Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2015, Article ID 273131, 16 pages
http://dx.doi.org/10.1155/2015/273131

Research Article

Sleepwalk: Scalable and Energy-Efficient Processing of Continuous Range Queries for Location-Aware Mobile Computing

MoonBae Song

Mobile Communications Division, Samsung Electronics Co., Ltd., 416 Maetan-dong, Yeongtong-gu, Suwon, Gyeonggi-do 443-742, Republic of Korea

Correspondence should be addressed to MoonBae Song; mbsong@gmail.com

Received 13 February 2015; Revised 30 June 2015; Accepted 6 July 2015

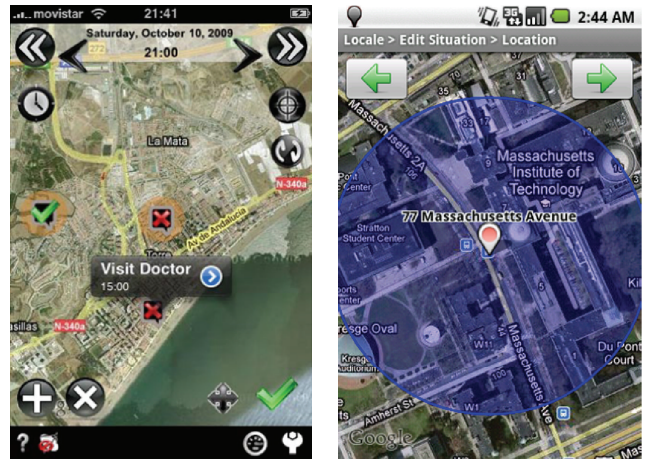
Academic Editor: Jinsong Han

Recently, monitoring queries are getting attention for various real-life applications such as safety, security, and personalization services. This work proposes a distributed sensing and monitoring technique (called *Sleepwalk*) for continuous range queries with energy- and computation-efficient optimizations. In our scheme, each mobile client (MC) is aware of its nearby monitoring queries by leveraging its processing power. The proposed Sleepwalk has three major contributions. First, with *piecewise* linear movement assumption and motion vector $\overline{v}$, it can locally preevaluate every possible query result in advance *in bulk* and sends them to the server at once. We also provide a timestamp-based invalidation technique for efficiently removing failed preevaluated results by computing the smallest valid timestamp. Second, an energy-conserving technique that repeatedly sleeps off MCs whenever possible is proposed by calculating the *safely sleepable* time. Third, we provide a set of localized query optimization techniques for MCs' local query subset using plane-sweeping, which effectively minimize search space. Extensive experiments indicate that Sleepwalk technique remarkably outperforms existing state-of-the-art techniques in terms of server scalability, communication cost, and energy consumption of MCs.

1. Introduction

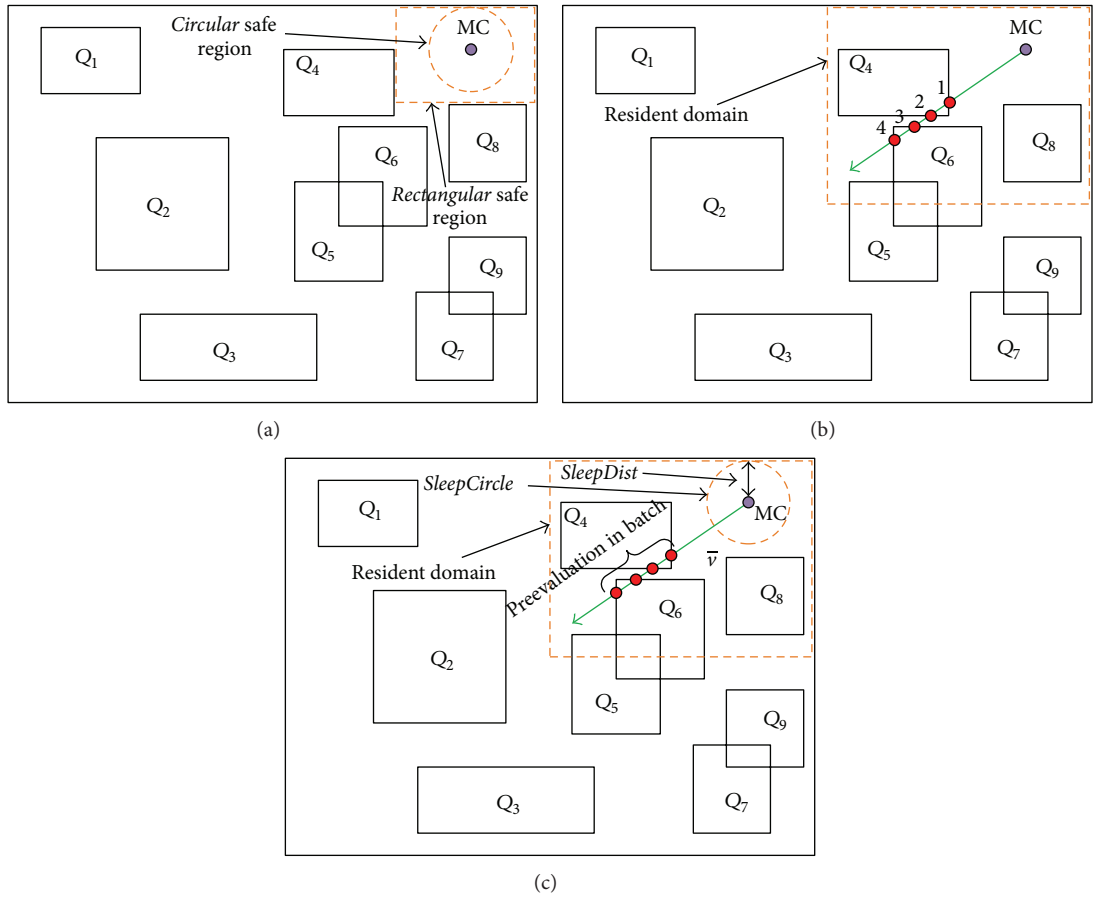
With emergence of GPS-capable mobile devices and introduction of 3G and 4G mobile broadband networks, location-based services (LBSs) are rapidly becoming all-pervasive [1, 2]. More importantly, *location-based sensing and monitoring* (LBSM) services, which monitor mobile objects in a geographic region of interest, have only recently started to receive attention. Representative applications of LBSM are traffic monitoring for a specific region, area monitoring for sending mobile coupons, child-caring with real-time monitoring in a dangerous area, and triggering special tasks with a given region (see Figure 1). LBSM can be formalized as continuous range queries processing with a large number of mobile objects (e.g., mobile clients (MCs)). Such services impose heavy workloads on the central server for recomputing and managing real-time query results continuously. Moreover, moving objects (in the work, moving objects and mobile clients are used interchangeably) should transmit their location continuously in order to support up-to-date query results. This incurs a huge communication cost and excessive energy consumption of battery-powered mobile clients.

In order to address this problem, numerous works have proposed in a centralized setup (e.g., see [3, 4]) and a distributed setup (e.g., see [5, 6]). Centralized approaches only can improve server's scalability while distributed approaches can also do the communication cost of MCs by leveraging their processing power. As representative techniques of the latter, Figure 2 shows the conceptual differences between the proposed technique and two existing works such as Q-index [7] and Monitoring Query Management (MQM) [8]. Similarly, these introduce some kind of spatial region called *safe region* and *resident domain* (RD), respectively, for reducing communication costs. In Q-index technique [7], an MC filters out every location update within its safe region



(a) (b)

FIGURE 1: Real-world example from practical and real mobile applications: aToDo Map app for iPhone (a), LOCALE app for Android (b).



(a) (b)

(c)

FIGURE 2: Conceptual differences: (a) Q-index technique with the concept of safe region, (b) MQM technique with the concept of resident domain, and (c) the proposed technique with preevaluation and selective sleep.

and sends location update only if it exits its safe region (cf. Figure 2(a)). In this work, however, both processing range queries and computing safe regions (with $\mathcal{O}(n)$ to $\mathcal{O}(n \log^3 n)$ computations) are performed in server. Consequently, the server is not very scalable, and each MC's processing power is not fully utilized. In MQM technique [8], the server effectively distributes the query workload over a number of MCs by leveraging their processing power; for example, in Figure 2(b), a query set $\{Q_4, Q_5, Q_6, Q_8\}$ with RD is allocated and cached as MC's local query subset by server based

on its location and processing capability. By referring to the *cached* local query subset, MC is capable of evaluating query results locally and autonomously. During that, all unnecessary movements irrelevant to the query results can easily be filtered out; then no updates are triggered. MC sends an update only when it exits its RD or its query results are updated.

Existing distributed monitoring techniques [7, 8] have two major shortcomings. One is the lack of consideration of MC's motion vector; that is, most of previous works focused only on point location with oversimplified mobility assumption. Hence, the performance improvement in query processing is limited. For instance, 4 updates of MQM technique in Figure 2(b) can be reduced into a single update which includes 4 predictive results in advance by linear prediction. The other one is no harnessing of *sleep mode* (or power-saving mode) for reducing energy consumption, which is supported in every modern mobile processor. As long as accurate results are maintained, mobile devices should transit themselves into sleep mode as possible proactively. To our knowledge, there are only limited researches on proactively exploiting the sleep mode in continuous range query processing.

Motivation. In the light of the weakness of existing techniques, we propose a scalable and energy-efficient technique (called *Sleepwalk*) for processing continuous range queries. In addition to the virtue of MQM technique, we present three major improvements: (1) bulk preevaluation of query results with linear prediction, (2) energy-conserving strategy by selective sleep-off, and (3) localized query optimizations for MCs. First, by exploiting MC's motion vector $\overline{v}$, we preevaluate all possible query results in advance in bulk. Thus, four separate intersecting points (i.e., query results) in Figure 2(b) can be preevaluated in advance and aggregated into a single query result set including them. Failed preevaluated results are efficiently removed by a timestamp-based invalidation technique. The query processing cost and communication cost will be greatly reduced as MCs' movements are close to linear movements and the number of intersecting points increases. Second, in order to conserve energy consumption of MCs, we propose a selective sleep-off technique that repeatedly transits MCs into sleep mode whenever possible; the safely sleepabletime (called *SafeSleepTime*) can be calculated by considering maximum velocity $v_{\max}$ and minimum perimeter distance to the nearest query (*SleepDist*) (cf. Figure 2(c)). Third, we provide a set of localized query optimization techniques using plane-sweeping over MCs' local query subset, which effectively minimize search space. Basic idea is to sort local query subset along an axis and maintain lower- and upper-bound of it for computing query results in a computationally efficient manner. Through these improvements, our Sleepwalk proposal can achieve low server maintenance cost and processing delay, minimized energy consumption of MCs, minimized communication cost, and low computational cost in MCs in comparison to the existing state-of-the-art techniques.

Organization. The rest of the paper is organized as follows: Section 2 provides the details of *Sleepwalk* techniques with its data structures and distributed query processing. In Section 3, we discuss advanced techniques for further improving the proposed approach. Section 4 presents the results of a performance evaluation of the proposed and previous studies. Section 5 gives the related work on the subject. Finally, we conclude in Section 6 with directions for future work.

2. Sleepwalk: Efficient Processing of Range Monitoring Queries

2.1. The Basic Idea and Design Goals. In a systematic viewpoint, the proposed system architecture is a set of a single central server (as *mediator*) and numerous mobile clients (as distributed query processors) with positioning, computational, and communication capabilities. The basic idea is to make the utmost use of MCs' processing power to improve system's performance in terms of *server scalability*, *communication cost*, and *the energy consumption of MCs*. To this end, based on the virtue of distributed query processing discussed in Section 1, we propose the following techniques.

Bulk Preevaluation of Query Results with Linear Prediction. By utilizing the linear prediction technique with MC's motion vector, we can precompute every possible query result in advance in bulk. If such prediction is correct within the location error threshold δ (location error is inherent due to sensor error, continuous movement of moving objects, and communication delay, and its threshold δ is application-specific. The distance between actual location $p(t)$ and the estimated location $\widehat{p}(t)$ does not exceed the threshold δ; i.e., $|p(t), \widehat{p}(t)| \leq \delta$, where $|\cdot,\cdot|$ is defined as distance between two points), query processing can be performed without any message transmission; simply the oldest result is consumed and influenced to query results. If it fails, however, the results should be recomputed totally. Assuming the *piecewise linear* movement of real-life moving objects (The movement of real users cannot be described as a single line (i.e., a linear function) but as a set of line segments, which is called piecewise linear function (i.e., a function composed of line segments). The approach is proved to be effective in the area of research with various works such as [9].), this technique will greatly decrease the number of transmissions between server and MCs, so that the communication cost between them is minimized.

Selective Sleep-Off. The *sleep mode* is mainly designed for energy saving. In the Hobbit chip from AT&T, for instance, the ratio of energy consumption in the active mode to the sleep mode is 5,000 [11]. Thus, most of energy-efficient approaches largely depend on the policy of "switching into sleep mode whenever possible" without interfering MC's main task. At least to our knowledge, only few works have been done so far on proactively exploiting the sleep mode for continuous range query processing. Assuming each MC's maximum velocity, we can define safe period of time (called *SafeSleepTime*) which ensures the correctness of query results during sleep.

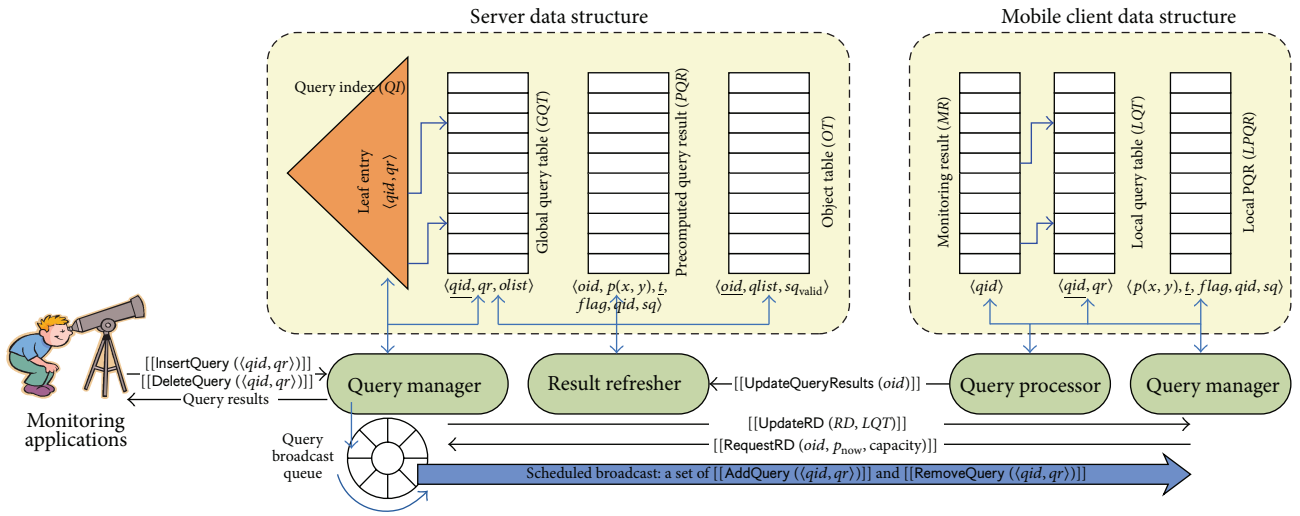


Figure 3: Client/server-side data structures and interchanged messages between them.

Localized Query Optimizations. Despite various woks on minimizing the communication cost and energy consumption of MCs, there is still a lack of research in optimizing the localized query processing of MCs. Based on a plane-sweeping algorithm, we study a lightweight optimization technique by pruning unnecessary search space of MC's local query table.

2.2. Client/Server Data Structure Design

2.2.1. Server Data Structures. As illustrated in Figure 3, the server acts as a mediator of a large number of mobile clients; it manages monitoring queries and their up-to-date results and allocates optimal query subset to each MC:

(i) Global query table (GQT): GQT is a table of all registered *stationary* monitoring queries $\langle \underline{qid}, qr, olist \rangle$ in a rectangle shape qr, indexed by a query identifier qid. Each tuple's $olist$ is the list of oid intersected with its query rectangle qr (i.e., $oid \in OT$) (for the sake of simplicity, monitoring queries are abstracted as rectangles in this work. However, these can be extended to any kinds of shape including circles and polygons).

(ii) Preevaluated query result (PQR) is a sorted list (or heap) of all preevaluated results $\langle oid, p(x, y), \underrightarrow{t}, flag, qid, sq \rangle$ from mobile object oid in ascending order of time t ($\underrightarrow{t}$ means that each entry in PQR is sorted in ascending order of t). Each tuple means that object $oid \in OT$ will enter or exit the monitoring query $qid \in GQT$ at point $p(x, y)$ at time t ($flag$ is either "`enter`" or "`exit`"). Since each PQR entry is predictive, the failed prediction should be purged efficiently; in the case it is called *expired*. To minimize the purging cost, every PQR entry is timestamped by the sequence number sq generated by each MC's sequence generator *sqGen*.

(iii) Object table (OT) is a table of all registered objects in forms of tuple $\langle \underline{oid}, qlist, sq_{\text{valid}} \rangle$ indexed by object-id oid. The list of all intersected queries is $qlist$, and sq_{valid} is the smallest sequence number of all preevaluated results in PQR. Thus, expired PQR entries can be identified by referring to sq_{valid} (cf. Section 2.3.3).

(iv) Query index (QI) is a spatial index over GQT. It can be either an R-tree [12] or a grid-file. Thanks to simplicity and efficiency of MQM technique, and for fair comparison, we exploit its BP-tree index in this work (cf. Section 2.3.1).

2.2.2. Mobile Client Data Structures. Each client maintains a local subset of the global query database (called GQT) in server:

(i) Resident domain (RD) is a unique spatial region which guarantees that MC's current location $\dot{p}$ is always enclosed by its RD. Local query table (LQT) is a cached subset of GQT with respect to RD. More clearly, $LQT = \{q \in GQT \mid q.qr \cap RD \neq \emptyset\}$ and each of its entries is form of $\langle qid, qr \rangle$. Each MC's *capacity* indicates its processing power ($capacity \geq |LQT|$).

(ii) Monitoring result (MR) is a set of queries intersecting with MC's location $\dot{p}$ at current time $\dot{t}$ (we assume that every MC and server share synchronized clocks $\dot{t}$).

(iii) Locally preevaluated query result ($LPQR$) is oid's original source of PQR and a sorted list of *locally* preevaluated results (or predicted points) in ascending order of t. Each point is computed by linear prediction using motion vector $\overline{v}$, and similar to PQR entry it is represented as a tuple $\langle p(x, y), \underrightarrow{t}, flag, qid, sq \rangle$ with the same meaning.

For supporting server to efficiently filter out "expired" PQR entries, each MC timestamped its LPQR entries using its *sqGen*, a monotonic increasing sequence generator. Actually,

```
// Performed upon receiving the following messages
ServerMessageHandler()
(1) Upon receiving [[InsertQuery (q⟨qid, qr⟩)]]:: // from applications
(2)     Insert q⟨qid, qr⟩ into QI and GQT;
(3)     BroadcastMsg([[AddQuery (q⟨qid, qr⟩)]]);

(4) Upon receiving [[DeleteQuery (q⟨qid, qr⟩)]]:: // from applications
(5)     Delete q⟨qid, qr⟩ from GQT and QI;
(6)     BroadcastMsg([[RemoveQuery (q⟨qid, qr⟩)]]);

(7) Upon receiving [[RequestRD (oid, p, capacity)]]:: // from an MC with oid
(8)     Perform a top-down search until domain node's size ≤ capacity; Choose RD and LQT;
(9)     SendMsg(oid, [[UpdateRD (RD, LQT)]]);
```

Algorithm 1: Query management with query index in server.

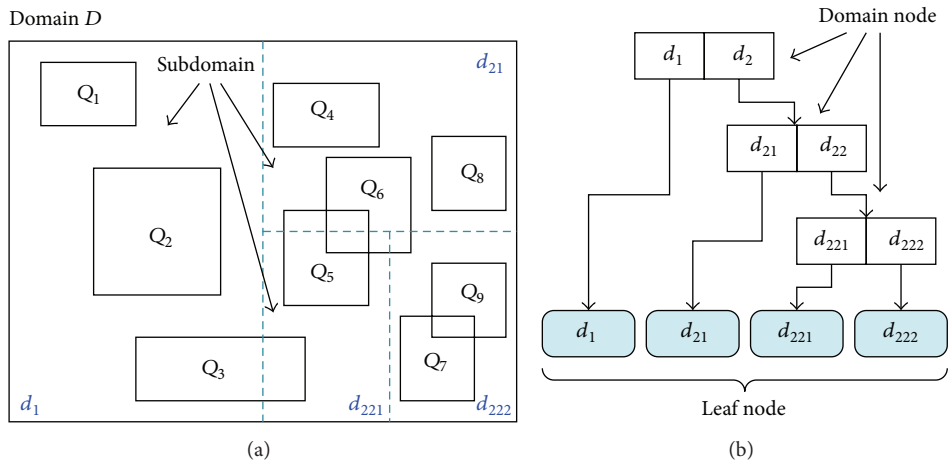


Figure 4: BP-tree with center split (the leaf node's capacity is 4): (a) domain partitioning, (b) BP-tree directory structure.

PQR is the aggregated global view of all LPQRs that are reported to server and appended into PQR whenever updates occur. Each MC synchronously consumes $p_i \in LPQR$ and updates its MR, whenever server does $p_i \in PQR$. For denoting the end of preevaluation, the last LPQR entry's $flag$ should be "`end.`"

2.3. Server-Side Processing. In server, query manager manages monitoring queries in GQT and QI, while result refresher does the up-to-date query results from MCs.

2.3.1. Query Management with Query Index. Algorithm 1 describes ServerMessageHandler() needed to deal with 3 messages for query management with query index (in the following, messages handled by MessageHandler() are denoted as [[MessageName $(arg_1, arg_2, \ldots)$]]). Once a monitoring query is issued (or deleted) by a user using [[InsertQuery $(q\langle qid, qr\rangle)$]] (or [[DeleteQuery $(q\langle qid, qr\rangle)$]]), it should be registered (or deleted) in the server's GQT and QI and then notified to the relevant MCs via broadcasting (without loss of generality, we assume one-to-all broadcasting via wireless channel without base station-level filtering discussed in [13]. We also do not consider communication delay and failures). In order to efficiently manage monitoring queries, as our query index (QI), we exploit BP-tree scheme with *center split* approach proposed in [8]. As shown in Figure 4, domain D is recursively partitioned along x- and y-axis in turn, into a set of subdomains until each subdomain contains fewer queries than leaf node's capacity. Capacity of BP-tree's leaf node is predefined, which is strongly related to MC's minimum processing capacity. Once a new query $q\langle qid, qr\rangle$ is inserted, its fraction $qr \cap D_i$ should be inserted into every subdomain D_i intersecting with it. When an MC requests its effective subset (RD and LQT) with its location and computational capacity ([[RequestRD $(oid, p, capacity)$]]), the server should be able to respond with [[UpdateRD (RD, LQT)]] by searching QI in a top-down manner. See [8] for more details about BP-tree.

2.3.2. Scheduled Broadcasting. Broadcasting every update in GQT is required to maintain consistency between server and MCs. In order to optimize the broadcasting frequency, each update is scheduled to be broadcasted in a batch with a certain

```
// Performed upon receiving the following messages
ServerMessageHandler()
(1) Upon receiving [[UpdateQueryResults (oid, MR, LPQR)]]::
(2)    foreach qid ∈ OT[oid].qlist do
(3)        GQT[qid].olist ← GQT[qid].olist \ {oid};
(4)    OT[oid].qlist ←MR;
(5)    foreach qid ∈ OT[oid].qlist do
(6)        GQT[qid].olist ← GQT[qid].olist ∪ {oid};
(7)    foreach p ∈ LPQR do // Enheap all LPQR entries into PQR
(8)        PQR.enheap(⟨oid ‖ p⟩);
(9)    OT[oid].sq_valid ← min(e.sq, ∀e ∈ LPQR); // update OT

ServerProcessing() // Result refreshing in server
(10) while PQR.top.t ≤ t_now and PQR ≠ ∅ do
(11)     e ← PQR.deheap();
(12)     if OT[e.oid].sq_valid > e.sq then continue; // expired PQR entry
(13)     if e.flag = enter then
(14)         GQT[e.qid].olist ← GQT[e.qid].olist ∪ {e.oid};
(15)         OT[e.oid].qlist ← OT[e.oid].qlist ∪ {e.qid};
(16)     else if e.flag = exit then
(17)         GQT[e.qid].olist ← GQT[e.qid].olist \ {e.oid};
(18)         OT[e.oid].qlist ← OT[e.oid].qlist \ {e.qid};
(19)     end-if
(20) end-while
```

Algorithm 2: Management of query results in server.

broadcast interval ω. With the expected latency of $\omega/2$, this can reduce the number of broadcasts as well as the communication cost. As a primitive, BroadcastMsg (*msg*) enqueues the update message *msg* into server's broadcast queue which maintains all updates in time window of $[\dot{t}-accQ+1, \dot{t}]$ (called *broadcast window*) (lines 3, 6 in Algorithm 1). Determining *accQ* and ω is an optimization problem with communication cost, query responsiveness, and disconnection time by selective sleep-off (cf. Section 2.4.3).

2.3.3. Management of Query Results. As a distributed query processor, each MC asynchronously sends its query results, [[UpdateQueryResults (*oid*, *MR*, *LPQR*)]] where *MR* and *LPQR* are current and predictive results, respectively, whenever its new location invalidates the previous results. Algorithm 2 shows how to maintain up-to-date query results at server by receiving preevaluated results from MCs. Current query result (*MR*) at $\dot{t}$ is registered in the cross-referenced $OT[oid].qlist$ and $GQT[qid].olist$ (lines 1–6, Algorithm 2). Predictive query results (*LPQR*) for $t > \dot{t}$ are enheaped into PQR (lines 7-8, Algorithm 2). At each time $\dot{t}$, ServerProcessing() will process all PQR entries with $t \leq \dot{t}$ and refresh the cross-referenced query results in OT and GQT (lines 10–20, Algorithm 2).

Receiving [[UpdateQueryResults (*oid*, *MR*, *LPQR*)]] and inserting new LPQR entries keep/sustain expired PQR entries. The purging cost for removing all previous PQR entries immediately is very expensive. Alternatively, based on Lemma 1, we can minimize the purging cost by adopting a timestamp-based approach.

Lemma 1 (expired PQR entry). *A PQR entry e is expired (and should be removed) iff its sequence number e.sq is smaller than its* sq_{valid} *in OT (i.e.,* $OT[e.oid].sq_{valid} > e.sq$*).*

Proof. It is omitted for brevity. □

Every PQR entry is timestamped with a unique sequence number *sq* generated by each MC's *sqGen*, and the minimum *sq* of all valid PQR entries is preserved in $OT.sq_{\text{valid}}$ (line 9, ServerMessageHandler()). By Lemma 1, those *expired* PQR entries with smaller *sq* than $OT.sq_{\text{valid}}$ can easily be removed whenever they are deheaped; the purging cost is thus minimized negligibly (line 12, ServerProcessing()).

2.4. Mobile Client-Side Processing. Each MC's query manager keeps its LQT up to date according to $\dot{p}$ and server's GQT, and query processor evaluates and sends query results in a real-time manner as necessary.

2.4.1. Query Management. Algorithm 3 describes query management of MC. Each MC initializes its local query subset by invoking MobileClientInit() which sends $\dot{p}$ and *capacity* to the server (line 1, Algorithm 3) and then receives *RD* and *LQT* (lines 2, 8-9, Algorithm 3). After that, MC computes *MR* and *LPQR* and then sends them back to the server (lines 3-4, 5–7, Algorithm 3). MC's LQT is always synchronized by the broadcasted messages from server that modify the GQT of server (lines 10–19, Algorithm 3).

2.4.2. Query Processing with Movement Handling. Each MC continuously monitors the spatial relationship between its

```
MobileClientInit() // Invoked upon power on
(1) SendMsg(server, [[RequestRD (oid, ṗ, capacity)]]);
(2) ReceiveMsg(server, [[UpdateRD (RD′, LQT′)]]);
(3) MR ← ∅; LPQR ← ∅;
(4) SendUpdateMsg();

SendUpdateMsg()
(5) MR ← computemr(ṗ);
(6) LPQR ← computelpqr(ṗ);
(7) SendMsg(server, [[UpdateQueryResults (oid, MR, LPQR)]]);

// Performed upon receiving the following messages
MobileClientMessageHandler()
(8) Upon receiving [[UpdateRD (RD′, LQT′)]]::
(9)      RD ← RD′; LQT ← LQT′;

(10) Upon receiving [[AddQuery (q⟨qid, qr⟩)]]::
(11)     if q.qr ∩ RD ≠ ∅ then
(12)         if |LQT| + 1 > capacity
(13)             then Invoke MobileClientInit();
(14)             else LQT ← LQT ∪ {q};
(15)                  if q.qr ∩ l_DR ≠ ∅ then SendUpdateMsg();

(16) Upon receiving [[RemoveQuery (q⟨qid, qr⟩)]]::
(17)     if q ∈ LQT then
(18)         Delete q⟨qid, qr⟩ from LQT;
(19)         if q.qr ∩ l_DR ≠ ∅ then SendUpdateMsg();
```

Algorithm 3: Query management in MCs.

location $\dot{p}$ and its local query subset (i.e., LQT). If the relationship breaks the preevaluated $LPQR$, MC should recompute MR and $LPQR$ and then sends them to server.

Integrity Requirements. As a distributed query processor, each MC has to meet the following requirements properly:

(i) **R1**: $\dot{p} \in RD$.

(ii) **R2**: $LPQR \neq \emptyset$.

(iii) **R3**: $LPQR.top.t > \dot{t}$, where $LPQR.top$ is LPQR's top entry.

(iv) **R4**: during $LPQR.top.t > \dot{t}$, MR should be maintained.

(v) **R5**: at $LPQR.top.t = \dot{t}$, $|LPQR.top.p, \dot{p}| \leq \delta$.

Algorithm 4 describes MC-side query processing, and this can be explained in corresponding actions for holding each requirement. First, **R1** is clearly satisfied by reinvoking MobileClientInit() every time an MC gets new RD (line 4, Algorithm 4). **R2** and **R5** are also done by invoking SendUpdateMsg() which recomputes MR/LPQR and sends them to the server (lines 11-12, Algorithm 4). **R3** does hold by deheaping every LPQR entry with $LPQR.top.t \leq \dot{t}$ (lines 9–17, Algorithm 4). During $LPQR.top.t > \dot{t}$, in order to hold **R4**, SendUpdateMsg() is invoked if MR is changed (lines 5–7, Algorithm 4). Selective sleep-off (lines 2-3, Algorithm 4) will be discussed in Section 2.4.3.

Bulk Preevaluation by Dead-Reckoning (DR). Query preevaluation performed by ComputeLPQR(p) (lines 20–27) is the key of the proposed technique for improving the query processing power. Dead-Reckoning (DR) is the most popular method to estimate moving objects' location, and its underlying principle is the *piecewise* linear movements [9]. The object's estimated location at any future time t can be obtained as $\widehat{p}(t) = p_{\text{ref}} + \overline{v} \cdot (t - t_{\text{ref}})$, where p_{ref} is a reference point at some reference time t_{ref} ($t > t_{\text{ref}}$) and $\overline{v}$ is a movement vector. As illustrated in Figure 5, MC computes all intersected points with the predicted line segment l_{DR} which is always cropped by RD, timestamps them with increasing sq, and then sends them. At this point, the smallest sq is preserved as sq_{valid} (1 for l_{DR} and 5 for l'_{DR}). In 2nd preevaluation with l'_{DR}, all expired points that have smaller sq than l'_{DR}'s sq_{valid} will efficiently be removed by Lemma 1 (e.g., they are 3rd and 4th points of Q_2).

2.4.3. Selective Sleep-Off. In order to reduce energy consumptions, MC selectively sleeps off whenever possible (lines 2-3, Algorithm 4). Each MC is able to compute the amount of maximum *sleepable* time which does not break the abovementioned 5 requirements (line 28, ComputeSafeSleepTime()).

```
MobileClientProcessing()
(1) forever do // at each discrete time slot i
(2)    t_sleep ← ComputeSafeSleepTime(ṗ);
(3)    Sleep min(t_max_sleep, t_sleep) times, then wake up;
(4)    if ṗ ∉ RD then MobileClientInit();
(5)    if (LPQR.top.t > i) then
(6)       MR' ← ComputeMR(ṗ);
(7)       if (MR ≠ MR') then SendUpdateMsg();
(8)    else // LPQR.top.t ≤ i
(9)       while LPQR.top.t ≤ i and LPQR ≠ ∅ do
(10)         e ← LPQR.deheap();
(11)         if e.flag = end or |e.p, ṗ| > δ then
(12)            SendUpdateMsg();
(13)         else // e.flag = enter/exit and |e.p, ṗ| ≤ δ
(14)            if e.flag = enter
(15)               then MR ← MR ∪ {e.qid};
(16)               else MR ← MR \ {e.qid};
(17)      end-while
(18) end-forever

ComputeMR(p)
(19) return {qid | e⟨qid, qr⟩ ∈ LQT and p ∩ qr ≠ ∅};

ComputeLPQR(p)
(20) Let P be a heap, P ← ∅;
(21) p_end ← the point intersected with RD; P.enheap(p_end);
(22) Let l_DR be a line segment consiting of two points p_ref and p_end;
(23) foreach LQT[i] ∈ LQT intersecting with l_DR do
(24)    Compute p_enter and p_exit of LQT[i] if exists;
(25)    P.enheap({p_enter, p_exit});
(26) end-foreach
(27) return P;

ComputeSafeSleepTime(p)
(28) return min(minperdist(p, R), R ∈ {LQT ∪ RD})/v_max;
```

Algorithm 4: Query processing with movement handling in MCs.

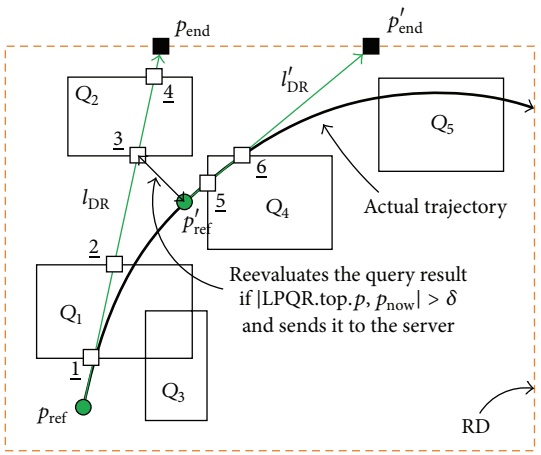


Figure 5: Query preevaluation by Dead-Reckoning (DR).

Definition 2 (MINPERDIST). The minimum perimeter distance of a point $p = (p_1, p_2, \ldots, p_d)$ to the perimeters of a rectangle $R = [l_1, u_1] \times \cdots \times [l_d, u_d]$, denoted as $minperdist(p, R)$, is

$$minperdist(p, R) = \begin{cases} \min_{i=1}^{d} \min\left(|u_i - p_i|, |p_i - l_i|\right), & \text{if } q \in R; \\ mindist(p, R), & \text{otherwise.} \end{cases}$$

$$= \begin{cases} \min_{i=1}^{d} \min\left(|u_i - p_i|, |p_i - l_i|\right), & \text{if } q \in R; \\ \sqrt{\sum_{i=1}^{d} \left(\begin{cases} |l_i - p_i|, & \text{if } p_i < l_i; \\ |p_i - u_i|, & \text{if } u_i < p_i; \\ 0, & \text{otherwise.} \end{cases} \right)^2}, & \text{otherwise,} \end{cases} \quad (1)$$

where d is dimensionality and $mindist(p, R)$ is the minimum distance of rectangle R from point p [14]. We only consider the case of $d = 2$.

Lemma 3 (SafeSleepTime). *With the known maximum velocity v_{max} of an MC at current location $\dot{p}$, it is guaranteed that during SafeSleepTime, MC's movement does not break any requirement for maintaining accurate query results. Thus, it is safe to switch into sleep mode for* $\min(t_{max\_sleep}, SafeSleepTime)$ *times, and SafeSleepTime is defined as the amount of time to the nearest minperdist query with v_{max}:*

$$SafeSleepTime = \frac{SleepDist}{v_{max}} = \frac{\min\left(minperdist\left(\dot{p}, R\right), R \in \{LQT \cup RD\}\right)}{v_{max}}, \quad (2)$$

where SleepCircle in Figure 6 is a circle centered at the current location $\dot{p}$ with radius SleepDist.

Proof. It is omitted for brevity. □

Based on Lemma 3, MCs can sleep off for *SafeSleepTime* to save considerable energy consumption only if $SafeSleepTime > t_{min\_sleep}$ (t_{min_sleep} is the minimum amount of time to compensate the energy consumption for sleep-off/wake-up, and this can simply be 0 for brevity), and this may result in deteriorated query responsiveness by missing newly incoming queries from server's broadcasting channel. As we already discussed in Section 2.3.2, the server periodically broadcasts updates in GQT with interval ω, and each broadcast covers *accQ* times back from $\dot{t}$ (see Figure 7). To control the deterioration by selective sleep-off, t_{sleep} should be limited by defining its lower- and upper-bound, $t_{sleep} \in [t_{min\_sleep}, t_{max\_sleep}]$, being optimized between energy efficiency and query responsiveness.

To ensure the consistency between server's *GQT* and MCs' *LQT*s, each MC must obey the following rules. (1) The safely sleepable time $SafeTime \leq t_{max\_sleep}$. (2) Once woken up, each MC cannot sleep off before it received the next broadcasted message. The worst latency for the next broadcast is simply its interval ω. (3) Thus, the size of broadcast window (*accQ*) should be big enough to cover the maximum sleepable time (t_{max_sleep}) and the worst case latency (ω); that is, $accQ \geq t_{max\_sleep} + \omega$ (ω can be different for various real-life applications).

3. Localized Query Optimizations in MCs

As long as the above 5 requirements (in Section 2.4.2) hold, we can defer any kind of operations for reducing communication and computation costs. In Sleepwalk technique, query processing is performed on MCs in a fully distributed manner. For MCs, the most frequently performed operation is MobileClientProcessing() where three iterative searches over *LQT* for *MR*, *LPQR*, and *SafeTime*, ComputeMR(), ComputeLPQR(), and ComputeSafeSleepTime(), are performed. A popular way to improve search efficiency is to create an additional index (e.g., R-trees [12]) over *LQT*. However, this is not feasible on resource-scarce mobile devices since it surely incurs high maintenance costs. Rather,

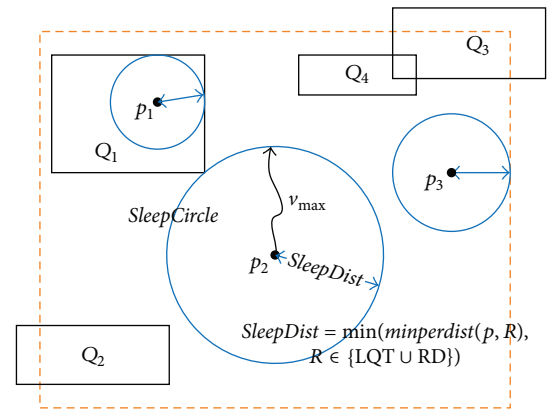


Figure 6: Illustration of *SleepDist*, *SleepCircle*, and *SafeSleepTime* (= $SleepDist/v_{max}$).

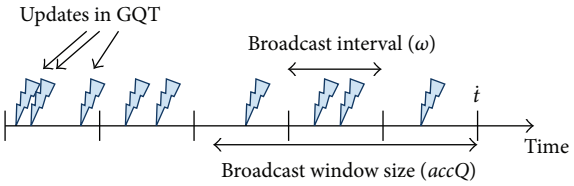


Figure 7: Scheduled broadcasting with interval ω.

increasing $|LQT|$ can be more profitable from a performance perspective. In this section, we will discuss a plane-sweep based optimization technique for these operations without any additional indexes over *LQT*.

In order to exploit plane-sweeping technique, we assume that *LQT* is sorted, by the server when it is chosen, along x-axis (resolving ties by sorting y-axis). The ith query in *LQT* is denoted as $LQT[i]$ where $i \in [0, |LQT| - 1]$. Only insertions and deletions in *LQT* are performed on mobile clients. Consequently, the overall computational overhead is quite low as $\mathcal{O}(|LQT| \log |LQT| + |LQT|)$ where $|LQT|$ is the number of queries stored in the mobile client.

Definition 4 (nearest lower/upper rectangle index). Given a point $p \in RD$, its nearest lower rectangle index NLR(p) and nearest upper rectangle index NUR(p) are formally defined as follows:

$$\begin{aligned} \mathrm{NLR}(p) &= \max \left(\arg\min_{i \in [0, |LQT|-1]} \left(p.x - LQT[i].hx\right) \right) \\ &\text{subject to} \quad p.x - LQT[i].hx > 0, \\ \mathrm{NUR}(p) &= \min \left(\arg\min_{i \in [0, |LQT|-1]} \left(LQT[i].lx - p.x\right) \right) \\ &\text{subject to} \quad LQT[i].lx - p.x > 0. \end{aligned} \quad (3)$$

Given a point p, its NLR/NUR can be computed as follows: first, it performs a forward scan from $LQT[0]$ until $LQT[i].lx < p.x$; then $\mathrm{NUR}(p) \leftarrow (i + 1)$. During this operation, it repeatedly updates $\mathrm{NLR}(p) \leftarrow i$ only if

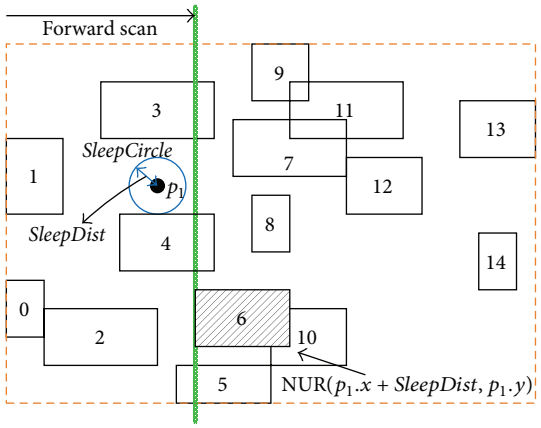


Figure 8: Search boundary for ComputeSafeSleepTime ().

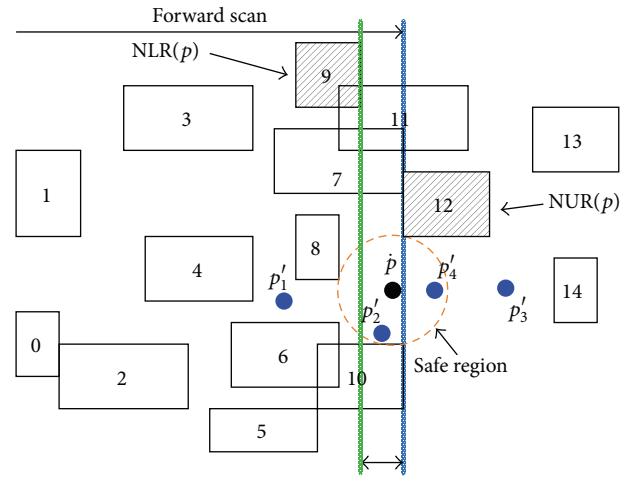


Figure 9: Search boundary *MRB* and safe region *SR* for ComputeMR().

$LQT[i].hx \in (LQT[\text{NLR}(p)].hx, p.x)$. For example, NLR($\dot{p}$) and NUR($\dot{p}$) in Figure 9 are $LQT[9]$ and $LQT[16]$, respectively. By definition, NLR and NUR are greatly impacted by the extent/sizes of queries. For more optimized search boundary for NLR/NUR, narrow extent is preferred. From computational cost viewpoint for computing NLR(p) and NUR(p), optimal axis is

$$opt\_axis = \arg\min_{i\in\{x,y\}} \left(\frac{\sum_{LQT[j]\in LQT} (LQT[j]\text{'s } i\text{th extent}/RD\text{'s } i\text{th extent})}{|LQT|} \right). \quad (4)$$

3.1. Optimizations for ComputeSafeSleepTime(). ComputeSafeSleepTime() is to choose $LQT[i] \in LQT$ which minimizes *SleepCircle* and its radius *SleepDist*. As depicted in Figure 8, it can be optimized by computing NUR of the rightmost point of *SleepCircle*. Initially, we set $SleepDist = minperdist(\dot{p}, RD)$ and then perform a forward scan until $SleepDist + \dot{p}.x < LQT[i].lx$ simultaneously updating *SleepDist* as $\min(SleepDist, minperdist(\dot{p}, LQT[i]))$. Finally, $SafeTime = SleepDist/v_{\max}$. Note that *SafeTime* is always recomputed due to its dependency on $\dot{p}$.

3.2. Optimizations for ComputeMR(). The most frequently invoked procedure is ComputeMR() which returns a set of queries intersecting with given point p (line 21, Algorithm 4). It should be invoked at each time t to hold **R4**. By exploiting the notion of NLR/NUR, we can minimize the search space for computing *MR*. By definition, the monitoring query result (*MR*) only exists in $LQT(\text{NLR}(\dot{p}), \text{NUR}(\dot{p}))$. We call the interval $[LQT[\text{NLR}(\dot{p})].hx, LQT[\text{NUR}(\dot{p})].lx]$ *MR*-boundary; based on this, we can optimize ComputeMR() by minimizing its search space:

(i) Initially, compute NLR($\dot{p}$) and NUR($\dot{p}$).

(ii) Then, by referring to current location $\dot{p}$ and next location p', it is possible to have proper new *MR* and *MR*-boundary by searching optimized search space (*MR*-boundary ≡ *MRB*):

(a) Case of $p'.x < MRB.lx$ (p'_1): perform a forward scan from $LQT[0]$.

(b) Case of $p'.x \in MRB$ (p'_2): perform a forward scan from $LQT[\text{NLR}(\dot{p})]$.

(c) Case of $p'.x > MRB.hx$ (p'_3): perform a forward scan from $LQT[\text{NUR}(\dot{p})]$.

3.2.1. Deferring the Update of MR and MR-Boundary with Safe Region. Although this optimization effectively minimizes the search space for ComputeMR(), it should be still performed always and sometimes performed for unchanged MR (in the worst case). In order to better computational efficiency, we can defer the update of *MR* and *MR*-boundary by exploiting a *circular-shaped* safe region which is the same as in [7] geometrically. Its uniqueness came from the fact that our safe region is computed not in the server but in MCs; moreover it is not for reducing communication cost, but for computational cost of MCs. Simply speaking, safe region and MR-boundary are maintained unless $p' \notin safe\_region$. As depicted in Figure 9, the safe region can go beyond the MR-boundary. Moreover, the notion of safe region is capable of relaxing **R2** ($LPQR \neq \emptyset$); that is, each MC simply does nothing without recomputing *MR* when $LPQR = \emptyset$ and $p' \in safe\_region$. This will be very effective in a huge free space and the skewed distribution of monitoring queries.

3.3. Optimizations for ComputeLPQR(). As illustrated in Figure 10, optimization of ComputeLPQR() is quite similar to that of ComputeMR(). NLR and NUR of line segment $l_{\text{DR}} = (s, e)$ can be obtained from its start and end points (s and e, resp); that is, $\text{NLR}(l_{\text{DR}}) = \text{NLR}(s)$ and $\text{NUR}(l_{\text{DR}}) = \text{NUR}(e)$. For correctness and simplicity, swap s and e when e precedes s along with the sorting axis. Based on the interval $[LQT[\text{NLR}(s)].hx, LQT[\text{NUR}(e)].lx]$

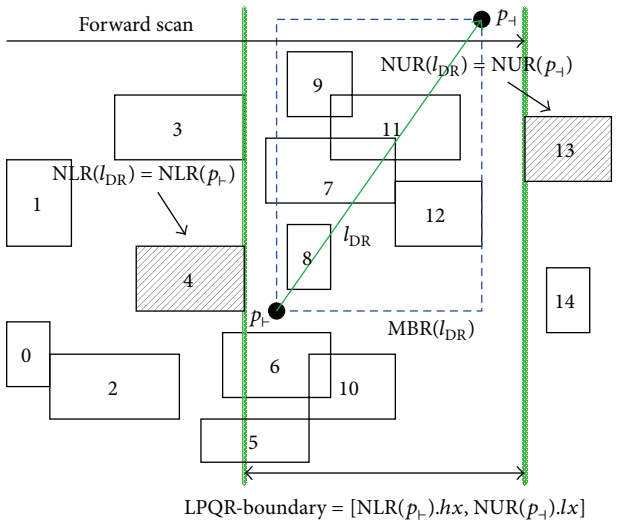


Figure 10: Search boundary for ComputeLPQR().

(called *LPQR-boundary*), we can optimize ComputeLPQR() by minimizing its search space:

(i) Initially, compute NLR(l_{DR}) and NUR(l_{DR}).

(ii) Then, by referring to current line segment l_{DR} and next line segment $l'_{DR} = (s', e')$, it is possible to have proper new *LPQR* and *LPQRB* by searching optimized search space.

  (a) Case of $s'.x < LPQRB.lx$: perform a forward scan from $LQT[0]$.
  (b) Case of $s'.x \in LPQRB$: perform a forward scan from $LQT[\text{NLR}(l_{DR})]$.
  (c) Case of $s'.x > LPQRB.hx$: perform a forward scan from $LQT[\text{NUR}(l_{DR})]$.

Owing to the forward scan-based searches over *LQT*, this optimization is related with the start point *s* only. As shown in Figure 10, another optimization is to check only $LQT[i]$ intersecting with l_{DR}'s minimum bounding rectangle ($MBR(l_{DR})$).

4. Performance Evaluation

This section describes the experiments we have carried out to evaluate the proposed technique.

4.1. Experiment Setup. We evaluate the performance of the existing *Q*-index [7], MQM techniques [8], and the proposed Sleepwalk technique in terms of the following performance metrics:

(i) Power consumption of mobile clients: this cost is measured as the number of messages sent by the client and the amount of time for sleep-off during processing continuous range queries. As the sleep-off time increases, the CPU of client can be in the sleep mode for conserving battery consumption. For measuring power consumption, we use the reference values in Table 1 describing the power consumption of representative components [10]. The power consumption is measured by the *unit energy* which is the amount of power consumption per second in sleep mode. In this experiment, the unit energy is assumed as 0.16 mW [10].

Table 1: The components of example MC (in mW) [10].

	Model	Sleep	Active	Receive	Transmit
CPU	StrongARM SA-1100	0.16	400	—	—
Network	RangeLAN2 7401/02	25	—	750	1,500
GPS	μ-blox GPS-MS1E	33	462	—	—

(ii) Server processing cost: this cost is defined as the total number of data accesses (including query index, global query table, precomputed query table, and object table) in server in order to evaluate the same number of range queries. The size of the BP-tree leaf node is set to be 50 rectangles as in [8]. For realistic experiment, LRU cache is applied for the BP-tree of the server; this makes most of nonleaf nodes reside in the buffer when query pattern is uniform. In this experiment, the access cost of the BP node is assumed to be the same as that of other tables.

In this work, the concept of safe region exploited in *Q*-index technique [7] is implemented as follows: With a given location of a mobile object, we first search query index (i.e., BP-tree) for finding the subdomain containing the location. Then we compute the largest *rectangular* region as the safe region for the object, such that the region does not overlap with the boundaries of any query [7].

For more realistic experiments, we use two different ways of generating mobility patterns of mobile objects. One is to exploit mobility simulator which simulates the movements of real-world objects. We use the network-based generator of moving objects [15] (described as NETWORK dataset). Another is to trace the movements of real user (described as TRACE dataset). For the NETWORK dataset, every object moves along the road network Oldenburge, Germany, in the normalized data space $[0, 100\,\text{k}]^2$. The number of mobile objects varies from 100 to 10 k, and their speed varies from 10 to 30 per simulation time (default speed 10). The number of query rectangles varies from 1 k to 100 k, and the size is from $[10 \times 10]$ to $[500 \times 500]$. And the overall experiment time is 10 k unit times. For the TRACE dataset, we trace the real movement of a user for 6 hours in Suwon, Korea. The tracked trajectory is normalized into the same data space of $[0, 100\,\text{k}]^2$. The number of query rectangles is the same as for the NETWORK dataset. And the overall experiment time is also normalized 10 k unit times. Experiment parameters and their default values, in bold, are summarized in Table 2.

All experiments were conducted on a Windows PC with an Intel Core2 Duo 3 GHz CPU and 4 GB memory. All techniques were implemented in Java language. In our implementations, every hash table was implemented using

Table 2: Experimental parameters and their values.

Parameters	Value used (default) [DATASET]
Data space	[0, 100 k] × [0, 100 k]
Number of mobile objects	100~10,000 (**1,000**) [NETWORK] 1 [TRACE]
Speed of mobile objects	10~30 (**10**) [NETWORK]
Number of monitoring queries	1,000~100,000 (**10,000**)
Size of monitoring queries	10 × 10~500 × 500 (**50 × 50**)
Simulation time	**10,000**

the `HashMap` class, and disk I/Os are performed using the `RandomAccessFile` class in Java 2 SE library.

4.2. Experiment Results. This section describes the experiment results as a function of the number of monitoring queries and that of moving objects.

4.2.1. Effect of the Number of Monitoring Queries. Figure 11 shows the result of evaluating the server cost and energy cost of mobile clients varying the number of monitoring queries from 1 k to 100 k with both NETWORK and TRACE datasets.

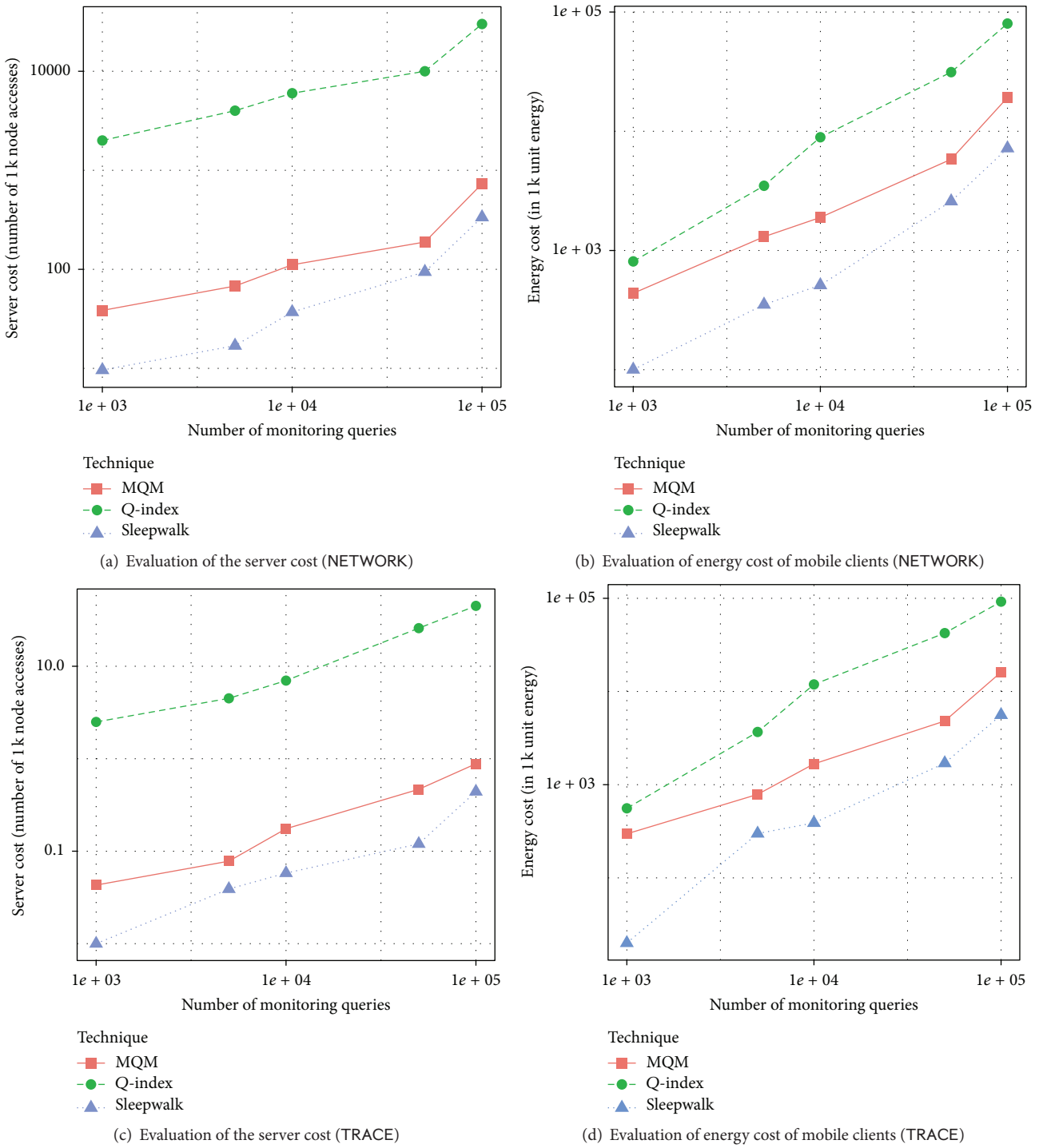
Regarding NETWORK dataset, Figures 11(a) and 11(b) show the result of evaluating the server processing cost and the average energy consumption of mobile clients, respectively. The queries are uniformly generated; their sizes are fixed as [50 × 50]. We also generated 1,000 moving objects and their initial locations are uniformly distributed in the road network. As depicted in Figure 11(a), all three techniques incur higher server processing and communication costs as the number of queries increases. Particularly the costs of *Q*-index are many times higher than those of MQM and Sleepwalk techniques. With the increased number of queries, the average size of RD adpoted in MQM and Sleepwalk techniques decreases and the number of [[RequestRD]] and [[UpdateRD]] messages interchanged increases proportionally. Overall, Sleepwalk technique clearly outperforms the existing *Q*-index and MQM techniques in all cases. This is due to the fact that it performs query processing in bulk predictively and sends the results at once. As depicted in Figure 11(b), all three techniques suffer performance degradation with high power consumption as the number of queries increases. Due to the batch processing capability and sleep-off behavior, the proposed technique shows graceful degradation and outperforms the existing techniques. With smaller number of queries, Sleepwalk can minimize the energy cost by increasing *SafeSleepTime* with the increased average distance between queries.

Regarding TRACE dataset, Figures 11(c) and 11(d) show the result of evaluating the server processing cost and the average energy consumption of mobile clients, respectively. TRACE dataset shows a similar trend of performance degradation when the number of queries increases. There is an important difference from NETWORK dataset; that is, in real-world situations, people tend to have large amount of inactivity time. This means people stay in some specific places such as home or work. In that case, the proposed technique has a huge benefit of maximizing sleep-off time. As already mentioned in Lemma 3, $SafeSleepTime = SleepDist/v_{\max}$ can be maximized to $t_{\max\_sleep}$, where $t_{\max\_sleep}$ is set as 1,500 unit time (approximately 30 min) in this experiment. Thanks to this nature, the proposed technique extremely outperforms previous works especially when the smaller number of queries is applied.

4.2.2. Effect of the Number of Moving Objects. Figure 12 shows the result of evaluating the server cost and energy cost of mobile clients varying the number of moving objects from 100 to 10 k with the NETWORK dataset (in this scenario, we omit the experiment with TRACE dataset since it can only provide a single object). Figure 12(a) shows the result of evaluating the server cost varying the number of moving objects from 100 to 10 k. Their initial locations are uniformly distributed in the road network. We also generated 10 k monitoring queries, and their sizes are fixed as [50 × 50]. As the number of objects increases, all three techniques incur higher server and communication costs proportionally. In this setting, Sleepwalk technique clearly outperforms the existing *Q*-index and MQM techniques in all cases. Sleepwalk technique scales well because of its localized behavior of processing monitoring queries; that is, it processes the monitoring queries in bulk predictively and selectively sleeps off whenever possible. Figure 12(b) shows the result of evaluating the average energy consumption of mobile clients. The result shows that the number of objects has no effect on the average energy cost.

4.2.3. Effect of the Size of Monitoring Queries. Figure 13 shows the result of evaluating the server cost and energy cost of mobile clients with 10 k queries varying their size from [10 × 10] to [500 × 500].

Regarding NETWORK dataset, Figures 13(a) and 13(b) show the result of evaluating the server processing cost and the average energy consumption of mobile clients, respectively. We generated 1,000 moving objects uniformly distributed in the road network. As depicted in Figure 13(a), all three techniques suffer higher server and communication costs as the size of monitoring queries increases. This is because the probability of intersecting query rectangles increases. *Q*-index will frequently send update requests for safe region, and MQM technique sends frequent query updates to server. Sleepwalk technique scales well because its cost is less affected by the size of queries and more affected by the motion pattern of clients. Figure 13(b) shows the result of evaluating the average energy consumption of mobile clients with various size of queries. The result shows that all three techniques degrade their performance due to the frequent update request to the server. Overall, Sleepwalk technique is superior to the existing *Q*-index and MQM techniques in all cases. This is due to the fact that it performs query processing in bulk predictively and sends the results at once. More importantly, it keeps mobile clients sleep-off whenever possible. This incurs better energy cost with increased size of



(a) Evaluation of the server cost (NETWORK)

(b) Evaluation of energy cost of mobile clients (NETWORK)

(c) Evaluation of the server cost (TRACE)

(d) Evaluation of energy cost of mobile clients (TRACE)

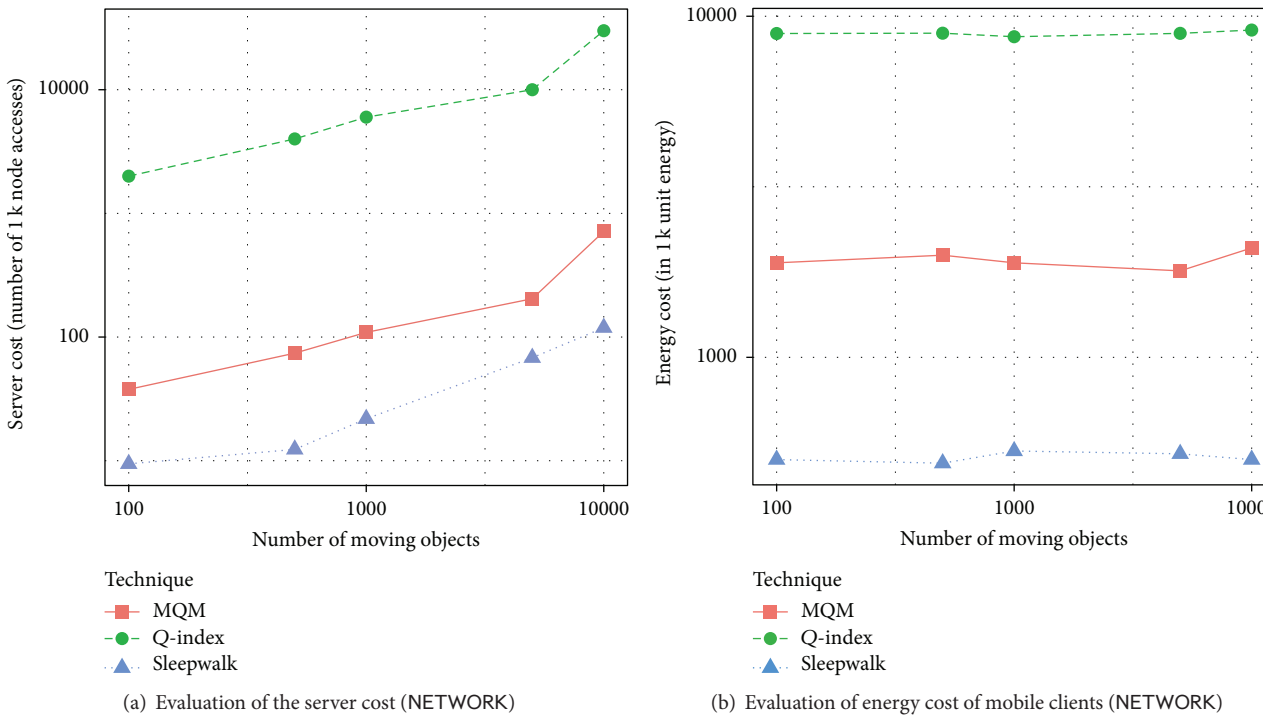
Figure 11: Effect of the number of monitoring queries.

queries ironically due to the increased *SafeSleepTime* to the perimeter of query rectangles.

Regarding TRACE dataset, Figures 13(c) and 13(d) show the result of evaluating the server processing cost and the average energy consumption of mobile clients, respectively. TRACE dataset also shows a similar trend as NETWORK dataset when the number of queries increases.

5. Related Work

Location-based sensing and monitoring (LBSM) is essential to various real-life applications such as location-based advertisement, traffic monitoring, surveillance, and many context-enriched services [16]. Recently, there are many studies for scalable monitoring [7, 8, 17–29].



(a) Evaluation of the server cost (NETWORK)

(b) Evaluation of energy cost of mobile clients (NETWORK)

FIGURE 12: Effect of the number of objects.

These studies are classified into two categories based on the type of data to be managed. One category is for modeling and indexing massive number of moving objects effectively [25–29] (called *moving object indexing*), and the other category is for indexing queries and their results efficiently [7, 8, 17–24, 30] (called *continuous query indexing*).

The former is proposed to minimize the maintenance and update costs for moving object databases (MOD) [28] in the server; various approaches are proposed such as a hash-based index structure [27], bottom-up update approach for R-trees [29], and buffer-based approaches for R-trees [25, 26].

The latter is to design specialized index structures for both objects and queries for the sake of scalability; most of works focused on how to localize query processing and manage the results both in clients and in server effectively. In [7], Prabhakar et al. firstly introduce the concept of query indexing (*Q*-index) and *safe region* where an R-tree-like index on the queries instead of the objects is built. In [19], a grid-based index structure for queries is proposed to achieve the minimized server cost. The concept of safe region has huge impact on this study, and it is improved and reused in [18, 23]. Each object maintains its own safe region and is only updated if its new location invalidates the results of any of the queries. In [17], Mokbel et al. present a centralized approach called SINA based on shared execution and incremental evaluation. Shared execution is achieved by a spatial join between the objects and the queries. In [8], Cai et al. propose MQM (Monitoring Query Management) technique which aims to reduce the communication and server costs in a distributed manner through the concept of resident domain. Similarly, Jung et al. [22] propose Query Region tree (QR-tree) to achieve efficient cooperation between server and moving objects by leveraging the computational resources. In [21], Do et al. consider the hybrid communication architecture unifying the two communication paradigms, wide-area and local-area wireless networks. They propose P2MRQ (peer-to-peer technique for moving range queries) technique which exploits peer-to-peer communication for maximizing the scalability of continuous spatial query systems. In [20], Farrell et al. propose a quality-based compensation method for reducing power consumption of moving objects. The method controls when an object should acquire and update its location with user-defined error or tolerances. In [30], Song et al. propose a privacy-preserving continuous query processing technique called anonymity of motion vectors (AMV) which utilizes motion vector to minimize cloaking region; however it does not consider power consumption and sleep-off mode.

To the best of our knowledge, none of these studies reviewed above effectively utilizes movement patterns of moving objects and their sleep-off mode (power conserving mode).

6. Concluding Remarks

This paper presents a distributed sensing and monitoring technique (called *Sleepwalk*) for processing monitoring queries that significantly improves the scalability of server and the energy efficiency of MCs. The proposed technique can be seen as a resource-constraint workload distribution scheme. Extensive experiments indicate that the proposed technique remarkably outperforms existing techniques in

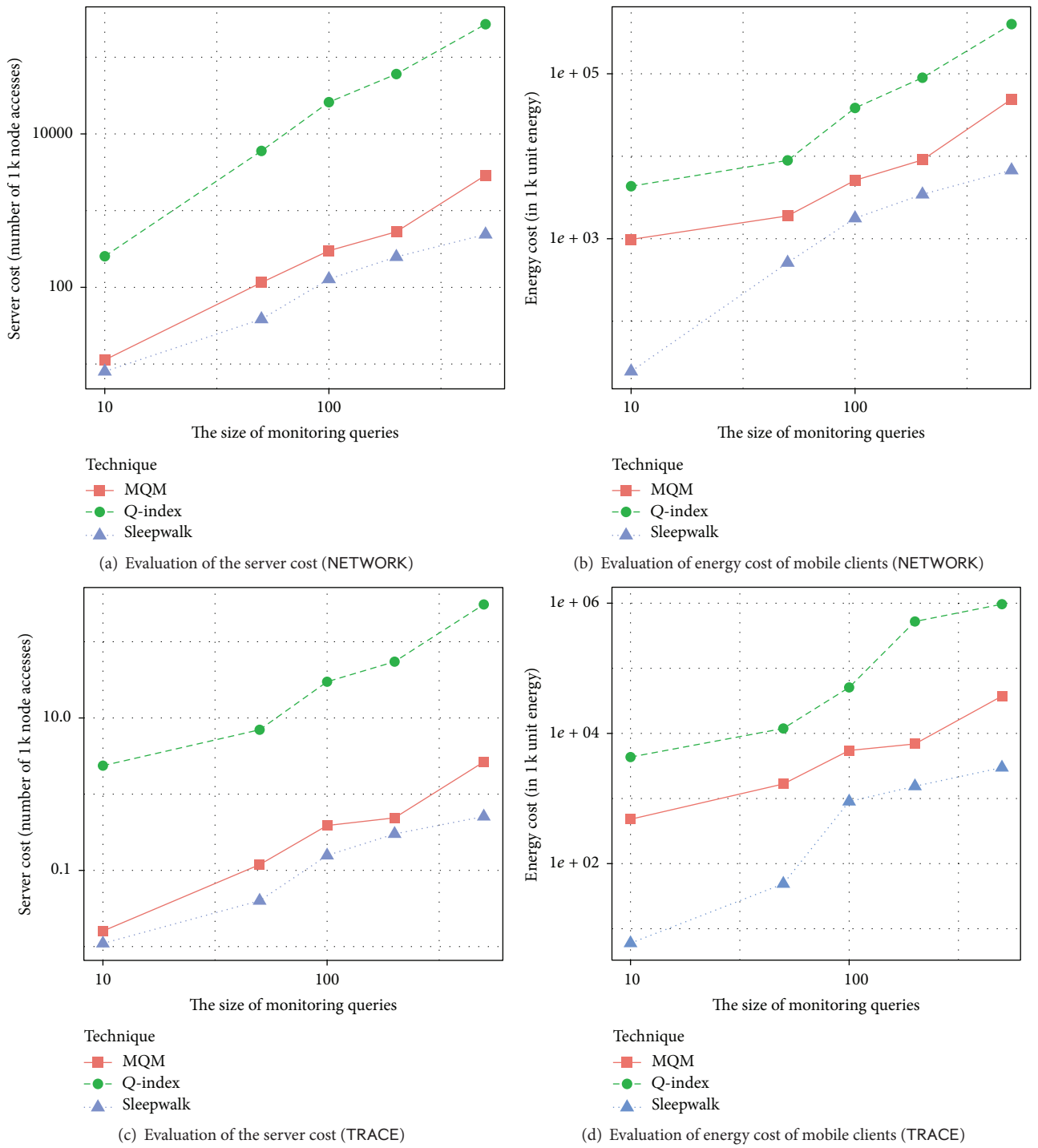


FIGURE 13: Effect of the size of monitoring queries.

terms of server scalability, communication cost, and energy consumption of MCs with various settings. This is mainly because the proposed scheme maximizes the localized processing for computing query results and conserves battery power efficiently.

For future research, we will investigate the problem in indoor space with noncoordinate based location models. Different query types such as kNN (k-nearest neighbor) and other sophisticated queries will be investigated. We will focus on many different environments such as Internet of Things (IoT) in order to support realistic scenarios and services.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

References

[1] M.-A. Dru and S. Saada, "Location-based mobile services: the essentials," *Alcatel Telecommunications Review*, no. 1, pp. 71–76, 2001.

[2] I. A. Junglas and R. T. Watson, "Location-based services," *Communications of the ACM*, vol. 51, no. 3, pp. 65–69, 2008.

[3] K.-L. Wu, S.-K. Chen, and P. S. Yu, "Incremental processing of continual range queries over moving objects," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 11, pp. 1560–1575, 2006.

[4] K.-L. Wu, S.-K. Chen, and P. S. Yu, "Efficient processing of continual range queries for location-aware mobile services," *Information Systems Frontiers*, vol. 7, no. 4-5, pp. 435–448, 2005.

[5] W. Wu, W. Guo, and K.-L. Tan, "Distributed processing of moving K-nearest-neighbor query on moving objects," in *Proceedings of the 23rd International Conference on Data Engineering (ICDE '07)*, pp. 1116–1125, April 2007.

[6] H. Wang, R. Zimmermann, and W.-S. Ku, "Distributed continuous range query processing on moving objects," in *Proceedings of the International Conference on Database Expert Systems Applications (DEXA '06)*, pp. 655–665, 2006.

[7] 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 Transactions on Computers*, vol. 51, no. 10, pp. 1124–1140, 2002.

[8] Y. Cai, K. A. Hua, G. Cao, and T. Xu, "Real-time processing of range-monitoring queries in heterogeneous mobile databases," *IEEE Transactions on Mobile Computing*, vol. 5, no. 7, pp. 931–942, 2006.

[9] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, "Moving objects databases: issues and solutions," in *Proceedings of the 10th International Conference on Scientific and Statistical Database Management (SSDBM '98)*, pp. 111–122, Naples, Italy, July 1998.

[10] O. Kasten, Energy Consumption.

[11] T. Imielinski, S. Viswanathan, and B. R. Badrinath, "Data on air: organization and access," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 3, pp. 353–372, 1997.

[12] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*, pp. 47–57, ACM, 1984.

[13] B. Gedik and L. Liu, "MobiEyes: a distributed location monitoring service using moving location queries," *IEEE Transactions on Mobile Computing*, vol. 5, no. 10, pp. 1384–1402, 2006.

[14] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*, pp. 71–79, ACM, San Jose, Calif, USA, May 1995.

[15] T. Brinkhoff, *Network-Based Generator of Moving Objects*, 2005.

[16] M. Song, G. Marreiros, H. Ko, and J.-H. Choi, "Context-enriched and location-aware services," *Journal of Computer Networks and Communications*, vol. 2012, Article ID 649584, 2 pages, 2012.

[17] M. F. Mokbel, X. Xiong, and W. G. Aref, "SINA: scalable incremental processing of continuous queries in spatio-temporal databases," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*, pp. 623–634, ACM, 2004.

[18] H. Hu, J. Xu, and D. L. Lee, "A generic framework for monitoring continuous spatial queries over moving objects," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, pp. 479–490, June 2005.

[19] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch, "Main memory evaluation of monitoring queries over moving objects," *Distributed and Parallel Databases*, vol. 15, no. 2, pp. 117–135, 2004.

[20] T. Farrell, R. Cheng, and K. Rothermel, "Energy-efficient monitoring of mobile objects with uncertainty-aware tolerances," in *Proceedings of the 11th International Database Engineering and Applications Symposium (IDEAS '07)*, pp. 129–140, September 2007.

[21] T. T. Do, F. Liu, and K. A. Hua, "When mobile objects' energy is not so tight: a new perspective on scalability issues of continuous spatial query systems," in *Proceedings of the International Conference on Database & Expert Systems Applications (DEXA '07)*, pp. 445–458, 2007.

[22] H. Jung, Y. S. Kim, and Y. D. Chung, "QR-tree: an efficient and scalable method for evaluation of continuous range queries," *Information Sciences*, vol. 274, pp. 156–176, 2014.

[23] H. Al-Khalidi, D. Taniar, J. Betts, and S. Alamri, "Monitoring moving queries inside a safe region," *The Scientific World Journal*, vol. 2014, Article ID 630396, 13 pages, 2014.

[24] H. AL-Khalidi, D. Taniar, and M. Safar, "Approximate algorithms for static and continuous range queries in mobile navigation," *Computing*, vol. 95, no. 10-11, pp. 949–976, 2013.

[25] M. Song and H. Kitagawa, "Managing frequent updates in R-trees for update-intensive applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 11, pp. 1573–1589, 2009.

[26] M. Song, H. Choo, and W. Kim, "Spatial indexing for massively update intensive applications," *Information Sciences*, vol. 203, pp. 1–23, 2012.

[27] Z. Song and N. Roussopoulos, "Hashing moving objects," in *Mobile Data Management: Second International Conference, MDM 2001 Hong Kong, China, January 8-10, 2001 Proceedings*, Lecture Notes in Computer Science, pp. 161–172, Springer, Berlin, Germany, 2001.

[28] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez, "Cost and imprecision in modeling the position of moving objects," in *Proceedings of the 14th International Conference on Data Engineering (ICDE '98)*, pp. 588–596, February 1998.

[29] M.-L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo, "Supporting frequent updates in R-trees: a bottom-up approach," in *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB '03)*, pp. 608–619, 2003.

[30] D. Song, J. Sim, K. Park, and M. Song, "A privacy-preserving continuous location monitoring system for location-based services," *International Journal of Distributed Sensor Networks*. In press.