A Filter-based Protocol for Continuous Queries over Imprecise Location Data

Yifan Jin† Reynold Cheng† Ben Kao† Kam-Yiu Lam‡ Yinuo Zhang§*
†University of Hong Kong ‡City University of Hong Kong §University of Southern California
†{yfjin, ckcheng, kao}@cs.hku.hk ‡cskylam@cityu.edu.hk §yinuozha@usc.edu

ABSTRACT

In typical location-based services (LBS), moving objects (e.g., GPS-enabled mobile phones) report their locations through a wireless network. An LBS server can use the location information to answer various types of *continuous queries*. Due to hardware limitations, location data reported by the moving objects are often uncertain. In this paper, we study efficient methods for the execution of *Continuous Possible Nearest Neighbor Query* (CPoNNQ) that accesses imprecise location data. A CPoNNQ is a standing query (which is active during a period of time) such that, at any time point, all moving objects that have non-zero probabilities of being the nearest neighbor of a given query point are reported. To handle the continuous nature of a CPoNNQ, a simple solution is to require moving objects to continuously report their locations to the LBS server, which evaluates the query at every time step. To save communication bandwidth and mobile devices' batteries, we develop two *filter-based protocols* for CPoNNQ evaluation. Our protocols install "filter bounds" on moving objects, which suppress unnecessary location reporting and communication between the server and the moving objects. Through extensive experiments, we show that our protocols can effectively reduce communication costs while maintaining a high query quality.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications

General Terms

Algorithms, Performance

Keywords

continuous queries, uncertain database, communication cost

*Work done in the University of Hong Kong

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

1. INTRODUCTION

With the advances in mobile computing devices and mobile communication technologies, various location-based services (LBS), such as navigation and map-based services, have emerged. Fig. 1(a) shows a simple location-based system in which a location server maintains a database that continuously registers the most updated locations of a set of moving objects. The location of the moving objects can be acquired by positioning devices, e.g., *Global Positioning System* (GPS). Once the location of an object is captured, it is reported as a location update to the location server through a wireless network. The location information collected at the server can support different spatial queries, for instance, "Tell me whether one of my friends is in the same building as me".

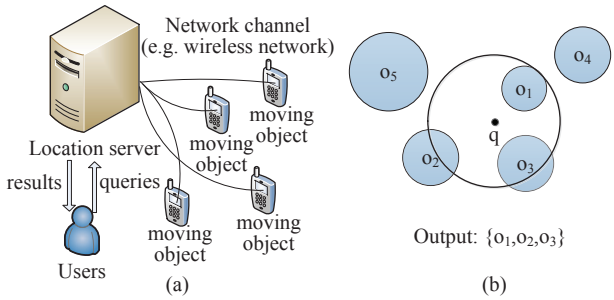

Figure 1: (a) A simple location-based system. (b) An example of *CPoNNQ*.

Due to the imprecision of location sensing devices, location values captured by the moving objects are only approximation at best [1, 2, 3, 4]. The actual locations of objects are therefore uncertain. Various studies have been conducted on how to improve the quality of queries' answers given that object locations are uncertain. The most prominent approach is to impose an uncertainty model on objects' locations and evaluate queries based on this model [2, 5, 6, 7]. A typical uncertainty model defines an *uncertainty region* for each object, which is a closed region that confines the possible location of that object. For example, in Fig. 1(b), the location uncertainty of objects is modeled using circular uncertainty regions.

One of the most studied spatial queries is the nearest neighbor query (NNQ). Given a query point q, the NNQ on q returns the object that is the closest to q in space. With location uncertainty, the answer to an NNQ is uncertain as well. An approach is to evaluate the possibility of

each object being the NNQ's answer. Efficient methods have previously been put forward to solve this problem [2, 3, 6, 8]. We note that existing solutions to the NNQ problem assume that a snapshot of the objects' locations (and their uncertainty models) is given. The answer returned, therefore, is good only for a particular time instant. In many cases, however, we are interested in answering an NNQ *continuously* over an extensive period of time. For example, one might want to be continuously informed about which battalion is the closest to a military base while the troops are in motion. In this case, the NNQ is a standing query that is evaluated continuously.

We remark that existing methods for answering NNQ are inadequate in handling continuous NNQ. This is because a snapshot of the moving objects' locations have to be reported at every single time step. This imposes big demands on the communication bandwidth and battery power on the mobile devices, as well as on the computation overhead on the location server. To solve this problem, we consider two ideas: (1) Objects that have a zero chance of being the answer of an NNQ could be *muted* (i.e., they do not have to report their locations by each time step and their locations need not be considered by the location server in computing NNQ answers). (2) Location updates that do not change the NNQ answer need not be sent. Note that these two ideas help reducing the number of location updates (messages) sent from the moving objects as well as reducing the number of objects that have to be processed in determining the nearest neighbor. Continuous NNQ can therefore be answered more efficiently.

To realize the two ideas, we study the evaluation of *Continuous Possible Nearest Neighbor Query* (or *CPoNNQ*). Given a query point q, the *CPoNNQ* on q returns all moving objects that have non-zero probabilities of being the closest neighbors of q. As an example, in Fig 1(b), we see that the location of object o_4 is always farther away from q than that of object o_1. Therefore object o_4 is not a possible nearest neighbor of q. The same can be said for object o_5. The answer set of executing *CPoNNQ* on q is therefore $\{o_1, o_2, o_3\}$. Solving *CPoNNQ* realizes the first of the above two ideas. In particular, objects that are not in the answer set of a given *CPoNNQ* need not be handled and their location updates can be safely ignored.

We realize the second idea by studying message-efficient methods for executing *CPoNNQ*. Specifically, we propose *filter-based* protocols to avoid *redundant* location-update messages. A filter-based protocol installs *filter bounds* on each object. Essentially, a filter bound is a simple distance constraint that can be easily verified by the moving object. As long as an object satisfies its filter bounds, its location updates can be suppressed because the protocol ensures that those updates do not affect the query answer. We propose two filter-based protocols. For the first one, the *basic filter protocol* (BFP), we focus on designing effective filter bounds whose violations precisely capture the moments at which the *CPoNNQ* answer could be changed. When such violations occur, moving objects are probed for their most updated locations and new revised filters are installed in them. For the second protocol, the *optimized filter protocol* (OFP), we focus on identifying objects whose locations need not be reported and filter bounds that need not be updated even when filter violations occur. OFP can therefore save even more communication bandwidth and battery power compared with BFP.

The rest of paper is organized as follows. Section 2 presents the related work. We introduce the system model and define the *CPoNNQ* in Section 3. The Basic Filter Protocol (BFP) and the Optimized Filter Protocol (OFP) are presented in Sections 4 and 5, respectively. We give our experimental results evaluating the performances of BFP and OFP in Section 6. In Section 7 we conclude the paper with a brief discussion on the future work.

2. RELATED WORK

Filter bound algorithms. To support continuous query evaluation in LBS and sensor networks, location and sensor data are constantly acquired from the external environments. Due to the limited battery power of the sensing devices (e.g., mobile phones and temperature sensors), as well as network bandwidth, it is crucial to maintain a low communication cost during query evaluation. Recently, the concept of *filter bounds* have been proposed, which controls when a sensor should report its value. In [9, 10, 11], the authors proposed filter bound protocols for nearest neighbor queries. The authors in [12, 13, 14] developed filter bound protocols for range and top-k queries. The work in [10] studied the use of filter bounds in a peer-to-peer environment. However, all these work do not consider the important issue of *data uncertainty*, which can lead to incorrect results if it is not managed carefully. So far, few filter bound algorithms have considered data uncertainty. The authors in [7, 15] studied filter bound algorithms for continuous range queries on uncertain data. These protocols treat the user query range as a filter bound, and send it to the sensors involved. It is not clear how this protocol can be used to handle a *CPoNNQ*, since the query range is not defined in a *CPoNNQ*. In this paper, we explain how to design filter bounds for supporting this query, which, to our best knowledge, has not been studied before.

Nearest neighbor queries for uncertain data. In [2, 3, 16], the problem of efficiently evaluating a nearest-neighbor query on uncertain data was studied. They assume a *snapshot* query model; that is, the query is evaluated only once. A *CPoNNQ* stays in the system for an extensive amount of time, during which query answers need to be constantly refreshed. In [6], the authors studied the evaluation of a continuous NNQ. That paper focuses on how to incrementally recompute a query answer, after a new data value is received. However, it does not control when a sensor should report its value, and therefore cannot reduce communication costs. In this paper, we examine the use of filter bound protocols, with the goal of reducing communication and energy costs. Since the server receives less updates from sensors, the computation cost between the server and the sensors, as demonstrated in our experiments, can also be reduced.

3. PROBLEM DEFINITION AND THE BASELINE PROTOCOL

We now describe the architecture of an LBS system, and formally define *CPoNNQ* (Section 3.1). In Section 3.2, we describe a simple protocol for maintaining the answer of a *CPoNNQ*. Table 1 lists the symbols used in the paper.

Table 1: Description of symbols

Symbol	Description
q	The query point of a *CPoNNQ*
k	The number of moving objects
o_i	A moving object
$n_i(t)$	The distance from q to the closest point on o_i's uncertainty region at timestamp t
$f_i(t)$	The distance from q to the farthest point on o_i's uncertainty region at timestamp t
t_u	The time at which the most recent filter bounds are set by the location server.
t'_u	The time at which filter bounds were set just before t_u
$[l(n_i(t), u(n_i(t))$	A filter bound on $n_i(t)$ at timestamp t.
$[l(f_i(t), u(f_i(t))$	A filter bound on $f_i(t)$ at timestamp t.
$o_{\sigma(t)}$	The min-max object at timestamp t
$M(t)$	The set of objects with non-zero probabilities to be the NN of q, excluding $o_{\sigma(t)}$, at timestamp t
$F(t)$	The set of objects that cannot be NN of q at timestamp t
$A(t)$	The answer of *CPoNNQ* at timestamp t, i.e., $M(t) \cup \{o_{\sigma(t)}\}$

3.1 System Architecture

Fig. 2 depicts an LBS system. It consists of three parties: users, a location server, and a number (k) of moving objects maintained in the database (DB). We assume that each moving object is equipped with a positioning device, which captures the location of the object periodically with a periodicity of once every dt seconds. Different objects could have different periodicities (dt). In our simplest protocol (called *baseline* in Section 3.2), the location of a moving object is sent to the server as a location update message immediately after it is generated. Once the server receives an update message (from any object), it registers the update in the uncertain database together with the timestamp at which the update is generated. We can consider time being marked by a sequence of such timestamps $< t_0, t_1, ..., >$. In this paper, we define a timestamp to be a time instant at which an update is generated by an object. Between two consecutive timestamps, no location updates are made and the answer to a *CPoNNQ* is considered unchanged. We assume that the uncertainty region of a moving object is represented by a circular region similar to the one shown in Fig. 1(b) and the server can deduce the uncertainty region of an object based on the received location value.

A user submits a *CPoNNQ* to the location server, which processes the query via a Query Manager. This module accesses the uncertain database, and computes the answer based on the latest locations of the objects and their uncertainty regions. We now formally define a *CPoNNQ* below:

Definition 1. Given a query point q, a time interval $[t_s, t_e]$, a **Continuous Possible Nearest Neighbor Query** (or **CPoNNQ** in short) returns an answer set $A(t)$ for each timestamp $t \in [t_s, t_e]$ such that all and only those objects with non-zero probabilities of being the nearest neighbor of q at timestamp t are in $A(t)$.

As we have mentioned in Section 1, we aim to achieve savings in communication bandwidth by applying the filter-based protocols. In Fig. 2, the server employs a Filter Manager, which determines the filters to be sent to and installed

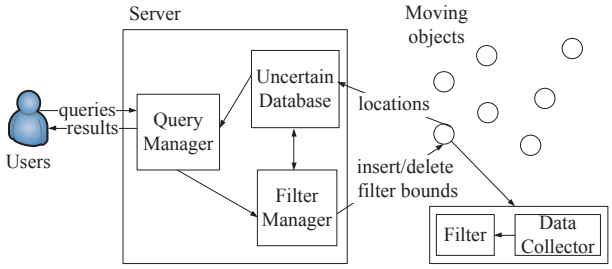

Figure 2: System architecture

at the moving objects. These filters are computed based on the standing *CPoNNQ* and the objects' locations. Once a *CPoNNQ* terminates (i.e., whose end time t_e is passed), the Filter Manger removes the filters that are associated with the query from the objects. The details of the Filter Manager will be covered in Sections 4 and 5.

3.2 The Baseline Protocol

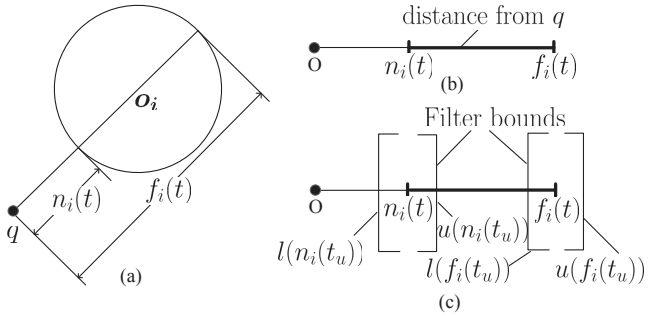

Figure 3: (a)Illustrating $n_i(t)$ and $f_i(t)$. (b) Distance of o_i from q. (c) Filter bounds for $n_i(t)$ and $f_i(t)$.

We now discuss a *baseline* protocol for processing *CPoNNQ*. A *CPoNNQ* with an interval $[t_s, t_e]$ submitted to the server becomes *active* at t_s. At t_s, the server broadcasts a message to all objects to acquire their most updated locations and activates their periodic location updates. The server then computes the answer set $A(t_s)$ by executing the procedure ComputePoNNQ (Algorithm 1. We will elaborate on this procedure shortly.) After that, each moving object o_i reports its location to the server once every dt_i seconds, where dt_i is the periodicity of o_i. Whenever a location update arrives at the server, ComputePoNNQ is executed again to refresh the answer. If the answer differs from the one of the previous timestamp, the revised answer will be reported to the user. At the end time of the query, t_e, if there are no more standing *CPoNNQ*s, the server instructs the moving objects to turn off location update messages.

At a given timestamp t, the procedure ComputePoNNQ first computes $n_i(t)$ and $f_i(t)$, which are the distances from q to the nearest point and the farthest point of each object o_i's uncertainty region, respectively. Fig. 3(a) illustrates these quantities. Then, the object with the smallest $f_i(t)$ is identified as the *min-max object*. Let $\sigma(t)$ be the index of this object and so $f_{\sigma(t)}(t) = \min_{i=1}^{k}\{f_i(t)\}$ is the min-max distance. It is easy to observe that for any object o_j, o_j is in

$A(t)$ if and only if $n_j(t) \leq f_{\sigma(t)}(t)$. All objects that satisfy the inequality are thus collected into the answer set $A(t)$.

```
1  A(t) ← ∅;
2  ∀1 ≤ i ≤ k, compute n_i(t) and f_i(t) ;
3  σ(t) ← 1 ;
4  for i ← 2 to k do
5      if f_i(t) < f_σ(t)(t) then
6          σ(t) ← i;
7  Add o_σ(t) to answer set A(t);
8  for each object o_i do
9      if n_i(t) ≤ f_σ(t)(t) then
10         Add o_i to answer set A(t);
11 Return A(t);
```

Algorithm 1: ComputePoNNQ(timestamp t)

Although *baseline* maintains a correct answer set for *CPoNNQ*, it is very expensive because each object has to report its updated locations at every timestamp. To reduce the communication cost for sending location updates, we next propose two filter-based protocols, namely BFP and OFP.

4. BASIC FILTER PROTOCOL (BFP)

With the Basic Filter Protocol (BFP), at either the start time of a query or when a location update is received at the server, the server computes filter bounds that are sent and installed at each moving object. At any timestamp t, let t_u ($t_u \leq t$) be *the most recent* timestamp at which filter bounds are made. For each object o_i, recall that $n_i(t)$ and $f_i(t)$ are the nearest and the farthest distances of the query point q from the uncertainty region of o_i (see Figure 3). The filter bounds installed by the server on object o_i at timestamp t_u are two intervals: $[l(n_i(t_u)), u(n_i(t_u))$and $[l(f_i(t_u)), u(f_i(t_u))$[1], which constrain the values of $n_i(t)$ and $f_i(t)$, respectively[2] (see Figure 3(c)). As object o_i moves, its location and hence its uncertainty region change. However, as long as $n_i(t)$ and $f_i(t)$ are within the intervals defined by their filter bounds, any location updates generated by o_i are suppressed. This gives us the following filtering rule:

> **Filtering Rule:** When object o_i generates an update at timestamp t, if $n_i(t) \in [l(n_i(t_u)), u(n_i(t_u))$ and $f_i(t) \in [l(f_i(t_u)), u(f_i(t_u))$, then the update is not sent.

By suppressing update messages, our filter-based protocols save communication bandwidth. The key to the protocols is how to set the filters correctly so that the answer set to a *CPoNNQ* remains correct despite missing updates. In the rest of this section, we will first describe BFP by assuming that filters are correctly set. Then, in Section 4.1, we show how filters are derived and prove their correctness.

BFP is shown in Algorithm 2. It consists of an initialization phase, a maintenance phase, and a cleanup phase. The initialization phase is similar to that of the Baseline Protocol (Section 3.2). At the start time, t_s, of a *CPoNNQ*, the server broadcasts a message to all the objects to acquire their locations and computes the answer set $A(t_s)$. Then, the server derives the filter bounds for all the objects and sends the bounds to them. After that, the server switches to the maintenance phase. During the maintenance phase, when an update arrives (because a filter bound is violated by an object), the server probes all the objects to acquire their locations and recomputes the answer set. It revises the filter bounds and sends the bounds to the objects. At the end time of the query, the server enters the cleanup phase. It instructs objects to remove their filter bounds that are associated with the query.

```
1  Initialization:
2  Broadcast to all objects to acquire their most updated
   locations;
3  A(t_s) ← ComputePoNNQ(t_s);
4  for each object o_i do
5      Compute filters [l(n_i(t_s)), u(n_i(t_s))) and
       [l(f_i(t_s)), u(f_i(t_s))) ;
6      Send filters to o_i;
7  t_u ← t_s;
8  Maintenance:
9  Upon receiving an update at timestamp t, trigger:
10 A(t) ← ComputePoNNQ(t);
11 for each object o_i do
12     Compute filters [l(n_i(t)), u(n_i(t))) and
       [l(f_i(t)), u(f_i(t))) ;
13     Send filters to o_i;
14 t_u ← t;
15 Cleanup:
16 Broadcast to all objects to remove filters associated with
   the query;
```

Algorithm 2: BFP

4.1 Setting Filters

In Section 3.2, at timestamp t, we define $o_{\sigma(t)}$ to be *the min-max object* at t, i.e., $f_{\sigma(t)}(t)$ is the smallest among those of all objects. We have argued that for any object o_j, $o_j \in A(t)$ iff $n_j(t) \leq f_{\sigma(t)}(t)$. The object $o_{\sigma(t)}$, the objects that satisfy the above inequality, and the objects that do not satisfy the inequality thus form three groups of objects. We denote them as $o_{\sigma(t)}$, $M(t)$, and $F(t)$, respectively. Our protocol BFP defines three sets of filter bounds, one for each group.

Definition 2. **$o_{\sigma(t)}$, $M(t)$ and $F(t)$**
At any timestamp t, the min-max object $o_{\sigma(t)}$ is the one such that $f_{\sigma(t)}(t) = \min\{f_i(t) \mid o_i \in DB\}$.[3] Then, the set $M(t) = \{o_j \in DB - \{o_{\sigma(t)}\} | n_j(t) \leq f_{\sigma(t)}(t)\}$, and the set $F(t) = \{o_i \in DB - \{o_{\sigma(t)}\} | n_i(t) > f_{\sigma(t)}(t)\}$.

Let t_u be the timestamp at which the most recent update was received at the server. By our definition, $o_{\sigma(t_u)}$ is the min-max object at timestamp t_u. Its min-max distance $f_{\sigma(t_u)}(t_u)$ determines the sets $M(t_u)$ and $F(t_u)$, and together, they define the answer set $A(t_u)$ at timestamp t_u. We note that the three groups of objects $o_{\sigma(t)}$, $M(t)$ and $F(t)$ are defined based on the three conditions mentioned in Definition 2. Specifically, they are:

[1]To simplify our correctness proof, each interval is closed on the left and open on the right.

[2]The parameter t_u shown in the intervals indicates that the filter bounds were set at t_u, which is the timestamp of the most recent update. Since there are no location updates between two consecutive timestamps, the filter bounds set at t_u remain valid and are in effect at t, the current time instant.

[3]If there are more than one such objects, we randomly pick one as $o_{\sigma(t)}$.

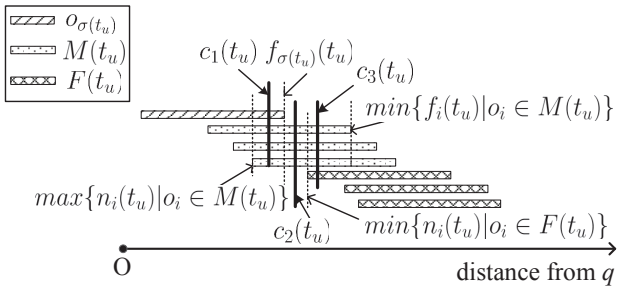

Figure 4: Illustration of three cutoff values at t_u.

(1) σ-condition: The object $o_{\sigma(t)}$ has to satisfy $f_{\sigma(t)}(t) = \min\{f_i(t) \mid o_i \in DB\}$.
(2) M-condition: All objects $o_j \in M(t)$ have to satisfy $n_j(t) \leq f_{\sigma(t)}(t)$.
(3) F-condition: All objects $o_i \in F(t)$ have to satisfy $n_i(t) > f_{\sigma(t)}(t)$.

Now, at any timestamp $t > t_u$, if we can guarantee that the three conditions hold when we substitute $\sigma(t) = \sigma(t_u)$, $M(t) = M(t_u)$, and $F(t) = F(t_u)$, then we have $A(t) = A(t_u)$. That is, the answer set remains unchanged.

To ensure that the conditions hold, BFP determines three demarcations. The purpose of these demarcations is to detect any potential violations of the three conditions. As an example, we can set a demarcation at a cutoff value c such that at any timestamp $t > t_u$, the min-max object at timestamp t_u, i.e., $o_{\sigma(t_u)}$, submits an update message only if $f_{\sigma(t_u)}(t) < c$ and any object $o_j \in M(t_u)$ submits an update message only if $n_j(t) \geq c$. Then, if no messages are received from them, we know that $n_j(t) < c \leq f_{\sigma(t_u)}(t)$ for all objects $o_j \in M(t_u)$. If we can further show that $\sigma(t) = \sigma(t_u)$, then $f_{\sigma(t_u)}(t) = f_{\sigma(t)}(t)$, and hence $n_j(t) < f_{\sigma(t)}(t)$ for all objects $o_j \in M(t_u)$. As a result, the M-condition stays valid if we substitute $M(t)$ by $M(t_u)$.

BFP uses three *cutoff values*, $c_1(t_u)$, $c_2(t_u)$, and $c_3(t_u)$ to detect the violations of the M-condition, the F-condition, and the σ-condition, respectively. Specifically, $c_1(t_u)$ is some value between $\max\{n_i(t_u)|o_i \in M(t_u)\}$ and $f_{\sigma(t_u)}(t_u)$, i.e.,

$$c_1(t_u) = (1-\alpha)\cdot\max\{n_i(t_u)|o_i \in M(t_u)\}+\alpha\cdot f_{\sigma(t_u)}(t_u) \quad (1)$$

The exact value of $c_1(t_u)$ is controlled by α, which is a real value between 0 and 1. Here we assume that $\alpha = 0.5$, i.e., $c_1(t_u)$ is in the middle of $\max\{n_i(t_u)|o_i \in M(t_u)\}$ and $f_{\sigma(t_u)}(t_u)$. We also tested different values of α in our experiments. The cutoff values $c_2(t_u)$ and $c_3(t_u)$ are similarly defined:

$$c_2(t_u) = (1-\alpha)\cdot f_{\sigma(t_u)}(t_u)+\alpha\cdot\min\{n_i(t_u)|o_i \in F(t_u)\} \quad (2)$$

$$c_3(t_u) = (1-\alpha)\cdot f_{\sigma(t_u)}(t_u)+\alpha\cdot\min\{f_i(t_u)|o_i \in M(t_u)\} \quad (3)$$

Fig. 4 illustrates $c_1(t_u)$, $c_2(t_u)$, and $c_3(t_u)$. We define the filter bounds of each object based on these values, as shown in Table 2. For example, for $o_{\sigma(t_u)}$, the bounds of $f_{\sigma(t_u)}(t)$ is $[c_1(t_u), \min\{c_2(t_u), c_3(t_u)\})$; for $o_j \in M(t_u)$, the bounds of $n_j(t)$ is $[0, c_1(t_u))$. So, if no filter bounds are violated, we have, $\forall o_j \in M(t_u), n_j(t) < c_1(t_u) \leq f_{\sigma(t_u)}(t)$. From our previous discussion, this is exactly what we wanted to achieve with the demarcation c. Note that it is easy to verify that the intervals shown in Table 2 are well-defined. For example, we can show that $c_1(t_u) \leq \min\{c_2(t_u), c_3(t_u)\}$.

Table 2: Filter bounds for $o_{\sigma(t_u)}$, $M(t_u)$ and $F(t_u)$ at timestamp t

Type	$l(n_i(t))$	$u(n_i(t))$	$l(f_i(t))$	$u(f_i(t))$
$o_{\sigma(t_u)}$	0	$f_i(t)$	$c_1(t_u)$	$\min\{c_2(t_u), c_3(t_u)\}$ [4].
$M(t_u)$	0	$c_1(t_u)$	$c_3(t_u)$	$+\infty$
$F(t_u)$	$c_2(t_u)$	$f_i(t)$	$n_i(t)$	$+\infty$

Hence the interval constraint for $f_{\sigma(t_u)}(t)$ is a valid interval. We now prove the correctness of BFP.

LEMMA 1. *Let t_u be the most recent timestamp at which filter bounds of objects are set. For any timestamp $t > t_u$, if no filter violations have occurred since t_u, $A(t) = A(t_u)$.*

Proof: Following our discussion, it suffices to show that the three conditions, namely, σ-condition, M-condition, and F-condition all hold if we substitute $\sigma(t)$ by $\sigma(t_u)$, $M(t)$ by $M(t_u)$, and $F(t)$ by $F(t_u)$.

σ-condition: If we substitute $\sigma(t)$ by $\sigma(t_u)$, the σ-condition becomes $f_{\sigma(t_u)}(t) = \min\{f_i(t)|o_i \in DB\}$, i.e., $o_{\sigma(t_u)}$ remains the min-max object at timestamp t. Since no filter violations have occurred, from Table 2, we know that
(1) $f_{\sigma(t_u)}(t) < \min\{c_2(t_u), c_3(t_u)\}$[5],
(2) $\forall o_j \in M(t_u), f_j(t) \geq c_3(t_u)$, and
(3) $\forall o_i \in F(t_u), f_i(t) > n_i(t) \geq c_2(t_u)$.
Hence, $f_{\sigma(t_u)}(t) < f_j(t)$, $\forall o_j \in (M(t_u) \cup F(t_u)) = DB - \{o_{\sigma(t_u)}\}$. The σ-condition thus follows.

M-condition: We have already shown that the M-condition holds if $\sigma(t) = \sigma(t_u)$. With the proof that the σ-condition holds, the M-condition follows.

F-condition: With the three substitutions, the F-condition reads: $\forall o_i \in F(t_u), n_i(t) > f_{\sigma(t_u)}(t)$. Again, with no filter violations, according to Table 2, we have (1) $f_{\sigma(t_u)}(t) < \min\{c_2(t_u), c_3(t_u)\}$ and (2) $\forall o_i \in F(t_u), n_i(t) \geq c_2(t_u)$. Hence, the F-condition follows.

In summary, if no filter violations have occurred since t_u, then at any timestamp $t > t_u$, $A(t) = A(t_u)$. Thus, BFP correctly maintains the answer set at timestamp t.

COROLLARY 1. *Given a* CPoNNQ *with start time t_s and end time t_e, BFP correctly maintains the answer set of the query at any timestamp $t \in [t_s, t_e]$.*

Proof: Let there be m and only m filter violations within the interval $(t_s, t_e]$, which occur at timestamps t_{u_1}, t_{u_2}, ... t_{u_m}. Let us use t_{u_0} to denote t_s, and $t_{u_{m+1}}$ to denote t_e. At any timestamp t_{u_i} $(0 \leq i \leq m)$, the server broadcasts to all the objects to acquire their locations. Hence the answer sets at those timestamps are correctly computed. For any other timestamps t, such that $t_{u_i} < t < t_{u_{i+1}}$ for some $0 \leq i \leq m$, protocol BFP returns $A(t_{u_i})$ as the answer. From Lemma 1, we know that $A(t) = A(t_{u_i})$. Hence BFP maintains correct answer sets at all timestamp $t \in [t_s, t_e]$.

5. OPTIMIZED FILTER PROTOCOL(OFP)

The main drawback of BFP is that whenever an update is received from an object, a large maintenance cost is involved:

[4]If the cardinality of $M(t_u)$ is zero, there is no $c_1(t_u)$ and $c_3(t_u)$ exist, here $u(f_i(t))$ is $c_2(t_u)$.
[5]Recall that the filter bound intervals are closed on the left and open on the right.

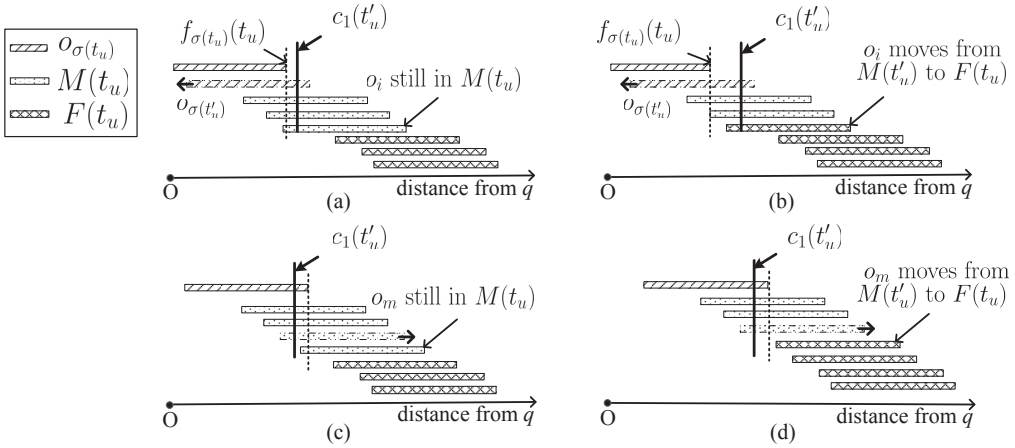

Figure 5: (a)e_σ case 1: no object in $M(t'_u)$ is assigned to $F(t_u)$. (b) e_σ case 2: some object $o_i \in M(t'_u)$ is assigned to $F(t_u)$. (c)e_M case 1: no object in $M(t'_u)$ is assigned to $F(t_u)$. (d)e_M case 2: $o_m \in M(t'_u)$ is assigned to $F(t_u)$.

the server has to request for the locations of all objects, recompute their filter bounds, and send these bounds to the objects. This is not always necessary. We now show how the maintenance overhead for two types of updates in BFP can be reduced. In this improved solution, *Optimized Filter Protocol (OFP)*, updates other than those related to o_σ and M are handled in the same way as in BFP. In Sections 5.1 and 5.2, we explain how to efficiently handle updates related to object o_σ and M, respectively.

5.1 Handling Updates of o_σ

We again let t_u be the timestamp at which the most recent filter bounds are set by the server. Let t'_u be the timestamp at which filter bounds were set just before t_u. We define the *update event*, called e_σ, for object $o_{\sigma(t'_u)}$, which occurs at timestamp t_u:

> **Event** e_σ: At timestamp t_u, the filter bound condition, $f_{\sigma(t'_u)}(t_u) \geq l(f_{\sigma(t'_u)}(t_u))$, is violated.

When e_σ occurs, $o_{\sigma(t'_u)}$ sends its location to the server. Instead of undergoing the costly maintenance phase of BFP, the server executes Algorithm 3. The algorithm has two goals: (1) it produces the correct answer set $A(t_u)$ (Steps 1-8), and (2) assigns filters correctly to the objects (Steps 9-20), which is proved by the following two lemmas.

LEMMA 2. *Steps 1-8 of Algorithm 3 correctly derive the query answer (i.e., $A(t_u)$) at timestamp t_u.*

Proof: We first claim that $\sigma(t'_u) = \sigma(t_u)$. Observe that:
(1) $o_{\sigma(t'_u)}$ is the min-max object from time $t \in [t'_u, t_u)$,
(2) $f_{\sigma(t'_u)}(t_u) < f_{\sigma(t'_u)}(t'_u)$ (since e_σ occurs), and
(3) No filter violations have occurred at any timestamp $t \in [t'_u, t_u)$.

Hence, $o_{\sigma(t'_u)}$ is the min-max object at timestamp t_u, and Step 3 correctly assigns $\sigma(t'_u)$ to $\sigma(t_u)$.

We next show that $M(t_u)$ and $F(t_u)$ can be correctly obtained. Step 2 fetches the latest values of $M(t'_u)$ and $o_{\sigma(t'_u)}$. Step 3 copies the information obtained at timestamp t'_u to the sets defined at timestamp t_u. In Steps 4-8, we check whether it is possible for some object $o_i \in M(t'_u)$ to be moved to $F(t_u)$. This is illustrated in Fig. 5(b), where after $o_{\sigma(t'_u)}$ (dotted-line rectangle) has moved to the left at t_u, $f_{\sigma(t'_u)}(t_u)$ is smaller than $n_i(t_u)$ of some object o_i. Hence, this o_i becomes a member of $F(t_u)$, as handled by Steps 6-7. Notice that all objects in $F(t'_u)$ remain in $F(t_u)$, since (1) for all $o_i \in F(t'_u), n_i(t_u) \geq c_2(t'_u)$; and (2) $c_2(t'_u) > c_1(t'_u) > f_{\sigma(t'_u)}(t_u)$. Thus, $n_i(t_u) > f_{\sigma(t'_u)}(t_u)$. Hence, Steps 1-8 correctly derives $A(t_u)$.

In Step 8, the variable *m2f* indicates whether an object o_i, originally in $M(t'_u)$, now belongs to $F(t_u)$. It is used to compute filter bounds, as discussed next.

```
1  m2f ← false;
2  Broadcast to o_i ∈ (M(t'_u) ∪ o_σ(t'_u)) to acquire their
   newest locations;
3  σ(t_u) ← σ(t'_u); M(t_u) ← M(t'_u); F(t_u) ← F(t'_u);
   A(t_u) ← A(t'_u);
4  for each object o_i in M(t_u) do
5      if n_i(t_u) > f_σ(t_u)(t_u) then
6          Move o_i from M(t_u) to F(t_u);
7          Delete o_i from A(t_u);
8          m2f ← true;
9  Obtain c_1(t_u) and c_3(t_u) using Equations 1 and 3;
10 if m2f = true then
11     c_2(t_u) ← (1 − α) · f_σ(t_u)(t_u) + α · min{n_i(t_u)|o_i ∈
       (M(t'_u) − M(t_u))};
12 else
13     c_2(t_u) ← c_2(t'_u);
14 for each object o_i in o_σ(t_u) or M(t_u) do
15     Compute filters [l(n_i(t_u)), u(n_i(t_u))) and
       [l(f_i(t_u)), u(f_i(t_u)));
16     Send filters to o_i;
17 if m2f = true then
18     for each object o_i in F(t_u) do
19         Compute filters [l(n_i(t_u)), u(n_i(t_u))) and
           [l(f_i(t_u)), u(f_i(t_u)));
20         Send filters to o_i;
```

Algorithm 3: OFP: maintenance for e_σ at t_u

LEMMA 3. *Let t_u be the most recent timestamp at which filter bounds of objects are set by Steps 9-20 of Algorithm 3. For any timestamp $t > t_u$, if no filter violations have occurred since t_u, then $A(t) = A(t_u)$.*

Proof: We now prove that σ-condition, M-condition, and F-condition all hold if we substitute $\sigma(t)$ by $\sigma(t_u)$, $M(t)$ by $M(t_u)$, and $F(t)$ by $F(t_u)$.

σ-condition: Lemma 2 shows that we correctly obtain $\sigma(t_u)$ and $M(t_u)$. In Step 3, we get their latest values. Hence, $c_1(t_u)$ and $c_3(t_u)$ can be set in the same way as BFP (Step 9). However, we cannot use Equation 2 to set $c_2(t_u)$, since the values of objects in $F(t_u)$ are *not* acquired by the algorithm. Our solution is to assign another value to $c_2(t_u)$, by considering two scenarios:

• $M(t_u) = M(t'_u)$. This is illustrated in Fig. 5(a), where after $o_{\sigma(t'_u)}$ (dotted rectangle) has moved to the left at timestamp t_u, $f_{\sigma(t'_u)}(t_u)$ is larger than all the $n_i(t_u)$ values for all $o_i \in M(t'_u)$. Hence, $M(t_u) = M(t'_u)$, and $F(t_u) = F(t'_u)$. Step 13 assigns $c_2(t'_u)$ to $c_2(t_u)$. This is correct, because:
(1) $f_{\sigma(t'_u)}(t_u) = f_{\sigma(t_u)}(t_u)$ ($\because \sigma(t_u) = \sigma(t'_u)$ as shown in Lemma 2),
(2) $f_{\sigma(t'_u)}(t_u) < c_2(t'_u)$ ($\because$ no filter violation for the upper bound of $f_{\sigma(t'_u)}(t_u)$), and
(3) $\min\{n_i(t_u)|o_i \in F(t_u)\} \geq c_2(t'_u)$ ($\because$ no filter violation for $F(t'_u)$, and $F(t'_u) = F(t_u)$).
We can thus see that:

$$c_2(t'_u) \in (f_{\sigma(t_u)}(t_u), \min\{n_i(t_u)|o_i \in F(t_u)\}] \quad (4)$$

Observe that $c_2(t'_u)$ is within the same range $(f_{\sigma(t_u)}(t_u), \min\{n_i(t_u)|o_i \in F(t_u)\}]$ of $c_2(t_u)$ (Equation 2). We thus replace $c_2(t_u)$ by $c_2(t'_u)$ in Table 2, and use arguments similar to those of Lemma 1 to show that the σ-condition holds.

• $M(t_u) \subset M(t'_u)$. There exists object o_i, where $o_i \in M(t'_u)$ but $o_i \in F(t_u)$. This is shown in Fig. 5(b). In Step 11, $c_2(t_u)$ is assigned the value of

$$\begin{aligned} c_2(t_u) \leftarrow & (1-\alpha) \cdot f_{\sigma(t_u)}(t_u) + \\ & \alpha \cdot \min\{n_i(t_u)|o_i \in (M(t'_u) - M(t_u))\} \end{aligned} \quad (5)$$

Observe that:
(1) $F(t_u) = F(t'_u) \cup (M(t'_u) - M(t_u))$,
(2) $\min\{n_i(t_u)|o_i \in (M(t'_u) - M(t_u))\} < c_1(t'_u)$ ($\because$ no violation in $M(t'_u)$),
(3) $\min\{n_i(t_u)|o_i \in F(t'_u)\} \geq c_2(t'_u)$ ($\because$ no violation in $F(t'_u)$).

Since $c_2(t'_u) > c_1(t'_u)$, we have $\min\{n_i(t_u)|o_i \in F(t_u)\} = \min\{n_i(t_u)|o_i \in (M(t'_u) - M(t_u))\}$. If we substitute this to Equation 2, we obtain Equation 5. Following the proof of Lemma 1, we see that the σ-condition holds.

M-condition: Step 2 of Algorithm 3 acquire the newest values of $o_{\sigma(t'_u)}$ and $M(t'_u)$. Step 9 uses these results to compute $c_1(t_u)$ and $c_3(t_u)$. Since these two values are used to compute the bound of $M(t_u)$ in Table 2, we can use the same argument of Lemma 1 to show that the M-condition holds.

F-condition: We have argued that the new value of $c_2(t_u)$, set in Steps 10-13, leads to the satisfaction of the σ-condition. We can use a similar argument to show that the F-condition holds. The details are skipped due to space limitation. Notice that when *m2f* is false, the old values of $c_2(t'_u)$ are used as $c_2(t_u)$. According to Table 2, $F(t_u)$ only depends on $c_2(t_u)$, with is equal to $c_2(t'_u)$. Hence, there is *no need* to send the filters to $F(t_u)$ (Steps 14-16). When *m2f* is true, the filter bounds computed based on $c_2(t_u)$ obtained in Step 11 are sent to all objects in $F(t_u)$ (Steps 17-20).

The advantage of Algorithm 3 is that it does not require the updated location of $F(t'_u)$. In case *m2f* is false, $F(t'_u) = F(t_u)$, no filter bounds are sent to $F(t_u)$. If the size of $F(t_u)$ is large, this can save a significant amount of communication cost.

```
1  m2f ← false;
2  Broadcast to o_i ∈ (M(t'_u) ∪ o_{σ(t'_u)}) to acquire their
   newest locations;
3  σ(t_u) ← σ(t'_u); M(t_u) ← M(t'_u); F(t_u) ← F(t'_u);
   A(t_u) ← A(t'_u);
4  if n_m(t_u) > f_{σ(t_u)}(t_u) then
5      Move o_m from M(t'_u) to F(t_u);
6      Delete o_m from A(t_u);
7      m2f ← true;
8  Obtain c_1(t_u) and c_3(t_u) using Equations 1 and 3;
9  if m2f = true and n_m(t_u) < c_2(t'_u) then
10     c_2(t_u) ← (1 − α) · f_{σ(t_u)}(t_u) + α · n_m(t_u);
11 else
12     c_2(t_u) ← c_2(t'_u) ;
13 for each object o_i in o_{σ(t_u)} or M(t_u) do
14     Compute filters [l(n_i(t)), u(n_i(t))) and
       [l(f_i(t)), u(f_i(t))) ;
15     Send filters to o_i;
16 if m2f = true then
17     if n_m(t_u) < c_2(t'_u) then
18         for each object o_i in F(t_u) do
19             Compute filters [l(n_i(t_u)), u(n_i(t_u))) and
               [l(f_i(t_u)), u(f_i(t_u))) ;
20             Send filters to o_i;
21     else
22         Compute filters [l(n_m(t_u)), u(n_m(t_u))) and
           [l(f_m(t_u)), u(f_m(t_u)));
23         Send filters to o_m;
```

Algorithm 4: OFP: maintenance for e_M at t_u

5.2 Handling Updates of Objects in M

Another update event, related to the set M, also allows communication cost savings. Let o_m be an object in $M(t_u)$. We now define the update event, called e_M, for any object o_m, as follows:

> **Event** e_M: At timestamp t_u, the filter bound condition, $n_m(t_u) < u(n_m(t_u))$, is violated.

When this event occurs, o_m sends its updated location value to the server. Two scenarios occur:

• $M(t_u) = M(t'_u)$, as illustrated in Fig. 5(c).
• $M(t_u) \subset M(t'_u)$, as shown in Fig. 5(d).

Notice that the cases described here are exactly the same as the ones for event e_σ. Specifically, in Fig. 5(a) and Fig. 5(c), no object in $M(t'_u)$ move to $F(t_u)$, whereas in Fig. 5(b) and Fig. 5(d), one or more objects change their type from $M(t'_u)$ to $F(t_u)$. Moreover, the upper bound $u(n_m(t_u))$ mentioned in e_M is $c_1(t'_u)$, which is the same as the lower bound $l(f_{\sigma(t'_u)}(t_u))$ described in e_σ. Hence, the solution we developed for handling e_M, Algorithm 4, is a slightly changed version of Algorithm 3. The main benefit of this algorithm is that if *m2f* is false, the update and filter bound assignment effort for $F(t_u)$ is saved. Otherwise, if $n_m(t_u) < c_2(t'_u)$, we update $c_2(t_u)$ and send the new filters to $F(t_u)$ (Steps 17-20). The detailed proof of this algorithm, which is similar to that of Algorithm 3, is skipped here due to space limitation.

To summarize, OFP is an enhanced version of BFP. In particular, the filter violation events e_σ and e_M are handled

by Algorithms 3 and 4 respectively. To manage other kinds of filter violation, OFP borrows the event handling method of BFP. Next, we discuss the experimental results of our protocols.

6. PERFORMANCE EVALUATION

We now report the results obtained from the performance studies on the three protocols: the Baseline, the Basic Filter Protocol (BFP), and the Optimized Filter Protocol (OFP). In Section 6.1, we describe the experimental setup. We present the experimental results in Section 6.2.

6.1 Experimental Setup

We use VanetMobiSim [17], an open-source vehicular traffic generator, to simulate the movement of objects. This simulator can handle multi-lane roads, speed limits for different road categories, and traffic lights. We use the environment data provided by the TIGER data source [18], where the map used is a 10km × 10km region clipped from Washington DC. The simulation time (logical) is 300 seconds. In our experiments, an object's motion is based on the *Intelligent Driver Model with Lane Changing* model [19], which regulates an object's speed based on the movement of neighboring objects. The speed range of an object provided by this simulator is between 8.33 and 13.89m/s. We assume that every object is equipped with a GPS device, which can be used to obtain the current location of the object. The GPS device captures the location of the object once every second (i.e., periodicity $dt = 1$). The number of objects simulated is $1k$ by default, producing a total of $300k$ location values. The error of a location value is represented by a circle, with a radius of 10m. This number is based on the result of GPS error analysis provided by [20]. According to [21], the amount of energy required for sending (receiving) a message of each object is 77.4mJ (25.2mJ).

For each CPoNNQ, its query point is randomly generated within the map. Unless stated otherwise, each query has a lifetime of 300 seconds. In the latter part of the experiments, we will also explain the details of simulating multiple concurrent queries. We compare the performance of the three protocols studied here (i.e., baseline, BFP, and OFP) in terms of their communication and energy costs for executing a CPoNNQ. For communication costs, we measure their number of *uplink* and *downlink* messages. An uplink message is used by an object to send a location update to the server, while a downlink message is used by the server to send filter information to an object. We present the total number of these messages per second. We also analyze the rate of energy consumed by the objects for sending and receiving messages. Each data point presented in the figures is obtained by averaging over the results for 100 queries.

The simulation program is developed in Java, and the experiments are performed by using a PC, with a configuration of Intel E8300 2.83GHz CPU and 2048MB of main memory. Table 3 summarizes the parameters and their values used.

Table 3: Parameters used in the experiments (default values are marked with asterisks)

Parameter	Values
Number of Objects	$500, 1k^\star, 1.5k, 2k, 4k, 6k, 8k, 10k$
Uncertainty Region Size (m)	0, 10*, 50, 100, ..., 300
α	0.1, 0.2,..., 0.5*,..., 0.9
Number of Queries	50, 100*, 150, 200
Query lifetime	300 seconds

6.2 Experimental Results

1. Scalability. Fig. 6(a) illustrates the communication costs against the database size. For the baseline protocol, the number of messages increases linearly, because the larger the number of objects, the more uplink messages will be generated (Fig. 6(b)). Notice that BFP performs better than baseline: it saves about 56% of the number of messages created by baseline. This is because the filters installed to objects by BFP effectively reduces the number of uplink messages. As we can see from Fig. 6(b), when the number of objects, k, is 1,000, baseline generates 1,000 uplink messages per second, while BFP produces 117 of them, which is 88.3% less. The price of this improvement is the use of more downlink messages (Fig. 6(c)). While baseline does not produce any downlink messages, BFP needs 117 of them at $k = 1,000$. However, since the number of uplink messages saved by BFP (883) is much higher than that of the downlink messages it generates (117), the total number of messages required by BFP is still less than that of baseline.

Fig. 6(a) shows that OFP outperforms BFP. On average, OFP requires 22% less messages than BFP. This is because OFP uses better routines to handle events e_σ and e_M, which reduce the number of uplink messages required (Fig. 6(b)). Since fewer uplink messages are received by the server, it also has less chance to generate downlink messages. Hence, the number of downlink messages required by OFP is less than that of BFP (Fig. 6(c)). At $k = 1,000$, the number of downlink messages drops from 117 (for BFP) to 96 (for OFP). Consequently, OFP needs fewer messages than BFP.

We also study the energy consumption costs of the protocols. As we can see from Fig. 6(d) that the overall trend of energy consumption is similar to that of the communication costs in Fig. 6(a). They can be explained by the reasons stated above. Notice that there is an importance difference between these two figures. Particularly, BFP addresses an improvement of 71% over baseline in terms of energy consumption cost, which is *larger* than the savings in communication costs (56%). This is because the amount of energy required for sending an uplink message is about 3 times that of the downlink messages. Hence, the amount of energy saved by BFP, relative to baseline, is higher than the relative amount of communication costs it saves. We can make a similar observation about OFP. While OFP requires 22% less messages than BFP, it consumes 24% energy less than BFP.

2. Effect of Uncertainty Region Size. In the second set of experiments, we examine the effect of uncertainty region size on the performance of the protocols. Here we use the radius of the uncertainty region to denote its size. Notice that if the radius equals zero, it means that all locations of objects are exact. As shown in Fig. 7(a), the total number of messages remains unchanged for baseline; for BFP and OFP, the communication cost increases with the radius. To understand why, first observe that when the size of an uncertainty region increases, the value of $f_{\sigma(t)}(t)$ at timestamp t increases accordingly. Hence, more objects may become a member of the answer set (i.e., $M(t)$), and satisfy the M-condition at the time filters are installed at them. Secondly, with a larger uncertainty region, an object previously not in an an-

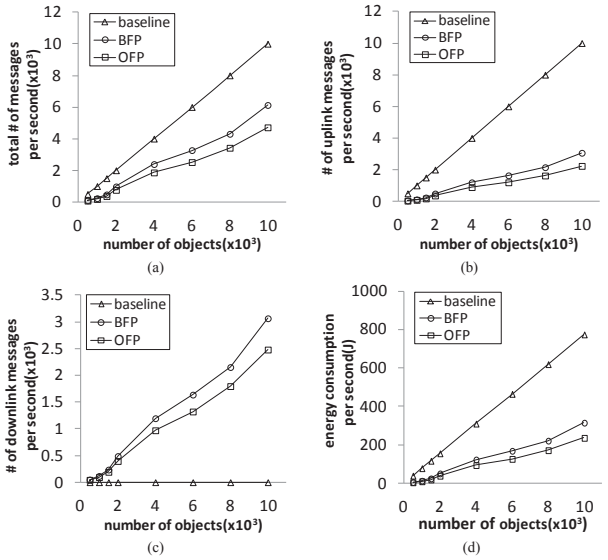

Figure 6: Effect of scalability: (a) Total # of messages; (b) # of uplink messages; (c) # of downlink messages; and (d) Energy consumption.

swer set can become a member of $M(t)$ more easily. Since the size of $M(t)$ is larger, the chance that the M-condition is later violated by any of the objects in $M(t)$ also increases. Finally, with a larger $M(t)$, the chance that any of these objects becomes o_σ later also rises. When this occurs, filters need to be regenerated for both BFP and OFP, thereby increasing their costs. Nevertheless, over a wide range of uncertainty region sizes, both protocols are still much better than baseline. When the radius is 300, for instance, BFP saves about 35% of messages required by baseline. In general, OFP performs better than BFP; when the radius is 300, it saves about 34% of BFP's communication cost. Fig. 7(b) illustrates the energy consumption costs of the three protocols. The performance is similar to Fig. 7(a), and the trend can be explained by using the same reason we just discussed.

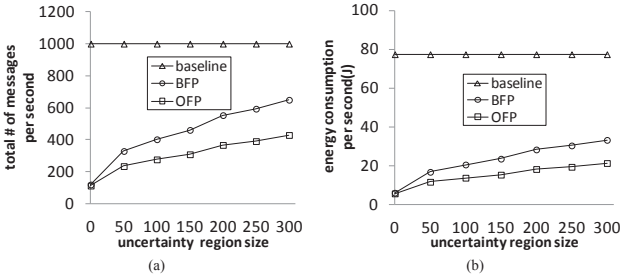

Figure 7: Effect of uncertainty region size: (a) Total # of messages; and (b) Energy consumption.

3. Effect of α. Recall that the parameter α, defined in Section 4, is used to determine the cutoff values (i.e., $c_1(t)$, $c_2(t)$, and $c_3(t)$). Fig. 8(a) shows how the communication cost of BFP is affected by this value. We can see that the number of messages is the minimum at around $\alpha = 0.6$, and is the highest when α is very small (0.1) or large (0.9). To understand why, recall from Table 2 that the three cutoff

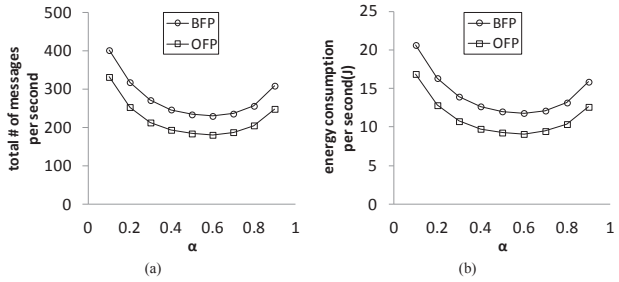

Figure 8: Effect of α: (a) Total # of messages; and (b) Energy consumption.

values determine the filter bounds used in BFP. When α increases, the three cutoff values, as well as the lower bounds of the filters, increase. When objects move towards the query point q, these filter bounds are violated more easily. For example, at timestamp t_u, for object $o_{\sigma(t_u)}$, the lower bound of $f_{\sigma(t_u)}(t)$ is $c_1(t_u)$. When α increases, so is $c_1(t_u)$. If $o_{\sigma(t_u)}$ moves towards q, $f_{\sigma(t_u)}(t)$ decreases, and so it can violate the lower bound more easily. When α approaches 1, even a slight decrease of $f_{\sigma(t_u)}(t)$ can cause a filter violation event. We can similarly argue that moving α closer to 0 decreases the upper bounds of filters, thereby increasing the chance of filter violation. If a filter violation occurs, a large maintenance cost can incur: collecting positions from objects, and sending filters to them, in Steps 10-13 of Algorithm 2. Hence, the performance of BFP suffers from the use of extremely small or large values of α.

In these experiments, we use $\alpha = 0.5$, which is just 1% worse than the optimal case ($\alpha = 0.6$). Moreover, observe that over a wide range of α (0.3 to 0.7), the performance of BFP is relatively stable. A similar observation can be made for OFP (Fig. 8(a)), as well as energy consumption costs (Fig. 8(b)). Without any knowledge about the movement patterns of objects (e.g., whether they tend to move towards or away from q), $\alpha = 0.5$ is a reasonable choice.

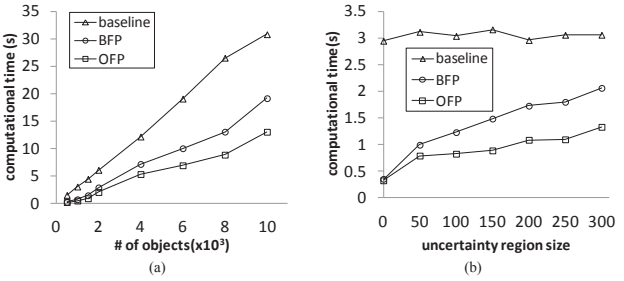

Figure 9: Computational Time vs. (a) # of objects; and (b) Uncertainty region size

4. Computational Time Analysis. Next, we measure the computational time required by the server for supporting the protocols studied here. Fig. 9(a) shows that this quantity increases with the database size, for all the protocols. To understand why, notice that the server needs to (1) retrieve the objects' locations and compute the initial query results; and (2) recompute query results and filter information upon receiving uplink messages from the objects of interest. When the number of objects increases, the burden

of handling activities (1) and (2) becomes heavier. Observe that BFP requires less time than baseline. For example, when $k = 10,000$, BFP requires 38% less computational time than baseline. This is because the filters employed in BFP significantly reduces the number of update messages to be sent to the server (c.f. Fig. 6). Hence, the computational effort required to handle (2) for BFP is also less than baseline. The computational time of OFP is smaller than that of BFP. For instance, when $k = 10,000$, OFP requires 32% less time than BFP. This is because in OFP, when events e_σ and e_M occur, the server does not need to handle the locations of objects in $F(t_u)$.

Fig. 9(b) shows the effect of the uncertainty region size. As explained before, a larger uncertainty region triggers more updates, and therefore the effort of handling activity (2) increases. Again, OFP is the winner here.

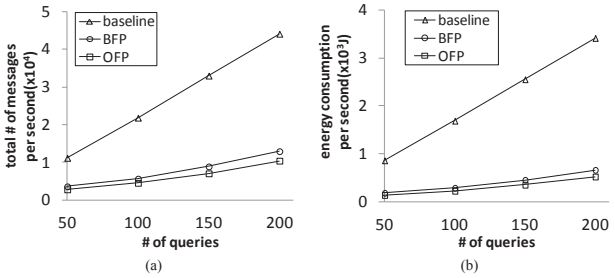

Figure 10: Results on multiple queries: (a) Total # of messages; and (b) Energy consumption.

5. Multiple Queries. In the final set of experiments, we examine the performance of our protocols in handling *multiple* queries, which are submitted to the system at different times. The simulation period spans a total of 1800 seconds. The starting time of each query is randomly distributed during the simulation period, and the lifetime of each query follows a uniform distribution in $[180, 600]$ seconds. To handle simultaneous query requests, we need to make some slight adjustments to our protocols. Specifically, the server has to maintain the filter information for every query; for each object, the filters for each query are stored. After a query has finished its execution, the server and the objects delete their filter information about this query. In these experiments, we assume that the object is capable of storing the filter information of all queries.

Figs. 10(a) and (b) show the performance result under this setting. On average, BFP saves 71% of messages and 81% of energy required by the baseline. Compared with BFP, OFP saves 21% of messages, and 22% of energy. Hence, under the multiple-user environment, OFP still performs the best.

7. CONCLUSIONS

Uncertainty management is an important issue in location-based services. In this paper, we study the communication cost of evaluating a CPoNNQ. We examine two *filter-based protocols*, namely BFP and OFP, where filter bounds are used to control when objects should report their updates. The OFP, which is an enhanced version of BFP, demonstrates superior performance in both communication and energy consumption costs. In this paper, we study how to improve the system's performance in handling events e_σ and e_M. Our next step is to consider other events (e.g., when an object changes from type F to M). We will also extend our approaches to support other complex queries in an LBS, such as k-nearest neighbor queries and skyline queries.

Acknowledgments

Reynold Cheng, Yifan Jin, and Yinuo Zhang were supported by the Research Grants Council of Hong Kong (GRF Projects 711110, 711309E, 513508). We would like to thank the anonymous reviewers for their insightful comments.

8. REFERENCES

[1] D. Pfoser and S. Jensen. Capturing the uncertainty of moving-object representations. In *SSD 1999*.
[2] R. Cheng et al. Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data. In *ICDE 2008*.
[3] H. Kriegel et al. Probabilistic nearest-neighbor query on uncertain objects. In *DASFAA 2007*.
[4] R. Cheng, D. Kalashnikov, and S. Prabhakar. Querying imprecise data in moving object environments. *TKDE 2004*.
[5] R. Cheng et al. Evaluating probability threshold k-nearest-neighbor queries over uncertain data. In *EDBT 2009*.
[6] J. Chen et al. Scalable processing of snapshot and continuous nearest-neighbor queries over one-dimensional uncertain data. *VLDBJ 2009*.
[7] Y. Zhang, R. Cheng, and J. Chen. Evaluating continuous probabilistic queries over imprecise sensor data. In *DASFAA 2010*.
[8] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB 2002*.
[9] J. Zhu et al. A probabilistic filter protocol for continuous nearest-neighbor query. In *YC-ICT 2009*.
[10] C. Chow, M. Mokbel, and H. Leong. On efficient and scalable support of continuous queries in mobile peer-to-peer environments. *Mobile Computing 2011*.
[11] K. Mouratidis et al. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *TKDE 2005*.
[12] R. Cheng et al. Adaptive stream filters for entity-based queries with non-value tolerance. In *VLDB 2005*.
[13] R. Cheng et al. Filtering data streams for entity-based continuous queries. *TKDE 2010*.
[14] C. Olston et al. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD 2003*.
[15] J. Chen et al. A probabilistic filter protocol for continuous queries. In *QuaCon 2009*.
[16] G. Trajcevski et al. Continuous probabilistic nearest-neighbor queries for uncertain trajectories. In *EDBT 2009*.
[17] J. Härri et al. Vanetmobisim: generating realistic mobility patterns for vanets. In *VANET 2006*.
[18] `http://www.census.gov/geo/www/tiger/shp.html`.
[19] M. Fiore et al. Vehicular mobility simulation for vanets. In *ANSS 2007*.
[20] Bradford W. and James J. Spilker. GPS error analysis. In *Global Positioning System: Theory and Applications 1996*.
[21] *MPR-Mote Processor Radio Board User's Manual*.