Cooperative Scalable Moving Continuous Query Processing

Xiaohui Li [1], Panagiotis Karras [2], Lei Shi[1], Kian-Lee Tan[1], Christian S. Jensen[3]

[1]*School of Computing, National University of Singapore*
[2] *Management Science & Information Systems, Rutgers University*
[3] *Computer Science, Aarhus University*

***Abstract*—A range of applications call for a mobile client to *continuously* monitor others in close proximity. Past research on such problems has covered two extremes: It has offered totally centralized solutions, where a server takes care of all queries, and totally distributed solutions, in which there is no central authority at all. Unfortunately, none of these two solutions scales to intensive moving object tracking applications, where each client poses a query. In this paper, we formulate the *moving continuous query* (MCQ) problem and propose a balanced model where servers cooperatively take care of the global view and handle the majority of the workload. Meanwhile, moving clients, having basic memory and computation resources, handle small portions of the workload. This model is further enhanced by dynamic region allocation and grid size adjustment mechanisms that reduce the communication and computation cost for both servers and clients. An experimental study demonstrates that our approaches offer better scalability than competitors.**

I. Introduction

A Moving Continuous Query (MCQ) is issued by a mobile client who needs to be continuously aware of other clients in its proximity. Several applications, such as massive multiplayer online games (MMOG) (e.g., World of Warcraft), virtual community platforms (e.g., Second Life), real-life friend locator applications, and marine traffic management systems employed by port authorities, require the efficient real-time processing of such queries. In all such applications, a large population of clients are moving around, and each client has to be aware of other clients in close proximity to its positions. Such a monitoring system typically consists of database servers, base stations, and a large number of moving clients.

In effect, each moving client issues a continuous query with a custom radius r and centered at its own location to the database servers that are monitoring the entire service region. Every moving client functions both as a query issuer in its own right, and a potential participant in other clients' query results. Under these circumstances, central servers have to facilitate large-scale tracking and communication among the moving clients; thus, the challenge is to keep both the server workload and the communication cost among servers and clients low.

Traditional techniques for continuous spatial query processing are based on a centralized client-server architecture or assume that there are significantly fewer queries than moving clients [11]–[13], [17]. Unfortunately, such techniques do not scale well to applications where each of a large number of mobile clients poses its own query. In addition, such techniques do not leverage the capabilities offered by modern ubiquitous computing environments, in which mobile clients can assume simple computational tasks themselves. The applications we target call for solutions designed for the particular scalability challenges they pose.

The solution to the scalability problem can be to buy a more powerful server or to buy more pieces of less powerful machines and then interconnect them to cooperatively handle the workload. The second solution is more affordable than the first one. Our design follows the second approach.

This paper proposes a framework that caters to these increased scalability requirements and leverages a ubiquitous computing environment. We propose a distributed architecture where servers take care of the global picture, tracking moving clients and communicating results to them; at the same time, mobile clients, having basic memory and computational resources, perform part of the required computational workload by updating their own results. In this framework, a cluster of inter-connected servers handle the majority of the workload in a cooperative manner. To that end, we partition the space into subregions (called service regions), each monitored by a separate server that is only responsible for computing the results for the moving queries in its service region. The server-server communication cost can be kept minimal by only exchanging information at the service region boundaries for cross-boundary queries. Three important metrics are considered here: client-server communication cost, client workload, and server workload. We assume that server-server communication is much less expensive than client-server communication because servers are inter-connected by high-speed cables.

The communication needs of mobile client monitoring applications can be divided into three components.

Location updates. In the framework, each server is indexing the mobile clients with an in-memory grid structure. Each server is allowed to freely have a different but coordinated grid cell side length depending on the distribution of mobile clients. This flexibility greatly reduces the total amount of update messages. To further reduce the number of update messages, each server negotiates a mobile region with every mobile client in its service region. A mobile region $\mathcal{R}$ of a mobile client o moves from the most recently updated position of o at o's most recently known velocity. Such regions are assigned by servers

to clients in their service region. A client o needs not send location updates as long as it does not exit its most recently assigned mobile region [10].

Probe messages. These messages are sent from a server to its clients when the server needs to know the exact location of a client.

Results update. In the framework, results are maintained by the clients which do not need to notify the server when result objects exit the query circles. We employ an adaptation of the *trigger time* concept [8]. A trigger time is the time when a moving client no longer satisfies another client's MCQ, and a *trigger event* is the event of exiting that MCQ region. Each mobile client maintains a queue of events, ordered by their trigger times. Servers can insert events into the event queues of each client so that each client verifies whether the pending trigger events in its queue have occurred at each time point and updates its local results accordingly. The amount of communication is reduced because no messages are being exchanged.

Our major contributions can be summarized as follows:

- We proposed Moving Continuous Queries (MCQ) that have wide application in both real-world and gaming applications. MCQs are distinct from existing queries in the following aspects: (1) Each client is allowed to issue at least one MCQ, and thus the number of queries can be larger than the number of moving clients. (2) Each client must have exact answer to its own query, while servers do not need to store the answer to each query.
- We proposed balanced computational models to efficiently answer MCQs. In these models, servers cooperatively take care of the global view and share the majority of the workload among them. Clients are only concerned with their vicinity and also carry out a small portion of the workload.
- We also propose a mechanism to dynamically allocate regions to handle the problem of skew so that it is possible to balance the workload among the servers even in the extreme case that all mobile clients cluster at one service region.
- We report on extensive experiments that offer insight into the efficiency of the proposed framework.

The rest of this paper is structured as follows. Section II reviews related work. Section III formally defines the problem that we address, and Section IV describes our system architecture. Section V details the operation of query processing in our framework. In Section VI, we examine how our system adapts to a highly dynamic environment. Section VII covers our experimental studies, and Section VIII concludes the paper.

II. Related Work

Early works on moving-object querying focused on developing indexes posed by the challenge to handle frequent location updates. These include data-partitioning indexes [14] and space-partitioning indexes [9], [11], [12]. These works, however, aimed at reducing the computational and I/O cost at server side, and not the communication cost between server and clients. Hu et al. [7] index both moving objects and queries in order to obtain good performance — an R-tree based index is used for moving objects and grid-based structure is used for queries. However, the work considers only static queries with one central server.

Amir et al. [1] consider moving objects form a *peer-to-peer* (P2P) network, where each object is a computing unit and no central server is present. Each pair of moving objects defines and maintains *safe region* information, capturing the region of their mutual proximity. Unfortunately, the performance of this method does not scale well as the number of objects, and hence peers for each of them, increases. On the other hand, in a centralized methodology [13], all proximity computations occur at the server, while each moving object sends location updates there. Each moving object also continuously checks whether its changing position falls within a moving sector region, defined by an angular threshold θ, a minimum speed V_{min}, and a maximum speed V_{max}. Farrel et al. [5] propose to relax query accuracy requirement for continuous range queries in order to cope with location uncertainty and reduce energy consumption of the clients. Chen et al. [4] study the efficient processing of predictive range queries, proposing spatio-temporal safe regions (STSR) to bound the movements of objects in order to reduce update costs. Adopting a centralized approach, STSRs are computed at the server side. Moving objects are only capable of sending update messages.

Both MobiEyes [6] and DKNN [16] aim at *moving* continuous range- and kNN-queries. Although a central server is employed in these works, computation occurs mostly on the moving clients, with the server and base station being mainly responsible for relaying messages among the clients, using an inflexible space-partitioning index that incurs high communication overhead. Wang et al. [15] proposed a distributed framework where multiple servers cooperatively handle *static* continuous range queries. Unlike our framework, computation occurs mostly on the moving clients with an in-memory grid structure, while servers mainly relay messages.

In a central setting, Yiu et al. [17] propose the Reactive Mobile Detection (RMD) algorithm to solve the *proximity detection* (PD) problem where, given a collection of moving objects and a distance threshold, one needs to find *each pair* of moving objects whose distance is within the distance threshold. Arguing that using predefined mobile region radius renders it difficult to control the messaging cost, they propose the RMD scheme that expands and contracts the radii of mobile regions when needed. The intuition is to contract the mobile regions to reduce the probing cost at the cost of more updates if the probing cost is too high, and vice versa, in order to minimize the overall cost. The MCQ problem is different from the PD problem. (a) In PD, a distance threshold is defined for each pair of clients, whereas in MCQ, each client can have its own distance threshold. (b) In PD, the server must have the exact solution. So it fails to exploit the computational power of clients, which can lead the way to higher scalability. As the RMD scheme cannot be directly applied to solve the MCQ problem, we need to adapt it for MCQ problem. Instead of

finding pairs in proximity, the scheme is modified so that the server finds the result for each client, with the contraction and expansion properties still kept.

III. Problem Definition

The section formalizes the problem we set out to solve. Important notation is summarized in Table I.

α	A grid cell side length
β	The ratio to determine overloading
γ	The ratio to determine skew in one service region
σ	The average number of moving objects per cell
$tr_{o_1}(o_2)$	The trigger time when o_2 is no longer o_1's result
t_d	The time threshold for dynamic region allocation
t_e	The time threshold for dynamic cell side length adjustment
R	Service Region

TABLE I
NOTATION USED IN THE PAPER

First of all, let $D(o_1, o_2)$ denote the Euclidean distance between two moving clients o_1 and o_2. We assume an environment where each moving client issues a *continuous range query*; one moving object issues only one query at a time. The problem is to keep track of all moving clients, and to update the results of the queries they issue accordingly.

Definition 1: (Moving Continuous Query) Given a moving client o with a location p, a *moving continuous query* (MCQ) of o is a circular range query $\odot(p, r)$ centered at p and with radius r that continuously retrieves all objects within that circle.

As this definition suggests, it is important for a *client* to have the result of its MCQ. The server may or may not have that result. A moving client is represented as $o_i = \langle oid, \vec{p}_i^{ref}, \vec{v}_i^{ref}, t_i^{ref}, r\rangle$, where oid is an object identifier, $\vec{p}_i^{ref}$ is the the most recently updated location of o at time t_{ref}, $\vec{v}_i^{ref}$ is the most recently updated velocity at time t_{ref}, and r is the radius of the query issued by o. To reduce the amount of location updates sent by mobile clients to the server, we use a linear function to model o's movement, based on the latest known location and velocity of o, at time t_{ref}. Thus, the location of o_i at time t is $\vec{p}_i(t) = \vec{p}_i^{ref} + \vec{v}_i^{ref} \cdot (t - t_i^{ref})$, where $t \geq t_i^{ref}$.

In our setting, space is partitioned into cells by a grid; furthermore, the server assumes the responsibility of sending to each client a set of *candidate results*, which may satisfy the client's query (circle with radius r centered at o). In effect, the server has to know which cells overlap with the query of each client o. We define that set of cells as o's *alert region*. The shaded grid cells in Figure 1 illustrate the alert region of o. The server has to keep track of the alert region of each object o by indexing that object with the grid cells. Therefore, for each cell c, the server maintains a list of queries to whose alert regions contain c.

To reduce the server workload, servers take a simple approach to computing alert regions. Servers implement grids by using two dimensional arrays. Knowing that alert regions are the intersection of grid cells and bounding rectangles of MCQs, servers only need to compute the indices of the intersected cells. Suppose the location of o is $p = (x_0, y_0)$. The server of o computes

$$x_l = \max(\frac{x_0 - r}{\alpha}, 0), \qquad y_l = \max(\frac{y_0 - r}{\alpha}, 0)$$
$$y_h = \max(\frac{y_0 + r}{\alpha}, maxIdx_x), \quad y_h = \max(\frac{y_0 + r}{\alpha}, maxIdx_y)$$

where $maxIdx_x$ and $maxIdx_y$ are the maximum x and y indices for the 2D array. The alert region of o is then $\{G[i][j] \,|\, x_l \leq i \leq x_h \wedge y_l \leq j \leq y_h \wedge G[i][j] \cap o.q\}$, where G is the grid.

Different from Iwerks et al. [8], a trigger time in our setting has the form $\langle t, oid\rangle$ indicating that the moving client with the identifier oid exits the query circle of another moving client at time t. Each client maintains a heap structure indexing on the trigger times for its results.

Definition 2: (Trigger Time) Given a moving client o_1, suppose o_2 is another moving client that satisfies $o_1.q$. The trigger time for o_2 is the earliest time t so that o_2 no longer satisfies $o_1.q$.

$$tr_{o_1}(o_2) = \min\{t \,|\, t > t_{ref} \wedge D(p_{o_1}(t), p_{o_2}(t)) > o_1.r\}$$

The queue is ordered on the trigger time and stored as a heap in each moving client.

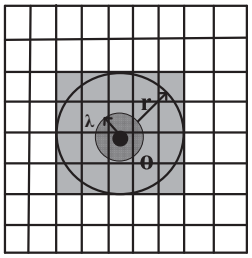


Fig. 1. Alert Region and Query Region Augmentation

IV. System Overview

We now proceed to present the system architecture that supports scalable MCQ processing. We assume that moving clients and their queries fit into main memory of the database server. This assumption is valid in most applications where real-time evaluation is needed.

As we have discussed, a single-server centralized solution cannot easily scale to a large number of MCQs in a highly dynamic environment. Thus, our system architecture employs a cluster of servers, dividing the service space into *service regions*.

A. Space Division Model

Each server is assigned to monitor a part of the space (called service region), and it is only responsible for computing the results for moving queries in its service region. The challenge with this design is that the distribution of the moving clients can be highly skewed. In the extreme case, all mobile clients

can be in a single service region. Such skew may cause unbalanced workloads among servers and significant overhead of client-server communication cost. In the framework, we introduce two mechanisms to handle skew in order to achieve a balanced server workload and minimize the client-server communication cost. The macro-level mechanism is to dynamically split and merge regions. When a server detects that its workload is significantly higher than the average workload, the server is able to further divide its service region for servers with less workload to take up. On the other hand, compatible regions with very few moving clients can be merged and then monitored as one single service region.

This model also allows the same server to monitor more than one, potentially non-adjacent, regions. In case any service region R has more than β times the average number of moving objects in each region, R is split into two new smaller service regions, one of which is then taken over by the server with the least number of moving objects in its service region. We use the term *configuration* for a space division. Obviously, there can be infinite many configurations in a system.

The micro-level mechanism is to allow each server to tune its own grid cell side length. As service regions may have different densities of clients, each server needs to pick the right granularity of the grid structure that reduces the amount of messages being exchanged between clients and the server.

B. Server Cluster Initialization

When the server cluster is initialized, a *bootstrap server*, which can be any server in the system, obtains the total number of available servers and partitions the entire service space using *binary space partitioning*. Ideally, each service region should contain almost the same number of moving objects. Each server is assigned the task of monitoring the moving objects in its service region, while the moving clients themselves are also informed about the server they should report their location updates to. We assume that each server knows the addresses of all other servers. For a small number of servers, the resulting routing tables can be stored in each server's main memory.

In the event that a new server S' requests to join an existing server cluster, the following steps occur.

1) The *bootstrap server* selects the service region R that contains the largest number of moving objects; assume R is currently assigned to server S.
2) R is divided into two regions of equal area, R_1 and R_2.
3) R_2 is assigned to S', which obtains from S the list of moving objects in R_2.
4) S informs the moving objects in R_2 that they should report to S' and removes them from its list.
5) S broadcasts messages about this event to other servers.

Each sever maintains its own space-partitioning grid as its own local index. Another important task for server-server initialization is that each server tunes its cell size to the density of objects in its region aiming at the same average object per cell ratio σ. Suppose a server is monitoring N moving clients in a space of area A. Recall that α is the length of a grid. We have the following relation between σ and α.

$$\sigma = \frac{\alpha^2}{A} \cdot N \quad (1) \qquad\qquad \alpha = \sqrt{\frac{A \cdot \sigma}{N}} \quad (2)$$

When the system initializes, the bootstrap server picks up a value for its cell length, calculates its σ, and broadcasts σ to all other servers. Upon receiving the σ value, each server computes its own cell length with equation 2, substituting its own number of moving clients and service region area.

Note that after initialization, the cell side lengths may still be adjusted in circumstances such as the distribution of the moving clients is highly skewed. In system optimization, we give a discussion on how to dynamically adjust cell side lengths when the system is running.

V. Query Processing

We now consider query processing on a single server S. We use terms *moving query* and *moving object* interchangeably, depending on the context, as each object issues one query. We use the term *query circle* to refer to the query issued by a moving object.

As discussed, the server maintains an in-memory grid index, in which each cell has side length α. We assume there is the maximum speed limit, V_m, for all objects. It should hold that $\alpha \geq tu \cdot V_m$, where tu is the basic time unit, so that no object enters more than one cell from one time point to the next.

For each cell, the server maintains a list of overlapping MCQs, as well as a list of moving objects. The moving objects themselves maintain information for computation. Each moving object also maintains the following data: its own id (oid), the query radius (r), its velocity (v), its current server ($currS$), its current location ($\overrightarrow{p}$), a list of that MOs satisfying the MCQ ($lRes$), and an event priority queue ($eventQ$). Query processing is divided into two phases, namely *initialization* and *continuous monitoring*.

A. Query Processing at Initialization

During initialization, a server S identifies all MCQs in its service region, and computes initial results for every MCQ with the following steps.

1) S broadcasts a request for data to its service region R.
2) Each moving object o in R sends $\langle oid, \vec{p}_0, \vec{v}_0, 0, r\rangle$ to S.
3) S receives the locations of all moving objects in R, populates each grid cell's list of overlapping MCQs, computes the initial result for every query, and sends it to the respective client o, along with the information on moving objects in o's alert region.
4) Upon receiving the message from S, a moving client o_i initializes its variables with the correct values and populates its event priority queue, $eventQ$.

The $eventQ$ is used to implement a concept similar to the *trigger time* [8], as follows: For each moving client o_i, the server calculates the expected time at which each current MCQ result will cease being a result. The server then sends to o_i a message containing the identifier, location, velocity, and expected exit time of each result object, in the form

$\langle t_j, oid_j, \overrightarrow{p}, \overrightarrow{v} \rangle$, signifying that object with identifier oid_j is expected to exit the query result at time t_j. These entries are indexed by $eventQ$ on the exit times. As will be illustrated later, the number of messages exchanged between clients and server is kept minimal with the help of the $eventQ$s.

Following the representation of a moving client in Section III, the squared Euclidean distance between o_1 and o_2 at time t, $t > t_{ref}$, can be computed as follows [2].

$$\begin{aligned} D_{o_1 o_2}(t) &= (\vec{p}_1(t) - \vec{p}_2(t))^2 \\ &= (p_1^{ref} + v_1^{ref}(t - t_1^{ref}) - (p_2^{ref} + v_2^{ref}(t - t_2^{ref})))^2 \\ &= (v_1^{ref} - v_2^{ref})^2 \cdot t^2 \\ &+ 2(v_1^{ref} - v_2^{ref})(p_1^{ref} - p_2^{ref} - v_1^{ref} t_1^{ref} + v_2^{ref} t_2^{ref}) \cdot t \\ &+ (p_1^{ref} - p_2^{ref} - v_1^{ref} t_1^{ref} + v_2^{ref} t_2^{ref})^2 \end{aligned}$$

We need to compute the time t when a result object exits the query circle, which is given by finding the positive root of the formula: $D_{o_1 o_2}(t) = r^2$. The calculation above is carried out by the function getExitTime().

B. Continuous Monitoring

We re-iterate that each client is responsible for monitoring its own MCQ result, preferably *without* updating the server, unless that is deemed necessary; this approach serves to reduce both the communication cost and the computational burden at the server, at the cost of a bit more local computation.

In the next sections, we show two computational models.

1) Monitoring without Mobile Regions: We first illustrate query processing in our system without using the mobile region concept.

Algorithm 1: Server Procedure without Mobile Region

```
1  foreach moving client o that sends a location update do
2      C1 ← newly contained cells by o.q;
3      C2 ← partially intersected cells in the alert region;
4      O ← {o'|o' ∈ C1.objSet ∪ C2.objSet};
5      if O ≠ ∅ then
6          O_res ← ∅;
7          foreach o' ∈ O do
8              if isResult(o', o) then
9                  t ← getExitTime(o', o);
10                 O_res.add(⟨t, o'⟩);
11         if O_res ≠ ∅ then sendMsg(O_res, o);
```

In this algorithm, as soon as a moving client o moves to a new location, it updates its server S_o about its new location and velocity. S_o computes the cells that appear in new alert regions of o and checks if there are moving clients in these new cells. Specifically, S_o does not need to send candidate messages if no other clients exist in new overlapping cells. However, it is not enough to only compute the difference between old and new alert regions. Some objects may move into o's query circle even if o does not move at all. For correctness, S_o has to send candidate objects located in the *partially* intersected cells in the new alert region, even if some of them might be already in the result of o. Note that S_o does not need to send moving objects in the *fully* contained cells in the new alert region, thanks to the maximum speed assumption. The detailed steps are summarized in Algorithm 1.

Algorithm 2: Client Procedure without Mobile Region

```
   input : O_res from the server
1  while eventQ.peek() ≤ currTime do
2      (t, o) ← eventQ.pop();
3      if O_res.has(o) then
4          t_s ← O_res.getNewExitTime(o);
5          eventQ.update(t_s, o);
6          O_res.remove(o);
7      else lRes.remove(o);
8  foreach (t, o) ∈ O_res do
9      lRes.add(o);
10     eventQ.add(t, o);
```

The procedure that takes place at the client side consists of two phases, namely the *deletion phase* and *addition phase*. The detailed steps are summarized in Algorithm 2.

The client procedure takes the message Q_{res} sent from server as input. In the deletion phase, from the $eventQ$, the algorithm pops out the result object that is expected to exit from the query circle (lines 1-2). If the result object is in Q_{res} (line 3), then the object is still in the query circle. The new expected exit time is taken from Q_{res} (line 4), and the $eventQ$ is updated with the new time (line 5). Q_{res} removes the result object to avoid duplicate operation (line 6). Otherwise, the result object is removed from the result set (line 7). In the addition phase, the result objects in Q_{res} are added into the result set, and into the $eventQ$ together with their expected exit times.

2) Monitoring with Mobile Regions: The procedure just outlined incurs high location update cost. To render our solution more efficient, we introduce the concept of a vector-based *mobile region*.

Definition 3: (Mobile Region) Given a mobile client o and a radius λ, a *mobile region* is the set of points in the circular range $\odot(o.\overrightarrow{p}, \lambda, o.\overrightarrow{v})$. The mobile region has the same velocity as the mobile client o.

In our setting, mobile regions are agreed upon by each moving client o and its server S_o at the most recent update of o. Subsequently, both parties are aware of the mobile regions. As long as o stays in its mobile region, it does not need to send location updates to S_o. We say the mobile region is *valid* for o. Since S_o does not have o's exact location, the query region of o has to be augmented by the mobile region (see Figure 1) to ensure correctness.

Lemma 5.1: Let mr_o be the mobile region for a mobile client o. Let λ be the radius of mr_o. As the server side, the new query region of o is guaranteed to contain the original query region if the new query region radius is $o.r + \lambda$. (Proof omitted)

Treating the augmented circular range as the new MCQ, S_o can compute the alert region exactly as the case without mobile regions. In this algorithm, servers keep track of the global picture, while details are left upon the clients. However,

Algorithm 3: Server Procedure with Mobile Region

```
1  foreach moving client o do
2      if receive message that o exits mr_o then
3          create new mr_o;
4          sendMsg(o, mr_o);
5      q ← augmentQuery(mr_o);
6      R ← ∅;
7      C1 ← newly contained cells in q;
8      C2 ← partially intersected cells in q;
9      O ← {o'|intersects(mr_o', C1 ∪ C2) = true};
10     foreach o' ∈ O do
11         sendMsg(o', PROBE);
12         cacheLoc(o', τ_oid);
13         if contains(q, o') then
14             R ← R ∪ {o'};
15     if R ≠ ∅ then sendMsg(R, o);
```

Algorithm 4: Client Procedure with Mobile Region

```
   input : R from the server
1  if exit(p⃗, mr) then
2      sendMsg(p⃗, v⃗);
3      mr ← mr from server;
4  while eventQ.peek() ≤ currTime do
5      (t, o) ← eventQ.pop();
6      if o ∈ R then
7          if isResult(p⃗, r, o) then
8              t ← getExitTime(p⃗, v⃗, o);
9              eventQ.update(t, o);
10             R.remove(o);
11     else lRes.remove(o);
12 foreach o ∈ R do
13     if isResult(p⃗, r, o) then
14         lRes.add(roid);
15         t ← getExitTime(p⃗, v⃗, o);
16         eventQ.add(t, o);
```

unlike in the procedures for continuous monitoring without mobile regions, S_o no longer has the exact location of the query object qo. Instead, knowing the augmented alert region of qo, it identifies cells that *may* contain candidate results for qo, followed by probing the exact locations of these candidates and caching their exact locations. S_o then sends qo a candidate message with the exact locations. Upon receiving the candidate results, qo has to filter them based on its own location and query radius.

If a moving client exits its mobile region, a new mobile region is created to continuously bound its movement. With the mobile regions, location updates can be reduced greatly. However, this comes at the expense of increasing probe messages. This tradeoff is investigated in the experimental studies.

It is important for server S_o to cache exact locations of some moving clients because these exact locations may be re-used for queries issued by other moving clients in future. Caching exact locations can potentially save a large number of probe messages. But it is also important to allocate a cache of a small size to save precious memory space. In our implementation, the server allocates a fixed amount of buffer space to cache exact locations, e.g., 2KBytes. In addition, the server assigns identical life time τ_{oid} to each moving client o. Every time the exact location of o is re-used, its life time is renewed. If by τ_{oid} the exact location of o is not re-used, it is removed from the buffer. The detailed server procedure is summarized in Algorithm 3.

In this setting, the objects must send updated locations to their server when they exit their mobile regions. Each client also monitors its own result changes based on the $eventQ$ and its current location. As in the procedure without mobile regions, this procedure has a deletion and an addition phase. In the deletion phase, the client removes any previous results that no longer satisfy the query. In the addition phase, the moving client verifies which candidates are true results according to candidate-message sent by the server and then updates both the $lRes$ and $eventQ$ accordingly. The detailed client procedure is summarized in Algorithm 4.

C. Cross Boundary Queries

When the query circle of a client o covers several service regions, the server S_o answer the query by communicating with respective servers.

Figure 2 illustrates a scenario where a moving object o has its query cover both service regions of S_1 and S_2. S_1 probes the exact location of o and sends it to S_2, together with the query radius $o.r$ and the mobile region $o.\lambda$. According to Lemma 5.1, to ensure the correctness of the result of o, S_2 overlaps the new query region of radius $(o.r + o.\lambda)$ and retrieves its overlapping cells. Recall that each cell maintains a list of identifiers of objects inside the cell. S_2 probes the locations and velocities of these clients and sends them to S_1. S_1 computes and sends the candidate set for each query object as usual. This approach can be easily generalized to the case where a query overlaps with multiple service regions.

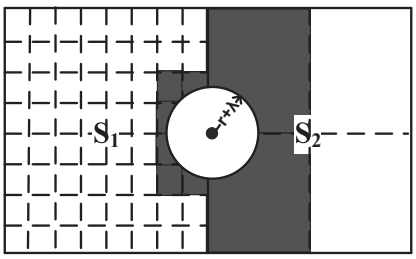


Fig. 2. A Cross Boundary Query

D. Client Handover

When an object moves across a region boundary, the server of the old region informs the server in the new region by sending it a handover message. Upon receiving a handover message, the server in the new region probes the exact location of the moving client, and send candidate results in its service region. The handover should occur seamlessly, in that the handover should not affect the correctness of the result of the object. Handling this type of event is particularly straightforward in our query processing scheme because the

moving clients compute the exact results. The only thing that the moving clients have to be aware of is to send location updates to the server responsible for the new service region.

VI. System Optimization

In this section, we outline how our system adapts itself to a highly dynamic environment by dynamically re-allocating service regions to servers.

In the applications that we target, the objects issuing queries and being queries move continuously. At times, a significant number of these objects may locate in the service region of a single server, possibly due to an event that has occurred at that site. The server in question is then overloaded. Meanwhile, some other servers may be idle because there are only few moving objects to serve in their regions. Under such circumstances, a necessity arises to re-balance the workload among servers so that the quality of service is kept stable.In order to detect this kind of circumstance, we follow a simple yet effective strategy to estimating each server's status. We use the number of moving clients as an indicator. Ideally, each server has $\frac{N}{M}$ moving clients if the system has N objects and M servers. A server is overloaded if it is serving more than $\frac{\beta N}{M}$ objects, where $\beta > 1$. If β is too large, a server may get overloaded and unnoticed, and the performance of the system deteriorates. But if β is too small, system re-balancing happens too often, which also wastes system resources. In our experiments, we set $\beta = 3$.

A. Adjusting the Service Region Allocation

When skew causes a server S to become overloaded, S divides its service region into two halves, and hands over the moving objects in the one half service region to a lightly utilized server S' so as to balance the workload. Next, the framework also seeks to merge *compatible* regions that are adjacent, and have the same width and height, in order to (1) prevent space fragmentation and (2) to further balance the workload.

Dynamic Allocation: In the server cluster, each server continuously monitors the number of clients in its service region. When a server S is overloaded for a period longer than t_d, Algorithm 5 is invoked to dynamically adjust the service region allocation, in order to balance the workload among the servers. Suppose server S with a service region R is overloaded.

In Algorithm 5, the function getServiceRegions returns the current set of service regions. The function numClients returns the number of clients in a service region as monitored by a server. Each server by definition should have one region, but the region may not be a connected region. In line 3, the function divideRegion divides a given region into two equal halves. In line 4, the server S gets another server, S', with the least clients by sending broadcast messages to the server cluster. Then the half region with less clients is allocated to S' (lines 5–8). The function allocate(S, S', R') allocates the region R' belonging to S to S'. Each object in R' then updates their server to S' (line 9).

Algorithm 5: DynamicAllocation

```
1  while S.numClients() > βN/M do
2      R ← getServiceRegions();
3      (R1, R2) ← divideRegion(R);
4      S' ← the server with the least clients;
5      if R1.numClients() > R2.numClients() then
6          R' ← R2;
7      else R' ← R1;
8      allocate(S, S', R');
9      foreach o ∈ R'.getObjSet() do o.currS ← S';
10 foreach Region R1 do
11     foreach Adjacent Region R2 do
12         n ← R1.numClients() + R2.numClients();
13         if compatible(R1, R2) ∧ (n < βN/M) then
14             S1 ← getServer(R1); S2 ← getServer(R2);
15             if S1 ≠ S2 then
16                 allocate(S1, S2, R1);
17                 R ← merge(R1, R2);
18                 S2.setServiceRegions(R);
19                 if S1.getServiceRegions() = ∅ then
20                     Rm ← the region with the most objects;
21                     (R1', R2') ← divideRegion(Rm);
22                     allocate(Sm, S1, R1');
23             else merge(R1, R2);
```

In next loop, every pair of adjacent service regions are compared to see if they can be merged (line 13). If any two compatible regions of different servers are merged, the new region is allocated to one server (lines 14–18). Next, if the other server is idle, the server with the heaviest workload allocates half of its service region to the idle server (lines 19–22). On the other hand, if two compatible regions belong to the same server, they are simply merged (line 23).

Extension to Dynamic Cell Side Lengths: As discussed above, when the system is initializing, the bootstrap server picks up a value for its cell length, calculates its σ, with which the remaining servers calculates their respective cell length α with Equations 1 and 2.

However, a need for dynamically adjusting the cell side length arises when data are highly skewed at certain servers. In our setting, each server monitors the distribution of the mobile clients in its service region. If a server detects skew, it is allowed to adjust its cell length. Specifically, if a server finds that a single cell holds more than γ of all objects in the service region for a period longer than a threshold t_e, the server halves its current cell side length and rebuilds its grid. This process continues until no cell has more than γ of total moving clients. Once the distribution of moving clients is no longer skewed, the server restores its original cell length, which was saved before the adjusting occurred. Note that cell side length adjust should not happen very often because of the expensive index rebuild. In the experiments, we empirically set γ to be 50% and set t_e to be 10 time points.

Extension to Multiple MCQs by One Client: We also consider relaxing the assumption that one moving client is issuing one MCQ at any time. Our system could also be used to satisfy the needs of a mobile client to issue many MCQs. In

this case, many queries may be sharing the same focal object or the same alert region, and a query grouping technique may be employed to reduce both the communication cost and the computation workload at server side. Specifically, if a moving client issues three MCQs at the same time, and thus three alert regions at server side. The server could group these MCQs and return the candidates in one message instead of three messages. Server could also use minimal encoding to indicate in which alert region a candidate is located in order to reduce client side processing.

VII. Experiments

In this section, we evaluate the effectiveness and efficiency of the proposed distributed architecture. All experiments are carried out on a Linux machine with an Intel(R) Core2 Duo 2.33 GHz CPU and 16GB memory. Each simulated server is assigned 1GB memory. With the data sets, our simulation replays the movement of the moving clients, and records the server workload and the messages sent by both clients and servers. We use the following metrics for our performance study: (1) query result change rate, (2) message rate between servers, (3) message rate between server and clients, (4) server workload (running time for processing the messages), and (5) workload at moving object side (average number of queries each moving object monitors). We adapt the RMD method [17] (see Section II), and use it as a baseline for our comparison. We set the parameters in the RMD algorithm according to the recommendations used in [17], namely the scale factor has default value 2.

The data sets were generated using Brinkhoff's the road-network-based moving-objects-generator [3]. The moving object datasets consist of the location update reports for each moving object, at each time point. Three default velocity values (slow, middle, and fast) are used to generate data of objects moving at different speeds. The Oldenburg map is used to generate trajectories. The generated data space has width 23,572 meters and height 26,915 meters. Our simulator generates cells that cover this area. As we have discussed, the cell side length α is an internal parameter for each server; its impact on the system performance is studied in our experiments. All parameters under investigation are gathered together in Table II.

Parameter	Default	Range
Query radius r (meters)	20	10–50
Mobile region radius λ (meters)	20	10–50
Cell side length α (meters)	40	20–100
Number Moving Clients N	300k	300k–500k
Number Servers M	8	4–12

TABLE II
Experimental Parameter Settings

A. Effectiveness of Server Architecture

In our system, servers divide their service regions among themselves according to the distribution of moving clients therein so as to balance their workloads. In our first experiment, we try various numbers of moving clients, and check the workload of each server. Figures 3(a) and 3(b) show our results on each server's workload (i.e., time taken) as a function of the number of moving clients. Server 1 is pre-selected as the bootstrap server, therefore it has a bit more workload than the other servers. But we still observe that the servers share the workload almost equally.

B. Varying Grid Side Length

We now investigate the effect of the (default) side length of grid cells. Remember that setting grid cell side length at the bootstrap server also determines the grid cell side length of other servers during initialization phase. In this set of experiments, we set various cell side lengths at the bootstrap server. This comparison is only between the two variants of scheme, NMR and MR, since these settings do not apply to RMD. Figures 3(c) and 3(d) show results for computational load and exchanged messages, respectively. We observe that performance is best with a cell length of 40 meters, which we choose as our default setting.

C. Varying Mobile Region Radius

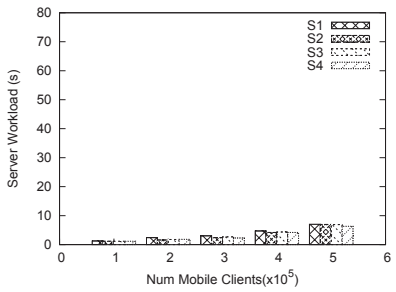
We now evaluate the effect of the mobile (safe) region radius on both servers' workload and communication cost with our MR variant. Figure 3(e) shows our results. As expected, as the size of mobile regions grows, the server has to probe and transmit message about more candidates that fall into them, hence the communication cost grows at the sever-side. On the other hand, as the size of safe regions grows, events in which a client exists its safe region become less frequent, and hence the communication cost of uplink message sent by clients drops.

D. Effect of Number of Servers

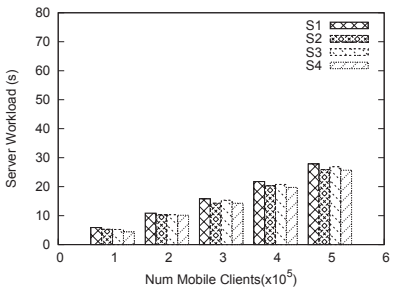
In this section, we also study the scalability when the number of servers increases. Figure 3(f) presents the amount average workload on each server when the number of servers increases from 4 to 12. It is observed that with the increase of number of servers, the average workload decreases *linearly* which indicates that the architecture is able to scale well.

E. Client Handover

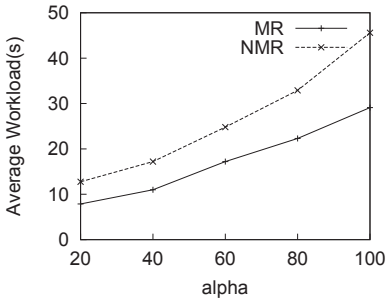
In this set of experiments, we examine the number of moving clients that actually cross the boundaries of two servers. Both the total number and the speed of moving clients affect the number of handovers. Therefore, we generate data sets with size varying from 100K to 500K. The speed for each data set can be either slow or fast, which is the default value of the data generator. From the result shown in Figure 4, we observe that for each data set with slow speed, the handover rate is around 1% with smaller variation. In contrast, for each data set with fast speed, the handover rate can be as high as 6%, with bigger variation. As expected, the speed of moving clients has a great impact on the rate of handovers. It also causes bigger variations because the servers have shorter time to respond to the event that many moving clients cluster in



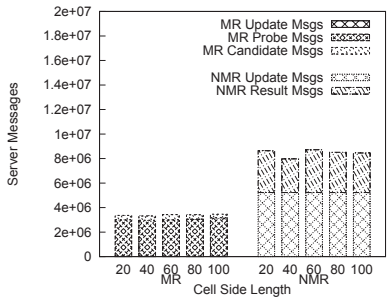
(a) Server Workload, MR



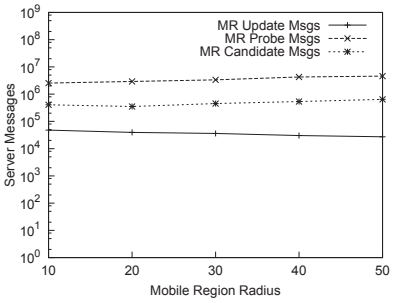
(b) Server Workload, NMR



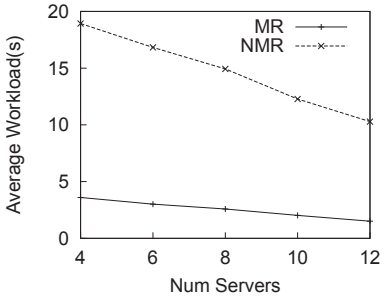
(c) Server Workload vs. Cell Length



(d) Server Messages vs. Cell Length



(e) Effect of Mobile Region Radius



(f) Effect of Number of Servers

Fig. 3. Server Workloads & Messages

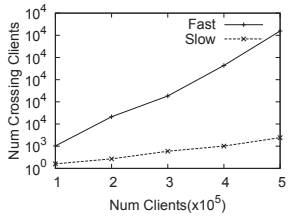


Fig. 4. Handovers

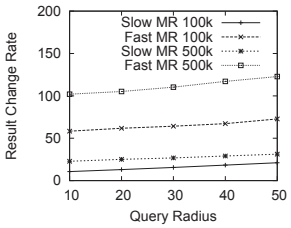


Fig. 5. Query Result Change Rate

one region. In summary, this set of experiments shows that the proposed architecture is effective in adapting to highly dynamic environment.

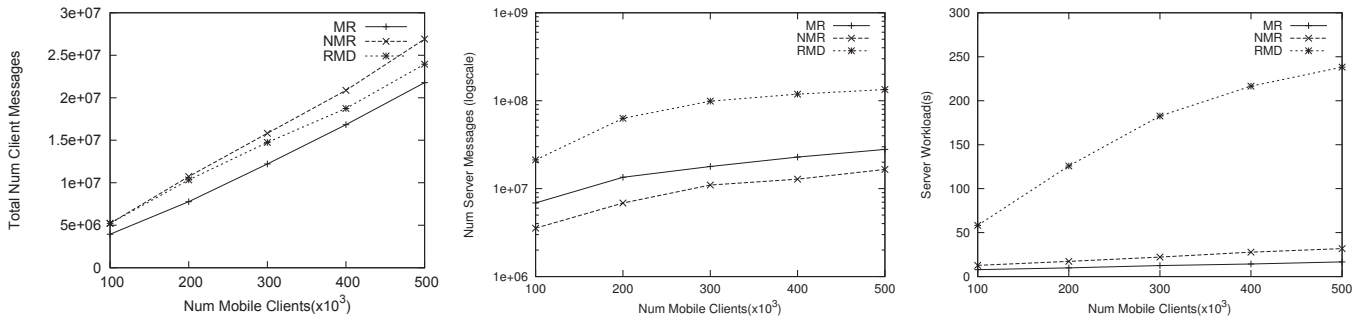
F. Query Result Change Rate

In this set of experiments, we conduct a sensitivity test on how the population size and the speed of moving clients affect the result change rate. We generated four different data sets: Slow100k, Fast100k, Slow500k, and Fast500k. As the names suggest, we have two data sets with 100k moving clients, and two with 500k moving clients. The speed for each data set can be either slow or fast which are the default values of the generator. We measure the average times that moving clients result changed for each time point against query radius. Figure 5 shows the results. We observe that both higher speed and larger population increase the result change rate. However, speed has greater impact than population.

We only show the result for MR algorithm. NMR exhibits the same results since speed and population affect only performance, not correctness. This set of experiments show that both MR and NMR can capture the frequent MCQ query result changes. Unlike centralized systems where the query processing is based on periodical location updates, MR and NMR perform true continuous monitoring. In MR and NMR, servers handle fast changes as they perform light computations, delegating part of the workload to the moving clients. Thus, they avoid the problem that the more frequently the query result changes, the more likely that it is the server gives wrong results as it occurs in a centralized solution due to the delay of updates.

G. Effect of Number of Moving Clients

We also study the scalability of our method in the number of moving clients. Figure 6 presents the results on client messages, server messages, and server computational workload. Figure 6(a) shows that NMR has the most client messages, whereas MR has the fewest client messages, thanks to the use of mobile regions. For the same reason, RMD has fewer messages than NMR. Figure 6(b) is logscaled on y-axis for better viewing. Both NMR and MR outperform RMD because of the push of computation to the client side. Quite a number of messages (about one order of magnitude) are actually saved in both MR and NMR. There are more messages in MR because of probe messages sent by servers. Figure 6(b) shows server workloads for the three methods. We compared total time of the four servers in MR and NMR with the time in RMD for fairness. Both MR and NMR outperform RMD by more than 50% in terms of computation time at the server side. These results indicate the scalability advantage of our methods in comparison to the RMD scheme. Again, these advantages are due to the exploitation of computational capacities at the client side.

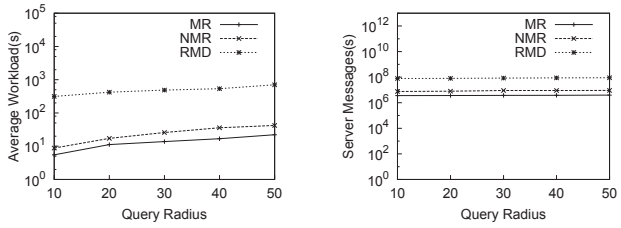


(a) Client Messages (b) Server Messages (c) Server Workload

Fig. 6. Effect of Number of Moving Clients

H. Varying Query Region Radius

We also experimented with tuning the query region radius. The advantage of our system is allowing each client to select its customized query radius. But in order to compare with RMD, which assumes the same radius for each moving client, we set query radius to be the same value. Results are shown in Figures 7(a) and 7(b). It is observed that as the query radius grows, the number of cells in each client's alert region also grows, hence the number of candidate message sent by servers increase with all algorithms. Still, both variants of our scheme outperform RMD in this experiment. This result verifies our expectations and reconfirms the advantages of our method. It also observed that MR improves the performance by a factor of 6 compared to RMD.



(a) Server Workload (b) Server Messages

Fig. 7. Effect of Query Region Radius

VIII. Conclusions

We address the problem of answering multiple moving continuous queries in an environment where each moving client poses its own range query. We propose a distributed client-server architecture for efficiently processing such queries. In our solution, a cluster of interconnected servers takes care of the global view, while part of the computation is relegated to the clients to verify actual results. We demonstrate the effectiveness and efficiency of our approach through extensive simulation experiments. Our method reduces both the server-side workload and the client-server communication cost in comparison to the most recent state-of-the-art scheme, while it is much more scalable to growing number of moving clients.

Acknowledgments

Xiaohui Li and Kian-Lee Tan would like to acknowledge the support of NExT Research Center funded by MDA, Singapore, under the research grant: WBS:R-252-300-001-490.

References

[1] A. Amir, A. Efrat, J. Myllymaki, L. Palaniappan, and K. Wampler. Buddy tracking - efficient proximity detection among mobile friends. *Pervasive and Mobile Computing*, 3(5):489–511, 2007.

[2] R. Benetis, C. S. Jensen, G. Karčiauskas, and S. Šaltenis. Nearest and reverse nearest neighbor queries for moving objects. *The VLDB Journal*, 15(3):229–249, 2006.

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

[4] S. Chen, B. C. Ooi, and Z. Zhang. An adaptive updating protocol for reducing moving object database workload. *PVLDB*, 3(1-2):735–746, 2010.

[5] T. Farrell, K. Rothermel, and R. Cheng. Processing continuous range queries with spatiotemporal tolerance. *IEEE Trans. on Mobile Computing*, 10(3):320–334, 2011.

[6] B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

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

[8] G. S. Iwerks, H. Samet, and K. P. Smith. Maintenance of *k*nn and spatial join queries on continuously moving points. *ACM Trans. Database Syst.*, 31(2):485–536, 2006.

[9] C. S. Jensen, D. Lin, and B. C. Ooi. Query- and update-efficient B^+-tree-based indexing of moving objects. In *VLDB*, pages 768–779, 2004.

[10] C. S. Jensen and S. Pakalnis. Trax: real-world tracking of moving objects. In *VLDB*, pages 1362–1365, 2007.

[11] 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, 2004.

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

[13] G. Treu, T. Wilder, and A. Küpper. Efficient proximity detection among mobile targets with dead reckoning. In *MobiWac*, pages 75–83, 2006.

[14] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.

[15] H. Wang, R. Zimmermann, and W. shinn Ku. Distributed continuous range query processing on moving objects. In *DEXA*, pages 655–665, 2006.

[16] W. Wu, W. Guo, and K.-L. Tan. Distributed processing of moving k-nearest-neighbor query on moving objects. In *ICDE*, pages 1116–1125, 2007.

[17] M. L. Yiu, H. U. Leong, S. Šaltenis, and K. Tzoumas. Efficient proximity detection among mobile users via self-tuning policies. *PVLDB*, 3(1):985–996, 2010.