An Enhanced Safe Region Technique for Continuous Queries over Moving Objects

Yu-Ling Hsueh[†] Roger Zimmermann[‡] Wei-Shinn Ku[§]

[†]Dept. of Computer Science, University of Southern California, USA

[‡]Computer Science Department, National University of Singapore, Singapore

[§]Dept. of Computer Science and Software Engineering, Auburn University, USA

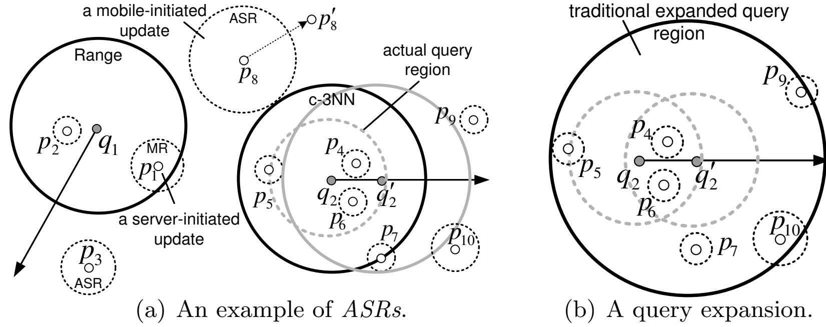
{hsueh@usc.edu, rogerz@comp.nus.edu.sg, weishinn@auburn.edu}

Abstract. Continuous spatial queries retrieve a set of time-varying objects continuously during a given period of time. However, monitoring moving objects to maintain the correctness of the query results often incurs frequent location updates from these moving objects. To address this problem, existing solutions propose lazy updates, but such techniques generally avoid only a small fraction of all unnecessary location updates because of their basic approach (e.g., safe regions, time or distance thresholds). Furthermore, most prior work focuses on a simplified scenario where queries are either static or rarely change their positions. In this paper, we introduce an *Adaptive Safe Region (ASR)* technique that retrieves an adjustable safe region which is continuously reconciled with the surrounding dynamic queries. The communication overhead is reduced in a highly dynamic environment where both queries and data objects change their positions frequently. In addition, we design a framework that supports multiple query types (e.g., range and c-kNN queries). In this framework, our query re-evaluation algorithms take advantage of *ASRs* and issue location probes only to the affected data objects, without flooding the system with many unnecessary location update requests. Simulation results confirm that the *ASR* concept improves scalability and efficiency over existing methods by reducing the number of updates.

1 Introduction

Significant research attention has focused on efficiently processing continuous queries and its extension work that supports location-based services during the recent past. The advancement of mobile technologies such as IEEE 802.11x networks, cellular communications and GPS sensors enables a server to track the positions of moving objects and provide various location-based services. Many existing techniques [5–7, 10, 11] have proposed continuous monitoring approaches without considering the cost of the communication overhead involved. Some prior work [4, 8, 9] has provided significant insight into these issues by assuming a set of computationally capable moving objects that cache query-aware information (e.g., thresholds or safe regions) and locally determine a mobile-initiated location update. However, the focus of these solutions is mainly on static queries or simple types of queries (e.g., range queries). If query movements are frequent, such systems suffer from repeated location detections to resolve location ambiguity (incurred on the objects that might become result points) and numerous down-link messages sent to refresh the query-aware information on those mobile objects.

In this paper, we propose a framework to support multiple types of dynamic, continuous queries. Our goal is to minimize the communication overhead in a highly dynamic environment where both queries and objects change their locations frequently. When a new query enters the system we leverage the trajectory information that it can provide by registering its starting and destination points as a movement segment for continuous monitoring. For example, a policeman might request the following



(a) An example of *ASRs*. (b) A query expansion.

Fig. 1. System overview.

query: "send me the closest 5 police cars on the road as I am moving from point A to point B." For simplicity, we assume a straight movement segment between two points. This assumption can be easily extended to a more realistic scenario which may approximate a curved road segment with several straight-line sub-segments. We propose an *adaptive safe region* that reconciles the surrounding queries based on their movement trajectories such that the system can avoid unnecessary location probes to the objects in the vicinity (i.e., the ones which overlap with the current query region). The basic concept of a *safe region* is that a moving object that stays within the given safe region does not affect any query results. Therefore, a location update is necessary only when a moving object exits its safe region. Furthermore, our incremental result update mechanisms allow a query to issue location probes only to a minimum area where the query answers are guaranteed to be fulfilled. In particular, to lower the amortized communication cost for c-kNN queries, we obtain extra nearest neighbors (n more NNs) which are cached and reused later to update the query results. Thus, the number of location updates incurred from the query region expansion due to query movement is reduced. An example

is shown in Figure 1 (a). The ASR of p_3 is determined based on the closest query q_1, since p_3 has a high probability of being covered by the query region of q_1 when q_1 moves in the future. The safe region of p_3 is adjusted to an appropriate size according to the trajectory information of q_1. The safe region for p_8 is simply set to the maximum non-overlapping area with the query region of q_2, because q_2 (due to its opposing movement direction) will never cover p_8. We buffer one extra NN for q_2 (a c-3NN query). When q_2 moves to q_2', and since the number of NNs is equal to 3, the query region remains unchanged. In the traditional approach (as shown in Figure 1 (b)), the query region is expanded to cover p_5 (the first closest object outside the query region) such that additional location probes to p_7, p_9, and p_{10} are issued. Overall, our approach reduces the number of query expansions to find sufficient NNs and the number of location probes. In summary, the contributions of this paper are as follows:

1. We propose a framework that supports multiple types of dynamic queries over moving objects and minimizes the communication overhead in a highly dynamic environments.
2. The concept of adaptive safe regions is introduced to enhance traditional safe regions. The numerous downlink location probes as a result of frequent query movements are reduced.
3. Our approach sends downlink ASR transmissions to a small set of objects affected by a query movement or an insertion of a new query.
4. We propose enhanced incremental query update algorithms that result in fewer location probes (e.g., through extra n NNs).

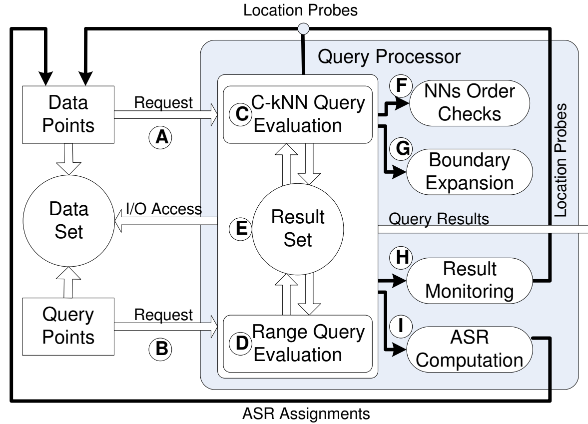


Fig. 2. System architecture overview.

Figure 2 shows the system framework. When a request arrives from a data point (A) or from a query point (B) (e.g., a location update, query insertion or deletion), the *ASR* query processor checks whether the point is part of a query result in modules (C) and (D). To incrementally update a query result, prior query results (E) are considered. For a c-kNN query, an NN order check (F) is performed during the query evaluation process. While there are less than k NNs in the result set, a query region expansion (G) is executed. Some server-initiated location probes might be needed to resolve location ambiguities. The points in the result set are monitored (H) through a passive mechanism – this result set is different from the non-result points that voluntarily issue location updates locally determined by the objects. Finally, an updated data point is assigned a new *ASR* based on the current query information in module (I). Detailed descriptions of the functionality of each component will be given in Section 3. The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 presents the adaptive safe region

computations and continuous query update mechanisms. We extensively verify the performance of our techniques in Section 4 and finally conclude the paper in Section 5.

2 Related Work

Continuous monitoring of queries over moving objects has become an important research topic, because it supports various useful mobile applications. Prabhakar et al. [9] designed two approaches named query indexing and velocity constrained indexing with the concept of *safe regions* to reduce the number of updates. The MQM solution [1] leverages the computational capabilities of moving objects for efficient processing of continuous range queries. Hu et al. [4] proposed a generic framework to handle continuous queries with safe regions through which the location updates from mobile clients are further reduced. However, these methods only address part of the mobility challenge since they are based on the assumption that queries are static which is not always true in real world applications.

By employing grid indices, a number of methods (e.g., LUGrid [12]) have been proposed to process dynamic continuous queries over moving objects. *MobiEyes* [3] presented a distributed infrastructure to process dynamic range queries where the server is acting as a mediator to coordinate query processing on both the server and moving objects. Mokbel et al. proposed the SINA algorithm [6] for evaluating a set of concurrent continuous spatio-temporal queries. With the incremental evaluation paradigm, SINA updates query results by computing and sending only in-

crements of the previously reported answer. Yu et al. [13] proposed an approach that continuously monitors a kNN query by defining a search region based on the maximum distance between the query point and the locations of current kNN query results. Moving object data are assumed to fit in main memory and are indexed with a regular grid. However, the algorithm suffers from high re-evaluation cost when the query point is highly mobile. The SEA-CNN framework designed by Xiong et al. [11] is based on the concepts of incremental evaluation and shared execution. Moving objects are stored in secondary memory and indexed with a regular grid. SEA-CNN continuously maintains the *search radius* of the query point to avoid recomputing the query results once the query point changes its location. The conceptual partitioning (CPM) [7] method assumes the same system architecture and indexing structures as SEA-CNN. CPM defines a conceptual partitioning of the space by organizing grid cells into larger rectangles. Location updates are handled only when objects move into the vicinity of queries and hence system throughput is improved. A threshold-based algorithm is presented in [8] which assumes that moving objects have some computational capabilities and aims to minimize the network cost when handling c-kNN queries. A threshold is transmitted to each moving object and when its moving distance exceeds the threshold, the moving object issues an update. However, the system suffers from many downlink message transmissions for refreshing the thresholds of the entire moving object population due to frequent query movements. Cheng et al. [2] proposed a time-based location update mechanism to improve the temporal data consistency for the objects relevant to queries. Data

objects with significance to the correctness of query results are required to send location updates more frequently. The main drawback of this method is that an object will repeatedly send location updates to the server when it is enclosed by a query region.

In contrast, our proposed techniques aim to reduce the communication cost of dynamic queries over moving objects and also support multiple types of queries. We utilize adaptive safe regions to reduce the downlink messages of location probes due to query movements. To further reduce the downlink messages, our incremental query update approach only probes a set of objects that might become part of the query results. Additionally, our approach allows for decoupled, query-aware information to be locally maintained by each moving object until the movement might affect the query results. Our *ASR*-based techniques surpass the aforementioned solutions with higher scalability and lower communication cost.

3 System Overview

The mobile units are partitioned into a set of dynamic query objects Q and a set of moving objects P. Each query object registers its movement trajectory with the server by uploading its starting and ending points (denoted by $\overrightarrow{q_j} = [q_j^s, q_j^e]$). Furthermore, all the data objects can move in a non-restricted fashion that allows them to move arbitrarily. We assume a maximum speed δ for both query and data objects. The server uses a main-memory grid G consisting of $w \times w$ cells of uniform length for each dimension to index the moving objects. The query processor evaluates

the queries based on the query types in an event-triggered manner. The locations of all queries are monitored by the server.

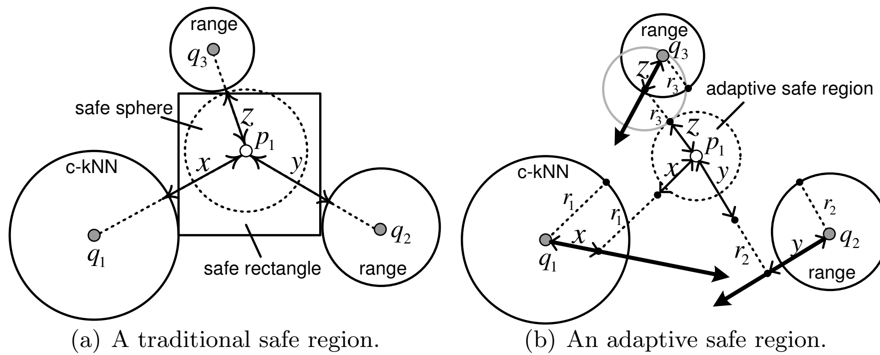
The location updates of a query result point (result point for short) and a non-result point (data point for short) are handled with two different mechanisms. An *adaptive safe region* (ASR) is computed for each data point. A mobile-initiated voluntary location update is issued when any data point moves out of its safe region. An example (p_8) is shown in Figure 1 (a). To capture the possible movement of a result point, we use a *moving region* (MR) whose boundary increases by the maximum moving distance per time unit. For the result points, the location updates are requested only when the server sends server-initiated location probes triggered when the moving regions of the result points overlap with some query regions. The details of the adaptive safe region computation are described in Section 3.1. We present an efficient continuous query update approach and propose a mechanism that uses location probes to solve location ambiguity in Section 3.2. Table 1 summarizes the symbols and functions we use throughout the following sections.

Symbol	**Description**
Q	A set of query objects
P	A set of moving objects
G	A $w \times w$ object grid where objects are hashed to the grid cells based on their locations
δ	Maximum speed for any object
$p_i.ASR$	Adpative safe region of object p_i
$p_i.MR$	Moving region of object p_i
$q_j.QR$	Query region of query q_j (the radius is denoted by $q_j.QR.radius$)
$\overrightarrow{q_j}$	Movement trajectory of q_j
q_j^s	Staring point of the movement trajectory for q_j
q_j^e	Ending point of the movement trajectory for q_j

Table 1. Symbols used in this paper.

3.1 Adaptive Safe Region Computation

The existing work adopts safe regions to reduce unnecessary location updates such that the communication cost between the server and moving objects is reduced. A safe region in a traditional system is simply an area of maximal size around an object such that no query regions overlap. Figure 3 (a) shows an example of two such safe region types (a safe sphere and a safe rectangle) for object p_1. However, this approach suffers from many location updates as a result of frequent query movements. When a query moves, the server initiates location probes to the data objects whose safe regions overlap with the query region to ensure the correctness of the query answers. In this paper, we propose a novel approach to retrieve an *adaptive safe region* (*ASR*), which is often smaller than a maximum non-overlapping region and yet is very effective in reducing the amortized communication cost in a highly dynamic mobile environment. The key observation lies in the consideration of some important factors (e.g., the velocity or orientation of the query objects) to reconcile the size of the safe regions. Figure 3 (b) illustrates the concept of an *ASR*. The on-demand



(a) A traditional safe region. (b) An adaptive safe region.

Fig. 3. Traditional safe region v.s. adaptive safe region.

location probes are issued as soon as any surrounding queries (q_1, q_2, or q_3) move. In this example, the distance z is the ASR radius of p_1, because in the worst case, after both q_3 and p_1 move by distance z and p_1 moves directly toward q_3, p_1 may become a result point of q_3. The following lemma establishes the ASR radius based on this observation.

Lemma 1:

$$p_i.ASR.radius = \min(CDist(p_i, q_j) - q_j.QR.radius), \forall q_j \in Q, \text{ where}$$

$$CDist(p_i, q_j) = \begin{cases} \overline{p_i f'} & \text{if } \theta_j \leq \frac{\pi}{2} \text{ and } \exists f', \text{ or} \\ \overline{p_i q_j^s} & \text{if } \theta_j > \frac{\pi}{2} \text{ or } \nexists f' \end{cases}$$

As an illustration of Lemma 1 (and to explain the symbol notation), consider Figure 4, where the set of queries $Q = \{q_j, q_k\}$ are visited for retrieving the adaptive safe region (the dashed circle) of the data point p_i. We measure the Euclidian distance between a query and a data point ($CDist$ in Lemma 1) and then deduct the query range. Lemma 1 captures two cases of *CDist*. The first case ($CDist(p_i, q_j)$) computes a distance $\overline{p_i f'} = \overline{q_j^s f}$ in the worst-case scenario where both p_i and q_j move toward

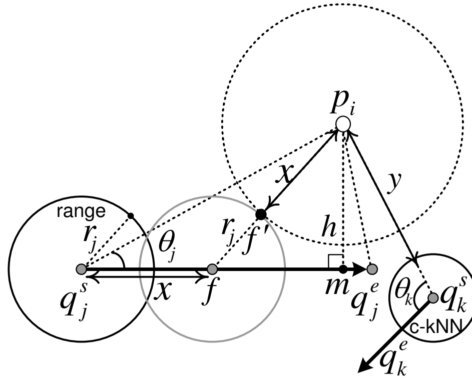


Fig. 4. An adaptive safe region.

each other (under the constraint of the maximum speed). f' represents the border point (on the border of $q_j.QR$ while q_j arrives at f on its movement segment), after which p_i would possibly enter the query region of q_j. f is the closest point to q_j^s on the trajectory of q_j, which satisfies the condition that the distance from p_i to f is equal to $\overline{p_i f'} + \overline{f' f}$, where $\overline{f' f} = q_j.QR.radius = r_j$. Let $\overline{p_i f'} = x$ for short. We can obtain the f and f' points by computing x first, which is considered the safe distance for p_i with respect to q_j. x can be easily computed with the trajectory information of q_j by solving the quadratic equation: $(x + r_j)^2 = h^2 + (\overline{q_j^s m} - x)^2$ (h is the height of triangle $\triangle p_i q_j^s m$). f on $\overrightarrow{q_j}$ exists only when θ_j ($\angle p_i q_j^s q_j^e$) is less or equal to $\frac{\pi}{2}$ and $(\overline{p_i q_j^e} - q_j.QR.radius) < \overline{q_j^s q_j^e}$ (triangle inequality). If the first case is not satisfied, we consider the second case ($CDist(p_i, q_k)$), which finds the maximum non-overlapping area with $q_j.QR$. Since $\theta > \frac{\pi}{2}$ in the second case, the query range of q_j can never cover p_i due to the opposing movement of q_j. In this example, the safe distance x (with respect to q_j) is smaller than y (with respect to q_k), so x is chosen as the radius of the adaptive safe region of p_i. In our system, since a c-kNN query can be considered an order-sensitive range query, we use the same principle to compute safe regions for each data object with respect to range queries and c-kNN queries. In case of a query insertion or query region expansion of a c-kNN query, the adaptive safe regions of the affected data objects must be reassigned according to current queries to avoid any missing location updates.

3.2 Query Evaluation with Location Probes

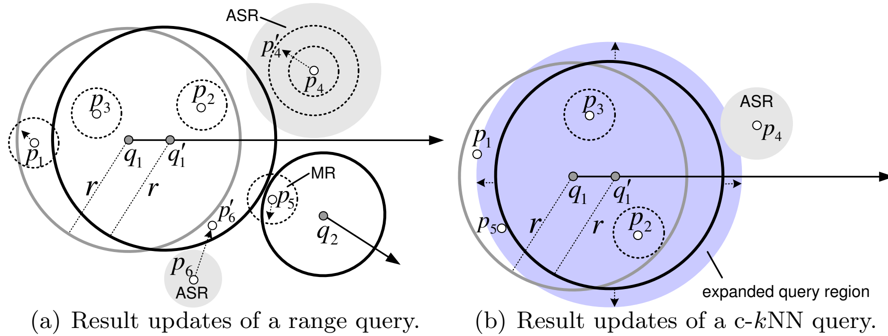
The initial query results of the range and c-kNN queries are obtained using *CPM* [4], and later the query results are updated in an event-driven fashion. Such events include the insertion or update of a query. In the following sections, we propose our incremental query re-evaluation algorithms for both range and c-kNN queries. While updating the query answers, on-demand server-initiated location probes are issued whenever any location ambiguity exists. Specifically, the cost of updating c-kNN queries is usually higher than updating range queries. The reason is that a c-kNN search is an order-sensitive query. The system executes more location updates to ensure the order of the result points. Furthermore, to make sure that at least k result points are found for a c-kNN query, the query region often needs to be enlarged in a situation where both query and data objects are moving, which leads to more location probes. In our approach, the strategy to handle such increasing unnecessary location updates incurred from a c-kNN query is that the query processor computes $(k + n)$ NNs for a c-kNN query instead of evaluating exactly k NNs. This approach helps to reduce the number of future query region expansions to retrieve sufficient NNs for the queries. Since a c-kNN query is treated as an order-sensitive range query, we adopt the same principle that is used for a range query to find the new answer set in the current query regions first. A query region is expanded only when there are less than k NNs in the result set. Finally, an order-checking procedure is performed to examine the order of the result points and determine necessary location probes.

Query Result Updates for Range Queries The query processor re-evaluates the range queries based on their current positions by the same principles as evaluating the initial query results. The traditional approach adopts the query region itself as the safe region for all the result points in the region to reduce the number of location updates. However, the approach incurs more network messages when a range query changes its position frequently, because the system needs to inform the result points of the new position of the query region to avoid missing location updates. An alternative approach basically monitors the entire set of result points to obtain the new correct results. However, such an approach is not scalable when there are large numbers of range queries. In this paper, we use a *moving region* (*MR*) for each result point to estimate the possible movement at the server side. The query processor sends the on-demand location probes to those result points that might move out of the current query regions. A *MR* is indexed on the grid and the boundary increases at each time step by the maximum moving distance until the result point is probed by the server. Since the number of result points are relatively small, indexing *MRs* does not significantly increase the overall server workload. In Figure 5 (a), when q_1 moves to q_1', the query processor checks p_1 and p_5, since their *MRs* intersect with $q_1'.QR$.

For a data point, in addition to its adaptive safe region, we also consider the current possible moving boundary to serve as an additional indicator for the server to determine a necessary location probe. Continuing the example in Figure 5 (a), the gray circle surrounding p_4 is its *ASR*, and the dashed circles represent the possible moving boundaries (the ra-

dius is equal to the maximum moving distance since the last update of p_4) for different time steps. p_4 is checked because its $p_4.ASR$ overlaps with $q'_1.QR$. However, the server does not need to issue a location probe since the current moving boundary does not overlap with $q'_1.QR$. p'_6 is a newly updated (p_6 moves out of its ASR) data point. The system also needs to check whether its current position is in the query region of q'_1. Algorithm 1 shows the pseudo code of the range query evaluation, where q'_j is the updated query of q_j. Lines 1-7 remove previous result points that are not in the the current query region $q'_j.QR$. Lines 2 and 4 compute the *mindist* and *maxdist* between a query point and a result point, respectively. If a result point with a MR is completely contained in the query range, a location probe is ignored. In Line 10, if p_i is a data point, the server uses the radius of ASR or the maximum moving distance since the last update, which ever is less to estimate its possible moving distance.

Query Result Updates for c-*k*NN Queries A c-kNN query is more complicated since it is order-sensitive. An intuitive solution enlarges a query region that covers at least all the previous result points (first k



(a) Result updates of a range query. (b) Result updates of a c-kNN query.

Fig. 5. Query result updates.

Algorithm 1 RangeQuery-Update(q'_j)

```
1: for (each d ∈ q_j.RangeNN) do
2:   if (dist(d, q'_j) − d.MR.radius) > q'_j.QR.radius) then
3:     remove d
4:   else if (dist(d, q'_j) + d.MR.radius) > q'_j.QR.radius) then
5:     probe d and remove d if its current position is outside of q'_j.QR
6:   end if
7: end for
8: for (each c ∈ G, which overlaps with the q'_j.QR) do
9:   for (each object p_i which resides in c or whose (1) ASR, or (2) MR overlaps with
     it) do
10:    let r = p_i.MR.radius, if p_i is a result point; else let r =
       min(p_i.ASR.radius, δΔt)
11:    if (dist(p_i, q'_j) − r < q'_j.QR.radius) then
12:      if(dist(p_i, q'_j) + r < q'_j.QR.radius), insert p_i into q'_j.RangeNN
13:      else probe the position of p_i and insert p_i into q'_j.RangeNN, if p_i is within
         q'_j.QR.
14:    end if
15:  end for
16: end for
```

NNs) to retrieve new result points. This approach greatly increases the number of location updates since such an expansion (the query region is expanded by the moving distance of the query and result points) often results in more location probes, even though in reality only a small fraction of queries and data objects move. Therefore, in our design for the c-kNN queries, we propose a server-initiated update strategy with an event-triggered update mechanism. Furthermore, the query processor retrieves $(n + k)$ NNs to avoid immediate and successive query region expansions. We relax the definition of the query region, that is, a query region does not necessary include exact k NNs only. The query region remains unchanged until a c-kNN query does not contain enough NNs in the region. We summarize the following steps to update a c-kNN query result incrementally:

Step 1: Assume that q'_j is a c-kNN query after it moves from q_j position. Initially, set $q'_j.QR.radius = q_j.QR.radius$. Perform a range query update (as described in the previous section) to update result points in $q'_j.QR$. If the number of NNs in $q'_j.QR$ is equal or larger than k, proceed to Step 3. Otherwise, continue to Step 2.

Step 2: Expand $q'_j.QR$ until there are $(k+n)$ NNs. Update $q'_j.QR.radius$ to the distance between q'_j to the $(k+n)^{th}$ NN.

Step 3: Sort the order of the result points and issue the necessary location probes.

Step 1 ensures that $q'_j.QR$ covers at least k result points. Note that during the process, some discarded objects that are not in $q'_j.QR$ might be useful in Step 2, because these objects are often very close to $q'_j.QR$ and might be already probed by the server. Finding new NNs from these points first in Step 2 helps the query processor to avoid expanding the safe region to a farther level of cells. In Step 2, while expanding the query region to cover $(k+n)$ result points, a location update is required from any data object p_i whose safe region overlaps with the query region of q'_j. A new ASR is computed for the updated p_i, if p_i is still a data object. We use the same approach (query region expansion) to handle a query insertion. In Step 3, sorting the order of the result points does not require the current positions of the entire result points. The processor performs an *OrderCheck* procedure that examines the possible actual moving distance of two consecutive NNs to determine the order of the NNs, and issues a location probe only if there is a location ambiguity.

Figure 5 (b) shows a query region expansion where $k = 3$ and $n = 1$. In Step 1, since p_1 and p_5 (probed during the process) are not in $q'_1.QR$, they are removed from the answer set and inserted into a buffer for "recycling" later. Step 2 is performed since there are only two result points in $q'_1.QR$. The query processor checks the data points in the buffer first, so the first two objects (sorted by the *mindist* to q'_1) are considered. The new $q'_1.QR.radius$ (the blue area) is set to the distance between q'_1 and p_1 to include at least 4 $(k + n)$ objects. p_4 is checked later since the safe region overlaps with $q'_1.QR$. Algorithm 2 shows the detailed process of a c-kNN query update. In Line 2, the *RangeQuery-Update* procedure inserts the discarded objects into buffer B sorted by *mindist* in the ascending order. Line 4 computes the number (v) of NNs missing in the current query region. Line 12 executes *CPM* to further expand the query region by checking the surrounding cells only when the buffer is empty. The *OrderCheck* procedure in Line 16 is performed after all the sufficient NNs are found. In the *OrderCheck* procedure, to determine a necessary location probe for kNN result points, we observe the following lemma. A proof of correctness is presented subsequently.

Lemma 2: Let q'_j be the last reported position of the query object q_j, and let $\ell = \delta \varDelta t$ be the maximum moving distance since the last update of q_j, where δ is the maximum speed and $\varDelta t$ is the time period from the last update time to the current time. $\forall i = 1$ to k, a result point p_i (the ith result point sorted by the *mindist to* q'_j) needs to issue a location update when the following condition is satisfied:

$$\ell \geq (mindist(q'_j, p_{i+1}) \text{ - } mindist(q'_j, p_i)) \times \tfrac{1}{2}$$

Algorithm 2 c-kNN-Update(q'_j)

1: let $B = \phi$ be a buffer
2: perform *RangeQuery-Update*(q'_j), which finds new NNs in the current query region and inserts discarded objects into B, if any
3: **if** ($q'_j.KNN.size < k$) **then**
4: $v = k + n - q'_j.KNN.size$
5: **while** ($v > 0$) **do**
6: **if** ($B.size > 0$) **then**
7: set $q'_j.QR.radius = dist(q'_j, V)$, where V is the v_{th} NN in B, if $B.size >= v$. Otherwise, set $dist(q'_j, L)$, where L is the last object in B.
8: empty B
9: perform *RangeQuery-Update*(q'_j) that inserts un-visited, discarded objects into B, if any
10: $v = k + n - q'_j.KNN.size$
11: **else**
12: perform $CPM(q'_j)$ that checks the objects in the surrounding cells of $q'_j.QR$, until $(k + n)$ objects are fulfilled, and terminate the loop.
13: **end if**
14: **end while**
15: **end if**
16: sort $q'_j.KNN$ by performing *OrderCheck*($q'_j.KNN$) that issues necessary location probes.

Proof: The proof is straightforward, since when the order of p_i and p_{i+1} changes, $mindist(p_i, q'_j) \geq mindist(p_{i+1}, q'_j)$. When considering the worst case that p_i moves in an opposing direction from q'_j and p_{i+1} moves toward q'_j directly, the following inequality holds true:

$$mindist(p_i, q'_j) + \ell \geq mindist(p_{i+1}, q'_j) - \ell$$

Therefore, we conclude that the order of p_i and p_{i+1} must change, when $\ell \geq (mindist(q'_j, p_{i+1})$ - $mindist(q'_j, p_i)) \times \frac{1}{2}$. It is necessary for the server to probe both locations of p_i and p_{i+1}. ∎

In Figure 6, the result set of q'_1 is $\{p_2, p_1, p_3\}$ sorted by the distance between q'_1 and their positions at the server since the last updates. The *OrderCheck* procedure first checks p_2 and p_1. Since $dist(q'_1, p_2) + r_2 >$

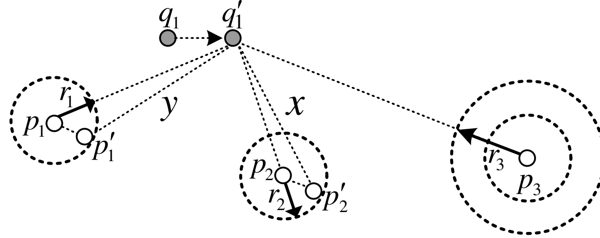


Fig. 6. The order checks of a c-kNN query.

$dist(q_1', p_1) - r_1$, the order of p_2 and p_1 might need to be switched. The system needs to probe p_2 and p_1. After the location probes, the order of the NNs becomes $\{p_1', p_2', p_3\}$. Thus, the procedure checks the next pair of p_2' and p_3. Since $dist(q_1', p_2') < dist(q_1', p_3) - r_3$, the location probe of p_3 is not necessary.

4 Experimental Evaluation

We evaluated the performance of the proposed framework that utilizes *ASRs* and compared it with the traditional safe region approach [4, 9] and a periodic update approach (*PER*). The periodic technique functions as a baseline algorithm where each object issues a location update (only uplink messages are issued in this approach) every time it moves to a new position. We extended the safe region approach (*SR**) to handle dynamic range and c-kNN queries where the result points are monitored the same way as in *ASR*. We preserve the traditional safe region calculations (maximum non-overlapping area) for the SR* approach. The simulation steps and the detailed simulation results are described in the following sections.

4.1 Simulation Steps

We use a main memory grid as the underlying index structure for all the three approaches. Our data sets are generated on a terrain service space of $[0, 1024]^2$. We assume a maximum speed for any moving object in the range of $[0.48, 1.25]$. The mobility (the percentage of objects that move from time step to time step) for the objects is set in a range from 10% to 50%. The length q_{len} of a range query is set in the range of [1,10] and k for the a kNN query is set from 5 up to 20. In the simulations, the main measurement is the cost of the communication overhead which includes uplink messages (e.g., a mobile-initiated location update) and downlink messages (e.g., a server-initiated location probe). The communication cost is measured by assuming that the cost of an uplink message ($c_{up} = 2$) is twice as costly as a downlink message ($c_{down} = 1$). Table 2 summarizes the default parameter settings in the following simulations.

Parameter	**Default**	**Range**
Number of objects (P)	100K	50K, 100K, 150K, 200K
Number of queries (Q)	100	50, 100, 150, 200
Mobility rate	50%	10%, 20%,30%, 40%, 50%
Number of NNs (K)	10	5, 10, 15, 20
Query length for range queries (q_{len})	5	1, 5, 10

Table 2. Simulation parameters

4.2 Number of Extra NNs

First, we test the efficiency of using extra NNs (n) for c-kNN queries by varying the number of n, since this factor greatly affects the number of downlink messages. The choice of the number of extra NNs is a trade-off.

If n is too large, the query processor evaluates more NNs for a query and the system is more likely to issue more location probes since a larger query region might overlap with more data objects for location probes. If n is too small, there are more query expansions which might also cause location probes. Figure 7 shows the number of overall communication cost (measured in thousands of messages) as a function of the number of extra NNs ranging from 0 to 20. When n is set to more than 10, the performance of *ASR* is degraded in terms of the communication cost. Therefore, we chose $n = 10$ for the rest of our experiments as this setting results in reduced communication cost.

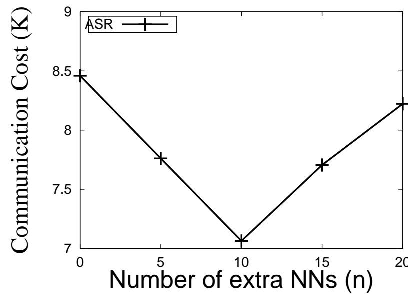
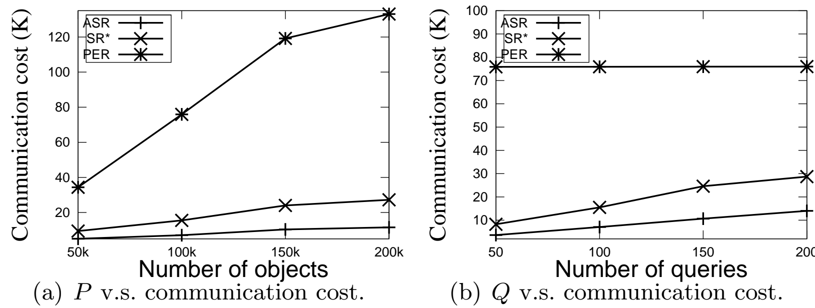


Fig. 7. Extra NNs v.s. communication cost.

4.3 Cardinality

We examined the effect of the query and object cardinality assuming that all query and object sets move with a mobility rate of 50%. Figure 8 (a) shows the communication overhead of *ASR*, *SR** and *PER* with respect to the object cardinality. *ASR* outperforms *SR** and *PER*. The difference increases as the number of objects grows. Since an *ASR* reconciles the surrounding moving queries, a query movement does not incur many

unnecessary location probes from the surrounding objects. *SR** on the other hand, triggers many location probes from the objects whose safe regions overlap with a query region once the query changes its position. As the density of objects increases, there are more objects in the vicinity area of a query region. Hence *SR** incurs an increasing number of location updates as the cardinality increases. Figure 8 (b) shows the impact of the number of queries. Our algorithm achieves about 50% reduction compared with *SR** and 90% reduction compared with *PER*.



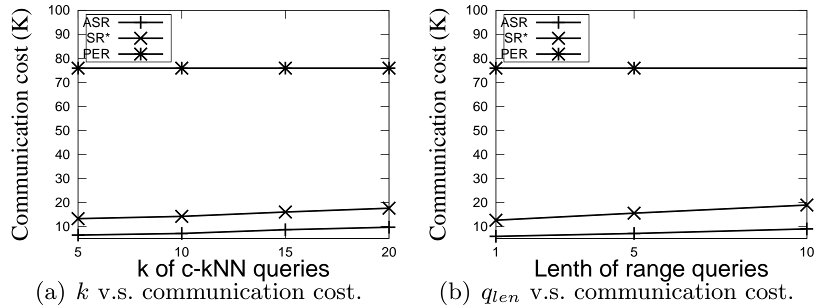
(a) P v.s. communication cost. (b) Q v.s. communication cost.

Fig. 8. Object and query cardinality.

4.4 Query Coverage

The query coverage varies with the number of NNs (for kNN queries) and the query length (for range queries). Figure 9 (a) shows the communication cost as a function of the number of NNs and Figure 9 (b) illustrates the effect of the query length. Overall, the communication cost increases as a function of k and q_{len}. However, since *ASR* and *PER* utilize the *OrderCheck* procedure to reduce the number of location probes from the objects which do not violate the order of result sets, the communication

overhead remains stable when k increases. This confirms the feasibility of the *OrderCheck* procedure as well as the c-kNN update mechanisms of our approach. The *PER* approach basically monitors all the moving objects. Therefore, the number of k is irrelevant to the communication cost; however, *PER* is not scalable when there is high query coverage.

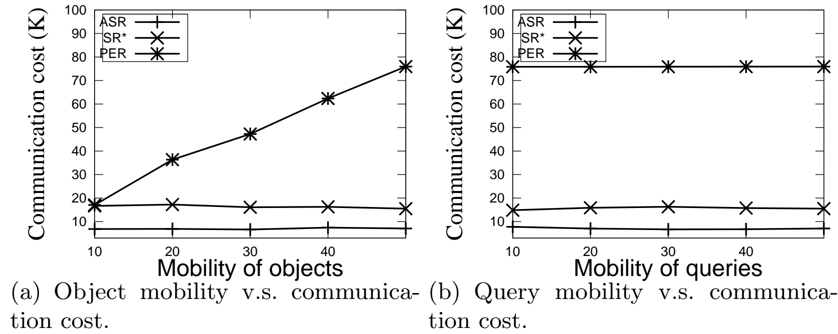


(a) k v.s. communication cost. (b) q_{len} v.s. communication cost.

Fig. 9. Effect of query coverage with k and q_{len}.

4.5 Mobility

Finally, we evaluated the impact of the mobility rate. Figures 10 (a) and (b) show the communication cost as a function of the object and query mobility, respectively. The *ASR* approach achieves a reduced location update rate compared to the other two approaches for all mobility rates. *PER* and *SR** have worse performance in terms of communication cost when the mobility rate is high. The degradation is caused by the location probes due to query movements.



(a) Object mobility v.s. communication cost. (b) Query mobility v.s. communication cost.

Fig. 10. Object and query mobility.

5 Conclusions

We have designed an *ASR*-based framework for highly dynamic environments where mobile units may freely change their locations. The novel concept of an adaptive safe region is introduced to provide a mobile object with a reasonable-sized safe region that adapts to the surrounding queries. Hence, the communication overhead resulting from the query movements is greatly reduced. To further decrease network traffic caused by c-kNN query region expansions to cover sufficient NNs for the result sets, our approach caches extra NNs. An incremental result update mechanism that checks only the set of affected points to refresh the query answers is presented. Experimental results demonstrate that our approach scales better than existing techniques in terms of the communication cost and the outcome confirms the feasibility of the *ASRs* approach.

References

1. Y. Cai, K. A. Hua, and G. Cao. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. In *Mobile Data Management*, pages 27–38, 2004.
2. R. Cheng, K. yiu Lam, S. Prabhakar, and B. Liang. An Efficient Location Update Mechanism for Continuous Queries Over Moving Objects. *Inf. Syst.*, 32(4):593–620, 2007.

3. B. Gedik and L. Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *EDBT*, 2004.
4. H. Hu, J. Xu, and D. L. Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *SIGMOD Conference*, pages 479–490, 2005.
5. C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *VLDB*, 2004.
6. M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD Conference*, pages 623–634, 2004.
7. K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *SIGMOD Conference*, 2005.
8. K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao. A Threshold-Based Algorithm for Continuous Monitoring of k Nearest Neighbors. *IEEE Trans. Knowl. Data Eng.*, 17(11):1451–1464, 2005.
9. S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Trans. Computers*, 51(10):1124–1140, 2002.
10. Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *VLDB*, pages 790–801, 2003.
11. 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, 2005.
12. X. Xiong, M. F. Mokbel, and W. G. Aref. LUGrid: Update-tolerant grid-based indexing for moving objects. In *MDM*, 2006.
13. X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-Nearest Neighbor Queries Over Moving Objects. In *ICDE*, pages 631–642, 2005.