

# An Efficient Location Update Mechanism for Continuous Queries over Moving Objects<sup>\*</sup>

Reynold Cheng<sup>1</sup>, Kam-Yiu Lam<sup>2</sup>, Sunil Prabhakar<sup>3</sup> and Biyu Liang<sup>2,4</sup>

Department of Computing<sup>1</sup>  
Hong Kong Polytechnic University  
Hung Hom, Kowloon,  
HONG KONG

Department of Computer Science<sup>3</sup>  
Purdue University  
West Lafayette, IN 47907, USA  
Email: cskcheng@comp.polyu.edu.hk, cskylam@cityu.edu.hk, sunil@cs.purdue.edu, bliang@cs.uvm.edu

Department of Computer Science<sup>2</sup>  
City University of Hong Kong  
Kowloon Tong  
HONG KONG

Department of Computer Science<sup>4</sup>  
University of Vermont  
Burlington, VT 05405, USA

## Abstract

In a moving-object database system that supports continuous queries (CQ), an important problem is to keep the location data consistent with the actual locations of the entities being monitored, in order to produce correct query results. This goal is often difficult to achieve due to limited network resources. However, if an object is not required by any query, its value need not be refreshed. Based on this observation, we redefine the notion of *temporal consistency* of data items with respect to the query result, where only data items that are relevant to the CQs need to be fresh. To exploit this correctness definition, we develop an adaptive time-based update technique called *Query-Result Update (QRU)*. The advantage of this technique is that it identifies objects with different levels of significance to the correctness of query results. Locations of objects that have more impact to the query result are acquired more frequently than the ones that do not.

To achieve this objective, queries are classified into *rank-based* (i.e., ranks of objects are critical to query results) and *non-rank-based*. For each query class, QRU decides the time instant that an object should send a location update based on the predicted impact of the object to the query result. Moreover, the location update frequency of each object is continuously adjusted in order to adapt to the accuracy of the prediction model used. We evaluate the effectiveness of QRU by simulating execution of CQs over synthetic and real data sets. We also compare QRU experimentally with existing location update policies, namely the distance-based method, the time-based method, the speed dead-reckoning method, as well as the safe region strategy.

*Keywords: moving objects, continuous queries, update generation schemes, temporal correctness*

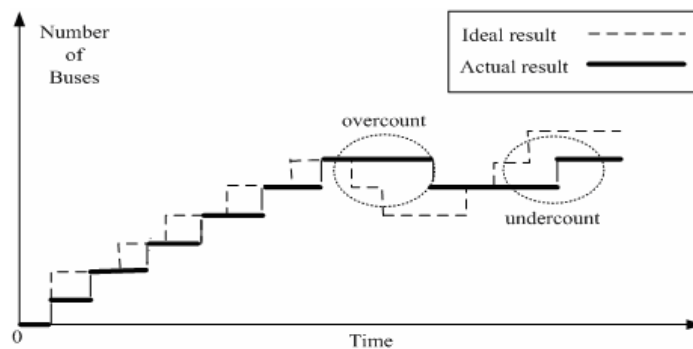
## 1 Introduction

Due to advances in mobile communication technologies, novel location-based applications have emerged in recent years. One important application is monitoring systems. These systems keep track of the locations of objects for control and management purposes in a real-time fashion [WCDJ97, WSCY99, GL04]. For example, a bus management system monitors the locations of buses and adjusts bus schedules dynamically to alleviate traffic congestion. In another example, the moving paths of courier service carriers are constantly tracked to facilitate optimal planning of job assignments.

---

<sup>\*</sup> The work reported in this paper was supported in part by the City University of Hong Kong under the strategic grant number 7001472.

In these systems, databases have to be used to store location data. In addition, *Continuous Queries* (CQs) are used to monitor remotely the status of the moving objects. These are queries that reside in the system for an extensive amount of time, with their answers being refreshed if necessary [GL04, OJW03]. An example CQ is a *Range Count Query* (RCQ) that reports the number of buses within a user-defined region. This number is updated whenever a location update from the objects being monitored is received. Figure 1 illustrates the results returned by this CQ during its lifetime in solid lines, where the number of buses increase or decrease at discrete time points.



**Figure 1 : Results from the ideal situation and the actual situation of a RCQ as a function of time.**

A location update can be obtained from moving objects through either the *pull* scheme (where the system requests objects to obtain their locations) or the *push* scheme (where moving objects send update messages to the system without being requested) [LR01]. Ideally, these update mechanisms should ensure that the location values in the database are the same as the actual locations of the moving objects. This is difficult to achieve in practice due to limited resources, such as limited network bandwidth, instability of network transmission (especially in wireless networks), and battery power of portable devices. Besides, a CQ may have to be recomputed every time a location report is received, and causes heavy burden to the system.

Hence the database values are often different from the current locations of the moving objects. This is termed *temporal inconsistency* in [R93]. A direct consequence of temporal inconsistency is that query results can be incorrect. This is illustrated in Figure 1, where the dotted line corresponds to the “ideal result” – the true number of objects satisfying the RCQ. It is different from the results reported by evaluating the database values since the locations of objects are not timely reported to the system. Two kinds of errors can occur: if an object is

included incorrectly in the answer, an *over-count* is generated; if it is wrongly excluded from the answer, an *under-count* is produced. These errors must be reduced in order to provide users with high quality answers.

Intuitively, the query result of evaluating the database can have higher quality by requesting moving objects to report location updates more frequently, so that the discrepancy between the database and the external environment can be reduced. However, this imposes a heavier communication load on the system. Thus, there is a trade-off between *the frequency of location update* and *the correctness of query results*. This is also known as the *location update* problem in [WSCY99], the focus of this paper.

An important observation is that if a data item is not accessed by any query, it does not matter whether it is temporally inconsistent or not. These data items can be allowed to report updates less frequently, in order to save more system resources. We thus propose temporal consistency to be studied from the *viewpoint of a query*: we achieve high quality results by only maintaining correctness of data that are relevant to the execution CQs. We adopt the notion of *fidelity* [DKPR01], defined as the probability that a CQ is correct during its execution time, in order to measure temporal consistency. We will see how such a definition allows us to design better protocols that reduce communication costs significantly.

In particular, we propose ***Query-Result Update (QRU)*** to control the rate of location updates for multiple concurrently executing continuous queries. The goal of QRU is to yield high fidelity with low communication costs. In QRU, the update generation rate is adaptively assigned based on the impact of the uncertainty in location of the objects to query answers, as well as accuracy of location prediction. QRU also handles different location management problems, such as network disconnection and loss of update messages.

The specific details of QRU depend on the *query class* to which the CQ belongs. Recall that QRU is designed to achieve a high level of fidelity, which measures query correctness. Each query class has its own definition of fidelity. More importantly, each query class has a different level of “tolerance” to temporal consistency. Consider an object  $O$  accessed by a RCQ that counts the number of objects inside an area. If there are thousands of objects inside the area, the failure to provide the accurate position of  $O$  to the query introduces little error to the result. However, if  $O$  is accessed by a query that returns the nearest neighbor of a point, and  $O$  happens to be the current nearest neighbor, a slight error in the position of  $O$  can render an incorrect result.

QRU recognizes these differences by mapping queries into two classes, namely *rank-based query* and *non-rank-based query* – the former performs some “ranking operations” on location data to produce results, while the latter does not. QRU then applies different update timing mechanisms to each query class.

Although the proposed techniques are applied to CQs, they may be used to support “past queries”, such as answering “Which vehicles were in the area of an accident that happened at 6:30pm yesterday?” If this area is constantly monitored, we can store all the versions of the updates that are generated by our mechanism. By interpolating the stored location data, the objects’ locations at the time specified by the past queries can be used to obtain the query answers.

To summarize, our main contributions are:

- (1) Use of fidelity to measure correctness of the results of CQs for different query classes. Fidelity measures are also developed for answers that return single values (value-based) or sets (entity-based);
- (2) Development of a novel time-based location-update scheme, called *Query-Result Update (QRU)*, to yield high fidelity results with low update costs;
- (3) Presentation of QRU algorithms for both non-rank-based queries (e.g., range query) and rank-based queries (e.g.,  $k$  nearest-neighbor query); and
- (4) Extensive experiments for comparing the effectiveness of QRU with existing location-update techniques.

The rest of this paper is organized as follows. Related work is presented in Section 2. Section 3 discusses the system model. Section 4 describes a query classification scheme and fidelity, and in Section 5 we present the details of QRU. Section 6 presents simulation results and Section 7 concludes the paper.

## 2 Related Work

The location update problem has been the subject of interest for mobile communication systems and location-dependent applications, and various update mechanisms have been proposed recently [PS01]. These methods can be classified into *push* (reporting) and *pull* (querying) schemes [LR01]. For continuous queries, it has been shown that push schemes are more efficient than pull methods [DKPR01]. The conditions for pushing information are usually based on the distance moved or the time elapsed since the last location update. In particular, the *time-based* (TB) scheme requires that an object reports its location at every fixed period. In the

*distance-based* (DB) scheme, an object generates a location update if the distance traveled from the last location is more than a pre-defined threshold.

To further reduce location update cost, various mobility prediction schemes were proposed [WA02, WSCY99]. In [SWCD97], the Moving Objects Spatio-Temporal (MOST) model was proposed for modeling object movement and predicting locations. In MOST, plain dead-reckoning (*pdr*) method [WCDJ97] was proposed where an update is generated to refresh the object's location and re-define its location function when the deviation of its current location is greater than a pre-defined threshold. In [WSCY99], three types of dead-reckoning policies were developed. First, the adaptive dead-reckoning (*adr*) extends *pdr* by computing the threshold not only based on deviation but also based on update costs. The second policy, known as the speed dead-reckoning (SDR) scheme, requires an object to produce an update if its current position is farther than the predicted location by a distance threshold. Finally, the disconnected detecting dead-reckoning (*dtldr*) policy handles network disconnection problems by continually decreasing the threshold after the latest update is received. In this scheme, the object should have more chance to send an update as time goes by; if no update is received after some extensive amount of time, the system knows that a network disconnection occurs.

While these methods succeed in bounding the location uncertainty of moving objects with lower cost, the information needed by continuous queries is not considered. Even if an object is not required by any CQ, its location information is still “pushed” to the system. As a result some objects are kept fresh unnecessarily. Our update mechanism exploits query information, and adjusts the update frequency based on the object's significance to the queries. This saves system resources (such as wireless network bandwidth and processing time), since mobile objects that are not relevant to any queries are given less amount of update bandwidth.

Some recent work, including [PX02], [OJW03] and [MPBT05], exploits the information about continuous queries in order to reduce the number of updates. The main idea of this work is to send a maximum bound (or region) to the object based on the conditions of the query. In these “query-aware distance-based policies”, an object only needs to report its location if it moves beyond the bound in order to guarantee correctness. Particularly, [PX02] computes a “safe region”, which is derived based on the boundaries of the range queries. The safe region is sent to the object, and as long as the object does not cross the boundary of the

safe region (i.e., stays outside any of the query ranges), it does not need to generate any update. The bounds of the objects for satisfying exact  $k$ -nearest-neighbor queries were studied in [MPBT05]. “Adaptive filters” were introduced in [OJW03], where the bound widths can be adjusted dynamically according to the system conditions while meeting precision requirements of queries. Recently, by assuming that there is sufficient amount of memory and computation power, [CHC04] proposed that range queries relevant to the object can be sent to it to judge directly whether an update should be sent.

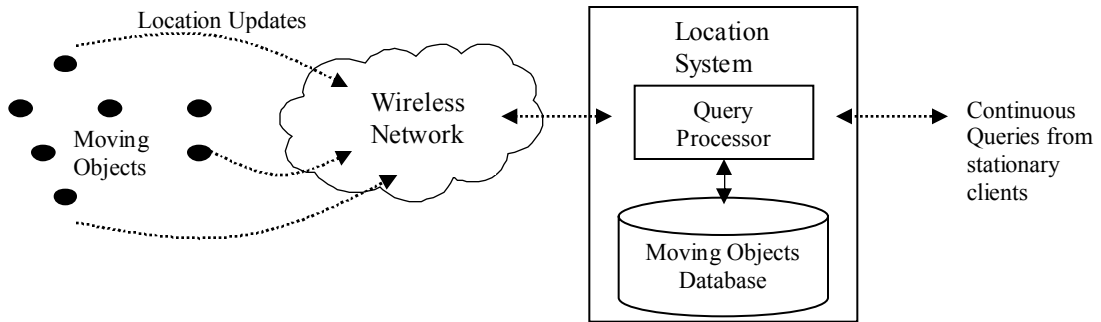
The main assumption of these “query-aware distance-based” algorithms is that if the system does not receive any updates from the objects, it assumes they do not move outside the bound, and the query answer is guaranteed to be correct. In reality, disconnection between data sources and the system in a wireless environment can happen, and also network packets can be lost [WSCY99]. Thus, query correctness can be violated without the system being notified. Instead of sending a bound to the object, the protocol presented in this paper sends a time value which indicates when the object has to send its next location update. Thus, if the system does not hear from the object at the expected time, it can suspect some network problem (e.g., network disconnection). We will discuss the advantages of this “time-based” approach in Section 4.1. Another assumption made by the existing algorithms is that transmission of messages in the network is instantaneous. As we have explained in Section 1, network packets often experience propagation delay in a wireless environment. When the object necessitates an update (e.g., its location moves beyond the maximum bound), its location may not be sent to the system immediately, and incorrect query results can be produced (e.g., Figure 1). We define “fidelity” in Section 4 in order to quantify the effect of network problems to query correctness.

Other ways for reducing re-evaluation cost of continuous queries in the server are semantic load shedding [DC02] and spatial query techniques [PX02]. They may be combined with QRU (that reduces communication costs) to further improve performance. In our approach, less updates are generated by the sources, and so less updates are received by the system, resulting in a lower re-evaluation effort.

### **3 System Model**

Our system model consists of a location database system and a number of moving objects connected by a mobile network, as shown in Figure 2. The location database system maintains a moving object database which

records the location information of moving objects. Periodically, the moving objects submit location updates to the location database system. From the locations in successive updates, the system derives information about the movement pattern for each object, e.g., its speed and direction of movement. The clients submit CQs to the location database system. Here we assume that the clients are stationary, and are connected to the system through a fixed network.<sup>1</sup> For example, a control center can submit queries to monitor the movement of buses and courier carriers. Each CQ, namely  $Q_i$  ( $i=1, \dots, n$ ), is associated with a *monitoring set*  $S_i$  of moving objects, as well an *activation period*  $[begin\_time_i, end\_time_i]$ , during which the CQ is being evaluated. We assume that at the start of a query  $Q_i$  its set  $S_i$  is known to the system, and  $S_i$  remains unchanged throughout the activation period. Whenever the system receives an update of an object in  $S_i$ ,  $Q_i$  is recomputed and returns the updated result to the user.



**Figure 2: The System Model.**

The model does not assume a moving object has high processing capability. It can generate location data with a positioning device, e.g., GPS, and perform simple calculations such as speed evaluation. However, it is not able to perform complex processing, such as evaluating a CQ or computing the traveling distance to a destination in a road/path network. These operations may be too complicated for common low-cost instruments like portable devices and location sensor networks. Moreover, the resources of these devices can be limited (e.g., the battery power of a sensor is scarce); by assuming that only simple operations are performed the amount of power consumed can be kept small.

<sup>1</sup> QRU needs to be revised when the movement of clients is considered, which we will consider in our future work.

## 4 Query Classification, Temporal Consistency, and Fidelity

In order to achieve high fidelity, QRU maps queries into classes. Each query class has its own definition of fidelity, and requires different treatment by QRU. This section describes how queries are classified, and also presents the formal definitions of temporal consistency and fidelity.

### 4.1 Classification of Continuous Queries

Continuous queries can be classified into two categories. First, we classify them based on whether ordering (ranking) operations are performed:

1. A **rank-based query** is one where a “partial-ordering” operation is performed on the data items to produce an answer. A typical example is a nearest-neighbor query (NNQ), where the identity of the object with the shortest distance to the query point is reported [KGT99].
2. A **non-rank-based query** is any query that is not rank-based, e.g., the range count query (RCQ), which returns the number of objects that fall within a specified range. In RCQ, whether each object is included in the answer is independent of its rank.

A query can also be classified based on the *nature of answer*, which can be a single value, or a set of objects [CKP03]:

1. A **value-based query** returns a single real value. For example, querying the distance of a particular object from a specified query point returns a single value.
2. An **entity-based query** returns a set of objects that satisfy the condition of the query. One example is the range query (RQ) that returns a set of objects with locations inside the query range.

Query Class	Value-Based	Entity-Based
<b>Non-Rank-Based</b>	Range Count Query (RCQ)	Range Query (RQ)
<b>Rank-Based</b>	Nearest Distance Query (NDQ)	Nearest Neighbor Query (NNQ), K-Nearest Neighbor Query (KNNQ)

Table 1: Classification of Continuous Queries.

Based on the above discussions, we obtain four query classes (Table 1). This classification does not capture every kind of queries (e.g., queries that evaluate sub-queries and return both value- and set-based answers). However, it applies to a wide range of common queries. We further assume the queries have certain “selection constraints” imposed on their answers (e.g., for range query, the answer is within a region; for

nearest-neighbor queries, the answer has to be the nearest-neighbor). Queries with no selection constraints like “what is the location of a particular object?” are not discussed here.

## 4.2 Temporal Consistency of Continuous Queries

Suppose at 3pm, John is located at (10,30), but his coordinates stored in the database are (10,25). The stored location value is said to be *temporally inconsistent* with the actual location value [R93]. Given a value  $v(t)$  of a data item and its corresponding actual value  $a(t)$  at time  $t$ , we can measure temporal consistency of  $v(t)$  with respect to  $a(t)$  by computing their differences. Maintaining a high degree of temporal consistency is important, since it affects the accuracy of query results.

As explained before, it is hard to ensure a high level of temporal consistency for every object in a resource-limited environment. To address this, [KLA03] defines temporal consistency of discrete objects, such as stock values from the *perspective* of a real-time query. We extend this notion to support continuously-evolving data for CQs. Examples of continuously evolving data include locations of moving objects, temperature, and other sensor data. Suppose each CQ informs the system about the set of objects that it is interested in at the time it is submitted to the system (for example, a query for monitoring the movement of certain types of vehicles in an area). Since an item not monitored by any CQ does not affect query correctness (even if it is not updated), the system then only has to focus on the temporal consistency of data that are relevant to the CQs.

Specifically, we define an **evaluated result** as a query result derived from database values, and a **true result** as the result computed by using the actual data values. In some models (e.g., [PX02, CHC04, MPBT05]), the true result may be obtained by dispatching the query information to the mobile objects where the queries are executed on the actual values. Alternatively, the true result may be approximated by techniques like prediction or interpolation. The *temporal error ratio* is defined as the relative difference between the **evaluated result** and the **true result**. The definition of temporal error ratio is different for each query class. An object is said to observe temporal consistency if the temporal error ratio is bounded by a threshold called **temporal consistency constraint (TC)**. It can be set by the query issuer depending on what degree erroneous answers are accepted

(alternatively, default values can be set by the system). As we will see next, the specific definition of TC depends on the query classes. In general, the lower the value of TC, the harder it is to achieve high fidelity.

#### 4.2.1 Temporal Consistency of Value-Based Queries

Suppose the system has  $n$  continuous queries actively running. Let  $Q_i$  be the  $i$ -th CQ ( $i=1, \dots, n$ ). Each  $Q_i$  is associated with a *monitoring set*  $S_i$  of objects that it is interested in, and an *activation period*  $[begin\_time_i, end\_time_i]$ , during which the CQ is active in the system.

The temporal consistency for value-based queries is defined as follows.

DEFINITION: *Temporal consistency in the result of a value-based query*  $Q_i$  is observed at time  $t$  if:

$$E_i(t) = \frac{|V_{ideal}(Q_i, t) - V_{dbase}(Q_i, t)|}{|V_{dbase}(Q_i, t)|} < \varepsilon_i \quad (1),$$

where  $E_i(t)$  is the *temporal error ratio* of  $Q_i$ ,  $\varepsilon_i$  is the TC of  $Q_i$  (in this case being a real-valued system parameter),  $V_{dbase}(Q_i, t)$  is the result of  $Q_i$  at time  $t$  by using the database information, and  $V_{ideal}(Q_i, t)$  is the result of  $Q_i$  at time  $t$  using perfect knowledge of the location data. That is,  $V_{ideal}$  is generated by assuming the system has instantaneous information about the moving objects. Essentially, Eqn. (1) expresses the difference between the true result ( $V_{ideal}$ ) and the result retrieved from the database ( $V_{dbase}$ ). The value of  $E_i(t)$  can be interpreted as the fraction of the value returned that is incorrect. The result of  $Q_i$  is considered temporally consistent only if  $E_i(t)$  is smaller than  $\varepsilon_i$ . An example query that uses this metric is RCQ in Figure 1. In order to have a small value of  $E_i(t)$  and maintain temporal consistency, the relevant items have to be refreshed appropriately.

#### 4.2.2 Temporal Consistency of Entity-Based Queries

While a value-based query returns a real-valued answer, the answer of an entity-based query is a set of objects. This section discusses temporal consistency for both rank-based queries and non-rank-based entity queries. For non-rank-based entity queries, the definition of temporal consistency is based on false positives and false negatives, similar to recall and precision in the Information Retrieval literature [HGP03].

DEFINITION: The ratio of false positives of  $Q_i$  at time  $t$ , denoted by  $f^+(Q_i, t)$ , is defined as:

$$f^+(Q_i, t) = \frac{|S_{dbase}(Q_i, t) - S_{ideal}(Q_i, t)|}{|S_{dbase}(Q_i, t)|} \quad (2),$$

where  $S_{dbase}(Q_i, t)$  is the result set of  $Q_i$  at time  $t$ , evaluated using the object database;  $S_{ideal}(Q_i, t)$  is the result set of  $Q_i$  at time  $t$ , evaluated using true information. Here,  $|S_{dbase}(Q_i, t) - S_{ideal}(Q_i, t)|$  represents the number of objects that appear in  $S_{dbase}$  but not in  $S_{ideal}$ . Thus,  $f^+(Q_i, t)$  measures the fraction of objects wrongly included in the answer of  $Q_i$  at time  $t$ .

**DEFINITION:** The ratio of false negatives, denoted by  $f^-$ , is defined as:

$$f^-(Q_i, t) = \frac{|S_{ideal}(Q_i, t) - S_{dbase}(Q_i, t)|}{|S_{ideal}(Q_i, t)|} \quad (3),$$

which measures the fraction of objects appearing in  $S_{ideal}$  but not in  $S_{dbase}$  of  $Q_i$  at time  $t$ . Using Eqn.(2) and Eqn.(3), we can define temporal consistency for non-rank-based entity queries.

**DEFINITION:** *Temporal consistency in the result of a non-rank-based entity query  $Q_i$  is maintained at time  $t$  if:*

$$E_i(t) = f^+(Q_i, t) + f^-(Q_i, t) < \varepsilon_i \quad (4),$$

where  $E_i(t)$  is the temporal error ratio of  $Q_i$  at time  $t$ , and  $\varepsilon_i$ , the temporal consistency constraint (TC), is a real-valued system parameter for  $Q_i$ . Temporal consistency is maintained if  $E_i(t)$  is lower than  $\varepsilon_i$ .

For rank-based entity queries, the rank of a query answer should deviate from the required rank by not more than a threshold. We denote  $rank(O_j)$  to be the true rank of object  $O_j$ . For instance, if  $O_j$  is actually the  $m$ -th nearest neighbor,  $rank(O_j)$  equals to  $m$ .

**DEFINITION:** *Temporal consistency in the result of a rank-based entity query  $Q_i$  is maintained at time  $t$  if:*

$$E_i(t) = \max_{O_j \in S_{dbase}(Q_i, t)} rank(O_j) < \varepsilon_i \quad (5),$$

where  $E_i(t)$  is the error of the query answer of  $Q_i$  at time  $t$ , and  $\varepsilon_i$ , the temporal consistency requirement (TC), is an integer-valued system parameter for  $Q_i$ . For a NNQ, since  $S_{dbase}$  contains only one item (say,  $O_{dNN}$ ), the above definition is equivalent to  $rank(O_{dNN}) < \varepsilon_i$ . For a KNNQ,  $S_{dbase}$  contains  $k$  items returned to the user, and the above definition requires the lowest rank of the object in  $S_{dbase}$  to be less than  $\varepsilon_i$ . Note that the definitions of

temporal consistency (Equations (1), (4), (5)) are based on a query, rather than on an object. Thus, data items not in  $S_i$  may have values in the database that deviate significantly from the values of the entities they model.

### 4.3 Fidelity of Continuous Queries

So far we have defined temporal consistency for query  $Q_i$  only at time instant  $t$ . Since  $Q_i$  is active during  $[begin\_time_i, end\_time_i]$ , it is useful to measure the fraction of  $Q_i$ 's activation period when temporal consistency is observed. Here we adopt the definition of *fidelity* [DKPR01]. Let  $F_i(t)$  be a function that returns 1 if  $E_i(t) < \epsilon_i$ , and 0 otherwise. Then the *fidelity* of  $Q_i$  is defined as:

$$fidelity_i = \frac{\int_{begin\_time_i}^{end\_time_i} F_i(t) dt}{end\_time_i - begin\_time_i} \quad (6).$$

We can view fidelity as the probability that a query is temporally consistent during its activation period. To measure the effectiveness of the system in achieving fidelity, we define *fidelity of the system* (FS):

**DEFINITION:** The *fidelity of the system* (FS) consisting of  $n$  queries is defined as:

$$FS = \frac{\sum_{i=1}^n fidelity_i}{n} \quad (7),$$

which is the average of fidelity of all continuous queries in the system. We also define the average error ratio ( $E_i$ ) for a query over its activation period:

$$E_i = \frac{\int_{begin\_time_i}^{end\_time_i} E_i(t) dt}{end\_time_i - begin\_time_i}$$

which is used to define the accuracy of the system:

**DEFINITION:** The *system accuracy* consisting of  $n$  queries is defined as:

$$\text{system accuracy} = \frac{\sum_{i=1}^n (1 - E_i)}{n} \quad (8),$$

where for every query  $Q_i$ ,  $1 - E_i$  is the accuracy of the query result throughout its activation period, and the system accuracy is an average over all  $(1 - E_i)$ 's.

#### 4.4 Sensitivity of Fidelity to False Information

The temporal consistency of rank-based and non-ranked queries exhibits a different level of sensitivity to false information (e.g., uncertainty in locations of moving objects). Observing this difference can help us design better location update policies in order to achieve higher fidelity. In general, more stringent conditions for generating updates should be adopted if the fidelity of a query is more sensitive towards false location information. Consider a RQ, which belongs to the non-rank-based entity class. If a database system uses the false location of an object, the effect on the answer is the inclusion of a false positive or a false negative;  $f^+$  and  $f^-$  may only change slightly and the temporal consistency of the query can still be maintained. On the other hand, the temporal consistency of a rank-based entity query may be much more sensitive, and may be heavily affected by even a single fault. For example, in a NNQ, if outdated location data of an object  $x$  is used, and the outdated location has the minimum distance from the query point, the query will wrongly return  $x$  as the answer. Temporal consistency may then be violated, even though all objects except  $x$  have reported accurate location values to the system.

A similar situation occurs in value-based queries. Consider a RCQ, which is non-rank based. If the RCQ uses false location data of one object, the result just differs from the true count by 1. Thus the temporal consistency of the RCQ may still be maintained. However, a NDQ can return a result that deviates a lot from the true minimum distance, since it may use a wrong object to compute the minimum distance. Its result may not observe temporal consistency, even though the false information of only one object is used.

Fidelity is also sensitive to the *number* of the objects that contribute to the result. For example, in a RCQ, if many objects satisfy the query, a false positive or a false negative may affect the fidelity of RCQ only lightly; if the number is small, the query may be more sensitive to false information. The number of objects that satisfy the RCQ also depends on the size of the query range. A RCQ with a smaller query range is more likely to have fewer numbers of objects than a RCQ with a large query range. Hence, the fidelity of a RCQ with a small query range may be more sensitive than one with a large query range.

The effect of false information on various query types is treated carefully by QRU. In particular, QRU is customized to handle each query class in a different manner – the location-update generation frequencies of

objects are different, according to the class of the query. For example, if an object is the current nearest neighbor in the result of a NNQ, it will be required to report its location more frequently than an object which is the 100th nearest neighbor, since the position of the nearest neighbor is more important to the result of the NNQ.

Let us now examine the QRU algorithm, which aims at maintaining high query fidelity.

## 5 Query-Result Update (QRU) Scheme

The *Query-Result Update (QRU)* scheme is designed to achieve high system fidelity by carefully controlling the generation of location updates. We will discuss the basic idea of QRU (Section 5.1), followed by explaining how to set the values of two key parameters of QRU, namely *Predicted Time Interval (PTI)* (Section 5.2) and *x-proportion* (Section 5.3). Section 5.4 proposes a modified scheme that improves the performance of QRU.

### 5.1 Design Overview

QRU is a *time-based* scheme i.e., an object is informed about the time interval between generating successive updates. Rather than using a *distance-based* scheme where an object notifies the system its location once it moves beyond a distance limit, we base our design on a time-based scheme because it can detect network disconnection and loss of update messages. If an object does not report to the system at the assigned report time, time-based schemes may conclude that the object is disconnected from the system. The system can then still return query results including information to the user that the result is not reliable. It may also include the maximum possible error in the reported results due to disconnection. In a distance-based scheme, if disconnection happens, the system may wrongly deduce that the object is within a distance threshold.

Unlike traditional time-based schemes where an object is assigned a fixed time threshold, QRU adjusts the time threshold assigned to an object dynamically. The threshold is based on: (1) *the impact of the object to query fidelity*; and (2) *the prediction accuracy of the object's movement*. We term a message sent by the system to an object a *control message*, and a message sent by the object to the system a *data message*. Two phases, namely *Initialization Phase* and *Refresh Phase*, are executed for each CQ. The Initialization Phase is executed once in order to prepare for a repeated execution of the Refresh Phase.

**Initialization Phase.** In this phase, the system initializes the state required for correct execution.

1. The system retrieves the set of items that  $Q_i$  is interested in, i.e., the monitoring set  $S_i$  (we assume when  $Q_i$  is submitted to the system, it has told the system what  $S_i$  is).
2. The system sends a control message to each object in  $S_i$ , and asks each object to: (i) send its current location value to the system, and (ii) wait for  $L_{time}$  time units before sending its next position, where  $L_{time}$  is a system parameter that specifies the minimum time threshold for generating the next update.
3. The objects in  $S_i$  respond by sending their location values to the system.
4. The system evaluates the CQ based on the values just obtained in  $S_i$ .

Notice that if  $S_i$  changes during its query execution period, the initial phase is re-executed to cope with the change in  $S_i$ .

**Refresh Phase.** Next, the system evaluates  $Q_i$  repeatedly during its activation period, as shown in Figure 3.

1. **While**  $begin\_time_i \leq current\ time \leq end\_time_i$  **do**
2. Upon receiving a location value from object  $o \in S_i$  **do**
  - a. Evaluate  $Q_i$  with the new location of  $o$  and return the result to the user.
  - b. Predict the *speed* and *direction* of  $o$ .
  - c. Compute the *predicted time interval (PTI)* and adjust the *x-proportion* of  $o$  based on (b).
  - d. Compute the new time threshold of  $o$  based on the results of (c).
  - e. Send a control message to  $o$  to inform it about the new time threshold.

**Figure 3: Algorithm - Refresh Phase**

In Step (a), the new result is computed by using the new value of  $o$ . Next, it predicts the velocity of  $o$  (Step (b)) by using the movement history of  $o$ . For example, the predicted speed can be obtained by dividing the difference in locations between the past samples by their difference in timestamps, while the direction can be predicted by considering the location reported in the current data message compared to the location in previous messages. The predicted speed and direction are used to compute a parameter called *predicted time interval (PTI)* and to adjust an error factor called *x-proportion* (Step (c)). Using these two parameters, a new time threshold is computed (Step (d)) and sent to  $o$ , which uses the threshold to calculate the time instant for generating the next update (Step (e)). We will explain Steps (c) and (d) in detail in Sections 5.2 and 5.3. Notice that if an object is accessed by more than one CQ at the same time, the time threshold of the object being accessed by each CQ may be different. In this case, QRU evaluates the object's time threshold values for all

CQs accessing it, and then generates a control message with the minimum of these values. For our purpose, we also define the *answer region* of a CQ:

**DEFINITION:** The *answer region* of a CQ is a close region such that if an object  $o$  is located inside it,  $o$  satisfies the CQ; otherwise  $o$  does not.

For example, the answer region of a RQ is simply the query region specified in the query: the objects satisfying the RQ are the ones inside the query region (answer region). The answer region is fixed throughout the lifetime of the RQ. On the other hand, the answer region of a NNQ is a circle, centered at the query point  $q$  specified by the query, with radius of length  $|q - \text{the location of the nearest neighbor of } q|$ . Any object that enters the answer region of the NNQ has to be included in the answer. If this happens, the answer region shrinks since the answer (the nearest neighbor) has changed. The answer region of a NNQ is expanded if the current nearest neighbor moves away from  $q$  and is still closer to  $q$  than other objects. In general, an answer region is defined when queries have certain “constraints” on answers, such as the user-specified range in a RQ.

We now discuss *PTI* and *x-proportion* in Figure 3 Step (c) which are used to decide on a good time interval for update generation.

**1. Predicted Time Interval (*PTI*).** This parameter is a prediction of:

- (1) the amount of time an object currently outside the answer region of a CQ needs to enter the answer region; or
- (2) the amount of time an object currently inside the answer region of a CQ needs to exit the region.

Whenever an object enters or leaves the answer region, the answer of a CQ changes. Therefore, *PTI* is the amount of time that the change of location of the object does not influence the result to query CQ. The value of *PTI* assists in the process of setting the next location report time, as will be explained next.

As an example, since the answer region of a RQ is exactly the user-specified range, the *PTI* of an object will be the predicted amount of time it takes an object to leave or enter the range. If the *PTI* of a moving object is large and the object currently satisfies the condition of the CQ, then it is assumed that it will not fail the condition in the near future. A corresponding assumption holds if the object currently fails the condition. Hence,

a large update time threshold may be assigned to this object without severely affecting the correctness of the answer. If an object's *PTI* is small, it is predicted to have a good chance of satisfying/failing the CQ soon; a smaller time threshold may be set to track the object more closely in order to have a more accurate query result.

The exact prediction method used for computing *PTI* is flexible and application-dependent. Many appropriate mobility prediction models like the stochastic model and Gauss-Markov model [LH99] can be used. Statistics about roads can also be applied to improve prediction accuracy [WA02]. Since our focus is not on the effectiveness of various prediction techniques, we choose a simpler prediction model – linear prediction – for illustration purposes.<sup>2</sup> It allows us to understand the behavior of QRU more easily.

In particular, we assume the *PTI* of object  $O_i$  is computed by the following formula:

$$\text{PTI of } O_i = \frac{\text{predicted distance for } O_i \text{ to exit or enter the answer region}}{\text{predicted speed of } O_i \text{ to exit or enter the answer region}} \quad (9)$$

where the value of the nominator (distance) depends on the roads/paths to the answer region from the current location of the object. The denominator (speed) can be decided by the speed and direction obtained in Figure 3 Step (b). The accuracy of this value depends on: (1) the movement characteristics of the object; and (2) the movement behavior of other objects on the paths/roads connecting the object to the answer region. We simplify the model by predicting objects move in straight lines, without using information about connections of roads and paths (in the experiments, however, objects move in a non-linear fashion). In Section 5.2, we develop algorithms for computing *PTI* for different query classes using this prediction model.

**2. *x-proportion.*** Can we simply set the update time threshold of an object based on the *PTI* value? The answer is negative, because the actual speed of an object may differ from its predicted value, and incorrect results may be produced. To explain this, suppose *PTI* is used to decide the next time the location of object  $O$  is received, which is accessed by an RQ. The system assumes that  $O$  arrives at the boundary of the query range of the RQ at about the time it receives the update. In reality,  $O$  moves faster than predicted and passes through the query region long before it sends back the update. The system can then fail to include  $O$  in its result.

---

<sup>2</sup> Section 5.2.3 demonstrates how *PTI* can be evaluated for NNQ using a more general prediction model.

The *x-proportion*, a real value between 0 and 1, reduces the negative impact of inaccurate prediction on query fidelity. It is used to fine-tune the time threshold according to prediction accuracy. Instead of using the whole PTI value to set an object’s threshold, QRU only uses a fraction of it, so that the system has a smaller chance of failing to include (exclude) the object into (from) the query result, even if prediction is inaccurate. Specifically, the new time threshold for an object  $O_i$ , upon receiving  $O_i$ ’s update in Figure 3 Step (d), is:

$$\text{Time threshold for } O_i = \text{current time} + \max(\text{PTI of } O_i \cdot x\text{-proportion of } O_i, L_{\text{time}}) \quad (10)$$

For a moving object with a *highly predictable* movement pattern, a large *x-proportion* suffices. If the prediction error is large, QRU adjusts *x-proportion* in order to alleviate the effect of poor prediction on fidelity. If the object moves *faster* than predicted in entering or exiting the answer region, *x-proportion* is reduced so that the location information of the object arrives earlier. On the contrary, a larger *x-proportion* is used if the movement is “*slower*” than predicted. We discuss the setting of *x-proportion* again in Section 5.3.

## 5.2 Evaluation of Predicted Time Interval (PTI)

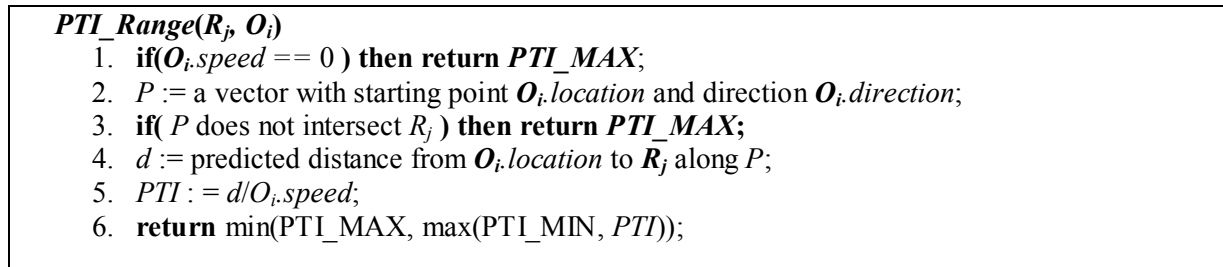
In Figure 3 Step (c) of the Refresh Phase, each time a location update is received, a new *PTI* is computed. A non-rank-based query sets the *PTI* in a manner different from a rank-based query. In Section 5.2.1, we discuss how to set *PTI* for non-rank-based queries, using RQ and RCQ as examples. Sections 5.2.2 and 5.2.3 illustrate the evaluation of *PTI* for two rank-based queries – NNQ and KNNQ, respectively. Section 5.2.4 extends the algorithms to support a generic movement prediction model.

### 5.2.1 Computing *PTI* for Non-Rank-Based Queries

For a non-rank-based query, the estimation of *PTI* is based on the answer region of the query, the current locations, predicted speeds and directions of the objects. We consider RQ and RCQ, which are non-rank-based queries deciding which objects fall within a user-specified range  $R$ . They use the same algorithm, *PTI\_Range* (Figure 4), to decide on the *PTI* value assigned to an object  $O_i$  in their monitoring sets.

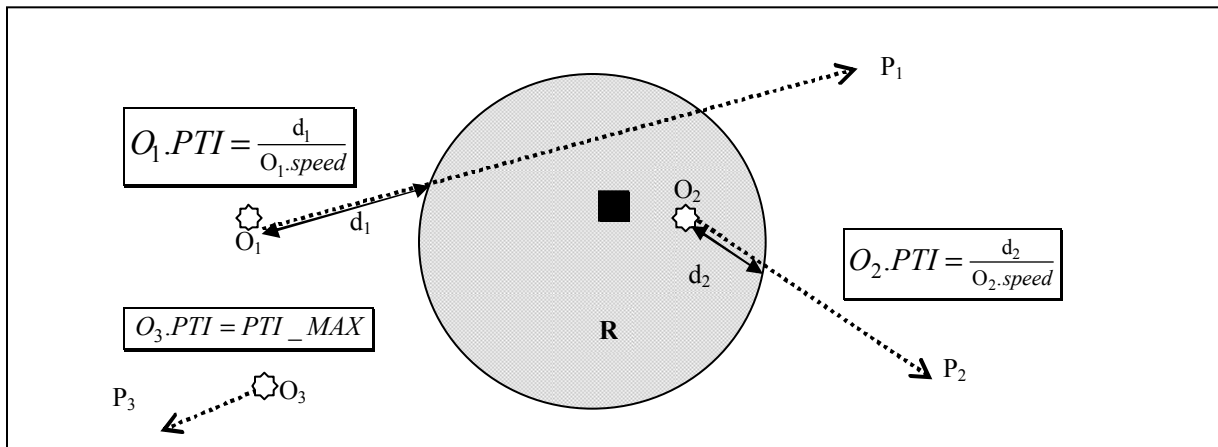
Algorithm *PTI\_Range* uses the range  $R_j$  of query  $Q_j$  and  $O_i$ ’s current location, predicted speed and direction (from Step (b) of Refresh Phase (Figure 3)) to decide on the *PTI* of  $O_i$ . If it is predicted that  $O_i$  is not moving, its *PTI* is assigned a value of *PTI\_MAX* to reduce its frequency of generating location updates (Step 1).

Otherwise, a vector  $P$  is generated to predict the moving path of  $O_i$ , and the shortest distance  $d$   $O_i$  can travel until it reaches the range of the query is computed (Steps 2-4). We then calculate the  $PTI$  based on Eqn.(9) (Step 5). The system parameters,  $PTI\_MAX$  and  $PTI\_MIN$ , are defined to ensure that the computed  $PTI$  is within  $[PTI\_MIN, PTI\_MAX]$  (Step 6). By default,  $PTI\_MIN$  is equal to  $L_{time}$ , the minimum time threshold. In Section 6, we investigate the impact of different values of  $PTI\_MIN$  and  $PTI\_MAX$  on the performance of QRU.



**Figure 4: Algorithm  $PTI\_Range$  - Computing the  $PTI$  of  $O_i$  for a RQ/RCQ.**

Figure 5 illustrates  $PTI\_Range$ . Three objects,  $O_1$ ,  $O_2$  and  $O_3$ , are members of the monitoring set of an RCQ, with range  $R$ . Vectors  $P_1$ ,  $P_2$  and  $P_3$  are generated for them correspondingly (Step 2). Since  $P_3$  does not intersect  $R$  nor travel towards  $R$ , its  $PTI$  is  $PTI\_MAX$  (Step 3). The shortest distance of  $O_1$  ( $O_2$ ) to cross  $R$  along the direction of  $P_1$  ( $P_2$ ) is  $d_1$  ( $d_2$ ) (Step 4). Their  $PTI$ s are computed in Steps 4 and 5.



**Figure 5: Calculating the  $PTI$  of objects for a RQ/RCQ.**

**Complexity.** All steps except Steps 3 and 4 need  $O(1)$  times. If  $R_i$  is a  $m_i$ -sided polygon,  $O(m_i)$  times are needed to find if  $P$  intersects  $R_i$  (Step 3), and which intersection gives the minimum distance (Step 4). This is done by testing the line segment of  $P$  against each side of  $R_i$ . The complexity of  $PTI\_Range$  is  $O(m_i) + O(1) = O(m_i)$ .

## 5.2.2 Computing *PTI* for Nearest-Neighbor Queries

Evaluating *PTI*s for objects accessed by a rank-based query is not trivial since its *answer region* is *not fixed*. For example, the answer region for an *NNQ* is a circle centered at the query point with  $d_{NN}$ , the distance of the current nearest neighbor  $N_i$  to the query point, as radius. The answer region may change due to the movement of  $N_i$  and other relevant objects. Since the evaluation of *PTI* involves answer regions (Eqn. (9)), the system has to keep track of the changes in the answer region. Thus, the system not only has to monitor the nearest neighbor, but also has to be aware of the objects that can modify the current answer region.

Algorithm *PTI\_NN* (Figure 6) computes the *PTI* of an object  $O_i \in S_i$ , given the latest nearest neighbor  $N_i$  and the query point  $q_i$  of query  $Q_i$ . If  $O_i$  is the current nearest neighbor, *PTI\_NN* quits (Step 1). Steps 2 and 3 initialize variables *CurrentNN* and  $N_i$ . Steps 4 and 5 compute for all objects in  $S_i$  their predicted velocities and point-to-point distances from  $q_i$ . Then, using the loop in Step 6, the time needed for the next object to become the nearest neighbor is predicted. The whole process is repeated until  $O_i$  is found to be the nearest neighbor (we will explain Step 6 again in more detail). Step 6(ii) computes the value of  $T$ , which is the time needed for  $O_i$  to become the nearest neighbor. If  $T$  is greater than *PTI\_MAX*, the search will stop (Step 7). The *PTI* of  $O_i$  is then assigned to be the minimum of *PTI\_MAX* and  $\max(\text{PTI\_MIN}, T)$  (Step 8).

***PTI\_NN***( $q_i, O_i, S_i, N_i$ )

1. **if** ( $O_i == N_i$ ) **then return** **PTI\_MIN**;
2. *CurrentNN* :=  $N_i$ ; // current nearest neighbor
3.  $T := 0$ ;
4. **for each object**  $obj \in S_i$  **do** { // initialize with last update location information
  - a.  $v_{obj}$  := component vector of predicted velocity of  $obj$  with direction towards the query point;
  - b.  $d_{obj}$  := point-to-point distance of  $obj$  from the query point;
5. }
6. **do** {
  - i. *Next\_Candidate* := The first object in  $S_i$  to take over *CurrentNN* as the new nearest neighbor;
  - ii.  $T := T +$  (time it takes for *Next\_Candidate* to take over *CurrentNN* as the new nearest neighbor);
  - iii. **for each object**  $obj \in S_i$  **do** { Update the values of  $v_{obj}$  and  $d_{obj}$  at time  $T$ };
  - iv. *CurrentNN* := *Next\_Candidate*; // shifting current nearest neighbor
7. } **while** (*CurrentNN*  $\neq O_i$  **and**  $T < \text{PTI\_MAX}$  )
8. **return**  $\min(\text{PTI\_MAX}, \max(\text{PTI\_MIN}, T))$ ;

**Figure 6: Algorithm *PTI\_NN* - Computing the *PTI*'s in an *NNQ*.**

To predict when an object becomes the nearest neighbor (Steps 6(i) and (ii)), we consider eight scenarios. As shown in Figure 7, in cases 1 to 5 the current nearest neighbor *NN* is moving towards query point  $q$ , while in cases 6 to 8, *NN* is moving away from  $q$ . In cases 5 and 8,  $O_i$  will not replace *NN* as the next nearest

neighbor if the moving speeds and directions of both  $O_i$  and  $NN$  remain unchanged. Otherwise,  $O_i$  may become the nearest neighbor.

$PTI\_NN\_next$  (Figure 8) refines Steps 6(i) to (iii) of  $PTI\_NN$  based on the observation in Figure 7. First, Step 2 defines  $T_{min}$  as the minimum time needed for a new object to become the nearest neighbor. Steps 3 and 4 define  $v_{NN}$  and  $d_{NN}$ , which correspond to the current nearest neighbor's velocity vector towards the query point  $q_i$  and the distance from  $q_i$  respectively. Step 5 computes the time needed for  $O_i$  to become the next nearest neighbor (i.e., implements  $PTI\_NN$ 's 6(i)), based on the eight different cases shown in Figure 7. We explain Step 5 in more detail. Step 5 calculates the time it takes each object  $O_i$  to become the nearest neighbor as needed for Step 6(ii) of  $PTI\_NN$ . Step 7 updates the values of  $v_{obj}$  and  $d_{obj}$  for each object as needed for Step 6(iii) of  $PTI\_NN$ . It considers whether the object has moved towards  $q_i$  (Step 7(A)(I)) or away from  $q_i$  (Step 7(A)(II)). The former case is divided into two subcases: (i)  $q_i$  has been reached by the object before  $T_{min}$  (Step 7(A)(I)(b)) and (ii)  $q_i$  has not yet been reached (Step 7(A)(I)(c)).

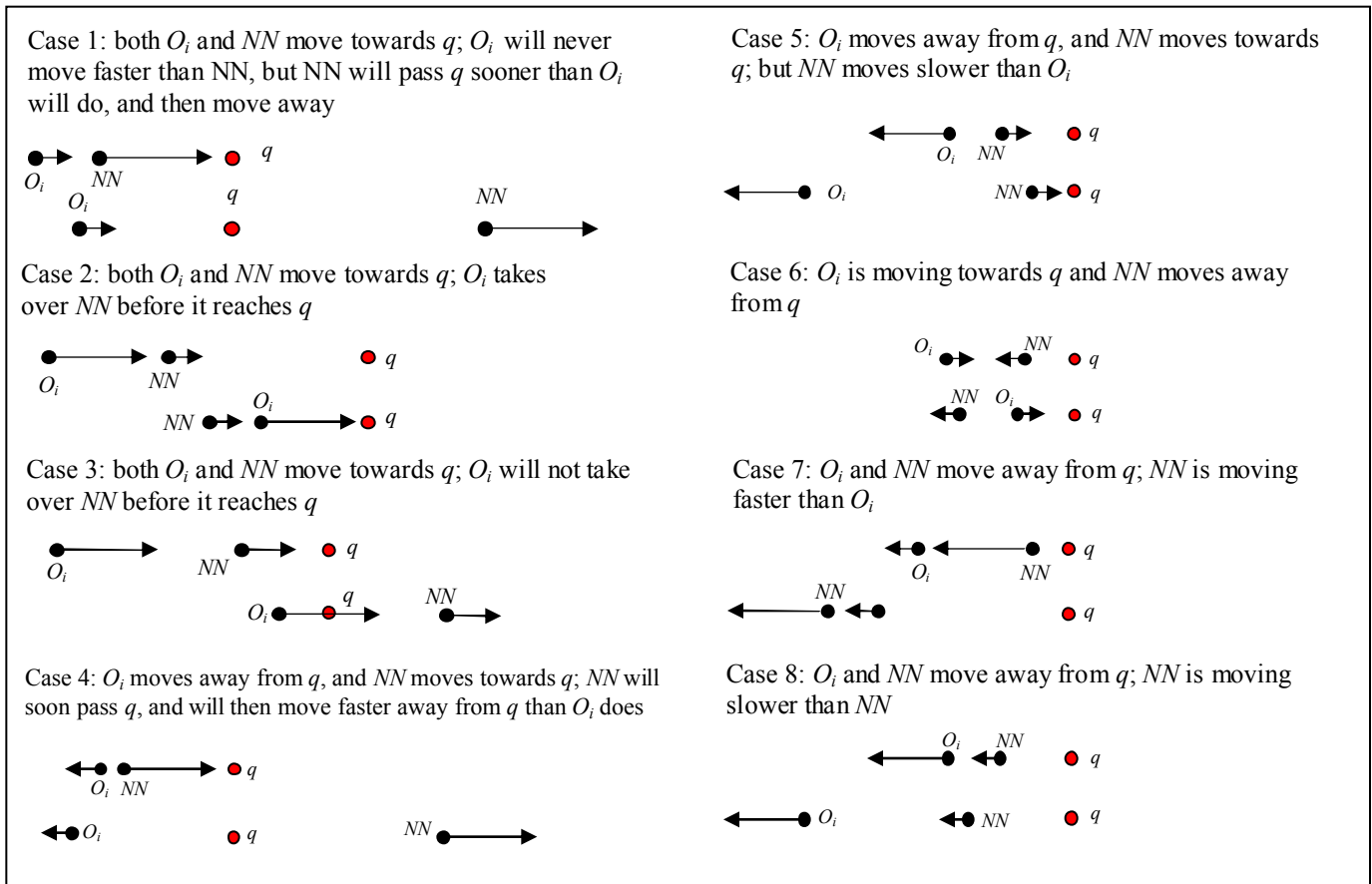


Figure 7: Different scenarios of the current nearest neighbor (NN) and an object  $O_i$ .

Step 5, the crucial step for computing the time that  $O_i$  takes over  $CurrentNN$  as the newest nearest neighbor, is divided into two sets of cases – whether the nearest neighbor is moving towards  $q_i$  (Step 5(A)) or away from  $q_i$  (Step 5(B)). Specifically, Step 5(A) implements Cases 1 to 5 and Step 5(B) takes care of Cases 6 to 8 in Figure 7. To understand how  $T_{obj}$  is computed for Case 1, notice that since  $v_{obj} < v_{NN}$ ,  $CurrentNN$  reaches  $q_i$  before  $obj$  does, after time interval  $T_{NN}$ . At that time,  $obj$  is  $(d_{obj} - v_{obj} \cdot T_{NN})$  units away from  $q_i$ . Let  $t$  be the amount of time needed for  $obj$  to become the next nearest neighbor after  $CurrentNN$  has reached  $q_i$ . Then,  $t$  is the time needed such that both  $obj$  and  $CurrentNN$  have the same distance from  $q_i$ , i.e.,

$$(d_{obj} - v_{obj} \cdot T_{NN}) - v_{obj} \cdot t = v_{NN} \cdot t$$

Hence,  $t$  is equal to  $(d_{obj} - v_{obj} \cdot T_{NN}) / (v_{obj} + v_{NN})$ , and the time needed for  $obj$  to be the next nearest neighbor,  $T_{obj}$ , is equal to  $T_{NN} + (d_{obj} - v_{obj} \cdot T_{NN}) / (v_{obj} + v_{NN})$ . These steps are detailed in Step 5(A). The reasoning for other cases is similar and is skipped here.

```

1. Next_Candidate := NULL;
2. T_min := infinity;
3. v_NN := component vector of velocity of CurrentNN with direction towards q_i;
4. d_NN := distance of CurrentNN from q_i;
// PTI_NN Step 6(i): find the first object in S_i to take over CurrentNN as the new nearest neighbor
5. for each object obj except CurrentNN do {
  A. if ( v_NN > 0 ) then { // the nearest neighbor is moving towards q_i
    I. T_NN := d_NN / v_NN; // time cost for CurrentNN to reach query point
    II. if ( -v_NN < v_obj < v_NN ) then
      a. // obj does not move towards or not move towards faster than CurrentNN
      b. // i.e. CurrentNN reaches q_i first. Case 1 or 4
      c. T_obj := T_NN + (d_obj - v_obj * T_NN) / (v_obj + v_NN);
    III. else if ( v_obj > v_NN ) then {
      a. T_obj := (d_obj - d_NN) / (v_obj - v_NN); // Case 2
      b. if ( T_obj > T_NN ) then T_obj := T_NN + (d_obj - v_obj * T_NN) / (v_obj + v_NN); // Case 3
    }
    IV. else continue; // Case 5. Check next object.
  V. if ( T_obj < T_min ) then {
      a. T_min := T_obj;
      b. Next_Candidate := obj;
    }
  }
  B. else { // the nearest neighbor is moving away from q_i
    I. if ( v_obj > v_NN ) then { // obj does not move away or does not move faster
      a. T_obj := (d_obj - d_NN) / (v_obj - v_NN); // Case 6 or 7
      b. if ( T_obj < T_min ) then {
          i. T_min := T_obj;
          ii. Next_Candidate := obj;
        }
    }
  }
}

```

```

    }
    II. else continue; // do nothing for case 8
  }
}

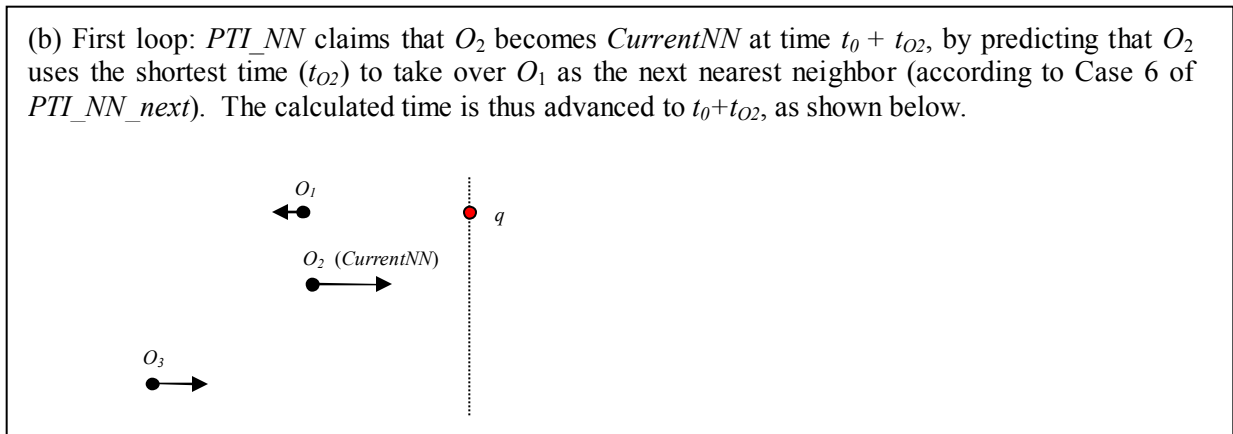
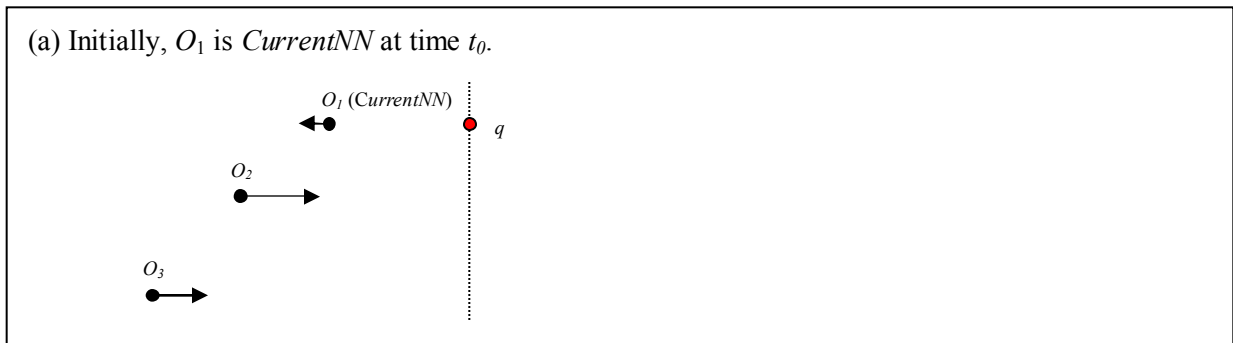
// PTI_NN Step 6(ii)
6.  $T := T + T_{min}$ ;

// PTI_NN Step 6(iii): update  $v_{obj}$  and  $d_{obj}$  of each object at time  $T+T_{min}$ 
7. if ( $Next\_Candidate \neq NULL$ ) then {
  A. for each object  $obj \in S_i$  do { // update velocity and distance
    I. if ( $v_{obj} > 0$ ) then { // the object is moving towards  $q_i$ 
      a.  $T_{obj} := d_{obj}/v_{obj}$ ;
      b. if ( $T_{min} > T_{obj}$ ) then { // reach query point before  $T_{min}$ 
          i.  $d_{obj} := |v_{obj} \cdot (T_{min} - T_{obj})|$ ;
          ii.  $v_{obj} := -v_{obj}$ ; // moving away
        }
      c. else
          i.  $d_{obj} := |d_{obj} - v_{obj} \cdot T_{min}|$ ;
        }
    II. else // moving away from query point
      a.  $d_{obj} := |d_{obj} - v_{obj} \cdot T_{min}|$ ;
    }
  }
}

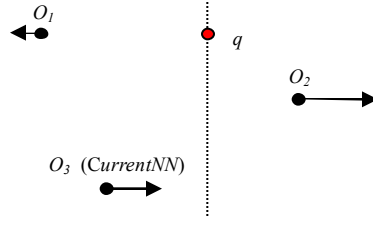
```

Figure 8: Algorithm  $PTI\_NN\_next$  - Computing the time for object  $O_i$  to become NN.

**Example** We now use an example to illustrate  $PTI\_NN$ . We suppose that a NNQ observes three objects ( $O_1$ ,  $O_2$  and  $O_3$ ), and  $PTI\_NN$  is evaluated at time  $t_0$ . The following explains how  $PTI$  is computed for  $O_3$ .



(c) Second loop:  $PTI\_NN$  predicts that  $O_3$  becomes  $CurrentNN$  at time  $t_0 + t_{O_2} + t_{O_3}$ , since it takes the shortest time ( $t_{O_3}$ ) to take over  $O_2$  after the time instant  $t_0 + t_{O_2}$  as the next nearest neighbor (according to Case 1 of Algorithm  $PTI\_NN\_next$ ).  $O_3$ 's  $PTI$  is thus equal to  $t_0 + t_{O_2} + t_{O_3}$ , which is the time needed for  $O_3$  to become the nearest neighbor, and  $PTI\_NN$  stops. The following figure shows the scenario at time  $t_0 + t_{O_2} + t_{O_3}$ .



**Computing  $PTI$  for the monitoring set** The  $PTI$ 's of all objects in the monitoring set of  $Q_i$  ( $S_i$ ) are usually updated at the same time. The performance of  $PTI\_NN$  can be improved by not computing  $PTI$  of the same object repeatedly. In particular, we have developed Algorithm  $PTI\_NN\_Batch$  (Figure 9), a slightly modified version of  $PTI\_NN$  (Figure 6), which saves computation time by evaluating  $PTI$  only once for each object. While  $PTI\_NN$  computes  $PTI$  for a single object,  $PTI\_NN\_Batch$  computes  $PTI$  for each object in  $S_i$ . As we can see in Step 4, once an object is examined, it is not further considered. Another important step is Step 4(B), which is modified to exclude objects that have been examined. Notice that  $PTI\_NN\_Batch$  has a time complexity of  $O(n^2)$ , the same as  $PTI\_NN$ .

**$PTI\_NN\_Batch(q_i, S_i, N_i)$**

1.  $N_i.PTI = PTI\_MIN$ ;
2.  $CurrentNN := N_i$ ; // current nearest neighbor;
3.  $T := 0$ ;
4. While (not all objects in  $(S_i - N_i)$  have their  $PTI$ 's computed) {
  - A. for each object  $obj \in S_i$  do { // initialize with last update location information
    - I.  $v_{obj} :=$  component vector of velocity of  $obj$  with direction towards  $q_i$  at time  $T$ ;
    - II.  $d_{obj} :=$  distance of  $obj$  from  $q_i$  at time  $T$ ;
- B.  $Next\_Candidate :=$  The first obj in  $S_i$  to take over  $CurrentNN$  as the new nearest neighbor and not examined yet;
- C.  $T := T +$  (time it takes for  $Next\_Candidate$  to take over  $CurrentNN$  as the new nearest neighbor);
- D.  $CurrentNN := Next\_Candidate$ ; // shifting current nearest neighbor
- E.  $CurrentNN.PTI = \min(PTI\_MAX, \max(PTI\_MIN, T))$ ;

**Figure 9: Algorithm  $PTI\_NN\_Batch$**

**Complexity.** The running time of  $PTI\_NN$  is  $O(n^2)$ , where  $n$  is the number of objects in a query. In the worst case, the *do-while* loop iterates  $O(n)$  steps; each step checks all  $n$  objects to find the next candidate. In practice, the number of steps is much smaller than  $O(n^2)$  since as shown in Section 6, the condition for  $PTI$  being larger than  $PTI\_MAX$  is easy to satisfy. For  $PTI\_NN\_Batch$ , Step 4 iterates for  $O(n)$  times for a

total of  $|S_i - N_i|$  objects. Inside the *While-do* loop, Step 4(A) takes  $n$  steps to complete. Steps 4(B) to 4(E) needs  $O(n)$  times to complete. Thus the running time of *PTI\_NN\_Batch* is  $O(n^2)$ .

### 5.2.3 Computing *PTI* for *k*-Nearest-Neighbor Queries

Algorithm *PTI\_NN* can be extended to compute *PTI* for KNNQ (a rank-based query). In Algorithm *PTI\_KNN* (Figure 10), the major change is the separation of the set of relevant objects into two sets:  $S_F$  (the set of objects whose distances are farther than the  $k$ th nearest neighbor) and  $S_N$  (the set of objects whose distances are nearer than the  $k$ th-nearest neighbor). Again, Step 4 computes  $v_{obj}$  (the velocity vector of  $obj$  towards  $q_i$ ) and  $d_{obj}$  (the distance of  $obj$  from  $q_i$ ) for each object  $obj$ . Step 5 is the main loop that repeats itself until  $O_i$  is the newest  $k$ -nearest neighbor (called *CurrentkNN*). Step 5(a) partitions the objects into two sets:  $S_F$  and  $S_N$ . Then, from  $S_F$ , we find the first object, *Next\_Candidate<sub>F</sub>*, that will take over *CurrentkNN* as the next nearest neighbor among the members in  $S_F \cup \{CurrentkNN\}$ , and the time that this happens,  $T_F$  (Step 5(b-c)). Also, from  $S_N$ , we find the first object, *Next\_Candidate<sub>N</sub>*, that will take over *CurrentkNN* as the next farthest neighbor among the members in  $S_N \cup \{CurrentkNN\}$ , and the time  $T_N$  (Step 5(d-e)). Notice that steps 5b to 5e can be implemented by changing *PTI\_NN\_next* (Figure 8) slightly. Steps 5(f) to 5(i) reassign *CurrentkNN* to *Next\_Candidate<sub>F</sub>* (*Next\_Candidate<sub>N</sub>*) if  $T_F < T_N$  ( $T_F \geq T_N$ ). Finally, step 7 returns the minimum of **PTI\_MAX** and  $\max(\mathbf{PTI\_MIN}, T)$ .

*PTI\_KNN*( $q_i, O_i, S_i, N_i$ )

1. **if** ( $O_i == N_i$ ) **then return** **PTI\_MIN**;
2. *CurrentKNN* :=  $N_i$ ; // current k-nearest neighbor
3.  $T := 0$ ;
4. **for each object**  $obj \in S_i$  **do** { // initialize with last update location information
  - a.  $v_{obj}$  := component vector of velocity of  $obj$  with direction towards  $q_i$ ;
  - b.  $d_{obj}$  := distance of  $obj$  from  $q_i$ ;
5. **do** {
  - a. Based on  $\{d_{obj}\}$ , partition the set of objects relevant to the query into two sets  $S_F$  and  $S_N$ ;  
//  $S_F$  is the set of objects whose distance is farther than *CurrentkNN*  
//  $S_N$  is the set of objects whose distance is nearer than *CurrentkNN*
  - b. *Next\_Candidate<sub>F</sub>* := The first object to take over *CurrentKNN* as nearest neighbor among  $S_F \cup \{CurrentkNN\}$ ;
  - c.  $T_F$  := time it takes for *Next\_Candidate<sub>F</sub>* to take over *CurrentKNN* as nearest neighbor among  $S_F \cup \{CurrentkNN\}$ ;
  - d. *Next\_Candidate<sub>N</sub>* := The first object to take over *CurrentKNN* as farthest neighbor among  $S_N \cup \{CurrentkNN\}$ ;
  - e.  $T_N$  := time it takes for *Next\_Candidate<sub>N</sub>* to take over *CurrentKNN* as farthest neighbor among

```

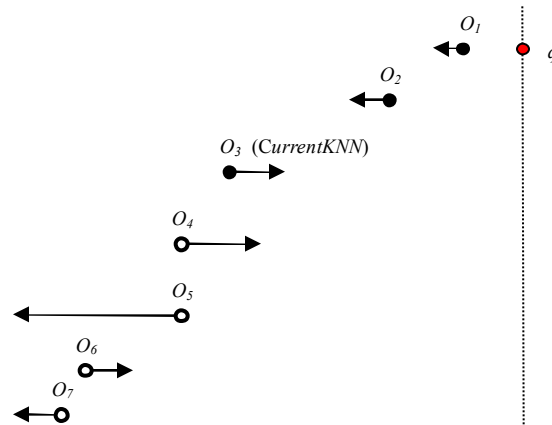
 $S_N \cup \{CurrentKNN\};$ 
f.  $T := T + \min\{T_F, T_N\};$ 
g. for each object  $obj \in S_i$  do { Update the values of  $v_{obj}$  and  $d_{obj}$  at time  $T$  };
h. if ( $T_F < T_N$ ) then
    1.  $CurrentKNN := Next\_Candidate_F$ ; // shifting current k-nearest neighbor
i. else
    1.  $CurrentKNN := Next\_Candidate_N$ ; // shifting current k-nearest neighbor
}
6. while ( $CurrentKNN \neq O_i$  and  $T < PTI\_MAX$ );
7. return  $\min(PTI\_MAX, \max(PTI\_MIN, T))$ ;

```

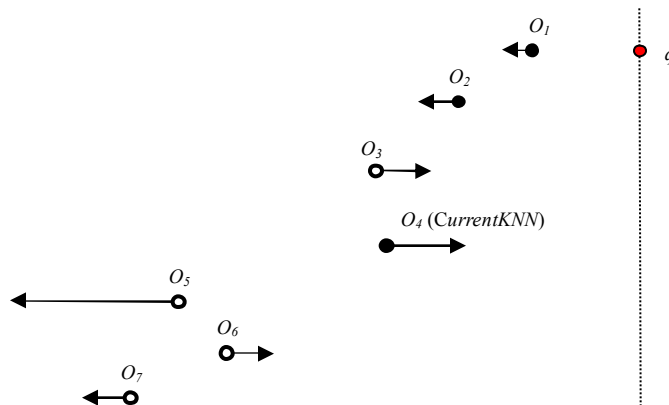
Figure 10: Algorithm  $PTI\_KNN(q_i, O_i, S_i, N_i)$  - Computing the  $PTI$ 's in KNNQ.

**Example** We now explain  $PTI\_KNN$  with an example. Suppose we have seven objects ( $O_1, O_2, \dots, O_7$ ), and we evaluate a KNN query with  $k = 3$ .  $PTI\_KNN$  is evaluated at time  $t_0$ . The following steps illustrate how  $PTI$  is computed for  $O_2$ , using Algorithm  $PTI\_KNN$ .

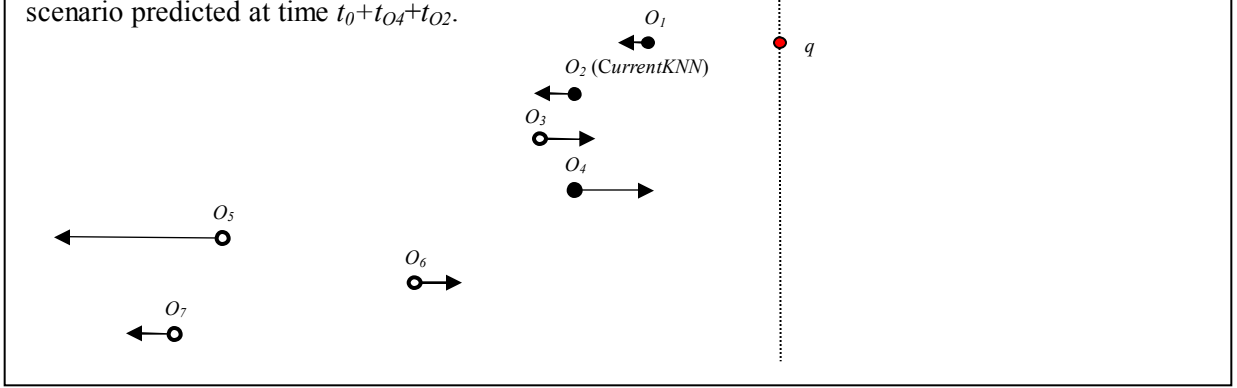
(a) Initially,  $O_3$  is  $CurrentKNN$  at time  $t_0$ .



(b) First loop: At time  $t_0$ ,  $PTI\_KNN$  evaluates that  $S_N = \{O_1, O_2\}$ ,  $S_F = \{O_4, O_5, O_6, O_7\}$ . Among the objects in  $S_N \cup \{O_3\}$ ,  $O_2$  will take over  $O_3$  as the next  $CurrentKNN$ , i.e.,  $Next\_Candidate_N = O_2$ . Among the objects in  $S_F \cup \{O_3\}$ , it is found that  $O_4$  will take over  $O_3$  as the next  $CurrentKNN$  i.e.,  $Next\_Candidate_F = O_4$ . Since  $T_F = t_{O_4} < T_N$ , we advance the time to  $t_0 + t_{O_4}$ , as shown below.



(c) Second loop:  $PTI\_KNN$  predicts that at time  $t+t_{O_4}$ ,  $CurrentKNN = O_4$ ,  $S_N = \{O_1, O_2\}$ , and  $S_F = \{O_3, O_5, O_6, O_7\}$ . Among the objects in  $S_N \cup \{O_4\}$ ,  $O_2$  will take over  $O_4$  as the next  $CurrentKNN$ , i.e.,  $Next\_Candidate_N = O_2$ . Among the objects in  $S_F \cup \{O_4\}$ , it is found that  $O_3$  will take over  $O_4$  as the next  $CurrentKNN$  i.e.,  $Next\_Candidate_F = O_3$ . Since  $T_N = t_{O_2} < T_F$ , the PTI of  $O_2$  can be computed as  $t_0 + t_{O_4} + t_{O_2}$ , and  $PTI\_KNN$  stops at this point. The following figure shows the scenario predicted at time  $t_0 + t_{O_4} + t_{O_2}$ .



**Complexity.** The running time of  $PTI\_KNN$  is  $O(n^2)$ , where  $n$  is the number of objects in a query. This is because in the worst case, the *do-while* loop iterates  $O(n)$  steps; each step checks all  $n$  objects to find the next candidate.<sup>3</sup>

#### 5.2.4 Computing $PTI$ Using a Generic Prediction Model

We have already presented how  $PTI$  is computed for different queries using a linear prediction model. These algorithms can, in fact, be extended to support a general prediction model, which is not necessarily linear. We use NNQ as an example, but the idea can be applied to other queries. Here we assume that the velocity function vector,  $v(s)$ , from a specific movement prediction model, is known. Specifically,  $v(s)$  predicts the velocity of the moving object at time  $s$ . Assuming the current time is  $T$ , Algorithm  $PTI\_NN\_next\_generic$  (Figure 11) is a modified version of the previously presented  $PTI\_NN\_next$  (Figure 8) for supporting a general prediction movement model. It uses the same framework as  $PTI\_NN\_next$  except that all the checking conditions incorporate the use of the instantaneous velocity vector  $v(s)$ , instead of a single distance vector value in the linear prediction model. In particular, the distance moved by object  $obj$  between the time  $[T, t+T]$ , namely  $\int_T^{T+t} v_{obj}(s) ds$ , is used extensively. For example, in Step

<sup>3</sup> We assume a randomized select algorithm is used to partition  $S_F$  and  $S_N$  with an average of  $O(n)$  time.

6(A)(I), the expression  $\min\left\{t \mid d_{NN} - \int_T^{T+t} v_{NN}(s) ds = 0\right\}$  is the minimum time that object *CurrentNN*

arrives at the query point. Other lines in *PTI\_NN\_next* are modified in the same manner.

```

1. Next_Candidate := NULL;
2.  $T_{min}$  := infinity;
3.  $v_{NN}$  := component vector of velocity of CurrentNN with direction towards  $q_i$ ;
4.  $d_{NN}$  := distance of CurrentNN from  $q_i$ ;
5. // PTI_NN Step 6(i): find the first object in  $S_i$  to take over CurrentNN as the new nearest neighbor
6. for each object obj except CurrentNN do {
  A. if exists  $t : d_{NN} - \int_T^{T+t} v_{NN}(s) ds = 0$  then {
    // the nearest neighbor is moving towards  $q_i$ 
    I.  $T_{NN} := \min\left\{t \mid d_{NN} - \int_T^{T+t} v_{NN}(s) ds = 0\right\}$ ; // time cost for CurrentNN to reach query point
    II. if  $\min\left\{t \mid d_{obj} - \int_T^{T+t} v_{obj}(s) ds = 0\right\} > T_{NN}$  AND exists  $t : d_{obj} - \int_T^{T+t} v_{obj}(s) ds < \int_{T+T_{NN}}^{T+t} v_{NN}(s) ds$  then
      // i.e. CurrentNN reach  $q_i$  first. Case 1, 3 or 4
      a.  $T_{obj} := \min\left\{t \mid d_{obj} - \int_T^{T+t} v_{obj}(s) ds < \int_{T+T_{NN}}^{T+t} v_{NN}(s) ds\right\}$ ;
      III. else if  $(\min\left\{t \mid d_{obj} - \int_T^{T+t} v_{obj}(s) ds = 0\right\} \leq T_{NN})$  then // Case 2
        a.  $T_{obj} := \min\left\{t \mid d_{obj} - \int_T^{T+t} v_{obj}(s) ds < d_{NN} - \int_T^{T+t} v_{NN}(s) ds\right\}$ ;
        IV. else continue; // Case 5. Check next object.
        V. if  $(T_{obj} < T_{min})$  then {
          a.  $T_{min} := T_{obj}$ ;
          b. Next_Candidate := obj;
        }
      }
    B. } else { // the nearest neighbor is moving away from  $q_i$ 
      I. if exists  $t : d_{obj} - \int_T^{T+t} v_{obj}(s) ds < d_{NN} - \int_T^{T+t} v_{NN}(s) ds$  then { // Case 6 or 7
        // obj does not move away or does not move away faster than CurrentNN
        a.  $T_{obj} := \min\left\{t \mid d_{obj} - \int_T^{T+t} v_{obj}(s) ds < d_{NN} - \int_T^{T+t} v_{NN}(s) ds\right\}$ 
        b. if  $(T_{obj} < T_{min})$  then {
          i.  $T_{min} := T_{obj}$ ;
          ii. Next_Candidate := obj;
        }
      }
      II. else continue; // do nothing for case 8
    }
  }
}

// PTI_NN Step 6(ii)
7.  $T := T + T_{min}$ ;

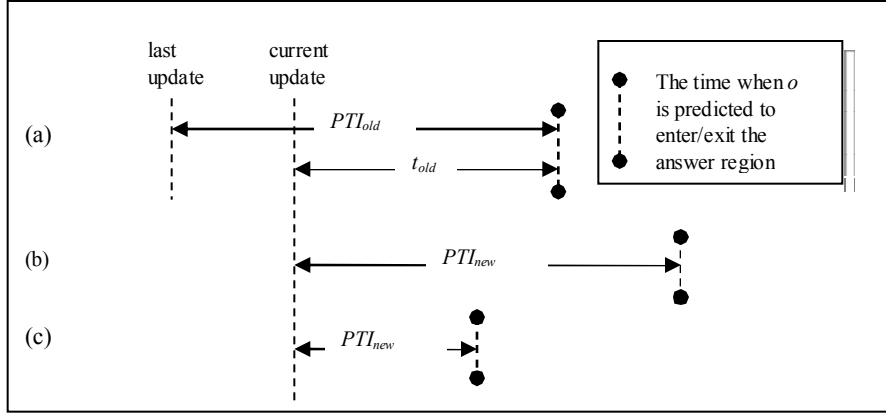
// PTI_NN Step 6(iii): update  $v_{obj}$  and  $d_{obj}$  of each object to time  $T+T_{min}$ 
8. if  $(Next\_Candidate \neq NULL)$  then {
  A. for each object  $obj \in S_i$  do // update distance
    I.  $d_{obj} := d_{obj} - \int_T^{T+T_{min}} v_{obj}(s) ds$ ;
}

```

Figure 11: Algorithm *PTI\_NN\_next\_generic* - Computing the *PTI* in NNQ using a generic prediction model.

### 5.3 Evaluation of $x$ -proportion

The value of  $x$ -proportion for an object is related to the amount of error in predicting the movement of the object. The initial value of  $x$ -proportion can be decided based on factors like temporal consistency constraints and the current network workload (for example, if the TC of a CQ is tight,  $x$ -proportion can be made smaller). When the prediction quality is poor,  $x$ -proportion is adjusted to achieve better prediction. QRU uses a *feedback approach* to perform this adjustment.



**Figure 12: Relationship between old  $PTI$ , new  $PTI$  and  $t_{old}$ .**

Specifically, let the  $PTI$  computed from the last update be  $PTI_{old}$ . Also, let  $t_{old}$  be the difference between  $PTI_{old}$  and the time elapsed between the last and the current update. Then  $t_{old}$  is the  $PTI$  evaluated at the time of the current update, but without using the current update information (Figure 12). In Figure 12 (a),  $O_i$  evaluates  $PTI_{old}$  after its last update. At the time the current update is completed,  $t_{old}$  is the predicted time of  $O_i$  entering/exiting the answer region of the CQ (using the information of  $PTI_{old}$  only). When the Refresh Phase is executed after the current update is completed, a new value of  $PTI$  (called  $PTI_{new}$ ) is actually computed – using the current location of  $O_i$ . In both Figure 12 (b) and (c), if  $PTI_{new}$  differs from  $t_{old}$ , the prediction is inaccurate. We can quantify the prediction error by  $PTI_{error}$ :

$$PTI_{error} = \frac{PTI_{new} - t_{old}}{t_{old}} \quad (11),$$

which is evaluated in Step (c) of the Refresh Phase (Figure 3). Observe that  $PTI_{error}$  becomes high when the movement of the object is very different from the predicted movement. Moreover,  $PTI_{error}$  is used in QRU to adjust the  $x$ -proportion. There are two cases to consider:

- (1)  $PTI_{error} > 0$ . This means the object is moving *slower* than expected (Figure 12 (b)). The time threshold may be slightly increased without reducing fidelity. By increasing the time threshold, the bandwidth utilization is also reduced. QRU increases the time threshold with the following rule: when  $PTI_{error}$  is larger than  $\gamma$ , a larger  $x$ -proportion,  $\min(x+\delta, 1)$ , is set, where  $\gamma$  and  $\delta$  are system parameters.
- (2)  $PTI_{error} \leq 0$ . This indicates the object is moving *faster* than predicted (Figure 12 (c)). Hence an object that moves across the answer region may not be noted by the system, leading to reduction in fidelity. To avoid this, the time threshold may be reduced. QRU reduces the time threshold with this rule: when  $PTI_{error}$  is less than  $-\gamma$ ,  $x$ -proportion is set to  $\max(x-\delta, 0)$ .

Hence, the value of the  $x$ -proportion is continuously tuned with the arrival of a new update.

#### 5.4 Reducing Control Message Load

In adaptive time-based schemes such as QRU, *control messages* may be sent from the system to the objects for coordination purposes (e.g., adjusting time thresholds). The network load of *control messages* can be high. To reduce control message workload, QRU sends time thresholds to objects only when the new time thresholds evaluated in Step (d) of the Refresh Phase (Figure 3) deviate from the old ones by more than a pre-defined value. The object keeps using the current threshold until it receives a new one from the system.

Usually, the bandwidth channel for sending control messages (called *downlink channel*) has much more bandwidth than the *uplink channel* (for sending information from objects to the system). Also, our experiments in the next section show that for many objects, there is no need to refresh the time thresholds: when they are far from entering/exiting the answer region, they are often assigned the maximum time threshold; when they are close to entering/exiting the answer region, they are often given the minimum time threshold. Thus, this small change is effective in reducing control message load.

We conclude this section by presenting a summary of QRU. Table 2 describes the parameters used in QRU. Table 3 presents the computation methods of  $PTI$  and  $x$ -proportion for each query class.

Parameter	Meaning	Comments
$L_{time}$	Minimum time threshold	Limits the update workload
$PTI_{MIN}, PTI_{MAX}$	Lower and upper bounds of $PTI$	The default value of $PTI_{MIN}$ is $L_{time}$
$\Gamma$	Error threshold of $PTI_{error}$	Controls when $x-proportion$ is changed by $\pm \delta$
$\delta$	Adjust $x-proportion$ by $\delta$ when $ PTI_{error}  > \gamma$	Magnitude affects sensitivity to $PTI_{error}$ and bandwidth utilization

Table 2: Parameters of QRU.

	Non-rank-based query	Rank-based query
Examples	RCQ, RQ	NNQ, KNNQ
Computing $PTI$	The answer region is fixed. Can estimate when $O$ will enter the answer region independently of other objects. For instance, <b><math>PTI_{Range}</math></b> predicts how long it takes for $O$ to reach the range boundary. <b>Complexity:</b> $O(m)$ where $m$ is the number of sides of the query range	The answer region is dynamic. Need to consider interactions between $O$ and other objects. For example, <b><math>PTI_{NN}</math></b> predicts how long it takes for $O$ to take over the nearest neighbor. <b>Complexity:</b> $O(n^2)$ (for both $PTI_{NN}$ and $PTI_{NN\_Batch}$ , where $n$ is the number of relevant objects of a query.)
Adjusting $x-proportion$	When $ PTI_{error}  > \gamma$ , $x-proportion$ is adjusted by $\pm \delta$ . <b>Complexity:</b> $O(1)$	

Table 3: Setting  $PTI$  and  $x-proportion$  of object  $O$  in QRU.

## 6 Performance Evaluation

We have conducted experimental evaluations to study the effectiveness of QRU. Section 6.1 presents our simulation model. In Section 6.2 we report the performance results of QRU compared with other location update schemes using synthetic data. Section 6.3 presents the results using a real dataset.

### 6.1 Simulation Model

We have conducted an experimental evaluation to compare the performance of QRU with three push-based update mechanisms: the time-based method (TB), the distance-based method (DB), and the speed dead-reckoning method (SDR).<sup>4</sup> We choose TB and DB for comparison because: (1) they represent two distinct classes of update schemes (using time and distance); (2) their simplicity facilitates better understanding of QRU’s properties; and (3) QRU is an adaptive time-based scheme while TB and DB are fixed threshold methods. A comparison with these methods allows us to illustrate the benefits of using an adaptive threshold assignment method. We remark that SDR is an extension of DB.

Initial experimental results revealed that the number of updates generated by any of these methods is much larger than those required by QRU. This is hardly surprising, as QRU only generate updates for

<sup>4</sup> Section 2 provides description of these update schemes.

objects that are needed by CQs, while in other methods *all objects* generate updates. To obtain more interesting results, TB, DB and SDR are modified to *MTB*, *MDB* and *MSDR* correspondingly (the word “M” stands for “Modified”). In these new protocols, an object only generates updates if it is required by a CQ. Control messages are employed to obtain the initial locations of the objects being accessed by CQs. Control messages are also used to inform objects about their update thresholds during initialization.

We use the model in Figure 2, where clients submit CQs to the system. We simulate two queries: RCQ (a non-rank-based value query) and NNQ (a rank-based entity query). The monitoring set for each CQ is generated randomly, i.e., all the objects are selected with the same probability. For a RCQ, the distances of objects from the query range, either inside or outside, are also generated randomly. For a NNQ, we randomly generate the nearest neighbor to the query point and the relative distances of the objects from the query point. We use *PTI\_NN* (Figure 6) to computing PTI for NNQ.

Our first set of experiments use synthetically generated movement data. Instead of simulating the behavior of moving objects in the service area, we model their movement relative to the answer regions of the queries. At each time unit, we update the distance traveled by the objects relative to the answer region. The distance traveled is computed as the product of two random numbers: *speed* and *direction*. The *speed* is used to simulate the distance traveled by the object in a time unit. It is randomly generated using a uniform function within a range of speeds. The *direction* is also produced by a uniform number generator with the range of  $-1$  to  $+1$ . Its purpose is to simulate changes in the direction of movement of an object, relative to the answer region. A negative value means the object is moving away from the answer region. The *speed* of an object is generated every time unit while the *direction* is computed once every random period. This simulates the non-linear movement of an object towards a destination, where its speed can be changed slightly (by varying the speed factor slightly during each time unit), and a change of destination (by changing the movement direction after a relatively large time period). Notice that although we are using a one-dimensional movement model, we can view it as a projection of a two-dimensional movement onto the  $x$ -axis or the  $y$ -axis.

After updating the movement of the objects, the queries are evaluated to obtain the true results (i.e., results that are obtained using the true location data). We present the fidelity of the system ( $FS$ ) in Eqn. (7) under different testing conditions. We also measure the *system accuracy*, which is the average accuracy of completed queries defined in Eqn. (8).

Table 4 summaries the model parameters and their baseline values. Since the purpose of the simulation is to investigate the general properties of QRU rather than its performance in a particular mobile application, we investigate the impact of changing the values of many of the parameters ( $x$ -proportion, PTI\_MIN, PTI\_MAX,  $X_{init}$  and communication delay between the system and moving objects) on QRU. This allows us to see how QRU performs under different environmental and workload characteristics. Values of other parameters, such as the number of objects and their speed, are fixed. They reflect realistic values as they can be found in real systems or were used in previous studies [GL04]. The simulation time for each run is 800,000 sec. The length of the simulation is determined by some trial runs using different simulation lengths until stable results are obtained, i.e., the results are within 0.5% accurate compared to longer simulation runs.

<b>System Parameters</b>	
<b>Name</b>	<b>Baseline Value</b>
Number of moving objects	10,000
Number of relevant objects in a CQ	80 to 100 (uniformly distributed)
Query length (end time – begin time)	3000 sec to 5000 sec (uniformly distributed)
Speed factor	5 to 10 m/sec (uniformly distributed)
Direction factor ( $w$ )	-1 to 1 (uniformly distributed)
Time period for changing direction factor	600 sec to 900 sec (uniformly distributed)
Query types	RCQ and NNQ
Temporal Consistency Constraint (TC)	RCQ: $\epsilon_i = 0.01$ ; NNQ: $\epsilon_i = 1$
Communication delay between object and system	0
<b>Parameters for QRU</b>	
<b>Name</b>	<b>Baseline Value</b>
Smallest time threshold ( $L_{time}$ )	50 sec
PTI_MIN	50 sec
PTI_MAX	500 sec
Adjustment factor ( $\delta$ ) of $x$ -proportion	0.1
Error threshold ( $\gamma$ ) of $x$ -proportion	0.1
Initial value for $x$ -proportion ( $X_{init}$ )	0.5

Table 4: Parameters and the baseline values.

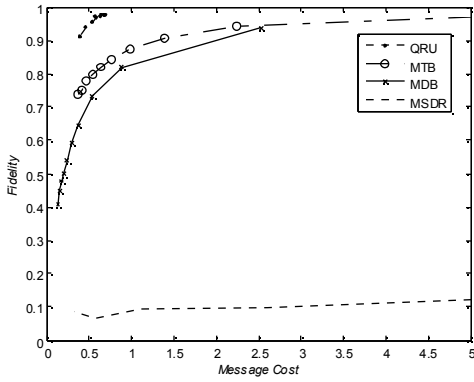


Figure 13: RCQ – Fidelity vs. Messages

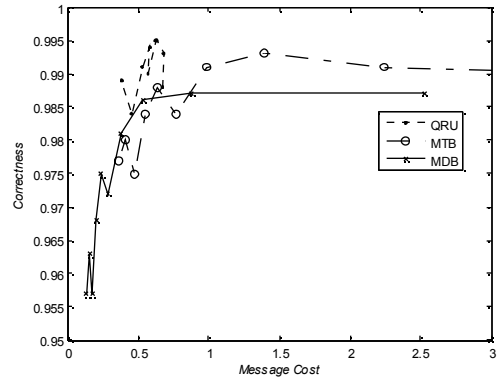


Figure 14: RCQ – System Accuracy vs. Messages

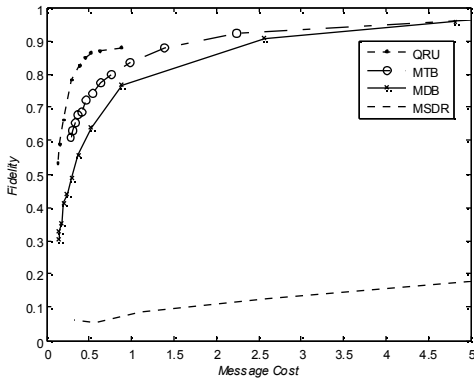


Figure 15: NNQ – Fidelity vs. Messages

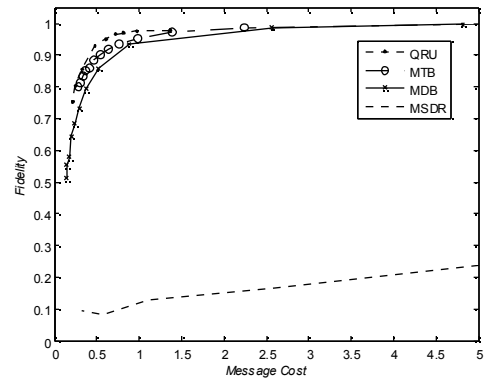


Figure 16: NNQ – Fidelity vs. Messages

## 6.2 Performance Results for Synthetic Data

Three sets of experiments have been performed. First, we compare the performance of QRU with MDB, MSDR and MTB. Then we test QRU for RCQ and NNQ. Since MTB outperforms MDB and MSDR by achieving a higher fidelity with the same update cost, the third experiment compares QRU and MTB with different parameter values ( $PTI\_MIN$ ,  $PTI\_MAX$ ,  $X_{ini}$  and  $TC$ ).

### 6.2.1 QRU vs. MDB, MSDR and MTB

Figure 13 shows the fidelity (FS) as a function of message cost (number of update messages per object for MTB, MDB and MSDR, and total number of update and control messages per object for QRU). For MTB, MDB and MSDR, we vary their update thresholds in order to obtain different update costs. For QRU, we vary the  $x$ -proportion from 0.1 to 0.9 to vary message costs. As shown in Figure 13, for RCQ, QRU provides high fidelity with little message overhead. The other mechanisms, in contrast, achieve high fidelity only at higher update costs (MTB, MDB), or almost not (MSDR). For a given message cost, MTB has slightly higher fidelity than MDB. While MTB is a synchronized update scheme, both MDB and

MSDR are not. For MTB, all objects required by a query generate location updates at the same time. This makes the database state consistent with the actual state at the update time. The performance of MSDR is relatively poor, because the system needs to predict the speed and movement, and, subsequently the distance, of each moving object. In our simulation, the speed and direction of moving objects change quite frequently, and so the prediction can be inaccurate which also leads to poor fidelity. However, if the change of the movement is less frequent, MSDR should have a better performance.

Figure 14 shows that the system accuracy of the three methods (QRU, MTB and MDB) is similar and is maintained at a high value. Increasing the message cost slightly improves the system accuracy. The curve of MSDR is not shown since it is very low (close to 0.1) in this experiment. Some curves in Figure 14 oscillate compared with Figure 13 (which measures fidelity). This is due to the fact that system accuracy is a function of  $E_i(t)$ , which can be any value between 0 and 1 (Equation 8), while fidelity is a function of  $F_i(t)$  (Equation 6), having either a value of 0 or 1. Therefore, the value of fidelity has less variation. Correspondingly, we can see Figure 13 yields smoother curves than Figure 14.

Figure 15 shows the results for NNQ. Again, QRU obtains higher fidelity with lower message costs than both MTB and MDB. Similar to the results for RCQ, MSDR has the worst performance. Comparing Figure 13 and Figure 15, higher fidelity is obtained with the same message cost for RCQ than NNQ. As explained in Sections 3.4 and 3.2.2, the evaluation of NNQ is more sensitive to errors, and it is more difficult to have an accurate prediction for NNQ than for RCQ. This also explains why the effect of  $x$ -proportion to QRU in RCQ is smaller than in NNQ as shown in Figure 17. One way to improve the fidelity for QRU is to use a smaller PTI\_MIN and PTI\_MAX i.e., a higher update frequency. Figure 16 shows the results when PTI\_MIN is 30, PTI\_MAX is 300. We can see that QRU attains a higher fidelity with a slight rise in message cost. In the same figure, the TC of NNQ (Eqn. (5)) is relaxed from 1 to 3 in order to further improve the fidelity. This relaxation not just improves the performance of QRU but of all methods (i.e., MTB, MDB and MSDR). We will use these parameter values (PTI\_MIN=30, PTI\_MAX=300, TC=3) in our subsequent experiments for NNQ.

The better performance of QRU is due to the adaptability of the time threshold assignment. The threshold value is generated based on the object movement and the answer regions of the CQs. If an object is unlikely to enter the answer region, its  $PTI$  and time threshold will be larger. On the contrary, a smaller  $PTI$  is assigned to an object entering the query region soon. The update is *effective* even though its  $PTI$  is small. By utilizing the query information, QRU can better decide which objects should be given more network bandwidth for propagating their updates. Figure 17 shows that a high percentage of objects uses  $PTI\_MAX$ . The figure shows the percentage of objects assigned  $PTI\_MAX$  for increasing  $x$ -proportion values ( $X_{init}$ ). For RCQ, around 90% of all objects have  $PTI\_MAX$  assigned, while for NNQ the fraction of objects using  $PTI\_MAX$  is between 0.72 and 0.85.

### 6.2.2 RCQ and NNQ

By comparing the fraction of objects using  $PTI\_MAX$  for RCQ and NNQ (Figure 17), we observe that the fraction is higher for RCQ and is less sensitive to  $x$ -proportion. The higher value for RCQ is due to fewer variable factors in the prediction. For example, the answer region of RCQ is fixed while that of NNQ is dynamic. Interestingly, the  $PTI\_MAX$  percentage of NNQ soars when  $X_{init}$  is small (0.1) and large (0.9). A small  $X_{init}$  implies close monitoring, and QRU discovers that some objects do not need close monitoring. When  $X_{init}$  is large, QRU does little monitoring and uses large  $PTI$  values for many objects.

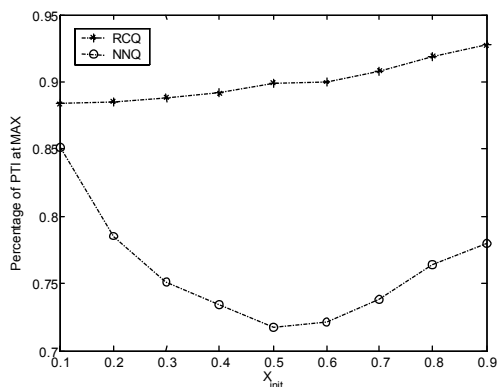


Figure 17:  $PTI\_MAX$  percentage vs.  $x$ -proportion.

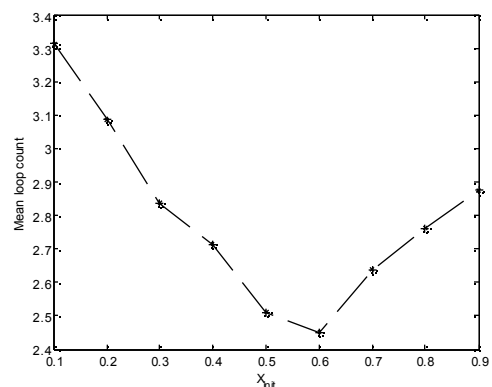


Figure 18: Average number of loops in NNQ.

The good performance with QRU may come at the price of high costs for evaluating  $PTI$  values. Section 5.2.1 explains that computation is not costly for RCQ (a complexity of  $O(1)$  since  $m = 1$  in our

experiments). The complexity of NNQ is theoretically higher ( $O(n^2)$ ). In practice, *PTI* computation cost in NNQ depends on the movement and the distribution of the objects. Figure 18 shows that this cost is low. The average number of loops (Figure 6 Step 6 of Algorithm *PTI\_NN*) is only between 2.4 and 3.3 for different *x-proportion* values.

We also studied the effect of different values for the time period after which an object changes its direction. We found that the longer the period, the better are the results for system fidelity and accuracy. This is not surprising, since the prediction of the movement becomes more accurate.

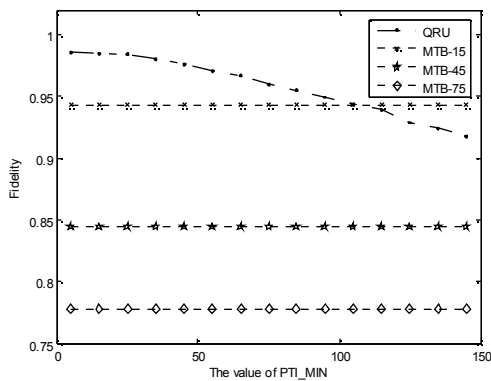


Figure 19: RCQ - PTI\_MIN vs. Fidelity

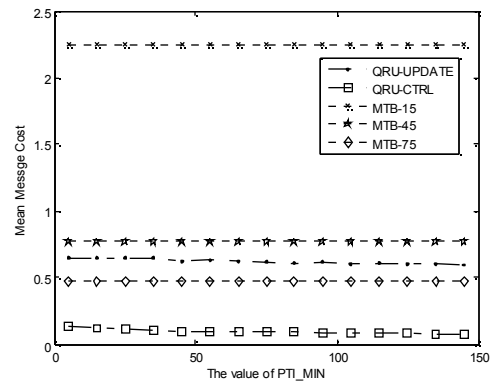


Figure 20: RCQ: PTI\_MIN vs. Message Cost

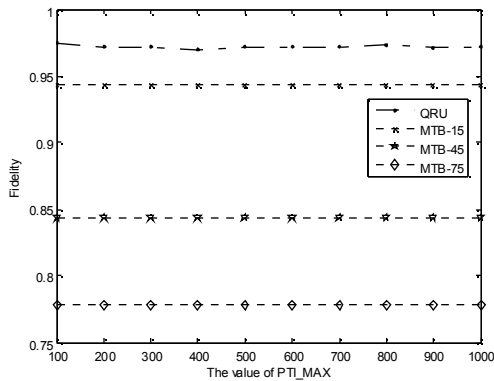


Figure 21: RCQ: PTI\_MAX vs. Fidelity

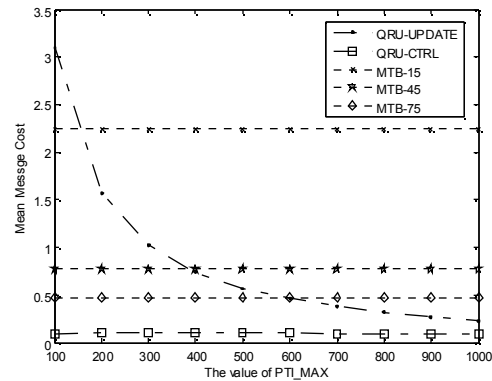


Figure 22: RCQ - PTI\_MAX vs. Message Cost

### 6.2.3 QRU vs. MTB

Since MTB shows better results than MDB and MSDR, we focus on QRU and MTB. Here we use the notation “MTB- $y$ ” to represent MTB using a  $y$ -second time threshold. We also use “QRU-UPDATE” and “QRU-CTRL” to denote the number of client update messages and system control messages respectively.

Figure 19 shows that fidelity decreases gradually with an increase in PTI\_MIN for RCQ, while the message cost is not affected much by changes in PTI\_MIN (Figure 20). The fidelity for QRU is still significantly better than that of MTB, except when MTB uses a small time update threshold (15 sec) – but its update cost rises (Figure 20). The better performance of MTB with small update threshold, however, comes at the price of high message costs (Figure 20). Figure 21 and Figure 22 show the results of varying PTI\_MAX. As shown in Figure 21, increasing the value of PTI\_MAX does not affect the fidelity but the message cost drops with larger PTI\_MAX values (Figure 22). The fidelity of QRU is significantly higher than that of MTB for different PTI\_MAX even when MTB uses a small update time threshold. The reader is reminded that, however, PTI\_MAX allows a disconnected object to be detected. Hence, its value should not be too large.

Figure 23 to Figure 26 show the effect of PTI\_MIN and PTI\_MAX for NNQ. We see that fidelity is much more sensitive to PTI\_MAX for NNQ than RCQ. Again, this is due to the presence of more variable factors in NNQ and the difficulty in prediction. For different PTI\_MIN and PTI\_MAX values, QRU is in general much better than MTB. To conclude, for RCQ, we can use a large PTI\_MAX and a small PTI\_MIN in order to have low message costs and high fidelity. For NNQ, smaller values should be used since it is more sensitive towards changes in PTI\_MIN and PTI\_MAX.

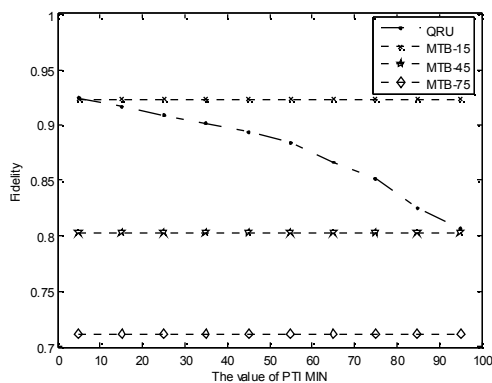


Figure 23: NNQ - PTI\_MIN vs. Fidelity

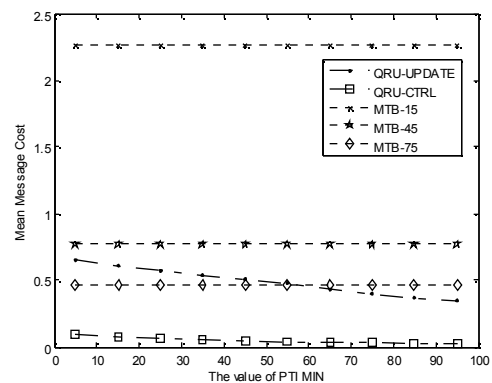


Figure 24: NNQ- PTI\_MIN vs. Message Cost

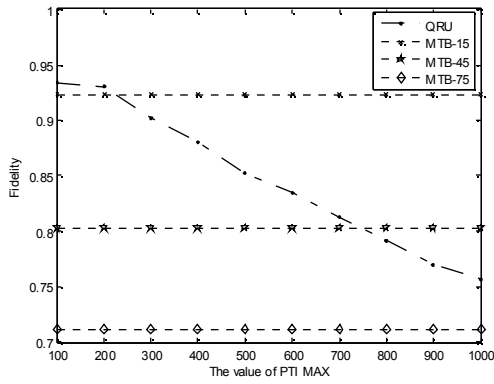


Figure 25: NNQ- PTI\_MAX vs. Fidelity

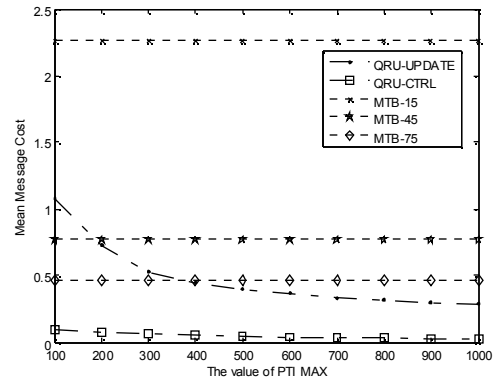


Figure 26: NNQ - PTI\_MAX vs. Message Cost

Figure 27 and Figure 28 show the effect of changing the communication delay between the objects and the database on QRU and MTB (for RCQ). Increasing the delay decreases the fidelity percentage of the system (FS) due to delayed installments of updates (Figure 27). Consistent with the results in Figure 19 to Figure 22, the fidelity of QRU in RCQ is better than that of MTBs with a lower message cost (Figure 27 and Figure 28). Similar results have been obtained for NNQ.

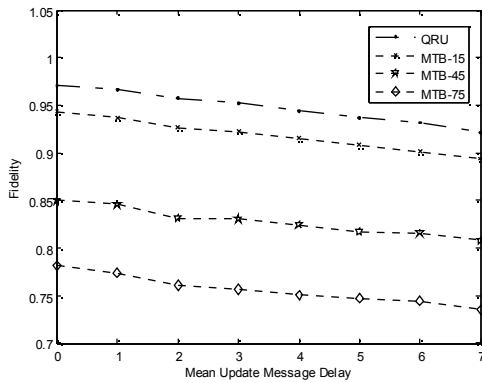


Figure 27: RCQ: Delay vs. Fidelity

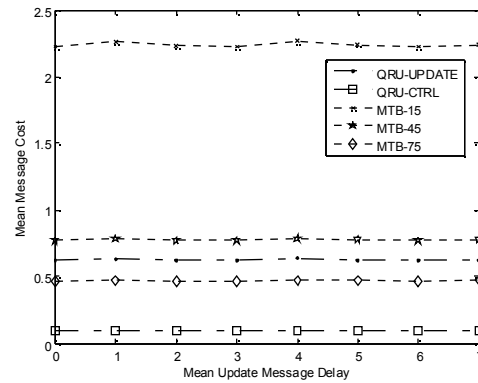


Figure 28: RCQ: Delay vs. Message Cost

The final experiment studies the temporal consistency constraint (TC) of queries. Since RCQ performs well even under a tight temporal consistency constraint, we only present the results for NNQ. In Figure 29, the fidelity of QRU rises with an increase in the TC value, due to a looser correctness constraint. The fidelity provided by *x-proportion* of 0.3 and 0.5 is similar. Although the fidelity of MTB is high if the TC is larger than 3, the advantage is paid by a higher update cost (Figure 30).

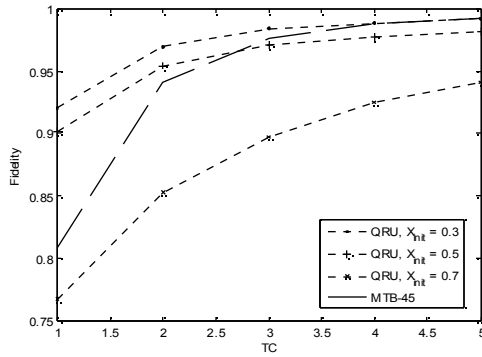


Figure 29: Fidelity vs. Temporal Consistency Constraint

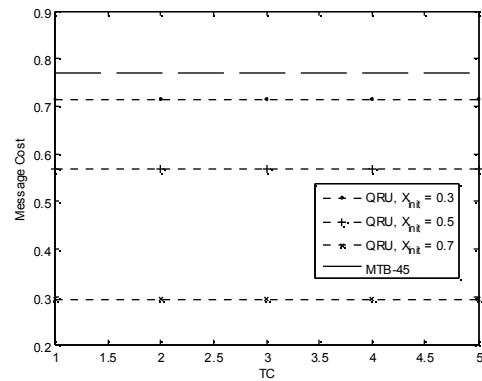


Figure 30: Message Cost vs. Temporal Consistency Constraint

### 6.3 Performance Results for a Real Dataset

In this section, we present results using a real dataset [P03]. The dataset contains the spatial-temporal information of school buses in the municipal area of Athens, Greece. The baseline value for the query length is 3600 seconds and we assume that multiple queries are being executed concurrently, at an arrival rate of 0.002 per second. The values of other parameters are the same as those stated in Section 6.1. Each data point is obtained by averaging over 100 trials. We also compare QRU with a technique that guarantees perfect fidelity, known as the “Safe Region” method [PX02].

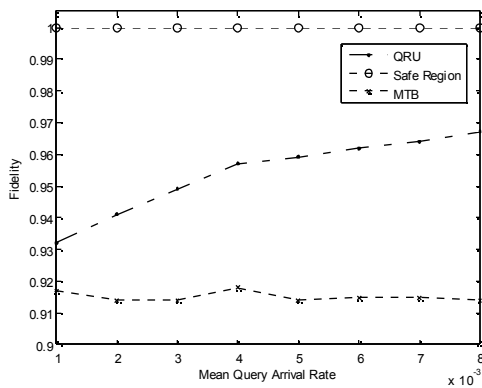


Figure 31: Fidelity vs. Query Arrival Rate

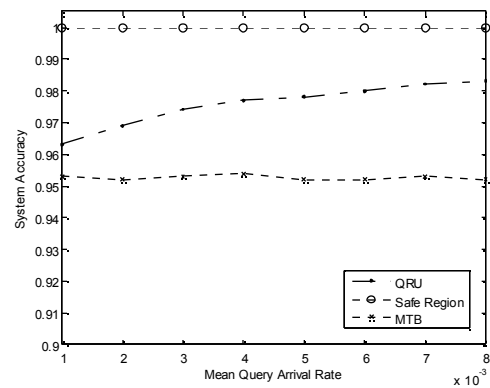


Figure 32: System Accuracy vs. Query Arrival Rate

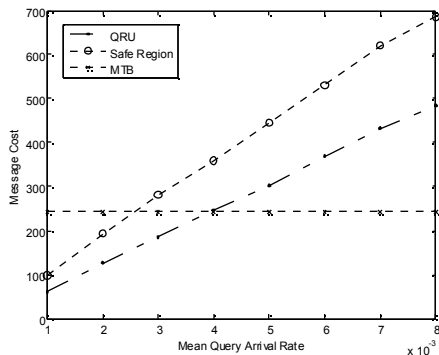


Figure 33: Message Cost vs. Query Arrival Rate

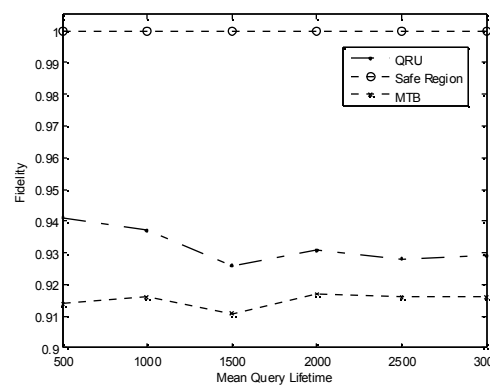


Figure 34: Fidelity vs. Mean Query Lifetime

**Multiple Concurrent Queries** We examine the effect of executing multiple continuous queries (RCQ) in the system. Figure 31 and Figure 32 show the fidelity and the system accuracy respectively. Under a wide range of mean query arrival rates, QRU achieves higher fidelity and system accuracy than MTB. In fact, both values increase with the query arrival rate. With more concurrent queries, there is a higher chance that the update thresholds are smaller, since it is more likely that an object is close to any of the query boundaries. Thus the database is kept fresh, and a higher fidelity and system accuracy can be achieved. On the other hand, this means a rise in the message cost. As shown in Figure 33, the message cost for QRU increases with the query arrival rate, while MTB is unaffected. For MTB, the time threshold is set independent of queries, and thus the message cost is unaffected by the increase in the number of concurrent queries.

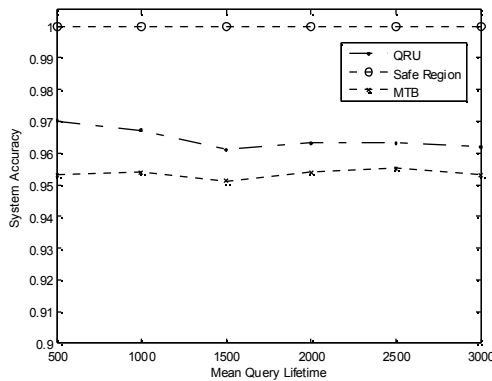


Figure 35: System Accuracy vs. Mean Query Lifetime

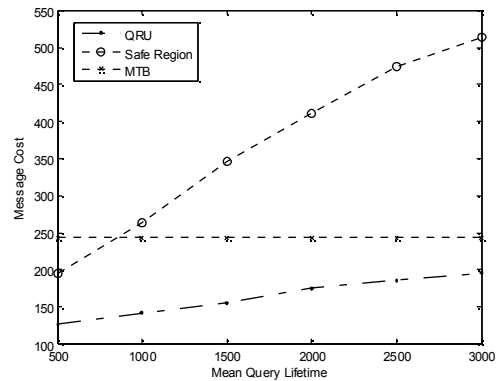


Figure 36: Message Cost vs. Mean Query Lifetime

**Safe Region** Next, we compare our approach with the “safe region” method proposed in [PX02]. As discussed in Section 2, a safe region guarantees that no update is necessary if an object does not travel across the boundary of the region. Here we implement the SafeSphere algorithm [PX02], which represents a safe region in a circular form. Since the safe region information is conveyed to the object, the object knows exactly when to send the update, and thus fidelity and system accuracy are always equal to 1, as shown in Figure 31 and Figure 32. The message cost of SafeSphere, as illustrated in Figure 33, increases with the number of concurrent queries. This can be explained by two reasons. First, the existence of more concurrent queries implies that a safe region is smaller, and therefore it is more likely that an object’s path cuts the safe region boundary; an update is necessitated in order to maintain perfect

fidelity and system accuracy. Secondly, when a new query is admitted to the system, the safe region has to be recomputed and sent to every object. In QRU, control messages are sent to the objects only if their new time thresholds deviate from the old ones significantly (Section 5.4). Therefore, although QRU has less fidelity and system accuracy than SafeSphere, it attains a lower message cost than SafeSphere when more concurrent queries are being executed.

**Query Activation Period** The final experiment examines the effect of query activation period (i.e., the time during which the CQ is being evaluated) on the performance of QRU, MTB and SafeSphere. As shown in Figure 34 and Figure 35, QRU achieves higher fidelity and system accuracy than MTB throughout the experiment, while SafeSphere attains a value of 1 for both fidelity and system accuracy. In terms of message costs, however, both the performance of QRU and SafeSphere degrades as the query activation period is increased (Figure 36). When a query has a longer activation period, it has a higher chance of running concurrently with other CQs. This renders more frequent exchange of messages for both QRU and SafeSphere. Similar to the situation in Figure 33, the message costs increase for both schemes. MTB is independent of the activation period, and its message cost is unaffected. Nevertheless, under the range of activation periods studied, QRU needs lower message costs than SafeSphere and MTB.

We remark that although SafeSphere provides better fidelity and system accuracy than QRU, it also requires comparatively higher message costs. With QRU, the fidelity and system accuracy is higher than 0.92 in most cases, which is achieved with a significant reduction of message costs. Moreover, while SafeSphere assumes the network is perfect, QRU can detect network disconnection.

## 7 Conclusions

We addressed the issues of maintaining temporal consistency of data for CQs in a moving-object environment. We used fidelity to measure the answer correctness of each class of CQ. We also proposed the QRU scheme, which is an adaptive time-based update generation scheme. In QRU, the time threshold assigned to an object is based on (1) the object's predicted movement, (2) the query class (rank-based or non-rank-based), and (3) prediction accuracy. We performed extensive experiments to study the

effectiveness of QRU compared with MTB, MDB and MS DR. In general QRU achieves a higher fidelity with lower update costs than these methods. As part of our future work, we plan to study how QRU can be adapted to “moving queries” i.e., the issuer of the query can move.

## References

- [CKP03] R. Cheng, D. Kalashnikov and S. Prabhakar, “Evaluating probabilistic queries over imprecise data”, in Proc. of ACM SIGMOD 2003 Intl. Conf. on Management of Data, pp. 551-562, 2003.
- [CHC04] Ying Cai, Kien A. Hua, Guohong Cao. "Processing Range-Monitoring Queries on Heterogeneous Mobile Objects," IEEE International Conference on Mobile Data Management 2004 (MDM'04), p. 27, 2004.
- [DC02] D. Carney, U. Centiemel, M. Cherniak, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – a new class of data management applications. In VLDB, 2002.
- [DKPR01] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, P. Shenoy “Adaptive Push-Pull: Disseminating Dynamic Web Data”, in Proc. of the 10th WWW Conference, Hong Kong, 2001.
- [GL04] B. Gedik and Ling Liu, “MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System”, in *Proc. 9<sup>th</sup> Intl. Conf. on Extending Database Technology*, Crete, Greece, March 2004.
- [HGP03] V. Hristidis, L. Gravano and Y. Papakonstantinou, “Efficient IR-style keyword search over relational databases”, in Proc. of VLDB, 2003.
- [KGT99] G. Kollios, D. Gunopulos and V. Tsotras, “Nearest Neighbor Queries in a Mobile Environment”, in Proc. of 1999 Intl. Conf. on Scientific & Statistical Database, Sep. 1999.
- [KLA03] B. Kao, K.Y. Lam, B. Adelberg, R. Cheng and T. Lee, “Maintaining Temporal Consistency of Discrete Objects in Soft Real-Time Database Systems”, IEEE Transactions on Computers, 52(3), 2003.
- [LH99] Ben Liang, Zygmunt J. Haas, “Predictive Distance-based Location management for PCS Networks”, in Proc. of IEEE INFOCOM'99, New York, NY, March 21-25, 1999.
- [LR01] A. Leonhardi and Kurt Rothermel, “A Comparison of Protocol for Updating Location Information”, in Clustering Computing, vol. 4, no. 4, pp. 351-367, 2001.
- [MPBT05] K. Mouratidis, D. Papadias, S. Bakiras and Y. Tao, “A Threshold-Based Algorithm for Continuous Monitoring of  $k$  Nearest Neighbors”, vol. 17, no. 11, November 2005.
- [OJW03] C. Olston, J. Jiang, and J. Widom, “Adaptive filters for continuous queries over distributed data streams”, in Proc. of ACM Intl. Conf. on Management of Data (SIGMOD 2003), June 2003.
- [P03] Pfoser et al., “School bus tracking data”, in ChoroChronos.com, Data and Knowledge Engineering Group, Research Unit 3, Research Academic Computer Technology Institute, 2003. URL: <http://www.chorochronos.com>
- [PS01] Pitoura, E. and Samaras, G., “Locating Objects in Mobile Computing”, IEEE Transactions on Knowledge and Data Engineering, vol. 13, no. 4, pp. 571-592, 2001.
- [PX02] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch, “Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects”, IEEE Transactions on Computers, Vol. 51, No. 10, October 2002, pp. 1124-1140.
- [R93] K. Ramamritham, “Real-time Databases”, Journal of Distributed and Parallel Databases, 1(2), 1993.
- [SWCD97] A. Sistla, O. Wolfson, S. Chamberlain, S. Dao, “Modeling and querying moving objects”, Proc. of the 13th Intl. Conf. on Data Engineering, pages 422-432, Birmingham, UK, April 1997.
- [WA02] W. Wang and I.F. Akyildiz, “On the Estimation of User Mobility Pattern for Location Tracking in Wireless Network”, in Proceedings of 2001 INFOCOM, Anchorage, April 2001.
- [WSCY99] O. Wolfson, P. Sistla, S. Chamberlain, Y. Yesha, “Updating and Querying Databases that Track Mobile Units”, Distributed and Parallel Databases, vol. 7, no. 3, pp. 257-287, 1999.
- [WCDJ97] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang “Location Management in Moving Objects Databases”, in Proceedings of WOSBIS'97, pages 7-13, Budapest, Hungary, October 1997.
- [WL01] V. Wong, V. Leung, “An Adaptive Distance-Based Location Update Algorithm for Next Generation PCS Networks”, IEEE Journal on Selected Areas in Communications, 19(10), 2001.