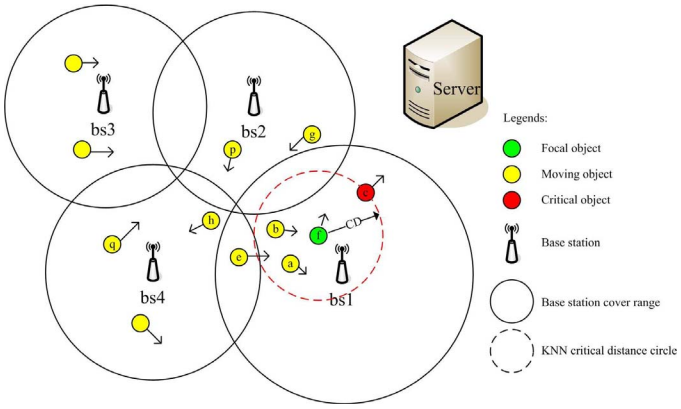


Figure 1. System Model

The server in the system lets the moving objects monitor a MKNN query by informing them the locations and velocity information of the query's owner object and the kth result object. An object determines whether it is changing the result by calculating and comparing its distance to the query owner object and the distance between the given two objects. Since the MKNN queries are processed in this distributed manner, the server does not monitor all objects' locations, and the objects do not need to report their locations to the server periodically.

In this paper we call a query owner object as a *focal object*, and the kth result object of a query as a *critical object*. The locations and velocity information of focal objects and critical objects are used in real-time query monitoring, thus if an object is a focal object or a critical object, it sends a message to the server when its velocity has changed. For an object that is neither focal or critical, it sends a message to the server only when it finds itself is changing a query's result or has moved into a new cell. An object needs to send a message to the server when it moves into a new cell because the queries it should monitor is related the cell that it is in.

A moving object issues a continuous KNN query to the server via the base station it registers with. Receiving the MKNN query, the server begins the distributed processing procedure of this query. The whole processing procedure is divided into two phases: an *initial process* phase and a subsequent *continuous process* phase. In the initial process phase, the server finds the initial result set for the KNN query. After the initial process, the continuous process procedure is launched, and this procedure will keep the query result updated every time when the motion of objects causes query result change.

The initial query result of a MKNN query is found by broadcasting the query owner's location to the nearby ob-

jects (in nearby cells). When a moving object o_i issues a MKNN query q_i, the server knows which cell this initiating object is in, because o_i submits the query through the base station that it registers with. We denote the cell where o_i is in as $cell(o_i)$. To find the initial query result for q_i, the server lets the base station of cell $cell(o_i)$ broadcast o_i's location so that each object in $cell(o_i)$ can reply with its distance to o_i. If there are fewer than k objects in $cell(o_i)$, o_i's location is broadcast to the neighboring cells of $cell(o_i)$ until k neighbors of o_i is found. The kth shortest distance among the relies is used to determine the cells that may have nearer objects. These cells are checked to find out the true query result for query q_i. We denote the query result of query q_i as $RS(q_i)$, and define the distance between o_i and its kth NN as the *critical distance* of this query, noted as $CD(q_i)$.

After the initial KNN result of q_i is discovered, $CD(q_i)$ is used to determine the set of cells that need to monitor this query. We call these cells as the *influenced cells* of query q_i, and are notated as $IC(q_i)$. The *influenced cells* of a query is the cells that intersect with the circle determined by the query owner object's location and the critical distance of the query (this circle is called as the *critical circle* of q_i). The objects in these cells may change the query result. The server lets the objects in $IC(q_i)$ monitor the query by telling them the locations and velocity vectors of the query owner and the Kth result object. When an object finds that its distance to the query owner is the same as the present critical distance, it reports this event to the server. Then the server will conduct the necessary operation to let the distributed monitoring process continue.

To facilitate the detailed discussion of the system, we define some notations (as we used in the description above). They are listed in table 1.

We note that the locations of a query's focal object and the critical object are changing with time, and this is captured with their (p, v, ts) vectors. When we use p_i, we sometime refer to the position of o_i at time ts_i, e.g. in (p_i, v_i, ts_i), and in other times refer to the current position of o_i, e.g. in $circle(p_i, CD(q_i))$. We hope the reader can distinguish the meaning from the context. We also want to point out that $RS(q_i)$, $CD(q_i)$, $IC(q_i)$ also change with time. Unlike p_i, when we use $RS(q_i)$, $CD(q_i)$, $IC(q_i)$, we always refer to their values at the present time.

Table 1. Notations

Notation	Meaning
o_i	a moving object with unique ID i.
p_i	position of object o_i, i.e. (x, y).
v_i	velocity vector of o_i, i.e (v_x, v_y).
ts_i	time point when o_i reported its location and velocity vector.
$bs(o_i)$	the base station that o_i registers with.
$cell(o_i)$	the cell where o_i is, i.e. the cell that $bs(o_i)$ covers.
q_i	a MKNN query from moving object o_i.
$RS(q_i)$	result set of query q_i.
$dis(o_i, o_j)$	the distance between object o_i and o_j.
$CD(q_i)$	critical distance for query q_i. Defined as $Max\{dis(o_i, o)\|o \in RS(q_i)\}$.
$circle(p, r)$	a circle with center point p and radius r.
$CriCircle(q_i)$	the critical circle of query q_i, defined as $circle(p_i, CD(q_i))$.
$IC(q_i)$	the influenced cells of query q_i. Defined as the cells that intersect with $circle(p_i, CD(q_i))$.
$neighbors(cell)$	the set of neighboring cells of the given cell(s).

3. Distributed MKNN Processing

3.1. Initial Processing

The purpose of the initial process procedure is to find the initial result set of a newly issued continuous KNN query. This process will also gather necessary information for the subsequent continuous processing of this query. We assume that the message delay in a query's initial processing phase is short enough to be ignored.

The idea in this step is that if we find any k neighbors of o_i and have their corresponding distances to o_i, we are sure that the longest distance among them must be equal or larger than the distance from the o_i's true kth nearest neighbor. So, we can take the longest distance from any k neighbors of o_i as a tentative critical distance, and then this distance can be used to confine the search region for finding the true KNN of o_i. Given such a distance d and the complete information of cells, the server can find out the cells that intersect with the circle $circle(p_i, d)$. Only these cells may contain objects whose distance to o_i is less than d. By searching these cells, we can find the true KNN for this query.

Here we see that any distance that is larger than the true critical distance can be a tentative distance. However, we do not know the true critical distance until we find the true KNN result. And since the value of the tentative distance determines the covered region of $circle(p_i, d)$, which determines the number of cells that we need to check to find the true KNN, a good tentative distance helps to reduce the number of cells we need to check, and thus helps to reduce the number of communication messages.

To have a good tentative distance, we begin looking for any k neighbors of o_i from the cell $cell(o_i)$, the cell where o_i is in, because the objects that are in the same cell as o_i

are more likely to be nearer to o_i than those in other cells.

Since the objects register with base stations, so each base station knows the number of objects in its cell. The server knows the number of objects in each cell by asking the base stations once and maintains the information incrementally: recall that in the system when an object moves to a new cell, it needs to send a message to the server (with its previous cell and current cell).

If k or more than k objects (exclusive o_i) are in $cell(o_i)$, we only need to check the objects in $cell(o_i)$ to have a tentative critical distance; if fewer than k objects (exclusive o_i) are in $cell(o_i)$, the neighboring cells of $cell(o_i)$ are checked. This process can be summarized as follows: $cell(o_i)$ is first checked to find k neighbors of o_i, if fewer than k neighbors are found, the neighboring cells of the checked cell(s) are checked until k neighbors are found. Note the purpose of finding k neighbors of o_i here is to get a good tentative critical distance. This tentative distance will be used to find the true KNN of o_i.

To "check" the objects's distance to o_i in a cell, the server lets the base station in that cell broadcast a message so that the objects in that cell may reply. Here we define four kinds of messages that are used in the initial process.

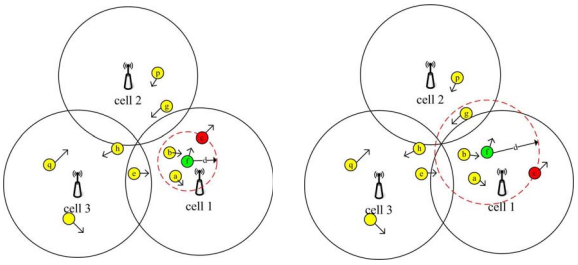
- *o2sInstallQuery(oid, qid, k, location, velocity)*: an object sends this *o2sInstallQuery* message to the server when it submits a new query to the system. The attached information in the message is the ID of this object (oid), the ID of this query (qid), the value of k, and this object's current location and speed velocity.

- *s2oNewQuery(qid, location, distance)*: the server lets the base station in a cell broadcast this *s2oNewQuery* message to the objects in that cell when the server wants to know the distances from these objects to the given location. The attached information is a query ID, the location of the query owner, and a distance value.

- *o2sReply(oid, qid, location, velocity)*: the object receiving a *s2oNewQuery* message will reply this *o2sReply* message to the server (via the base station) if its distance to the *given location* is less than the *given distance*.

- *s2oQueryResult(qid, resultSet)*: after the server finds out the initial result set for a MKNN query, it sends the query result to the query owner object via the base station with which the owner object registers with.

A moving object issues a MKNN query by sending a *o2sInstallQuery* message to the server. Receiving a new query, the server finds out a tentative critical distance by letting base station(s) broadcast *s2oNewQuery* message and receiving *o2sReply* messages. During the process of finding k neighbors of o_i, the distance field in the *s2oNewQuery* messages is set to ∞ so that every object receiving the *s2oNewQuery* message will reply.



(a) $IC \subseteq Checked$ (b) $IC - Checked \neq \emptyset$

Figure 2. Find true KNN.

After finding k or more than k neighbors of the query owner o_i, the tentative distance is picked as follows: for all the reply messages received, the server calculates the distance from each replying object to o_i, and then sorts the distances and picks the kth distance. This is the best tentative distance the server finds from the objects in the checked cell(s). Having a good tentative distance, the server begins to find out the true KNN result of the query.

Using the tentative distance d, the server finds out the cells that intersect with $circle(p_i, d)$. Only the cells that intersect with $circle(p_i, d)$ may have objects that are nearer to o_i than d, and once we find these objects, we find the true KNN query result.

Since we have checked some cells in the process of finding the tentative critical distance and every object in these cells reply, this time we only need to check the cells that intersect with $circle(p_i, d)$ but have not been checked yet. Thus if the cells that intersect with $circle(p_i, d)$ have all been checked, we are sure that the k nearest neighbors to o_i from the checked cells are the true KNN result, otherwise, the cells that have not been checked need to be checked to find possible nearer objects. To check these cells, the server lets the base station in these cells broadcast a *s2oNewQuery* message. Since we have had a tentative distance d and we are only interested in the objects whose distance to o_i is less than d, this time we set the *distance* field in the *s2oNewQuery* to d such that only the nearer objects will reply. If no reply message is received, it means that no object is nearer than the tentative critical distance, and the tentative KNN is the true KNN, otherwise, nearer objects are found and the true KNN is the new top k answers. After this step, the server is sure that the found KNN result is the true initial result of the query, because the true critical distance cannot be larger than the tentative one. The server then sends the initial query result to o_i by a *s2oQueryResult* message.

Figure 2 shows two examples of the initial process. In both examples (a) and (b), object o_f issues a moving 3NN query, 3 neighbors are found in $cell(o_f)$, and the tentative

critical distance is marked as d. The $cirlce(p_f, d)$ is used to find the cells where nearer objects to o_f may exist. In (a), the only cell that intersects with $cirlce(p_f, d)$ is cell 1, which is $cell(o_f)$ and has been checked, so the 3 neighbors that have been found are the true result of the query. In (b), the critical circle intersects not only cell 1, but also cell 2 and cell 3, and since cell 1 has been checked, cell 2 and cell 3 need to be checked because they may have objects whose distances to o_f is less than d. The server lets the base stations in cell 2 and cell 3 broadcast a *s2oNewQuery* message with distance filed set to d. Only object o_g in cell 2 will reply, because only its distance to o_f is less than d. After receiving the reply message from o_g, the true initial 3NN result is found.

One routine in the initial process procedure is to find the cells that intersect with a given circle – the circle determined by the location of the query owner and a distance to it. As we shall see, this routine is also a regular operation in the continuous processing phase. Since it is a regular operation, we must have an efficient algorithm for this task.

One important observation here is that if a cell intersects with a circle $circle(p, r)$, then this cell is either the cell where location p is located in, or it must be a neighbor of one or several other cells that intersect with the circle. (Here we assume that the cells cover the whole area, i.e. there is no region that is not covered by any cell.) So, to find the cells that intersect with a circle $circle(p, r)$, we only need to test the neighboring cells of the found cells that intersect with this circle. Since we know that the cell where p is in must intersect with the circle that centers at p, we can take it as a beginning cell and then test its neighboring cells and the neighboring cells of found intersecting cells. When no neighboring cell of the found intersecting cells intersects with the circle, we can be sure that we have found all the cells that intersect with the circle.

From the idea described above, we see that knowing the neighboring cells of each cell is important for finding the cells that intersect with a given circle. Since the cells configuration is static, the server does a simple pre-processing to find all neighboring cells for each cell and store them in a hash table.

To test whether a cell intersects with a given circle $circle(p, r)$, we compute the distance between the point p and the location of the base station of that cell. If that distance is less than $(r + R)$ where r is the radius of the circle, and R is the transmission range of that cell, we say the cell intersects with the circle, otherwise, it does not intersect with the circle.

The algorithm for finding the cells that intersect with a circle $circle(p, r)$ is given in Figure 3.

The whole initial process of a MKNN query q_i can be summarized as follows: from the cell where the query owner o_i is located in (and its neighboring cells if necessary), we find a tentative KNN set of o_i, then the distance d between o_i and the kth NN in the tentative KNN set is used to determine the cells we need to check to find the true KNN of o_i. The cells that intersect with $circle(p_i, d)$ but have not been checked for q_i yet will be checked to finalize the true KNN result. This KNN result of q_i is the initial KNN result of the MKNN. The algorithmic description of this process is presented in Figure 4.

FindIntersectingCells(C, $startCell$, p, d)
// Input=C: cells configuration,
$startCell$: the cell where p is, p: circle center point,
d: circle radius
// Output: a set of cells that intersect with $circle(p, d)$

1. $IC = \{startCell\}$ // intersecting cells
   $ctc = neighbors(startCell)$; // cells to check
   $cc = \{startCell\}$; // cells checked
   $cnf = \emptyset$; // intersecting cells newly found
2. While ($ctc \neq \emptyset$)
3. For each cell in ctc
4. If $intersect(cell, p, d)$ put cell to cnf
5. End For
6. If $cnf \neq \emptyset$
7. $IC = IC \cup cnf$
8. $cc = cc \cup ctc$
9. $ctc = neighbors(cnf) - cc$
10. End If
11. End While
12. RETURN IC

Figure 3. Find cells intersecting with a circle

After the initial process of a query q_i, the server has discovered the initial query result for the query. In addition, it has collected the following information that are needed for the subsequent continuous processing phase. These information will be stored in a data structure called Query Table.

- (q_i, k): query ID and value of k.
- (i, p_i, v_i, ts_i): query owner object ID i, position and velocity vector of o_i, and the time stamp of these information.
- $RS(q_i)$: the initial result set of this query.
- (c, p_c, v_c, ts_c): the ID, position, velocity of the critical object of q_i, and the time stamp of these information.
- $IC(q_i)$: the influenced cells of this query.

3.2. Continuous Processing

The continuous processing phase of a query is launched once the query's initial processing is completed. The pur-

InitialProcess(C, q_i, K, $cell(o_i)$, p_i)
// Input=C: cells configuration, q_i, K, $cell(o_i)$, p_i

1. $ctc = \{cell(o_i)\}$; // cells to check
   $cc = \emptyset$; // cells checked
   $cjc = \emptyset$; // cells just checked
   $replies = \emptyset$; $numReplies = 0$; $cd = \infty$;
   $IC = \emptyset$ // intersecting cells
2. While ($numReplies < K$)
3. let the base station in each cell in ctc broadcast $newQuery(q_i, p_i, cd)$ message.
   (each time a reply message for q_i is received, put it to $replies$, and increase $numReplies$ by 1)
4. $cc = cc \cup ctc$.
5. $cjc = \emptyset \cup ctc$.
6. (after time out)
   If ($numReplies < K$)
7. $ctc = neighbors(cjc) - cc$
8. End If
9. End While
10. sort the $replies$ and retain top K
11. cd=Kth distance; $numReplies = K$
12. IC = FindIntersectingCells(C,p_i,cd)
13. $ctc = IC - cc$
14. If ($ctc \neq \emptyset$)
15. let the base station in each cell in ctc broadcast $newQuery(q_i, p_i, cd)$ message.
    (each time a reply message for q_i is received, put it to $replies$, and increase $numReplies$ by 1)
16. (after time out)
    If ($numReplies > K$)
17. sort the $replies$ and retain top K
18. cd=Kth distance;
19. IC = FindIntersectingCells(C,p_i,cd)
20. End If
21. End If

Figure 4. Initial Process

pose is to keep the query result updated when the movement of the objects cause a change to the query result. This is done incrementally.

The idea employed in the distributed continuous processing of a query is that given an object o_i's position and velocity, an object o_j who knows its own position and velocity can monitor its distance to the given object o_i. Thus, for any query, if an object knows the query owner's location and velocity, and this query's critical object's location and velocity, it knows both its distance to the query owner object and the present critical distance. Knowing the two distances, the object knows whether it is changing the query result.

For an object that is in the query result, if its distance to the query owner becomes larger than the current critical distance, then it is becomes the new critical object of the query. For an object that is not in the query result, if its distance to the query owner becomes smaller than the current critical distance, then it becomes one of the query result, and is the new critical object of the query. In both cases, the critical object of the query is changed.

Therefore, to make the objects monitor a query by themselves, what the server needs to do is to keep the objects informed about positions and velocity vectors of this query's owner object and critical object.

To reduce the workload at the object side, an object only monitors a query when necessary. In other words, for each query, we should minimize the objects that monitor the query but ensure that we do not miss any object that may change the query result. The idea used here is similar with the one used in the initial process phase: for a query q_i, only the cells that intersect with the circle $circle(p_i, CD(q_i))$ may have objects that may change the query result, so we only need to let the objects in these cells monitor the query.

The challenge here is that both the center point of the circle and radius of the circle may be changing because the query owner and critical object may be moving. Their movement changes the area covered by the critical circle, which may lead to the changes of the set of cells that intersect with the circle. To make sure that every object that may change the query result is monitoring the query, the server needs to keep updating the IC for each query. Since the critical circle is changing with time (due to the movement of query owner and critical object), it seems that the server needs to continuously calculate the cells that intersect with the current critical circle, for each query, which is an "impossible mission". Fortunately, given the positions and velocity vectors of the query owner and of the critical object, the server is able to determine the time point when the IC of this query will change. Once this time point is computed out, it can be stored as a trigger. This is done as follows. Given an object o_i's information (p_i, v_i, ts_i) where ts_i is the time stamp when the information was obtained, the location of this object at time t is $(p_i.x + v_i.x * (t - ts_i), p_i.y + v_i.y * (t - ts_i))$, so its distance to any cell at time t can be represented as a function of t. Suppose the location of the base station of a cell is at (x_b, y_b), then the distance from o_i to the base station at time t is written as:

$$\sqrt{(p_i.x(t) - x_b)^2 + (p_i.y(t) - y_b)^2} \quad (1)$$

where

$$p_i.x(t) = p_i.x + v_i.x * (t - ts_i) \quad (2)$$

$$p_i.y(t) = p_i.y + v_i.y * (t - ts_i) \quad (3)$$

Similarly, given the location and velocity (p_c, v_c, ts_c) of critical object o_c, the critical distance at time t is:

$$\sqrt{(p_i.x(t) - p_c.x(t))^2 + (p_i.y(t) - p_c.y(t))^2} \quad (4)$$

Suppose the transmission range of the cell is R, at the time point when the cell changes from intersecting to non-intersecting (or vice versa) with the critical circle , we have the following equation.

$$\sqrt{(p_i.x(t) - p_c.x(t))^2 + (p_i.y(t) - p_c.y(t))^2} = \sqrt{(p_i.x(t) - x_b)^2 + (p_i.y(t) - y_b)^2} - R$$

We can turn this into a quartic equation about variable t. Quartic equation can be solved in constant time. If the quartic equation has real number root(s), the most nearest future one is the value of t that we are looking for. Once the value of t is computed out, the server knows when the cell will change its intersecting status with the critical circle. At time t, the server needs to update IC.

Note the value of t depends on the velocity vectors of query owner and critical object, thus when the velocity vector of any of them changes, the previously computed t is obsolete, and the server needs to re-compute the value of t.

It is important to note that only the cells in the current IC can become non-intersecting with the critical circle, and only the neighboring cells of cells in the current IC may become intersecting with the critical circle. So we can update IC incrementally. We only compute the value of t for the cells in IC and their neighboring cells. This is very important for improving the server performance and achieving real-time update of IC. If we compute the value of t for each cell, most of the computation is wasted when the velocity of the query owner or critical object changes, because all the previously computed values of t for the query become obsolete.

We compute the value of t for the cells in IC and their neighboring cells, because we want to update the IC in real-time. The process works as follows: after the initial process of a query q_i, the server computes the value of t for each cell in $IC(q_i)$ and $neighbors(IC(q_i))$. This gives us an ordered set of time triggers. If the velocity of query owner or critical object changes, the server re-computes the triggers. When the earliest trigger time arrives, the server updates $IC(q_i)$ accordingly, and this also leads to a change of $neighbors(IC(q_i))$, then for the changed elements in $IC(q_i)$ and $neighbors(IC(q_i))$, the server computes the trigger time for them.

At the moving object side, given the updated information (p_i, v_i, ts_i) and (p_c, v_c, ts_c) of a query q_i, it always knows whether it is in the result set of this query (by comparing its distance to o_i, and the distance between o_i and o_c) and whether it is changing the result set. These distances can be monitored continuously. Like the trick we used on the server for updating $IC(q_i)$, the object can compute the time when it will change the query result, by solving the following equation (let (p_j, v_j, ts_j) denote the position and velocity of this object).

$$\sqrt{(p_i.x(t) - p_c.x(t))^2 + (p_i.y(t) - p_c.y(t))^2} = \sqrt{(p_j.x(t) - p_i.x(t))^2 + (p_j.y(t) - p_i.y(t))^2}$$

This can be turned into a quadratic equation and solved in constant time. The value of t computed from this equation depends not only the velocities of o_i and o_c, but also the object's own velocity. The value of t should be updated if any of the velocities changes.

In the cellular networks, an object moves from cell to cell. In this distributed MKNN processing system, the queries an object needs to monitor is determined by the cell it is in, because the server assign queries to cells rather than individual objects. Thus when an object moves from one cell to another cell (it detects this event by looking at the base station ID it registers with), it should send a message to the server so that the server can tell the difference between the monitoring queries of the two cells.

So far, we have described the idea of the continuous processing phase of a MKNN query q_i, and how the server keeps updating $IC(q_i)$ (only the objects in the cells in $IC(q_i)$ need to monitor the query), and how a moving object monitor a query. Now we present the messages we defined for the continuous processing phase. They are:

- *s2oMonitorQuery(qid,(oid, location, velocity, ts),(oid, location, velocity, ts))*: the server lets the base station broadcast this message when it wants objects in a cell to monitor a query. Query ID, position and velocity vector of both focal object and critical object are given.
- *o2sFocalVelocity(oid, location, velocity, ts)*: when a query owner object changes its velocity, it sends this message to the server.
- *s2oFocalVelocity(qid, location, velocity, ts)*: receiving an *o2sFocalVelocity* message from a focal object, the server lets the base station of each cell in $IC(qid)$ broadcast this *s2oFocalVelocity* message to update the focal object information at the moving objects.
- *o2sCriticalVelocity(oid, location, velocity, ts)*: when a critical object changes its velocity, it sends this message to the server.
- *s2oCriticalVelocity(qid, location, velocity, ts)*: similar to *s2oFocalVelocity*.
- *o2sCriticalObjectChange(oid, qid, location, velocity)*: when an object finds that it is becoming the new critical object of a query, it sends this message to the server.
- *s2oCriticalObjectChange(qid, oid, location, velocity, ts)*: when receiving an *o2sCriticalObjectChange* message from an object, the server lets the base

station of each cell in $IC(qid)$ broadcast this *s2oCriticalObjectChange* message to update the critical object information at the moving objects.

- *o2sEnterCell(oid, cid, cid)*: when an object finds that it has moved from one cell to another cell, it sends this *o2sEnterCell* message to the server.

- *o2sRemoveQuery(qid)*: when a focal object wants to remove its continuous KNN query from the system, it sends this o2sRemoveQuery message to the server.

- *s2oDeleteQuery(qid)*: when the server wants the objects in a cell stop monitoring a query, it lets the base station in that cell broadcast a s2oDeleteQuery message so that the moving objects receiving this message will remove this query from their local monitoring query list.

The whole continuous processing procedure works as follows: after the initial processing phase of a query q_i, the server broadcasts (via base station) a *s2oMonitorQuery* message to the cells in $IC(q_i)$. Receiving the *s2oMonitorQuery* message, the objects begin monitoring this query. At any time when the focal object changes its velocity, it sends an *o2sFocalVelocity* message to the server, and the server broadcasts a *s2oFocalVelocity* message to the cells in $IC(q_i)$. The same thing happens with the velocity change of critical object. At any time when a moving object finds it has become the new critical object of the query, it sends an *o2sCriticalObjectChange* to the server, and then the server broadcasts a *s2oCriticalObjectChange* to the corresponding cells. The focal object updates its query result also by receiving the *s2oCriticalObjectChange* message (the query result is updated incrementally). During the whole continuous processing phase, the server keeps updating $IC(q_i)$ as we described before. When a cell is removed from $IC(q_i)$, a *s2oDeleteQuery* message is broadcast to that cell, because the objects in that cell do not need to monitor the query any more; when a cell is added into $IC(q_i)$, a *s2oMonitorQuery* message is broadcast into that cell, because the objects in that cell may become new critical object of q_i. When an object moves from a cell to another cell, it sends an *o2sEnterCell* to the server. Upon receiving the *o2sEnterCell* message, the server finds the difference between the monitoring queries of the two cells, and then sends (via point-to-point communication) the corresponding *s2oDeleteQuery* and *s2oMonitorQuery* to the object.

Due to the space limitation, we do not elaborate on the problem of handling simultaneous events here. Interested readers may refer to a longer version of this paper [12].

Table 2. System Parameters

Parameter	Default	Range
Number of objects (*no*)	50K	10,20,50,70,100(K)
Number of queries (*nq*)	2K	1,2,5,7,10 (K)
Number of NNs (*k*)	8	2,4,8,16,32
Object/Query speed	slow	slow, middle, fast
Cell radius (*R*)	500	200,500,1000,2000

4. Experimental Study

To study the performance of this distributed MKNN processing scheme, we developed a simulator in Java and a server program in C++. The input of the Java simulator is the moving objects and moving KNN queries generated using the Network-based Generator [1]. The simulator replays the movement of the objects and processes the queries using our proposed distributed MKNN processing scheme. It records all the messages generated during the whole simulation. Then the object-to-server (uplink) messages are feed to the C++ server program to study the workload incurred at the server.

The metrics used in the performance study are: query result change rate, message rate, server workload (running time for processing the messages), and workload at moving object side (average number of queries each moving object monitors).

Since the proposed scheme is the first distributed MKNN processing scheme and the motivation of our scheme is that an efficient distributed processing scheme will have better query result accuracy and lower messages and server load than centralized solutions, we study the performance of our scheme and compare it with the state-of-the-art centralized processing algorithm. The CPM [9] method is selected for comparison, because it is shown in [9] that CPM outperforms all other centralized continuous KNN query processing algorithms. We use the source code of CPM provided by the authors of [9]. The same sets of moving objects and moving KNN queries generated by [1] are used in running our scheme and CPM.

The moving object datasets generated by [1] are the location update reports for each time step. The default velocity values (*slow, middle, and fast*) are used to generate datasets of moving objects with different speed. The Oldenburg map is used to generate the datasets. The generated data space width and height is 23572 and 26915. In our simulator, we generate cells that cover the area. Since we do not know the real number of cells used in Oldenburg, we take cell size as a parameter and study its effect on system performance.

The parameters under investigation are summarized in Table 2.

4.1. Query result change rate

In this set of experiments, we show the average number of query result changes in each time step that our system captures in different datasets. The more frequently the query result changes, the more likely that the server in a centralized solution is giving the wrong query results. We use "query result change rate" to refer to the average number of times a query's result changes during one time step.

For a given set of MKNN queries, the following system parameters have effect on the query result change rate: number of moving objects, value of k, and moving speed of the objects. Among these factors, the moving speed of the objects has the largest impact.

Figure 5 depicts our observations. The more the data object population is (see the lines "slow (no=20k)" and "slow (no=50k)"), the more frequent the query result changes. The larger the value of k is, the more likely the query result will change. When the moving speed of the objects change from "slow" to "middle", the query result change rate increases significantly. As expected, for the "fast" scenario, the result change rate is very much larger than the "middle" case (> 250). As such we omit the results from the figure.

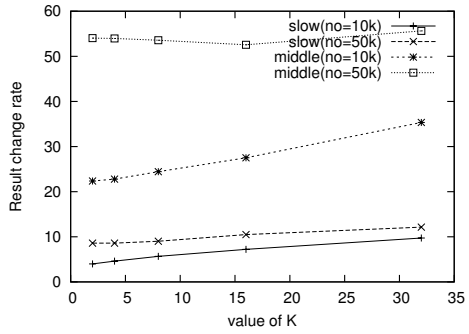
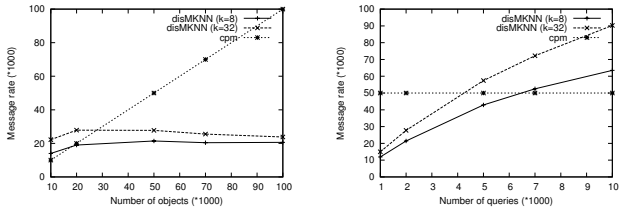


Figure 5. Query result changes.

This set of experiments shows that our disMKNN scheme can capture the frequent KNN query result changes, because the query monitoring is done in real-time, not based on periodical location report messages. This also exposes the potential problem of centralized moving KNN query processing solutions: if the moving objects do not send location update to the server very frequently, the server is likely to return the wrong KNN result, because the query result is based on the most recent location report message.

4.2. Messaging Cost

The purpose of this set of experiments is to investigate the effect of system parameters on the average number of messages that the server receives during each time step.



(a) Effect of object number (b) Effect of query number

Figure 6. Message rate.

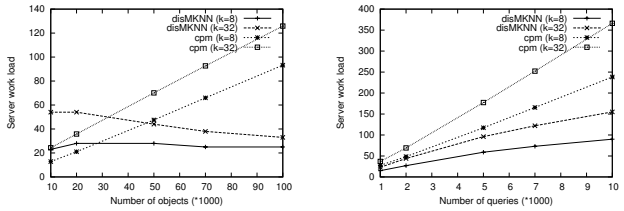
In CPM, the message rate is proportional to the number of objects because each object sends a location update message to the server in every time step. In our distributed processing solution, message is sent to the server only when necessary. Specifically, a message is sent to the server only when one of the following events happens: query result change, entering a cell, focal object velocity change and critical object velocity change. This means that in the proposed distributed processing scheme, the message rate is related to the number of queries and the query change frequency (the factors that affect query change rate is studied in section 4.1).

Figure 6 confirms our analysis. The message rate in CPM increases with the number of objects while the message rate in our disMKNN scheme increases with the number of queries. The value of k has impact on the message rate in disMKNN (because it has effect on query result change rate), but does not affect the message rate in CPM (thus only one "cpm" line is drawn).

4.3. Server Load

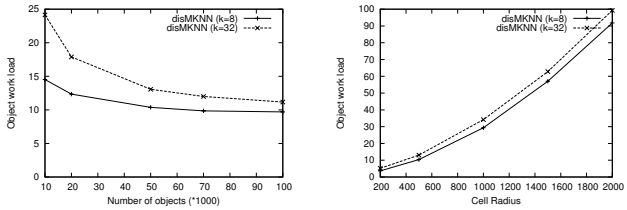
The task of the server in both CPM and disMKNN is to process the incoming message stream. We study the workload incurred by both schemes in this set of experiments. The metric we used here is the CPU time for processing the received messages. Note the workload is not only affected by the rate of messages, but also the operations that the server need to conduct for processing the messages.

Figure 7 shows the experiment results. From the figure, we see that number of objects has a big impact on the server workload in CPM but little impact on disMKNN, and number of queries and value of k have impact on the server workload in both schemes. And it is clear that disMKNN incurs much less workload at the server side (except under the situation where the number of objects is small). When the number of objects is small, the object density is low, thus the size of critical circle is larger, and more cells are in IC of each query. In this situation, the server in disMKNN spends more time on updating IC.



(a) Effect of object number (b) Effect of query number

Figure 7. Server workload.



(a) Effect of object number (b) Effect of cell size

Figure 8. Object workload.

4.4. Object Workload

In disMKNN, the moving objects help monitor the queries. Here the "object workload" is defined as the average number of queries that each moving object monitors. The effect of system parameters on object workload is studied in this set of experiments.

The number of queries an object needs to monitor is actually the number of queries that its registered cell is influenced by, i.e., the number of critical circles the cell intersects with. Besides the number of queries, which is an obvious affecting factor, object density (which is proportional to number of objects) and value of k also have impact on object workload. The effects of object density and value of k are shown in Figure 8(a). The higher the object density is, the smaller the critical circle is; the larger the value of k is, the larger the critical circle is. Cell size in the cellular networks also has great effect on object workload: the larger each cell is, the more critical circles it covers and intersects. The effect of cell radius is shown in Figure 8(b).

5. Conclusion

We have described a distributed strategy, disMKNN, for processing moving k-nearest-neighbor (MKNN) queries. In disMKNN, a server mediates the initial processing and the subsequent continuous distributed processing of the queries. Moving objects no longer need to send location update messages to the server periodically; instead, they monitor the MKNN queries. The proposed scheme achieves real-time processing of MKNN queries, and incurs much lower message rate and less workload on the server than centralized MKNN processing solutions do. It scales very well with the number of moving objects because the message rate and server load in the system are not affected by the object population much. Simulation study shows that the distributed MKNN processing solution is effective and efficient.

References

[1] T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2):153–180, 2002.

[2] Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *MDM*, 2004.

[3] B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, Heraklion, Crete, Greece, 2004.

[4] B. Gedik, K.-L. Wu, P. Yu, and L. Liu. Motion adaptive indexing for moving continual queries over moving objects. In *CIKM*, Washington, DC, USA, 2004.

[5] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD*, pages 479–490, Baltimore, Maryland, 2005. ACM Press.

[6] M. F. Mokbel and W. G. Aref. Gpac: generic and progressive processing of mobile queries over mobile data. In *MDM*, pages 155–163, Ayia Napa, Cyprus, 2005. ACM Press.

[7] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, Paris, France, 2004. ACM Press.

[8] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 17(11), 2005.

[9] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, Baltimore, Maryland, 2005. ACM Press.

[10] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB*, Seoul, Korea, 2006.

[11] H. Wang, R. Zimmermann, and W.-S. Ku. Distributed continuous range query processing on moving objects. In *DEXA*, 2006.

[12] W. Wu, W. Guo, and K.-L. Tan. Distributed processing of moving k-nearest-neighbor query on moving objects, http://www.comp.nus.edu.sg/ g0404403/papers/disMKNN-ext.pdf.

[13] X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654. IEEE Computer Society, 2005.

[14] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, 2005.