

MobiQuery: A Spatiotemporal Data Service for Sensor Networks

Chenyang Lu, Guoliang Xing, Octav Chipara, Chien-Liang Fok
Department of Computer Science and Engineering
Washington University in Saint Louis
Saint Louis, MO 63130
{lu, xing, ochipara, liang}@cse.wustl.edu

Abstract

Spatiotemporal query is a new type of data service that allows a user to periodically gather information from a geographic area that moves with the user. A defining feature of this new service model is that each query result is subject to a set of spatiotemporal constraints in terms of response time, data freshness, and changing locations of data sources due to user mobility. We have developed MobiQuery to support spatiotemporal query services in a wireless sensor network environment. MobiQuery can work with in-network aggregation, energy conservation, and motion prediction techniques. A novel just-in-time prefetching mechanism is employed to provide spatiotemporal performance guarantees despite extremely low node duty cycles and unpredictable communication delays in sensor networks, while still achieving significant energy conservation. We also provide theoretical analysis and ns-2 simulation results which demonstrate the robustness and efficiency of MobiQuery under a range of realistic settings.

1. Introduction

As large-scale wireless sensor networks make it possible to monitor physical environments at unprecedented spatial and temporal granularities, a key research challenge is to develop data services that deliver information to end users. Several promising data services for sensor networks have been proposed in the literature. For example, COUGAR [1], Directed Diffusion [2], and TinyDB [3] allow a user to query (or subscribe to) interesting events in a geographic area. More recently, SEAD [4] and TTDD [5] were developed to improve query performance when users are mobile. In this paper, we propose a new data service for sensor networks called *spatiotemporal query*. In contrast to existing data services that generally assume fixed areas of interest, spatiotemporal query is motivated by a fundamental need of a mobile user to continuously gather information within his vicinity, *i.e.*, a moving query area specified relative to the user's current location.

For example, a firefighter fighting a wild fire may request a periodic update of a temperature map within one mile around his location to maintain awareness of the fire condition. As the firefighter moves, the query area and the sensors should respond to the query change dynamically.

A defining feature of a spatiotemporal query is that each query result is subject to a set of spatiotemporal constraints associated with user mobility, *i.e.*, data needs to be served at the right time and also at the right location.

- *Spatial constraints:* The evolving query area constrains the valid data sources contributing to the query result. Specifically, only the nodes inside the current query area should contribute to the current query result. At the same time, it is desirable to aggregate data from as many nodes as possible to improve the integrity of the query result.
- *Temporal constraints:* At the same time, a spatiotemporal query is also subject to temporal constraints in terms of *delivery rate* and *data freshness*. A new query result must be delivered to the user before the end of each querying period. Furthermore, each query result must be aggregated from *fresh* sensor data, where the freshness constraint of a sensor datum is defined by its maximum validity interval after it is read from the sensor.

Meeting all the spatiotemporal constraints is crucial for many critical sensor network applications that depend on real-time context awareness. In the earlier example, a firefighter may be endangered by a quickly moving fire if the query results are aggregated from old sensor readings, are delivered too late or are aggregated from sources at wrong locations, or if too few nodes in the current query area contribute to the results.

At the same time, spatiotemporal query services must facilitate energy conservation, a paramount concern in many sensor network applications. For example, a structural health monitoring network on a bridge may require years of lifetime on limited power supplies. Recent research has found that significant energy savings can be achieved by controlling node duty cycles. In this approach, most nodes are scheduled to sleep (*i.e.*, turn off their radio interfaces) periodically, while a small number of nodes remain active to maintain continuous service.

Periodic sleep schedules have been supported by many existing energy conservation protocols [6, 7, 8, 9, 10].

Compared to traditional ad hoc networks, sensor networks often require much lower duty cycles due to their significant longer lifetime requirement [11]. For example, a MICA2 mote [12] can last about 3 days when it is continuously on. For a mote to remain operational for 5 years, the average duty cycle needs to be 0.17%. Thus, for every active time window of 25 milliseconds, the node needs to sleep for 15 seconds. Since a sleeping node can only communicate in its active windows, in this example the communication delay to a sleeping node may range from a few milliseconds to more than 15 seconds. Extremely low duty cycle introduces a key challenge to meet the constraints of spatiotemporal queries. In particular, the sleeping nodes located in a query area need to contribute to the result in time despite the long and unpredictable communication delays.

In this paper, we present a novel spatiotemporal query service called *MobiQuery*. *MobiQuery* meets stringent spatiotemporal constraints despite low duty cycles through a novel *just-in-time prefetching* mechanism. A prefetch message is disseminated to initiate query processing in a future query area before a user reaches it. A decision regarding when and where the prefetch message should be forwarded is taken based on the *motion profile* of the user, which may be generated from user input or from motion predictors. An important advantage of *MobiQuery* is that it can deal with imperfect motion profiles. In practice, a user may make an abrupt turn without notifying *MobiQuery* in advance. Motion profiles generated by history-based motion predictors usually lag behind actual motion changes. Due to the errors and delays in motion profiles, it is undesirable to greedily push the prefetch message too far ahead of the user. Greedy prefetching increases the network overhead related to the dissemination, maintenance, and storage of useless states in the nodes on the incorrectly predicted path. Through just-in-time prefetching, *MobiQuery* controls the distance between the prefetch message and the user in order to limit the negative effects of incorrect motion prediction without violating the spatiotemporal constraints.

The remainder of the paper is organized as follows. We first formalize the spatiotemporal query model and assumptions. We then present the detailed design of the *MobiQuery* architecture. Performance studies are then provided through both theoretical analysis and ns-2 simulations. We conclude the paper after a discussion on related work.

2. Problem Formulation

A spatiotemporal query consists of a mobile user traveling through a sensor field periodically collecting data from all sensors in a query area. A user initiates a spatiotemporal query using the following interface:

issueQuery(*attribute*, *F*, $A(P_u)$, T_{period} , T_{fresh} , $T_{duration}$)

attribute and *F* specify the query. *attribute* is the type of sensor data being queried. *F* is an aggregation function that is applied to the results prior to delivery. The in-network aggregation function can be used to reduce bandwidth consumption via in-network data aggregation — a well-investigated technique utilized by existing data services. Since *MobiQuery*’s primary contribution is providing spatiotemporal guarantees in a sensor network environment, we do not focus further on data aggregation in the rest of the paper. $A(P_u)$ is a function defining the query area relative to the user’s position P_u . The query area may span multiple hops. $A(P_u)$ is a general function that can return any spatial shape. For simplicity, however, we assume $A(P_u)$ is a circle with radius R_q centered around P_u in the rest of the paper.

T_{period} and T_{fresh} define the temporal constraints of the spatiotemporal query. T_{period} specifies the rate at which the user expects to receive query results. Query results must be delivered to the user by the end of each period. T_{fresh} specifies the maximum data age. That is, the k^{th} result must be received at or before $k \cdot T_{period}$, and the data in the result can be at most T_{fresh} old. This is necessary because the value of a sensor reading is often only useful when it is fresh. For example, in the fire fighting scenario, it is not useful to know last hour’s temperature. Finally, $T_{duration}$ is the lifetime of the query. After issuing a query, the user expects to receive query results every T_{period} for the next $T_{duration}$ seconds. Thus, $\lfloor \frac{T_{duration}}{T_{period}} \rfloor$ results are returned for each spatiotemporal query.

We make several assumptions about the underlying sensor network. First, all nodes have synchronized clocks. Second, we assume each node knows its own location through a localization service. Third, we assume the network runs an energy conservation protocol that selects a small subset of nodes to remain *active* while the remaining nodes operate in a duty cycle. The active nodes form a *backbone* network that allows messages to still be delivered between two nodes in the network with a latency in the order of one duty cycle. This assumption can be met since it is provided by a number of existing energy conservation protocols like SPAN [8], CCP [9], and GAF [13] which work alongside energy-conserving MAC protocols including SMAC [10], TMAC [14], and 802.11 PSM [15].

2.1. Motion Profile

MobiQuery relies on a motion profile to alert future query areas of an impending query. There are two ways of generating motion profiles: user input and motion prediction.

User Input: The user populates the motion profile with a sequence of motion vectors and times. In some scenarios, the user may be able to accurately predict his motion. For example, a user riding a train that travels a pre-defined route specified by the tracks can create the motion profile using the train route and speed information. In other scenarios, such as hiking primitive trails in a forest, motion may not be as predictable. In this case, the user can only create a partial motion profile.

When the user diverges from this motion profile, or when the profile expires, a new profile must be created.

Motion Prediction: A motion profile can also be produced using a motion prediction algorithm that relies on the recent movement history to predict the future motion of the user [16]. A motion history may be obtained through GPS.

There are advantages and disadvantages to each approach. The advantage of user input is that it allows the user to circumvent problems when the motion prediction changes by creating a new motion profile prior to the expiration of current motion prediction. The disadvantage is that it requires the user to manually enter his motion prediction, which is often inconvenient or infeasible. The history-based motion prediction approach can provide motion profiles transparently from the user, but cannot take advantage of the user’s anticipation of the current motion becoming invalid.

We assume motion profiles are specified by a *prediction event* that is represented by the 5-tuple $\langle P, t_s, T_v, t_g, T_a \rangle$, where the relationship between the fields are shown in Figure 1. P is the predicted motion profile of the user, it takes the following form:

$$\langle (\vec{v}_1, t_1), (\vec{v}_2, t_2), \dots, (\vec{v}_n, t_n) \rangle$$

where, \vec{v}_i is the velocity of the user at time t_i . The user is predicted to move along a straight line at a constant velocity during the time interval $[t_i, t_{i+1})$. For simplicity, we assume the user moves through a two dimensional space. t_s is the start time in which the prediction becomes active and T_v is how long the prediction is valid after t_s , e.g., the user is predicted to travel according to P during the interval $[t_s, t_s + T_v]$. t_g is the time when the motion profile is generated. $T_a = t_s - t_g$. It represents how early MobiQuery received the prediction event and is referred to as the prediction’s *advance time*.

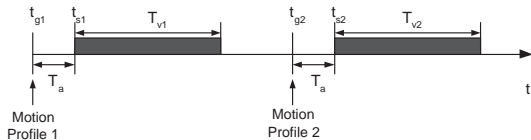


Figure 1: The motion profile model.

The above prediction event can model both of the above motion prediction schemes. For example, suppose the user inputs his motion profile periodically and T_v equals the period of the user input. In such a case, T_a is positive since the motion profile is always available from the user prior to the current motion profile expiring. If the motion profile is produced by a motion prediction algorithm, T_a may be negative due to the delay of motion estimation. That is, the motion profile within $[t_s + T_a, t_s]$ might have expired by the time the user receives the prediction event. T_a depends on the characteristics of the prediction algorithm.

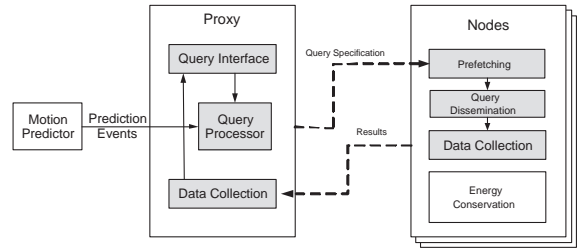


Figure 2: The architecture of MobiQuery (MobiQuery components are highlighted in gray)

3. System Architecture

The architecture of MobiQuery, as shown in Figure 2, is spread across two types of devices: a proxy, and nodes of the sensor network. A proxy is a mobile device external to the sensor network, such as a personal digital assistant or laptop. It is carried by the user as he moves through a sensor field and opportunistically forms wireless links with sensors in range. A user issues a spatiotemporal query through the proxy’s query interface. The query interface passes the specification to a query processor, which appends a motion profile, and passes the aggregate to the sensor network. The sensor network then performs the spatiotemporal query, periodically producing results at certain locations based on the motion profile. A data collection component within the proxy collects these results and passes them to the user via the query interface.

In addition to the proxy, MobiQuery also operates on the sensor network nodes. As shown in Figure 2, three MobiQuery components reside on the sensor nodes. They include prefetching, query dissemination, and data collection. MobiQuery can work with an energy conservation protocol that limits energy consumption by configuring the nodes to operate on a duty cycle, thereby extending network lifespan. The three components are discussed briefly below.

- **Prefetching.** The low duty cycles of sensor network nodes may result in poor data integrity due to sleeping nodes missing the query. To account for this, MobiQuery adopts a *prefetching* mechanism that forewarns nodes in future query areas allowing them to reconfigure their sleep schedules and prepare the results in time.
- **Query Dissemination.** The query dissemination component distributes the query to all nodes in each query area. This is done by creating a *query tree* that spans the entire query area, including the sleeping nodes.
- **Data Collection.** The data collection component is responsible for aggregating the query result for delivery to the user. When the data collection starts, all nodes in the query area perform a sensor reading and deliver the result to their parent. Intermediate nodes collect data from their children and send the aggregated data to their parent. The data is eventually aggregated by the root of the tree, which holds the data until the proxy arrives. When

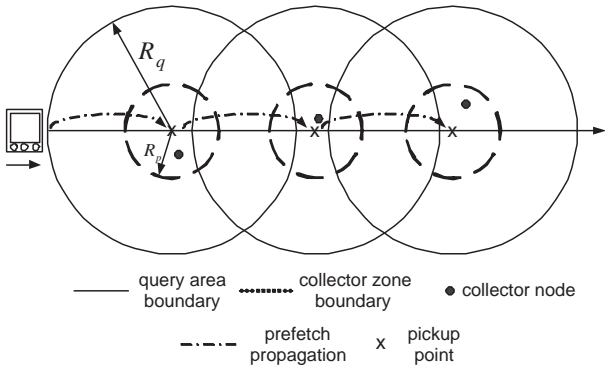


Figure 3: The prefetch message is anycasted to collector zones ahead of the user.

the proxy arrives, it broadcasts a request asking for the results, to which the root node responds with the aggregated query result.

We now discuss each of these components in detail.

3.1. Prefetching

MobiQuery uses prefetching to notify future query areas of an impending query as follows. The proxy first sends its query specification including the motion profile to the network. Next, the network uses the motion profile to predict a location, called *pickup point*, where the user expects to receive the next query result. The network then sends a *prefetch* message with the query specification to the pickup point using an area anycast [17]. The network repeats predicting the next pickup point and sending a prefetch message to it until the query expires. This process is shown in Figure 3. An area anycast is used since a node may not be located precisely at the pickup point. It delivers the prefetch message to a node in the *collector zone* which is within a certain distance, R_p , of the pickup point. Depending on the density of the sensor network, R_p may vary. The node within the collector zone that receives the prefetch message is the *collector node*. It is responsible for distributing the specifications throughout the query area and aggregating the results in time for delivery.

We now present two approaches, called *greedy* and *just-in-time* prefetching, under the assumption of perfect motion prediction. We then relax the assumption by showing how both approach can be generalized to account for errors in motion prediction.

3.1.1. Greedy Prefetching. In greedy prefetching, the proxy sends a prefetch message to the first collector zone. Each collector node forwards the prefetch message to the next collector node immediately after it receives the message. This process continues until $\lceil \frac{T_{duration}}{T_{period}} \rceil$ collector nodes have received the prefetch message. All query trees along the user’s route are set up as soon as possible. One advantage of this approach is it maximizes the slack time within each query area, allowing them to better handle unpredictable communication delays.

However, greedy prefetching also has several key disadvantages. 1) An incorrect motion profile (e.g., due to an unpredicted turn by the user) will result in the abandonment of many pickup points and query trees built on the original route. This leads to a waste of energy and bandwidth for forwarding prefetch messages and setting up query trees in abandoned query areas. It also leaves useless states on the nodes in those areas, which increases storage cost. 2) If the user moves slowly, or the query frequency is high, adjacent query areas will overlap. Since all trees are configured simultaneously, network congestion may occur. 3) The collector node and nodes on a query tree need to remember the query state (e.g., query parameters and the parental information) until the data collection is complete. In greedy prefetching, a query tree may be built long before the data collection, increasing memory overhead.

3.1.2. Just-In-Time Prefetching. Just-in-time prefetching addresses many of the disadvantages of greedy prefetching. Like greedy prefetching, the proxy sends a prefetch message to the first collector node, which forwards it to the next collector node, and so on. However, instead of forwarding the prefetch message immediately after it is received, a collector node holds the message before delivering it just-in-time to the next collector node. Ideally, the prefetch message should be forwarded at the latest possible time, such that the collector node can disseminate the specification to all nodes in the query area, including the sleeping nodes, for all the nodes to deliver the sensor data back to the collector in time for delivery to the proxy.

Just-in-time prefetching has several key advantages over greedy prefetching:

- As described in Section 3.1.3, just-in-time prefetching can potentially stop the creation of additional query trees when the user unexpectedly changes direction.
- Just-in-time prefetching also reduces the likelihood of multiple overlapping trees being formed concurrently, reducing wireless contention. The impact of greedy and just-in-time prefetching on network contention is formally analyzed in Section 4.4.
- Finally, creating the trees just-in-time reduces memory overhead since it reduces the amount of time for which memory is consumed for storing query states. We formally analyze the storage cost under greedy and just-in-time prefetching in Section 4.2.

3.1.3. Handling Motion Changes When the user changes its motion prediction, the proxy issues a new prefetch message based on a new motion profile. When just-in-time prefetching is used, the proxy can stop the previous prefetching process by sending a cancel message to the next collector node on the abandoned path. Similarly to greedy prefetching, each collector node immediately relays the cancel message to the

next one through area anycast until it reaches the furthest existing collector node. The cancel message prevents the node from forwarding the prefetch message further. This canceling mechanism reduces the number of query trees set up for an abandoned motion profile, and thereby saves energy, bandwidth, and storage cost associated with an expected motion change. Unfortunately, this mechanism is *not* effective for greedy prefetching. Since greedy prefetching forwards the prefetch message and creates query trees in an as-fast-as-possible fashion, all (or most of) the query trees corresponding to the abandoned motion profile may have already been created by the time a motion change occurs. Moreover, the cancel message may never be able to catch up with the prefetch message in order to stop the prefetching process!

A collector node removes its query result and query states when it receives a cancel message, or if it has not received a request for the query result from the proxy within a certain interval after the expected time, which is based on the motion profile.

Recall that the motion predictor provides motion events to the proxy whenever the user inputs a new motion profile, or when it detects an unpredicted change in movement. Occasionally, the motion predictor may fail to deliver a new motion profile before the current one expires. For example, a user may forget to input a new profile. To account for this, the proxy automatically renews the current motion profile T'_a seconds before the current profile expires. This extends the current direction and speed of the user until the predictor issues a motion event. Once the motion predictor issues a motion event, the new motion profile immediately overrides the old one. Figure 4 depicts the timing parameters of two motion profiles including a renewed profile. The black boxes are valid intervals of a motion profile, while the gray box represents a renewed motion profile composed by the proxy. Note that the valid interval of the second motion profile ($[t_{s2}, t_{s2} + T_{v2}]$) overlaps with the renewed profile, and overrides part of the renewed motion profile.

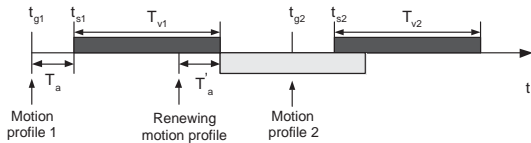


Figure 4: The renewing motion profile from the proxy.

A spatiotemporal query is effective immediately upon being issued. Since many nodes operate on low duty cycles, it may not be possible to notify them of the query fast enough to meet the first few deadlines. Thus, the first few queries after a new motion profile is issued may suffer from poor data integrity. This duration is called the *warmup interval*. Its length is analytically determined in section 4.

3.2. Query Dissemination

The query dissemination phase is responsible for distributing the query specification to all nodes in the query area, and to build a tree so that the query results can be efficiently routed back to the collector. The tree-building algorithm used by MobiQuery is as follows. Upon receiving the prefetch message, the collector delivers setup messages containing the query specification to all the nodes in the query area using a simple flooding technique where the message is repeatedly broadcast by backbone nodes further and further away from the collector node until it reaches the border of the query area. While flooding the query area, each backbone node remembers the first node from which it received the setup message as its parent and replies with an acknowledgement. The backbone nodes deliver the setup message to the sleeping nodes when they wake up (e.g., using 802.11 PSM). Upon receiving the specification, the sleeping nodes remember and acknowledge their parents, and reconfigure their sleep schedule to wake up just in time to contribute to the query. The process of selecting when to wake up is based on distance from the collector, and is covered in the next subsection on data collection. Once the sleeping nodes have been notified of the query, the result is a tree spanning the query area where the root is the collector node, all branches are composed of backbone nodes, sleeping nodes are leaves, and all parents know their children.

Sleeping nodes are purposely restricted to be leaves to allow them to quickly go back to sleep after performing a single sensor reading and transmission. This minimizes their energy consumption, and avoids problems that arise if they serve as a parent node. Specifically, if a sleeping node was allowed to be a parent node, it would have to remain awake in order to receive results from its children, and might also introduce significant delay to data collection process if it fails to wake up in time.

Note that MobiQuery utilizes a simple and efficient tree-building algorithm. More sophisticated algorithms were used by other data services such as TinyDB [18] and Directed Diffusion [2] to improve the quality of query trees. However, the user in these models remains stationary and queries the same tree numerous times justifying the greater overhead of building a more efficient tree. In contrast, MobiQuery consists of a mobile user who queries a tree at most once making it hard to justify using a more sophisticated tree-building algorithm.

3.3. Data Collection

Data collection is the process by which the nodes in the query area perform a sensor reading, and send their results through the tree back to the collector node. The earliest time any node can perform a sensor measurement for the query is T_{fresh} before the deadline. Thus, data collection must take less than T_{fresh} seconds to complete. A simple but naïve solution is to have all nodes wake up, perform a measurement, and transmit the results to their parent T_{fresh} before the dead-

line. The problem with this approach is that it will most likely result in network congestion and poor data integrity.

MobiQuery reduces contention by having each parent wait for their children before forwarding the data. This enables in-network data aggregation since a parent can send aggregated information after receiving data from all its children. By aggregating the data at each hop, fewer messages need to be sent, thus reducing contention. Furthermore, a parent will never send results at the same time as their children, further reducing contention. Once a parent receives data from all its children, it can immediately aggregate the data and send it to its parent. But what happens when a child fails to respond? This may happen for any number of reasons including an unreliable wireless network, contention, or loss of battery power. To address this, a parent times out at a sub-deadline and delivers the results regardless of whether all children have responded. This sub-deadline must be chosen so the results can still be delivered to the collector in time.

MobiQuery employs the following heuristic to assign sub-deadlines. After receiving a *setup* message node u sets its deadline to $k \cdot T_{period} - T_{fresh}$ and re-broadcasts the *setup* message. A node receiving this message sets u as its parent and sends back an acknowledgement. Once u determines it has at least one child, it changes its deadline to the *latest* deadline ensuring the child node has enough time to gather and report its measured data. The latest deadline is calculated according to the Euclidean distance between u and the root of the tree, using the root location included in the *setup* message. According to Section 3.1, a collector node always resides within R_p range of a pickup point and hence the maximum distance between the collector node and any node in the query area is $R_p + R_q$. The latest deadline of node u , denoted by d_u , is:

$$d_u = k \cdot T_{period} - \frac{|up|}{R_p + R_q} \cdot T_{fresh} \quad (1)$$

where p and $|up|$ represent the pickup point and the distance between node u and point p , respectively. The further a node is from the pickup point, the quicker it will timeout and forward the result to its parent. This sub-deadline assignment scheme increases the likelihood of delivering query results to the user in time, potentially at the cost of lower data integrity.

4. Analysis

In this section, we first derive an appropriate prefetch forwarding time for just-in-time prefetching, which is a key design parameter for MobiQuery to meet spatiotemporal constraints. We then analyze the duration of the warmup interval and the storage cost of a spatiotemporal query. Finally the impact of greedy/just-in-time prefetching on network contention is discussed.

4.1. Prefetch Forwarding Time

In a spatiotemporal query, the collector node must receive the prefetch message early enough to ensure that the query dissemination and data collection can complete before the query deadline. The time at which the prefetch message should be forwarded to the next pickup point such that the temporal constraints are met is defined as the *prefetch forwarding time*. The prefetch forwarding time depends on the delays involved in performing query dissemination and data collection at a pickup point. We first analyze the worst-case delays incurred by the query dissemination and data collection. We then derive an upper bound on the prefetch forwarding time.

In the derivation of the prefetch forwarding time we make use of two assumptions:

A.1 $T_{collect} \leq T_{fresh}$, where $T_{collect}$ is the maximum time it takes a node on a query tree to send its query result to the collector node. This condition is necessary for any service to meet both the freshness and the deadline constraints. We note that $T_{collect} \leq T_{fresh}$ is enforced by MobiQuery through the sub-deadline scheme during data collection as discussed in Section 3.3.

A.2 $v_{user} < v_{prefetch}$, where v_{user} and $v_{prefetch}$ denote the speed of the user and the prefetch message, respectively. We define the speed of the prefetch message sent from a source node u to a destination node v , denoted by $v_{prefetch}$, as the ratio of the Euclidean distance between u and v and the communication delay between the two nodes. Intuitively, this assumption is necessary for the network to be able to catch up with the user.

Let's consider the delivery of a prefetch message from the $(k-1)^{th}$ collector node to the k^{th} collector node. The goal is to derive the time t_k when the $(k-1)^{th}$ collector node should send the prefetch message to the k^{th} collector node. Let t'_k be the latest time by when the k^{th} collector node should receive the prefetch message in order to meet the deadline of the k^{th} query. Upon receiving the prefetch message, the node needs to perform two tasks: to set up the tree and to collect the data. Let T_{tree} be the time it takes to set up the tree. The total amount of work that a node has to do in order to deliver a query result is $T_{tree} + T_{collect}$. Consequently, the bound on the time by when the message should be received by k^{th} node is:

$$t'_k \leq k \cdot T_{period} - T_{tree} - T_{collect} \quad (2)$$

We approximate the location of every collector node with that of the corresponding pickup point. Then the time it takes for the prefetch message to be transmitted between the two considered collector nodes is $\frac{v_{user} \cdot T_{period}}{v_{prefetch}}$. Based on the assumption A.2, we know that $\frac{v_{user}}{v_{prefetch}} < 1$. Thus, we can upper bound the time it takes to relay the prefetch message by T_{period} . To guarantee that the k^{th} collector node can receive the prefetch message before t'_k , the following inequality must be satisfied:

$$t_k \leq t'_k - T_{period} \quad (3)$$

Considering (2) and (3), we can derive an upper bound on the time when the prefetch message should be sent by the $(k-1)^{th}$ delivery point:

$$t_k \leq t'_k - T_{period} \leq (k-1) \cdot T_{period} - T_{tree} - T_{collect} \quad (4)$$

Based on the first assumption we know that the time it takes to collect the data is upper bounded by T_{fresh} . Furthermore, the time it takes to set up the tree T_{tree} is upper bounded by the communication cost to set up the tree T_{setup} plus a maximum delay T_{sleep} due to the duty cycle. Accordingly, in MobiQuery a collector node computes the forwarding time using the following formula:

$$t_k \leq (k-1) \cdot T_{period} - T_{sleep} - T_{setup} - T_{fresh} \quad (5)$$

In (5) all quantities are known to the user except for T_{setup} . Since T_{setup} is the time it takes to set up the tree, T_{setup} is affected by several factors: the size of the query area, the one hop communication delay, and the contention level. For our experiments we assume that $T_{setup} \approx T_{collect} \leq T_{fresh}$. We made this assumption because of the similarities between the data dissemination and data collection processes: both exchange messages within the same query area and involve the same nodes. This assumption is reasonable because in the case of data collection, nodes must wait for the data to be delivered by their children to produce aggregates while the tree setup process does not require any synchronization. During the tree setup the query is disseminated as soon as possible. Thus, from the assumption $T_{setup} \approx T_{collect} \leq T_{fresh}$ and (5) we have:

$$t_k \leq (k-1) \cdot T_{period} - T_{sleep} - 2 \cdot T_{fresh} \quad (6)$$

We set t_k to the upper bound in (6) in the implementation of MobiQuery.

4.2. Storage Cost

In this section, we compare the storage costs of a spatiotemporal query under greedy prefetching and just-in-time prefetching. Intuitively, the information related to a spatiotemporal query that the network needs to remember is proportional to the total number of query trees ahead of the user. We refer to this number as *prefetch length (PL)* in the rest of paper. The storage cost of a query is roughly equal to the product of PL and the storage cost of a query tree which includes the parental information and parameters of the query (see Section 2). Since the storage cost of a query tree is fixed for a given query, we focus on the analysis of PL in the rest of this section.

We first derive PL under greedy prefetching and just-in-time prefetching when the user moves at a constant velocity

v_{user} . Under greedy prefetching, the number of query trees that are set up ahead of the user during an interval of t , denoted by $PL_{gp}(t)$, is as follows:

$$\begin{aligned} PL_{gp}(t) &= \left\lfloor \frac{(v_{prefetch} - v_{user}) \cdot t}{v_{user} \cdot T_{period}} \right\rfloor \\ &= \left\lfloor \frac{(v_{prefetch} - 1) \cdot t}{T_{period}} \right\rfloor \end{aligned} \quad (7)$$

We can see from (7) that the storage cost of greedy prefetching increases with the speed ratio of the user and the prefetch message. Furthermore, the storage cost is proportional to the duration of the query. We now derive PL under just-in-time prefetching. Let t_k represent the time instance when the $(k-1)^{th}$ collector node forwards the prefetch message. At t_k , the user is traveling between pickup points $\left\lfloor \frac{t_k}{T_{period}} \right\rfloor$ and $\left\lfloor \frac{t_k}{T_{period}} \right\rfloor + 1$. Hence all the query trees between $\left\lfloor \frac{t_k}{T_{period}} \right\rfloor$ and k have been set up. When t_k takes the upper bound in (6) (as in the implementation of MobiQuery), the prefetch length of just-in-time prefetching, denoted by PL_{jit} , satisfies the following equation:

$$\begin{aligned} PL_{jit} &= k - \left\lfloor \frac{t_k}{T_{period}} \right\rfloor \\ &= k - \left\lfloor k - 1 - \frac{T_{sleep} + 2 \cdot T_{fresh}}{T_{period}} \right\rfloor \\ &= \left\lfloor \frac{T_{sleep} + 2 \cdot T_{fresh}}{T_{period}} \right\rfloor + 1 \end{aligned} \quad (8)$$

We can see that the prefetch length of just-in-time prefetching is constant for given query parameters. From (7) and (8), it can be easily seen that PL_{jit} is smaller than PL_{gp} when the duration of a query exceeds a threshold t^* . We have (the rounding in (7) and (8) are ignored):

$$t^* = \frac{T_{sleep} + 2 \cdot T_{fresh} + T_{period}}{\frac{v_{prefetch}}{v_{user}} - 1} \quad (9)$$

We can see from (9) that when $v_{prefetch} \gg v_{user}$, t^* is small. This property of just-in-time prefetching is preferable since the memory resource of sensor nodes is highly constrained.

When the user changes its motion during a query, the prefetch length of MobiQuery can be analyzed similarly by applying the above derivations to every period of the user's movement with constant velocity. As discussed in Section 3.1.3, greedy prefetching incurs a much higher memory cost than just-in-time prefetching even when a cancel mechanism is employed.

4.3. The Warmup Interval

From the discussion in Section 4.1, we can see that the efficiency of just-in-time prefetching relies on the correctness of

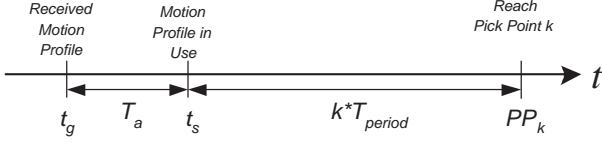


Figure 5: Temporal behavior of the prefetch message

the motion prediction. When the proxy is not forewarned about a motion change MobiQuery may transiently exhibit poor data integrity. This may occur in the case when the user inputs a change in the motion profile too late, or when the motion prediction algorithm updated the motion profile a certain interval after the motion change occurs. We refer to this transient interval caused by the late acknowledgement of a motion file as *warmup interval*. In this section we derive a bound on the warmup interval, denoted by T_{warmup} . This bound quantifies the tolerance of our system to errors in motion profile.

A collector node may detect that it has received a message too late *i.e.*, inequality (6) does not hold. In this case, MobiQuery attempts to catch up by forwarding the prefetch message as soon as possible. Since the speed of the message is higher than that of the user it is guaranteed that eventually inequality (6) will hold. During this catchup phase, MobiQuery uses greedy prefetching.

Suppose the proxy receives a motion profile with an advance time t_a . The motion profile starts being valid at t_s . We approximate the position of the collector nodes by the position of the pickup points. Let p_k denote the position of k^{th} pickup point as computed according to the motion profile. As shown in Figure 5, the earliest time that the collector node can send the prefetch message is when it receives the motion profile. The user needs to travel a distance of $v_{user} \cdot (k \cdot T_{period} + T_a)$ to reach p_k . Then, the time required for the prefetch message to reach the k^{th} pickup point, T_p , can be expressed as follows:

$$T_p = \frac{v_{user} \cdot (k \cdot T_{period} + T_a)}{v_{prefetch}} \quad (10)$$

To meet the deadline of p_k , the time taken by the user to reach p_k must be longer than the time taken to receive the prefetch message at p_k , set up the query tree and perform the data collection. Thus the following inequality must hold:

$$k \cdot T_{period} + T_a \geq T_p + T_{tree} + T_{fresh} \quad (11)$$

Solving k using (11) and (10):

$$k \geq \frac{T_{tree} + T_{fresh} - (1 - \frac{v_{user}}{v_{prefetch}}) \cdot T_a}{T_{period} \cdot (1 - \frac{v_{user}}{v_{prefetch}})} \quad (12)$$

k in (12) represents the upper bound on the number of queries that cannot meet the spatiotemporal constraints. Thus, the warmup interval lasts the first $\lfloor k \rfloor$ query periods and $T_{warmup} = T_{period} \cdot \lfloor k \rfloor$. The warmup interval becomes zero when $T_a = (T_{fresh} + T_{tree}) / (1 - \frac{v_{user}}{v_{prefetch}})$. That is, when the motion of the user can be predicted

early enough, the spatiotemporal queries do not incur any warmup interval. This result is also helpful for selecting the appropriate motion prediction scheme for the system to meet the spatiotemporal constraints of queries. The number of queries in the warmup interval reaches the minimum $(T_{tree} + T_{fresh} - T_a) / T_{period}$, when $v_{prefetch} / v_{user}$ approaches infinity.

Fig. 6 shows the warmup interval normalized to the query period T_{period} versus the $\frac{v_{prefetch}}{v_{user}}$. Both T_{setup} and T_{fresh} are set to $T_{period} / 2$ and the sleep period T_{sleep} is set to $2 \cdot T_{period}$. We can see that, as $v_{prefetch}$ approaches v_{user} , the duration of the warmup interval approaches infinity. This conforms to the intuition that the sleeping nodes in a query area may not be woken up before the user issues the query if the prefetch message travels very slowly.

At the beginning of a spatiotemporal query session, the proxy receives a motion profile from the motion predictor with zero advance time T_a , generation time T_g and start time T_s . In such a case, the number of queries in the warmup interval (from (12)) is:

$$k \geq \frac{T_{tree} + T_{fresh}}{T_{period} \cdot (1 - \frac{v_{user}}{v_{prefetch}})} \quad (13)$$

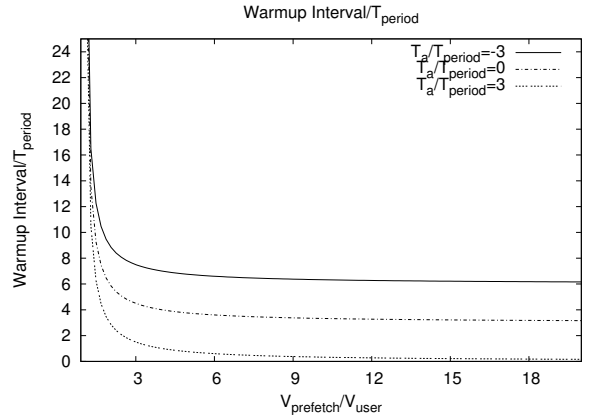


Figure 6: Warmup Interval Normalized to T_{period}

4.4. Network Contention

When the user moves slowly or the query frequency is high, adjacent query areas may overlap resulting in a high level of contention. Consequently, a query may suffer from packet loss or missed deadlines. In this section, we analyze the cause of network contention and show that the just-in-time prefetching scheme presented in Section 3.1 can reduce network contention by scheduling the setup times of different query trees just in time.

Fig. 7(a) and (b) illustrate the network traffic of the two prefetching mechanisms during the setup of query trees. The sleep schedule shown conforms to IEEE 802.11 PSM with the



Figure 7: Network Traffic in Creating Query Trees

extensions from [8]. A packet addressed to an active node can be sent immediately. On the other hand, a packet addressed to a sleeping node must be advertised by the sender inside the *advertisement* (ATIM) window before it is sent inside the *advertised traffic* (AT) window. A sleeping node turns its radio on at the beginning of the ATIM window and turns the radio off after receiving all the advertised traffic addressed to it inside the AT window. Hereafter, we refer to both the ATIM and AT windows as the *active window* without distinction, unless it is stated otherwise.

The channel is subject to significant contention when query trees are set up because communication with sleeping nodes must occur within short active windows. In contrast, the data collection process introduces less contention because all nodes are scheduled to wake up for data collection. Henceforth, we focus on analyzing the network contention caused by the tree setup process.

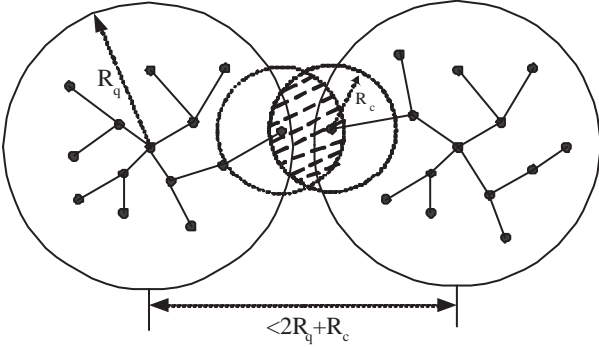


Figure 8: Two trees in two query areas may interfere with each other

We now consider the necessary conditions for two query trees to interfere with each other. First, the query trees must be located close to each other spatially. We approximate the position of the collector nodes by the position of the pickup points. Since the radius of a query area is R_q , as shown in Fig. 8, the Euclidean distance between the roots of the two trees must be less than $2R_q + R_c$. The shaded region in Fig. 8 depicts the contention area of two nodes in two query areas. Secondly, parts

of the set up processes of two query trees must occur concurrently. To quantify the level of contention, we define *overlapping factor* as the maximal number of query trees that overlap *both* spatially and temporally. A query tree is considered to overlap with itself, *i.e.*, the overlapping factor is at least one. Clearly, a higher overlapping factor indicates a higher level of contention.

The number of query trees that overlap spatially, denoted by M_s , remains the same for both prefetching schemes, since it only depends on the locations of pickup points.

For simplicity, we assume the user moves on a straight line in the rest of this section. Since the user issues a query every T_{period} s, the roots of the query trees align spatially at an interval of $v_{user} \cdot T_{period}$ on the route of the user. Note that a tree may interfere with the adjacent trees within a distance of $2R_q + R_c$ in two different directions. Thus, M_s satisfies the following inequality:

$$M_s \leq \left\lceil \frac{4R_q + 2R_c}{v_{user} \cdot T_{period}} \right\rceil \quad (14)$$

Let M_{t-JIT} and M_{t-GP} represent the upper bounds on the number of query trees that overlap temporally for just-in-time and greedy prefetching, respectively. Under greedy prefetching, the time taken for a prefetch message to travel between two consecutive pickup points is $\Delta_t = v_{user} \cdot T_{period} / v_{prefetch}$. That is, query trees are set up at an interval of Δ_t . Hence the maximum number of query trees whose setup processes overlap temporally is:

$$M_{t-GP} \leq \left\lceil \frac{T_{tree}}{\Delta_t} \right\rceil = \left\lceil \frac{T_{tree} \cdot v_{prefetch}}{T_{period} \cdot v_{user}} \right\rceil \quad (15)$$

where T_{tree} is the time taken to set up a query tree, as defined in Section 4.1. Hence the overlapping factor of greedy prefetching (denoted by M_{GP}) is as follows:

$$M_{GP} = \min(M_{t-GP}, M_s) \quad (16)$$

In the steady state of just-in-time prefetching, as discussed in Section 4.1, the query trees are set up with an interval of T_{period} . Hence the maximum number of trees whose setup processes overlap temporally, denoted by M_{t-JIT} , is $\left\lceil \frac{T_{tree}}{T_{period}} \right\rceil$. Thus the overlapping factor of just-in-time prefetching (denoted by M_{JIT}) can be derived as follows:

$$M_{JIT} = \min(M_{t-JIT}, M_s) \quad (17)$$

Since $v_{prefetch} > v_{user}$ (assumption A.2 in Section 4.1), we can see that $M_{t-JIT} > M_{t-GP}$. According to (16) and (17), $M_{JIT} \leq M_{GP}$.

According to (14)-(17), the following cases can be derived about the relationship between M_{GP} and M_{JIT} (the roundings in the expressions of M_{t-GP} , M_{t-JIT} and M_s are ignored):

Case 1: $M_{JIT} = M_s = M_{GP}$, when $v_{prefetch} > v_{user} > \frac{2Rc+4Rq}{T_{tree}}$.

Case 2: $M_{JIT} = M_{t-JIT} < M_s = M_{GP}$, when $v_{prefetch} > \frac{2Rc+4Rq}{T_{tree}} > v_{user}$.

Case 3: $M_{JIT} = M_{t-JIT} < M_{t-GP} = M_{GP}$, when $\frac{2Rc+4Rq}{T_{tree}} > v_{prefetch} > v_{user}$.

We can see that the overlapping factor of just-in-time prefetching is smaller than that of greedy prefetching as long as the user speed remains below the threshold $(2Rc + 4Rq)/T_{tree}$. This conforms to the intuition that when the speed of the user is high, the prefetch messages under just-in-time prefetching must be relayed by collector nodes very quickly, resulting in a behavior similar to greedy prefetching, where the prefetch messages are always relayed as fast as possible.

5. Simulation Results

We implemented MobiQuery on top of Coverage Configuration Protocol (CCP) and IEEE 802.11 Power Saving Mode (PSM) with the extension from [8] in the ns2 simulator. In this section, we present the simulation results.

5.1. Metrics

The performance metrics we used are as follows:

- **Data Integrity:** the ratio between the number of nodes that contribute to a query result and the total number of nodes in a query area. Data integrity represents how well the spatial constraints of queries are met.
- **Slack Time:** the difference between query deadline and the time instance at which a query result is received by the user. A negative slack time indicates a violation of the temporal constraints of queries. Due to the sub-deadline scheme discussed in Section 3.3, most of the query results meet deadlines.
- **Percentage of Successful Queries (PSQ):** the percentage of the results that meet deadlines and have data integrity higher than a threshold. PSQ indicates the overall quality of service received by the user. The threshold of data integrity is 95% in all simulations.
- **Energy Consumption:** We measure the average energy consumption per sleeping node during a spatiotemporal query.

5.2. Experimental Settings

In all simulations, 200 nodes are randomly distributed in a $450m \times 450m$ rectangular area. The Coverage Configuration Protocol (CCP) is run to choose a small number of active

nodes to maintain sensing coverage over the deployment region. We note that MobiQuery can also work with other energy conservation protocols that control node duty cycles. We set the communication range and sensing range of the network to $105m$ and $50m$ in all ns2 simulations. The radius of a query area is $150m$. Under these settings, query trees set up during query dissemination have about $2 \sim 4$ levels. The ATIM and AT windows in IEEE 802.11 are set to $100ms$ and $500ms$, respectively. The network sleep period varies from $3s$ to $15s$, which results in duty cycles of 20% to 4% for sleeping nodes.

At the beginning of each simulation, the user starts from a corner of the query area and moves in a random direction with a speed randomly chosen from the range $[v_{min}, v_{max}]$. The user may change its direction and choose a new speed from $[v_{min}, v_{max}]$ during its movement. Three speed ranges, $3 \sim 5m/s$, $6 \sim 10m/s$ and $16 \sim 20m/s$, corresponding to the speed of a walking human, a running human and a vehicle with moderate speed, respectively are used in the simulations. If the user hits the boundary of the deployment region before a simulation ends, it changes the movement direction toward a random point within the region.

Area anycast is implemented based on the greedy forwarding routing algorithm to deliver prefetch messages. A node always forwards a prefetch message to the neighbor that has the closest Euclidean distance to the pickup point. If a node cannot find a neighbor closer to the pickup point than itself, it then accepts the prefetch message and serves as the collector node of the query if its distance to the pickup point is shorter than R_q . We note that anycast can be implemented using more complex geographic routing algorithms like GPSR [19] that can handle routing voids and hence improve the ratio of successful queries. Tab. 1 summarizes our experimental settings.

5.3. Performance under Accurate Motion Profiles

In this section, we evaluate the impact of user mobility and network sleep schedule on the performance of MobiQuery. In the simulations, the user changes its direction and speed every 50 seconds and the user inputs its complete motion profile to the proxy at the beginning of each simulation. The cases where the proxy has only limited knowledge about the user's motion will be evaluated in Sections 5.4. The total time of each simulation is $400s$.

For performance comparison, we implemented a baseline algorithm called *No-Prefetching (NP)*. In NP, the user broadcasts a query to the network at the beginning of each query period and requires the query results by the end of the same period. The remaining operations are the same as MobiQuery except that the user, instead of the collector node, serves as the root of the query tree in each query area.

Fig 9 shows the percentage of successful queries of MobiQuery and NP under different network sleep schedules and user speeds. The results are averages over 3 runs with different network topologies. The implementations of MobiQuery with greedy prefetching and just-in-time prefetching are denoted as

Parameter	R_c	Topology	ATIM Window	AT Window	Query Period	Query Freshness	R_q
Value	110m	$450 \times 450m$	100ms	500ms	2s	1s	R_c

Table 1: Experimental Settings

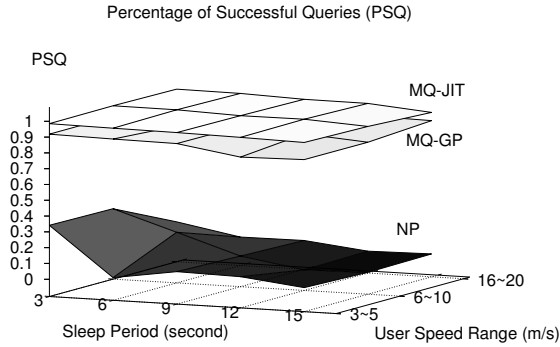


Figure 9: Performance Comparison between MobiQuery and Baseline Algorithms

MQ-GP and MQ-JIT respectively, in the figure. The PSQ under MQ-JIT achieves near 100% in all settings, even when the sleeping period is as long as 15 sec, *i.e.*, 7.5 times the query period. The PSQ of MQ-GP reaches about 90% when sleep period is less than 9s and decreases when sleep period becomes longer. In sharp contrast, the PSQ of NP remains below 35% in all settings. NP’s performance degrades when the sleep period increases. This result clearly indicates that prefetching is crucial and highly effective to meet spatiotemporal constraints in sensor networks with low duty cycles.

Interestingly, MQ-GP performs *worse* than MQ-JIT, despite the fact that MQ-GP sets up more pickup points more quickly. As discussed in Section 4.4, the number of query trees that are set up concurrently under greedy prefetching is large, resulting in high network congestion. Consequently, some sleeping nodes may not be woken up due to the loss of *setup* messages, which impairs the data integrity of queries. When the sleep period is longer, as discussed in Section 4.4, the network congestion becomes worse since more packets need to be sent to the sleeping nodes within the active window. Greedy prefetching performs slightly better when the user moves faster since the adjacent query areas become further apart, resulting in lower network congestion.

To examine the dynamic behavior of just-in-time prefetching and greedy prefetching, we plot the data integrity and slack time of MobiQuery at each pickup point in Fig. 10 and 11, respectively. The sleep period is 15s in all simulations. The speed range of the user is $3 \sim 5m/s$. As shown in Fig. 10, both implementations of MobiQuery suffer from an initial warmup phase in which about 5 queries have relatively low data integrity. This result is consistent with the worst-case analysis of warmup interval presented in Section 4.3. After the warmup phase, MQ-JIT achieves a data integrity of 100% in most query

periods.

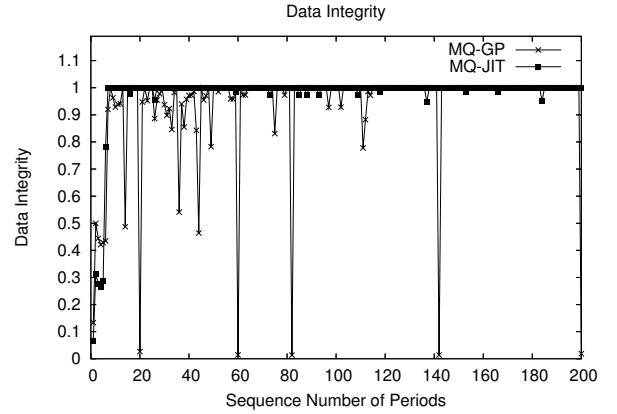


Figure 10: Data Integrity of Greedy/JIT Prefetching

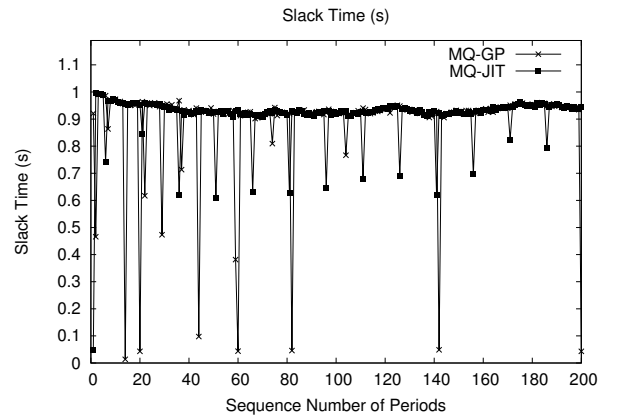


Figure 11: Slack Time of Greedy/JIT Prefetching

In contrast, the data integrity of MQ-GP incurs significant fluctuation due to packet loss caused by network congestion. Since the query period in the simulations is relatively short (two seconds), the consecutive query areas share a number of nodes. Hence, the query trees of adjacent query areas were set up roughly at the same time, resulting in high congestion. The fluctuation of the data integrity decreases over time since network congestion becomes less severe after the setup of some query trees is completed.

Fig. 11 shows the slack time of MobiQuery with different prefetching schemes. We see that, consistent with the result on data integrity shown in Fig. 10, the slack time of the queries has high fluctuation when greedy prefetching is used. There are two reasons for the fluctuation: 1) If the setup message

broadcast by the root node of a query tree is lost due to collisions, the resulting tree will have only the collector node. In such a case, both data integrity and slack time of the query approaches zero. This is because the collector node is unaware of the loss of the *setup* message and waits until the end of the query period before delivering its sensor data to the user. The results of the 20th, 60th and 82th results correspond to this case. 2) If a result message from a node is lost, the parent of the node will incur a delay waiting for the lost result. In such a case, although the slack time of a query is small, the data integrity may still be high. The results of the 22th and 29th queries correspond to this case. Note that most of the queries met their deadlines due to the timeout mechanism presented in Section 3.3.

On the other hand, MQ-JIT has a much smaller fluctuation in slack time since the network congestion is moderate. However, several queries have relatively small slack times (about 0.4 ~ 0.5s), compared to other queries that achieve much longer slack times. Such delays are caused by the network contention inside the active windows of the sleep schedule.

5.4. Effects of Imperfect Motion Prediction

Information about the user’s future movement is important for MobiQuery in order to meet the spatiotemporal constraints of queries through prefetching. However, the motion predictor varies significantly depending on how the profile is generated, *e.g.*, from the user input or a history-based motion prediction algorithm, as discussed in Section 2. In this section, we evaluate the performance of MobiQuery under different motion profile settings.

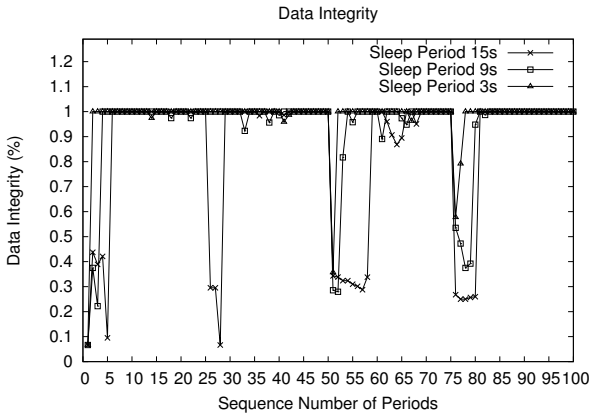


Figure 12: Date Integrity with Multiple Warmup Phases

In the following experiments, the user moves in a straight line at a random speed within 3 ~ 5m/s until it reaches a “change point” where it selects a new direction and sets a random speed within 3 ~ 5m/s. The user changes direction every 70 seconds. The total time of each simulation is 500 seconds.

The proxy receives a motion profile T_a seconds before it reaches a change point, indicating the prospective motion change. When T_a is negative, the motion profile is provided to MobiQuery after a motion change occurs. We vary T_a from $-3s$ to $9s$ in the simulations, thus which modeling different ways of generating a motion profile, *e.g.*, from the user input or a motion prediction algorithm. Each motion profile from the predictor has a validity interval of $T_v = 25$ seconds. Since the validity of each motion file is shorter than the actual duration of a segment of movement, as presented in Section 2, the proxy automatically extends the existing motion profile T'_a seconds before it expires. T'_a is set to be 3 seconds longer than the sleep period, so that the warmup costs (except the initial period) are solely caused by the motion changes.

Fig. 12 shows the data integrity of MQ-JIT when the advance time of the motion profile is 3 seconds. We can see that the warmup phase is not evident when the sleep period is 3s. In other settings, the warmup intervals are roughly proportional to the ratio between the sleep period and query period, which conforms to the analysis in Section 4.3.

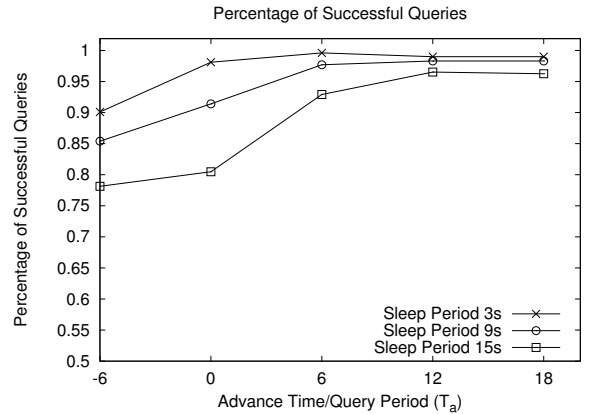


Figure 13: Performance of MobiQuery vs. Advance Times

Fig. 13 shows the average PSQ of MQ-JIT under different sleep schedules when T_a varies from $-6s$ to $18s$. We can see that when the sleep period decreases, the warmup interval becomes shorter and hence more queries succeed. For each sleep period, the PSQ increases with T_a . The additional benefit of a larger T_a diminishes when it reaches a threshold. In this case, the warmup interval approaches zero, and the PSQ converges to a level close to 100%. The difference between the convergence value and 100% is due to the occasional packet loss and initial warmup interval at the beginning of a simulation. For example, when the sleep period is 9s, MQ-JIT achieves a PSQ around 98% when a motion profile is provided only 12s before a motion change.

It is also important to note the performance of MQ-JIT when $T_a = -6s$, *i.e.*, a new motion profile is provided to MQ-JIT 6s after the motion change occurs. In this case, although its PSQ drops due to longer warmup intervals, MQ-JIT

still maintains reasonable spatiotemporal services. For example, MQ-JIT achieves a PSQ around 85% when the sleep period is 9s.

From Section 5.4, we can see that the user may receive degraded service when there are unexpected motion changes. In this section, we evaluate the impact of motion changes on the performance of MobiQuery .

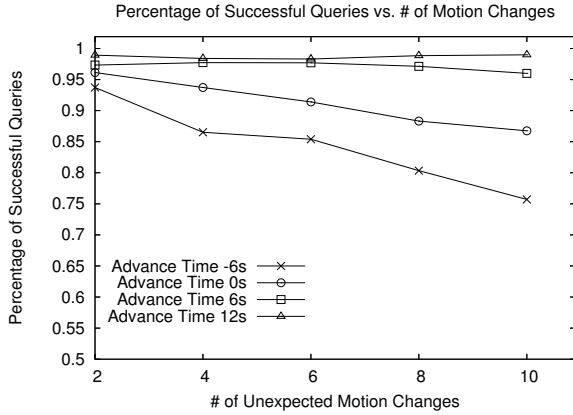


Figure 14: Performance of MobiQuery vs. Number of Unexpected Motion Changes

In this set of simulations, the motion pattern of the user is the same as described earlier in this section except that the number of motion changes varies from 2 to 10. Fig. 14 shows the PSQ under different advance times of motion profiles when the sleep period is 9 seconds. As expected, motion changes have no impact on the spatiotemporal performance of MQ-JIT when T_a is large. When the advance time is negative, the system performance drops slowly as the number of motion changes increases. However, MQ-JIT still achieves satisfactory performance even when the user changes motion at very high frequency. For example, MQ-JIT successfully returns about 75% of the requested results under desired spatiotemporal constraints even when the user changes his motion every 70s.

In summary, our results demonstrate the robustness of MQ-JIT when operating with imperfect motion profiles. MQ-JIT can take advantage of a small advance time in motion profile input to maintain perfect spatiotemporal services despite motion changes by the user. It can also tolerate motion prediction techniques with considerable delays, as well as high frequency motion changes by the user.

5.5. Power Consumption

In a network with a sleep schedule, MobiQuery wakes up a sleeping node only when necessary and the sleep schedule of the node is resumed immediately after it reports its sensor data. In this section, we evaluate the power consumption of MobiQuery under different sleep schedules.

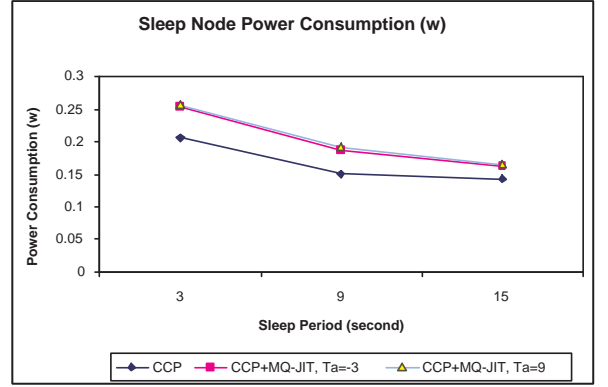


Figure 15: Power Consumption

In the simulations of this section, a small number of nodes in the network are activated by CCP and remain active all the time to maintain sensing coverage. Since these active nodes never turn off their radios before running out of energy, their power consumption is much higher than sleeping nodes and is not effected by the settings of sleep schedules. We focus on the average power consumption per sleeping node under different sleep schedules. The motion pattern is the same with that used in Section 5.4. The user changes its motion 6 times in each simulation. To test the impact of the latency of motion prediction on the power consumption, two advance times of motion profiles, $-3s$ and $9s$, are used in the simulations. For comparison, we also measured the power consumption of CCP without any query. In accordance with the measurement of the Cabletron IEEE 802.11 network card performed in [8], the power consumption of Tx (transmit), Rx (receive), Idle and Sleeping modes of the radio are set to 1400mW, 1000mW, 830mW and 130mW, respectively.

Fig. 15 shows the power consumption per sleeping node during a simulation time of 400s. The results are averages of 5 runs with different network topologies. We can see that the power consumption of both CCP and MQ-JIT decreases as the sleep period becomes longer. The difference between the power consumptions of CCP and CCP+ MobiQuery , which is the net power consumption of sole MobiQuery operations, remains below $0.05w$ under all sleep periods. The power consumption of MobiQuery becomes slightly lower when the advance time of motion profiles decrease from $9s$ to $-3s$. When the advance time is $-3s$, the proxy receives the user's new motion profile 3 seconds after a motion change. In such a case, as analyzed in section 4.3, MobiQuery incurs a warmup interval for each motion change of the user and hence less nodes are woken up to participate in the data collection, resulting in less energy consumption. The result of this section indicates that MobiQuery conserves the power consumption of the network by taking advantage of the sleep schedule.

6. Related Work

The notion of spatiotemporal constraints in sensor networks was identified in [20]. Mobicast[20] is a spatiotemporal multicast protocol designed for just-in-time data dissemination to a delivery area that changes over time. In contrast, MobiQuery is a spatiotemporal *data service* designed to query an area that changes over time. In addition, MobiQuery is designed to be sensitive to energy conservation issues. In particular it can effectively handle node sleeping schedules controlled by energy conservation protocols through just-in-time prefetching.

Early work on data services in sensor networks such as TinyDB[3] and Directed Diffusion[2] were not optimized for mobile sinks or query areas. Directed Diffusion uses a data-centric naming scheme to allow for in-network data aggregation. A disadvantage of Directed Diffusion is that it requires an interest to be flooded throughout a large, fixed area. MobiQuery avoids this problem by bounding the query area to the vicinity of a mobile user.

MobiQuery bears some resemblance to TinyDB in the way in which data aggregation is performed at each pickup point: in both cases a tree is setup and query results are routed and aggregated using an overlay tree. However, in contrast to TinyDB, in MobiQuery the building of the tree and the propagation of query results are subjugated to spatiotemporal constraints related to mobile users and changing sources. Unlike the other aggregation protocols that produce multiple results from the same sources, MobiQuery needs to deal with the change of data sources as the query area changes. Consequently, since the tree setup in MobiQuery is not being reused, the complexity of the tree setup has been kept to a minimum. Potentially, as long as the overhead is acceptable, MobiQuery may use TinyDB or Directed Diffusion to collect data for a query area.

TTDD[5] and SEAD[4] are data services that deal with sink mobility while the query area is fixed. Unlike these protocols, MobiQuery is designed to query an area that follows the movement of an user. Dealing with both the motion of the sink and with the change in data sources introduces new challenges to the design of data services.

In TTDD a virtual grid is built to optimize the delivery of data from sources to mobile sinks. By building a grid, a two tier hierarchy is established between nodes that are on the grid, and nodes that are within the grid. The messages are routed towards the sink by the nodes on the grid. The last part of the delivery is a flooding of the region in which the sink is located. Similarly, in MobiQuery, a two tier hierarchy is established between active and sleeping nodes. Even though the setup process differs from TTDD, MobiQuery uses the two tier hierarchy to bound the setup time for a pickup point. As such, during the tree setup phase, the active nodes first become a part of the tree and then later the sleeping nodes join the tree as leaves. SEAD deals with the issue of sink mobility by building a tree and placing caches in the network to minimize latency and energy consumption. The approach used by SEAD is not

directly applicable to spatiotemporal query when the sink is mobile *and* the sources change over time.

DCTC[21, 22] is an object tracking technique. It builds a tree around the source and tracks the source as it moves in the sensor field. The interesting aspects of DCTC are the two policies proposed for the tree expansion and pruning schemes. The two policies spring from considering different motion models. In the conservative scheme, the speed of the source is bounded. The source is able to move in any direction at any speed smaller than its speed bound. In the predictive scheme, it is assumed that the motion profile of the source is known, and thus the future position of the source can be predicated. The two policies decide which nodes are to be woken up in order to join the tree as the source moves. The predictive scheme bears resemblance to the approach taken by MobiQuery. Other than the functionalities, a key difference from DCTC is that MobiQuery is designed to meet stringent spatiotemporal constraints. While DCTC only provides a best effort scheme for waking up nodes, MobiQuery achieves predictable spatiotemporal performance by utilizing an analytical bound on the time when prefetch messages must be forwarded. The impact of long sleeping schedules is also not analyzed in DCTC. As shown in our experimental results, the long sleeping schedules have a significant effect on the prefetch forwarding time. Another aspect not explored by DCTC is the impact of imperfect motion prediction. In MobiQuery we introduce the idea of just-in-time prefetching to mitigate the impact of incorrect motion predication. Unlike DCTC, MobiQuery does not reconfigure the tree but establishes a new tree at each pickup point. We find that the techniques proposed in DCTC nicely complement our approach for optimizing query trees.

A method for bounding the delay of disseminating information in an area is presented in [20]. This result can be extended to provide a bound on T_{setup} . However it relies on compactness properties that require additional knowledge about network topology. [23] analyzed the worst-case bounds on the hop counts of several greedy forwarding routing algorithms. However, these results require sensing coverage and double range property in a sensor network.

7. Conclusions

We presented MobiQuery, a novel spatiotemporal query service that allows a mobile user to periodically gather information from a surrounding area through a sensor network. The key feature of MobiQuery is its capability to meet stringent spatiotemporal constraints despite a set of unique challenges in sensor networks including 1) extremely long sleeping schedules due to the need for energy conservation, 2) network congestion due to overlapping query areas, and 3) imperfect knowledge about user motion. A just-in-time prefetching mechanism was designed to minimize energy consumption while reducing the impact of incorrect motion profiles. We also offer a set of analysis that provides theoretic guarantees on the spatiotemporal constraints and bounded warm-

up phase for MobiQuery. NS-2 simulation results demonstrate that just-in-time prefetching enables MobiQuery to maintain desired spatiotemporal performance under a broad range of network and motion conditions, when non-prefetching and greedy-prefetching fail to provide acceptable services. Our results also demonstrate that MobiQuery can tolerate imperfect motion profiles with considerable delays and errors.

References

- [1] Yong Yao and Johannes Gehrke, "The cougar approach to in-network query processing in sensor networks," *SIGMOD Rec.*, vol. 31(2), pp. 9–18, 2002.
- [2] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking*, 2000, pp. 56–67.
- [3] Madden S., Franklin M., Hellerstein J., and Hong W., "Tag: a tiny aggregation service for ad-hoc sensor networks," in *Proceedings of the Fifth Annual Symposium on Operating Systems Design and Implementation*, 2002.
- [4] Hyung Seok Kim, Tarek F. Abdelzaher, and Wook Hyun Kwon, "Minimum-energy asynchronous dissemination to mobile sinks in wireless sensor networks," in *Proceedings of the first international conference on Embedded networked sensor systems*. 2003, pp. 193–204, ACM Press.
- [5] Fan Ye, Haiyun Luo, Jerry Cheng, Songwu Lu, and Lixia Zhang, "A two-tier data dissemination model for large-scale wireless sensor networks," in *Proceedings of the 8th annual international conference on Mobile computing and networking*. 2002, pp. 148–159, ACM Press.
- [6] J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris, "A scalable location service for geographic ad-hoc routing," in *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom '00)*, Aug. 2000, pp. 120–130.
- [7] D. Tian and N.D. Georganas, "A coverage-preserved node scheduling scheme for large wireless sensor networks," in *Proceedings of First International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, USA, Sep 2002, pp. 169–177.
- [8] Benjie Chen, Kyle Jamieson, Hari Balakrishnan, and Robert Morris, "Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks," in *Proceedings of the 7th ACM International Conference on Mobile Computing and Networking*, 2001, pp. 85–96.
- [9] Xiaorui Wang, Guoliang Xing, Yuanfang Zhang, Chenyang Lu, Robert Pless, and Christopher D. Gill, "Integrated coverage and connectivity configuration in wireless sensor networks," in *The First ACM Conference on Embedded Networked Sensor Systems(Sensys 03)*, 2003.
- [10] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient mac protocol for wireless sensor networks," in *Proceedings of INFOCOM 2002.*, 2002.
- [11] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM international workshop on wireless sensor networks and applications*. 2002, pp. 88–97, ACM Press.
- [12] Crossbow, "Mica2 wireless measurement system datasheet," 2003.
- [13] Ya Xu, John Heidemann, and Deborah Estrin, "Geography-informed energy conservation for ad hoc routing," in *Proceedings of the 7th annual international conference on Mobile computing and networking*. 2001, pp. 70–84, ACM Press.
- [14] Tijs van Dam and Koen Langendoen, "An adaptive energy-efficient mac protocol for wireless sensor networks," in *The First ACM Conference on Embedded Networked Sensor Systems(Sensys 03)*, October 2003.
- [15] IEEE, "Wireless lan medium access control (mac) and physical layer (phy) specifications," *IEEE Standard 802.11*, 1999.
- [16] A. Aljadhai and T. F. Znati, "Predictive mobility support for qos provisioning in mobile wireless environments," *IEEE journal on selected areas in communications (JSAC)*, vol. 19(10), October 2001.
- [17] Tian He, John A. Stankovic, Chenyang Lu, and Tarek Abdelzaher, "Speed: A stateless protocol for real-time communications in sensor networks," in *International Conference on Distributed Computing Systems (ICDCS 2003)*, 2003.
- [18] Samuel Madden, Michael Franklin, Joseph Hellerstein, and Wei Hong, "The design of an acquisitional query processor for sensor networks," in *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*. 2003, pp. 491 – 502, ACM Press.
- [19] Brad Karp and H. T. Kung, "GPSR: greedy perimeter stateless routing for wireless networks," in *Proceedings of the 7th ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2000, pp. 243–254.
- [20] Q. Huang, C. Lu, and G-C. Roman, "Spatiotemporal multicast in sensor networks," in *First ACM Conference on Embedded Networked Sensor Systems (SenSys'03)*, 2002.
- [21] Wensheng Zhang and Guohong Cao, "DCTC: Dynamic Convoy Tree-Based Collaboration for Target Tracking in Sensor Networks," to appear in *IEEE Transactions on Wireless Communication*.
- [22] W. Zhang and G. Cao, "Optimizing tree reconfiguration for mobile target tracking in sensor networks," in *Proc. IEEE INFOCOM*, March 2004.
- [23] Guoliang Xing, Chenyang Lu, Robert Pless, and Qingfeng Huang, "On greedy geographic routing algorithms in sensing-covered networks," in *Proc. Fifth ACM Symp. on Mobile Ad Hoc Networking and Computing (MobiHoc'04)*, Tokyo, Japan, Oct. 2004.