A Spatiotemporal Query Service for Mobile Users in Sensor Networks

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

Abstract

This paper presents MobiQuery, a spatiotemporal query service that allows mobile users to periodically gather information from their surrounding areas through a wireless sensor network. A key advantage of MobiQuery lies in its capability to meet stringent spatiotemporal performance constraints crucial to many applications. These constraints include query latency, data freshness and fidelity, and changing query areas due to user mobility. A novel just-in-time prefetching algorithm enables MobiQuery to maintain robust spatiotemporal guarantees even when nodes operate under extremely low duty cycles. Furthermore, it significantly reduces the storage cost and network contention caused by continuous queries from mobile users. We validate our approach through both theoretical analysis and simulation results 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 mobile users at the right time and right location. In this paper, we propose a new data service for mobile users in sensor networks called *spatiotemporal query*. In contrast to existing data services that generally assume fixed areas of interest [9, 14, 20, 10, 19], spatiotemporal query is motivated by a class of mission-critical applications in which mobile users need to continuously gather real-time information within their vicinities. For example, a fireman fighting a wild fire may request a periodic update of a temperature map within one mile around his current location to remain alert to the surrounding fire conditions. As the fireman moves, the query area changes accordingly. As another example, a robot in a search and rescue operation needs to continuously query surrounding sensors for information about the terrain and survivors as it moves in a unknown dynamic environment. Based on the query results, the robot can locate survivors and find the best rescue route through motion planning [13]. The queries in the above applications are subject to a common set of constraints.

- *Spatial constraints:* Only the nodes inside the query area corresponding to the user's current position should contribute to the query result. Involving more nodes wastes precious energy without improving the quality of service as their data are not interesting to the user at this point of time. In addition, it is desirable to aggregate data from enough nodes in the current query area to improve the fidelity of the query result.
- *Temporal constraints:* Meanwhile, a query is also subject to temporal constraints in terms of *query deadlines* and *data freshness*. A new query result must be delivered to the user by the end of each query period. Furthermore, each query result must be aggregated from *fresh* sensor data, where the freshness of a sensor datum is defined by its maximum validity interval after it is read from the sensor.

Meeting these constraints is crucial for the mission-critical applications that rely on surrounding sensor data to maintain spatiotemporal context awareness. In the examples discussed earlier, a fireman may be endangered by a quickly evolving wild fire if the query results are aggregated from old sensor readings, are delivered too late, are aggregated from sources at wrong locations, or if too few nodes in the current query area contribute to the results. Similarly, a search and rescue robot may fail to locate survivors in time and to find the best rescue route in a dynamic environment if any of these constraints are violated.

Meeting all the spatiotemporal constraints is especially challenging in wireless sensor networks due to their severe power and resource constraints. For example, although the temporal constraints require the nodes to respond to a query in a timely fashion, the low node duty cycles forced by the power constraint can significantly increase the communication delay. Compared to traditional ad hoc networks, sensor

Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICSCS'05)
1063-6927/05 $20.00 © 2005 **IEEE**

networks often require much lower duty cycles due to their significant longer lifetime requirement. For instance, for a MICA2 mote to remain operational for 450 days, the duty cycle needs to be lower than 1% [16], which corresponds to an active window of 150 milliseconds in every 15 seconds. Consequently, the delay in communicating with sleeping nodes can be as high as 14.85 seconds. Such a long latency is intolerable to mobile users that must maintain real-time context awareness in response to rapidly changing environments. In addition, a spatiotemporal query service faces the challenge of reducing storage cost and network contention caused by the continuous queries from mobile users since nodes in sensor networks typically have very limited memory and bandwidth. For example, MICA2 motes only have 4KB data memory and a 38.4 KBaud radio.

We make the following contributions in this paper. (1) We design a new spatiotemporal query service called *MobiQuery* that allows a mobile user to periodically query a surrounding area under spatiotemporal constraints (see Section 4). (2) We develop a novel just-in-time prefetching scheme that can meet stringent spatiotemporal constraints despite the long communication delays caused by low node duty cycles. Furthermore, our analysis shows that an advantage of just-in-time prefetching is that it significantly reduces storage cost and network contention caused by continuous queries from mobile users (see Section 5). (3) We validate the design of MobiQuery through extensive simulations with realistic settings. The results show that MobiQuery can deal with considerable location errors and inaccurate/late knowledge of user's motion. Over 85% of queries under MobiQuery achieve a data fidelity above 95% even when the user motion pattern changes every $70s$ and the location error is as large as $10m$ (see Section 6).

2 Related Work

Several data query service have been developed for sensor networks [9, 14, 20, 10]. Directed Diffusion [9] is a data-centric communication paradigm that allows for in-network data aggregation. TinyDB [14] is a energy-efficient query service for sensor networks. Unlike MobiQuery , both TinyDB and Directed Diffusion assume fixed query areas and are not designed to handle moving users or query areas. [10] shows that Directed Diffusion is not optimized for mobile users. TTDD [20] and SEAD [10] are data services that allow mobile users to collect data from fixed areas. TTDD builds a virtual grid to deliver the data to mobile sinks. SEAD maintains a routing tree for mobile users and uses data caches to balance latency and energy consumption. Unlike these protocols, MobiQuery is designed to query an area that moves with an user. Dealing with both user mobility and time-varying data sources introduces new challenges to the design of data services.

DCTC [22] is an object tracking protocol that maintains a tree around a moving target. DCTC wakes up the nodes ahead of the target based on motion prediction schemes. However, while DCTC only provides a best effort scheme for waking up nodes, MobiQuery achieves predictable spatiotemporal performance by waking up nodes just-in-time.

Our earlier work on Mobicast [8] is also related to this work. Mobicast is a spatiotemporal multicast protocol designed for disseminating data to a changing area just in time. Although both MobiQuery and Mobicast deal with spatiotemporal constraints in sensor networks, they differ in the following important aspects. First, they provide different types of data services. Mobicast deals only with disseminating (pushing) a message to an evolving area, while MobiQuery pulls data from a moving area to a mobile user in addition to disseminating the query. Second, Mobicast does not consider the duty cycles of nodes. In contrast, MobiQuery employs a novel data prefetching scheme to achieve spatiotemporal guarantees despite the long communication delays introduced by node duty cycles and hence can work effectively with existing power management protocols. Furthermore, we provide extensive analysis and simulations on storage cost, network contention and warmup cost introduced by node duty cycles. These issues were not systematically studied in previous work on Mobicast.

3 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 spatiotemporal query is specified with six parameters: $(\alpha,\ F,\ A(P_u(t)),\ T_{period},\ T_{fresh},\ T_d)$ where α is the type of sensor data being queried. F is an aggregation function that is applied to the results inside the network. In-network aggregation is a well-investigated technique utilized by existing data services [9, 14] to reduce bandwidth consumption. $A(P_u(t))$ is a function defining the query area relative to the user's position $P_u(t)$. $A(P_u(t))$ specifies a spatial constraint that requires all and only sensors within $A(P_u(t))$ to contribute to the query result. For simplicity, we assume $A(P_u(t))$ is a circle with radius R_q centered around the user in the rest of the paper, although our design can be easily extended to other types of query areas.

T_{period} and T_{fresh} define the temporal constraints of the query. T_{period} specifies the rate at which the user expects to receive query results. A new query result must be delivered to the user by the end of each period. T_{fresh} specifies the data freshness constraint, *i.e.*, a query result is acceptable only if it is aggregated from sensor readings no more than T_{fresh} seconds old. Hence the kth result must be received at or before $k \cdot T_{period}$, and the data in the result can be at most T_{fresh} old. T_d is the lifetime of the query.

1063-6927/05 $20.00 © 2005 **IEEE**

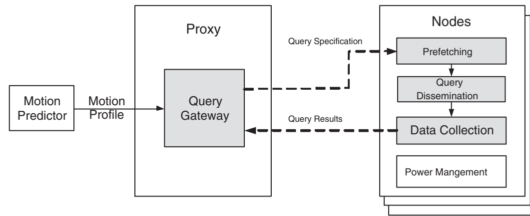

Figure 1. The architecture of MobiQuery (MobiQuery components are highlighted in gray)

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 a power management protocol that selects a small subset of nodes to keep *active* while the remaining nodes operate in a duty cycle. The active nodes form a *backbone* network that allows the communication delay between any two nodes to remain in the order of one duty cycle. A number of existing power management protocols like SPAN [3], CCP [17], and GAF [18] maintain such backbones [1].

4 System Architecture

As shown in Figure 1, MobiQuery is spread across two types of devices: a proxy and nodes of the sensor network. A proxy is a mobile device (such as a PDA or laptop) carried by the user as he moves through a sensor field. The query gateway in the proxy serves as the interface between the proxy and the nodes in the network. Three components of MobiQuery reside in the nodes: prefetching, query dissemination and data collection. In addition, MobiQuery also works with a motion predictor on the proxy and power management protocols on nodes.

Prefetching is a key component of our design that enables MobiQuery to meet the spatiotemporal constraints of a query despite the extremely low node duty cycles forced by power constraint of sensor networks. To understand why prefetching is necessary for spatiotemporal queries, we consider the following simple example. Assume nodes in a network keep active (turn on the radios) for $150ms$ in every 15 seconds. The user needs to query the nodes around him every 5 seconds. If the user disseminates the query to the network at the beginning of each query period, due to the sleep schedule, on average only $1/3$ nodes can be woken up in time to respond to the query by the end of the query period. Prefetching accounts for this by waking up the sleeping nodes at the right time to participate in the query.

[1]Our design and analysis can be extended to the case where no backbone is present and all nodes operate in a duty cycle. Only the analysis on the latency of query tree setup in Section 5.1 needs to be modified to consider the communication delay due to the duty cycle. For example, an analysis on the delay of S-MAC protocol is provided in [21].

A spatiotemporal query works as follows. The user issues the query through the query gateway which appends a motion profile (from the motion predictor) describing the predicted user path to the query and sends the aggregate to the network. The network then sends a *prefetch* message to forewarn nodes in future query areas (specified by the motion profile). Sleeping nodes receiving the prefetch message reconfigure their sleep schedules to wake up at the right time to participate in the query. Each node then takes a sensor reading before the query deadline and delivers the result to a collector node that sends the aggregated data to the user when he arrives.

In the rest of this section, we first discuss how motion profiles can be acquired in practice and then discuss the details of MobiQuery components.

4.1 Motion Profile

MobiQuery relies on motion profiles that specify the user's movement to predict future query areas and alert the nodes in them before the user arrives.

4.1.1 Generation of Motion Profiles

We discuss two ways of generating motion profiles.

Motion Predictor based on History. A motion profile can be generated based on the recent movement history of the user obtained from GPS in the proxy or a location service [12]. As a simple example, a motion profile P includes a velocity $\overrightarrow{v}$, assuming the user moves at $\overrightarrow{v}$ in future. $\overrightarrow{v}$ can be estimated based on two previous user positions (p_1, t_1) and (p_2, t_2), where p_i $(i = 1, 2)$ is the user position at time t_i. Let $\delta = t_2 - t_1$. Then $\overrightarrow{v}$ can be estimated as $\overrightarrow{v} = \frac{\overrightarrow{p_1 p_2}}{\delta}$. δ represents the sampling period of user positions. δ affects the accuracy of the motion prediction and is a system parameter of the motion predictor. Intuitively, when there are location errors in GPS readings (*e.g.*, p_1 and p_2), a small δ may result in high prediction error. The proxy periodically monitors the user's position and issues a new motion profile whenever the user diverges from the path predicted by the motion profile, by a system threshold. Our experiments (see Section 6.3) demonstrate that MobiQuery can achieve satisfactory performance when working with this simple and efficient motion prediction technique. We note that more complex techniques [1] can be used to improve the accuracy of the motion prediction. Further discussion on this topic is beyond the scope of this paper.

Motion Planner. Autonomous robots usually generate motion profiles based on motion planning [11] and control their future movement accordingly. In such cases, the motion planner can provide the motion profiles to MobiQuery . Hence MobiQuery does not introduce additional overhead for acquiring the motion profiles as they are already needed

1063-6927/05 $20.00 © 2005 IEEE

for the motion planning of the robot. We also note the motion profiles are generated by the proxy carried by the user, which typically has more processing power than the sensor network nodes.

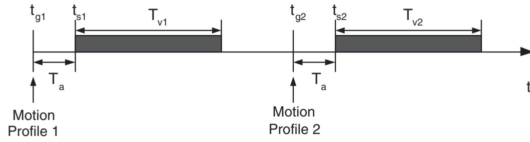

Figure 2. The motion profile model.

4.1.2 Motion Profile Model

Each motion profile P is associated with three timing parameters (t_s, T_v, t_g), as shown in Fig. 2. t_s specifies when P takes effect. T_v is the validity interval of P, *i.e.*, the user is predicted to travel according to P within $[t_s, t_s + T_v]$. t_g specifies when P is generated. Let $T_a = t_s - t_g$. T_a represents how early the proxy receives P before P takes effect and is referred to as the *advance time* of P.

The above model can accommodate both methods of generating P discussed in Section 4.1.1. When P is generated by a motion planner, T_a is positive since P is always created before the user takes the planned path. In contrast, when P is generated by a motion predictor, T_a is negative because the motion profile is available only after one sampling period (as discussed in Section 4.1.1). In this case, the part of P within $[t_s + T_a, t_s]$ $(T_a \leq 0)$ has expired by the time the proxy receives P. The sampling period is lower bounded by the delay in acquiring user locations. For example, obtaining an initial GPS reading takes about $8s$ [15]. The experiments on Leadtek GPS with Berkeley motes show that the lag of GPS readings is about $2 \sim 3s$ when the user walks briskly [6]. We evaluate the effect of T_a on the performance of MobiQuery in Section 6.3.

4.2 Prefetching

After receiving the query and the motion profile from the proxy, the network uses the motion profile to predict a location, called *pickup point*, where the user expects to receive the next query result. A *prefetch* message with the query specification and the motion profile is then relayed by the network to each future pickup point using an area anycast [7]. The area anycast delivers the prefetch message to a node within a certain distance, R_p, of the pickup point. To guarantee the delivery of the prefetch message, R_p may vary depending on the density of the sensor network. The node receiving the prefetch message is called a *collector node*. It is responsible for relaying the prefetch message to the next pickup point, distributing the query to the current query area, and aggregating the results in time for delivery. We present the following two prefetching schemes.

Greedy Prefetching. In greedy prefetching, each collector node forwards the prefetch message to the next collector node *immediately* after receiving the message. Consequently, the query is distributed to all future query areas along the predicted user path as soon as possible. One advantage of this approach is that it maximizes the slack time of following query areas in order to meet query deadlines.

Just-In-Time Prefetching. Different from greedy prefetching that forwards the prefetch message immediately, just-in-time prefetching holds the message for a certain amount of time before forwarding it to the next collector node. The right time to forward the prefetch message without violating the temporal constraints of the query is a key design parameter and will be analytically determined in Section 5.1.

Just-in-time prefetching has several key advantages over greedy prefetching. First, it reduces the likelihood of distributing query to multiple query areas concurrently, resulting in lower network contention. The network contention is formally analyzed in Section 5.4. Second, distributing the query to each query area just-in-time reduces the storage cost of query states. We formally analyze the storage cost in Section 5.2. Third, just-in-time prefetching can reduce the cost caused by motion changes of the user. When the user diverges from the path specified by the motion profile, the proxy can send a cancel message along the abandoned path to stop the previous prefetching process. However, this mechanism is not effective for greedy prefetching. Since the prefetch message is forwarded in an as-fast-as-possible fashion in greedy prefetching, due to which the query is distributed to most of the query areas on the abandoned path by the time a motion change occurs.

4.3 Query Dissemination

The query dissemination phase is responsible for distributing the query specification to a query area, and building a query tree rooted at the collector node for data collection. A query tree is built as follows. Upon receiving the prefetch message, the collector node floods a setup message with the query specification to all the nodes in the query area. Each backbone node sets the first node from which it receives the message as its parent. The backbone nodes deliver the message to the sleeping nodes when they wake up. Upon receiving the message, the sleeping nodes reconfigure their sleep schedule to wake up at $T_{period} - T_{fresh}$, which is the earliest time any node can perform a sensor measurement without violating the data freshness constraint. Sleeping nodes are purposely restricted to be leaves to allow them to quickly resume sleep after performing a single sensor reading and transmission. This minimizes their energy consumption. If a sleeping node is allowed to be parent node, it would have to remain awake for a long time in order to receive and route the results from its children.

4.4 Data Collection

Data collection is the process by which the nodes in a query area perform a measurement and send their data through the tree back to the collector node. To enable in-network aggregation and reduce contention, each parent waits for its children before sending the aggregated data. In case a child fails to respond, a parent times out at a sub-deadline and sends the data to its parent regardless of whether all children have responded. This sub-deadline must be chosen such that the results can still be delivered to the collector in time. We use the following heuristic to assign sub-deadlines. Leaf nodes set their deadline to $k \cdot T_{period} - T_{fresh}$ for the kth query. Each parent node u sets its deadline d_u as follows:

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

where $|up|$ is the distance between u and the collector node p. $R_p + R_q$ represents the maximum distance between a node in the query area and the collector node, since the collector node resides within R_p range of the pickup point. (1) ensures that 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 effective in-network aggregation and timely delivery of query results to the user.

We note that more sophisticated protocols [14, 9] could be used to disseminate the query and collect data. However, these protocols are designed for stationary users to query repeatedly and incur greater overhead. In contrast, each query area in MobiQuery is only queried once by the mobile user, which motivates the lightweight approach we adopted.

5 Analysis

In this section, we first derive the key design parameter of just-in-time prefetching – the right time a prefetch message should be forwarded in order to meet the spatiotemporal constraints. We then analyze several important practical issues which include storage cost, network contention and the warmup cost caused by low duty cycles.

5.1 Prefetch Forwarding Time

A collector node must receive the prefetch message early enough to ensure that query dissemination and data collection can be completed before the query deadline. We derive an upper bound on the time at which the prefetch message should be forwarded to the next pickup point such that the query deadline is met, which is referred as the *prefetch forwarding time*. We first make the following assumptions:

$$T_{collect} \leq T_{fresh} \tag{2}$$

$$T_{setup} \leq T_{fresh} \tag{3}$$

$$v_{user} < v_{prfh} \tag{4}$$

$T_{collect}$ is the time it takes the data collection process to complete. $T_{collect} \leq T_{fresh}$ is necessary for a network to meet both the freshness and the deadline constraints. Otherwise it is impossible for the network to deliver a query result to the user before the data becomes too old. This condition is enforced by the timeout scheme discussed in Section 4.4. Similarly, we assume $T_{setup} \leq T_{fresh}$ where T_{setup} is the time it takes to create the partial query tree composed of only backbone nodes in a query area. This assumption is made because T_{setup} is usually shorter than $T_{collect}$ due to the following two facts: (1) Unlike data collection, the query tree is set up as soon as possible and does not incur any in-network aggregation delay. (2) During T_{setup}, only backbone nodes communicate, involving fewer hops than in data collection.

v_{user} and v_{prfh} in (4) denote the user speed and the speed of the prefetch message, respectively. v_{prfh} is defined as the ratio of the distance between two consecutive collector nodes to the communication delay between them. Intuitively, (4) is necessary for the network communication to be able to catch up with the user movement.

Our goal is to derive the time $t_{send}(k-1)$ when the $(k-1)^{th}$ collector node should forward the prefetch message to the kth collector node such that the deadline of kth query result ($k \cdot T_{period}$) is met. We first derive the time (denoted by $t_{recv}(k)$) by when the kth collector node should receive the prefetch message in order to meet the deadline. Upon receiving the prefetch message, the kth collector node needs to set up the query tree and to collect the data before the deadline $k \cdot T_{period}$. Hence the query deadline will be met if the following inequality holds:

$$t_{recv}(k) \leq k \cdot T_{period} - T_{tree} - T_{collect} \tag{5}$$

where T_{tree} denotes the time it takes to create the query tree composed of all nodes in the query area. T_{tree} equals the sum of T_{setup} and the delay of waking up all sleeping nodes which is upper bounded by the sleep period T_{sleep}:

$$T_{tree} \leq T_{setup} + T_{sleep} \tag{6}$$

From (6) and (3), we have:

$$T_{tree} \leq T_{setup} + T_{sleep} \leq T_{fresh} + T_{sleep} \tag{7}$$

Based on (7), (2) and (5), the deadline $k \cdot T_{period}$ will be met if the following condition holds:

$$t_{recv}(k) \leq k \cdot T_{period} - T_{sleep} - 2 \cdot T_{fresh} \tag{8}$$

The time it takes the prefetch message to be transmitted between the two considered collector nodes is $\frac{v_{user} \cdot T_{period}}{v_{prfh}}$. Based on (4), $\frac{v_{user} \cdot T_{period}}{v_{prfh}} < T_{period}$. Hence, the prefetch

message will be received before $t_{recv}(k)$ by the kth collector node, if the following inequality holds:

$$t_{send}(k-1) \leq t_{recv}(k) - T_{period} \tag{9}$$

Based on (9) and (8), the deadline $k \cdot T_{period}$ will be met if the prefetch forwarding time of the $(k-1)^{th}$ collector node satisfies the following condition:

$$t_{send}(k-1) \leq (k-1) \cdot T_{period} - T_{sleep} - 2 \cdot T_{fresh} \tag{10}$$

A collector node in MobiQuery forwards the prefetch message to the next pickup point according to the upper bound of $t_{send}(k-1)$ in (10). Consequently, the prefetch message is forwarded between pickup points at an interval of T_{period}. As we show in the rest of this section, by delaying the forwarding of prefetch messages, MobiQuery can effectively reduce the storage cost and network contention caused by continuous queries from the mobile user.

5.2 Storage Cost

In this section, we analyze the storage cost of a query. The information related to a query that the network needs to remember depends the storage cost of a query tree (*e.g.*, parental information and query parameters), and the number of query trees set up ahead of the user by the prefetching process which we refer to as *prefetch length*. Since the storage cost of a tree is fixed for a given query, we focus on the analysis of prefetch length in the rest of this section.

Suppose the user moves at a constant velocity v_{user}. The query session lasts T_d seconds. We now derive the worst-case prefetch length under greedy prefetching and just-in-time prefetching, denoted by PL_{gp} and PL_{jit}, respectively. When greedy prefetching has set up all query trees in the query session, the number of pickup points the user has visited is $\left\lfloor \frac{v_{user} \cdot T_d}{v_{prfh} \cdot T_{period}} \right\rfloor$. Hence we have:

$$PL_{gp} = \left\lfloor \frac{T_d}{T_{period}} \right\rfloor - \left\lfloor \frac{T_d}{T_{period}} \cdot \frac{v_{user}}{v_{prfh}} \right\rfloor \tag{11}$$

(11) shows that the worst-case storage cost of greedy prefetching increases with the duration of the query. We now derive PL_{jit}. Let $t_{send}(k-1)$ denote the time instance when the $(k-1)^{th}$ collector node forwards the prefetch message to the kth collector node. At $t_{send}(k-1)$, the user is traveling between the ith and the $(i+1)^{th}$ pickup points where $i = \left\lfloor \frac{t_{send}(k-1)}{T_{period}} \right\rfloor$. All the query trees between the ith and the kth query areas have been set up. Hence we have:

$$PL_{jit} = k - \left\lfloor \frac{t_{send}(k-1)}{T_{period}} \right\rfloor = \left\lceil \frac{T_{sleep} + 2 \cdot T_{fresh}}{T_{period}} \right\rceil + 1 \tag{12}$$

where $t_{send}(k-1)$ takes the upper bound in (10). (12) shows that the storage cost of just-in-time prefetching is constant for given query parameters. From (11) and (12), it can be easily seen that $PL_{jit} < PL_{gp}$ when

$$T_d > \frac{T_{sleep} + 2 \cdot T_{fresh} + T_{period}}{1 - \frac{v_{user}}{v_{prfh}}} \tag{13}$$

where the rounding in (11) and (12) is ignored. All quantities in (13) are known for a given senor network and query specification except $\frac{user}{v_{prfh}}$.

As an quantitative estimation on $\frac{v_{user}}{v_{prfh}}$, let us consider the following simple example on MICA2 motes [5]. Suppose two consecutive collector nodes are $100m$ apart and there are 5 hops between them [2]. The size of a prefetch message is 60 bytes. The bandwidth of a mote is 38.4 Kbps[5]. To account for routing/MAC overhead and contention delay, we assume the effective bandwidth of a mote is 5 Kbps. Then v_{prfh} can be calculated as follows:

$$v_{prfh} = \frac{100m}{5 \times (60\,bytes \times 8)/5000\,bps} \times \frac{3600s}{1000 \times 1.6} \approx 469\,mph$$

Obviously, v_{prfh} is much larger than the velocity of a human or a vehicle. Hence, according to (13), greedy prefetching has a higher worst-case storage cost than just-in-time prefetch for queries with reasonable lifetime. We consider the following concrete example. The user is a human walking at $4m/s$, who issues a query every $10s$ in a duration of $600s$. The data freshness constraint is $5s$ and the sleep period of nodes is $15s$. Under these settings, the number of trees set up ahead of the user is 4 in just-in-time prefetching. However, it can be as high as 58 in greedy prefetching, which indicates a storage cost 14.5 times higher than in just-in-time prefetching. The analysis in this section shows that just-in-time prefetching is preferable since sensor nodes usually have very limited memory.

5.3 The Warmup Interval

The accuracy of motion profiles affects the effectiveness of prefetching. The user may diverge from a motion profile due to change of motion pattern (*e.g.*, turns, accelerations/decelerations) or error in the motion profiles. We refer to these cases as *unexpected motion changes*. The motion predictor issues a new motion profile whenever an unexpected motion change occurs. However, the first few queries after a new motion profile is issued may suffer from poor data fidelity because some sleeping nodes cannot be woken up before query deadlines due to delayed prefetching ((10) does not hold). In this case, MobiQuery attempts to catch up by using greedy prefetching and resumes just-in-time prefetching once (10) holds. We refer to this transient phase as *warmup interval*, denoted by T_w. We derive an upper bound on T_w. This bound quantifies the robustness of MobiQuery in presence of unexpected user motion changes.

[2] The radio range of a MICA2 mote is $1000ft$ in datasheets [5]. The actual range varies with the environment.

Suppose MobiQuery receives a new motion profile T_a seconds before the motion change occurs and the warmup interval lasts k query periods since the motion profile is issued. Let p_k denote the pickup point where the warmup interval ends. The user needs to travel a distance of $v_{user} \cdot (k \cdot T_{period} + T_a)$ to reach p_k from the point where he receives the motion profile. We approximate the locations of collector nodes with those of the corresponding pickup points. Then, the time it takes the prefetch message to reach the kth collector node, T_p, is as follows:

$$T_p = \frac{v_{user} \cdot (k \cdot T_{period} + T_a)}{v_{prfh}} \tag{14}$$

In the worst case, a query deadline in the warmup interval cannot be met, *i.e.*, the user reaches p_k before the completion of the data collection:

$$k \cdot T_{period} + T_a \leq T_p + T_{tree} + T_{fresh} \tag{15}$$

Solving k using (15), (14) and (7):

$$k \leq \left\lfloor \frac{T_{sleep} + 2T_{fresh} - (1 - \frac{v_{user}}{v_{prfh}}) \cdot T_a}{T_{period} \cdot (1 - \frac{v_{user}}{v_{prfh}})} \right\rfloor \tag{16}$$

Thus, $T_w = k \cdot T_{period}$. T_w becomes zero when $T_a = (2T_{fresh} + T_{sleep})/(1 - \frac{v_{user}}{v_{prfh}})$. That is, when the motion of the user can be predicted early enough, the query does not incur any warmup interval. In addition, since $v_{prfh} \gg v_{user}$ in practice as discussed in Section 5.2, $T_w \approx (T_{sleep} + 2T_{fresh} - T_a)$. This result is confirmed by the simulation results in Section 6.3.

5.4 Network Contention

In this section, we analyze the cause of network contention and show that just-in-time prefetching can effectively reduce network contention. Network contention may be caused by the communication with sleeping nodes during query dissemination since it must occur within short active windows. Moreover, due to low node duty cycles, the setup of a query tree may last multiple query periods resulting in interference among adjacent query areas. Consequently, a query may suffer from packet loss or deadline misses. In contrast, data collection incurs lower contention since it completes within the current period (to meet the freshness constraint) avoiding interference with other query areas. Hence, we focus on analyzing the network contention caused by the tree setups in query dissemination.

Since the contention level caused by a single tree is fixed, in order to quantify the network contention, it suffices to analyze how many trees may interfere with a tree T during its setup, which is referred to as *interference length*. We now derive the interference length of greedy prefetching and just-in-time prefetching, denoted by M_{gp} and M_{jit}, respectively. To simplify our analysis, we assume that the nodes in the network have a communication range of R_c meters (note that the design of MobiQuery does not depend on this assumption). We also assume the user moves in a straight line. As shown in Fig. 3, for a tree to interfere with T, its root must be located within $2R_q + R_c$ meters from the root of T. Let M_s denote the number of such trees. We approximate the locations of collector nodes with those of corresponding pickup points. Since the distance between two consecutive pickup points is $v_{user} \cdot T_{period}$, we have:

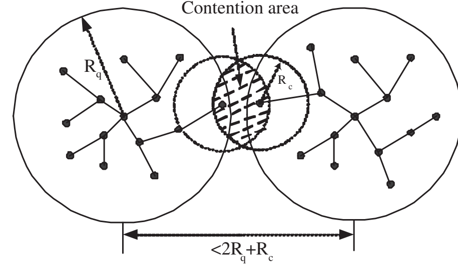

Figure 3. Two trees interfere with each other

$$M_s = \left\lceil \frac{4R_q + 2R_c}{v_{user} \cdot T_{period}} \right\rceil \tag{17}$$

Meanwhile, the setup of an interfering tree must overlap with the setup of T. Let M_{t-jit} and M_{t-gp} denote the number of such trees under two prefetching schemes. Clearly, $M_{gp} = \min(M_{t-gp}, M_s)$ and $M_{jit} = \min(M_{t-jit}, M_s)$. A prefetch message in greedy prefetching takes $\Delta_t = \frac{v_{user} \cdot T_{period}}{v_{prfh}}$ to travel between two consecutive pickup points, *i.e.*, trees are set up at an interval of Δ_t. We have:

$$M_{t-gp} \leq \left\lceil \frac{T_{tree}}{\Delta_t} \right\rceil \leq \left\lceil \frac{(T_{sleep} + T_{fresh}) \cdot v_{prfh}}{T_{period} \cdot v_{user}} \right\rceil \tag{18}$$

On the other hand, the query trees are set up in just-in-time prefetching at an interval of T_{period}. Hence $M_{t-jit} = \left\lceil \frac{T_{tree}}{T_{period}} \right\rceil$. The following cases can be derived on the relationship between M_{gp} and M_{jit} (rounding is ignored):

$$\begin{cases} M_{jit} = M_s = M_{gp}, & v_{prfh} > v_{user} > v^* \\ M_{jit} = M_{t-jit} < M_s = M_{gp} & v_{prfh} > v^* > v_{user} \\ M_{jit} = M_{t-jit} < M_{t-gp} = M_{gp}, & v^* > v_{prfh} > v_{user} \end{cases}$$

where $v^* = \frac{2Rc + 4R_q}{T_{sleep} + T_{fresh}}$. We can see that the network contention of just-in-time prefetching is lower than that of greedy prefetching as long as the user speed remains below v^*. On the other hand, the contention level of the two prefetching schemes becomes the same when the user speed exceeds v^*. This is because the prefetch messages under just-in-time prefetching must be forwarded very quickly in such a case, resulting in a behavior similar to greedy prefetching. As a concrete example, when $R_c = 50m$ and the query radius is $150m$, v^* approximates 131 mph if the sleep period is $9s$ and data freshness constraint is $3s$. We note the user is unlikely to travel at such a high speed in practice. In contrast, if the user is a human walking at a speed of $4m/s$, and a query is issued every $5s$, the number

of interfering trees is about 4 under just-in-time prefetching while it is 35 under greedy prefetching! The analysis in this section clearly shows that just-in-time prefetching can significantly reduce the network contention.

6 Simulation Results

We implemented MobiQuery in ns2. Coverage Configuration Protocol (CCP) [17] based on IEEE 802.11 Power Saving Mode (PSM) extended by [3] is used as the power management protocol. CCP maintains network connectivity and sensing coverage through a backbone. MobiQuery has also been implemented on MICA2 motes and demonstrated at a conference [2]. In this paper, we only present the simulation results. Evaluating the performance of MobiQuery on a mote testbed in real-world environment is our ongoing work and is not included here.

We use the following metrics in our performance evaluation: (1) Data fidelity, defined as the ratio of the number of nodes that contribute to a query result to the total number of nodes in a query area. (2) Success ratio, defined as the ratio of the number of queries that meet deadlines and have data fidelity above a threshold, to the total number of queries. Success ratio indicates the overall quality of service received by the user. We set the threshold of data fidelity to 95%. (3) Power consumption. We measure the average power consumption per sleeping node during a query.

6.1 Experimental Settings

In each simulation, 200 nodes are randomly distributed in a $450m \times 450m$ region. The active window in IEEE 802.11 PSM is $100ms$. The sleep period varies from $3s$ to $15s$, which results in duty cycles from 3.2% to 0.67% for sleeping nodes. The radius of a query area is $150m$. The communication and sensing range in CCP are set to $105m$ and $50m$, respectively. Under these settings, a query tree has about $2 \sim 4$ levels. Query period is $2s$ and data freshness constraint is $1s$. The node bandwidth is 2 Mbps.

6.2 Performance under Accurate Motion Profiles

In this section, we evaluate the performance of MobiQuery with the accurate knowledge about the user's motion. In each simulation of $400s$, the user starts from a corner of the region and moves in a random direction with a speed randomly chosen from a range. The user changes its direction and speed every 50 seconds. We simulate users moving at three speed ranges, $3 \sim 5m/s$, $6 \sim 10m/s$ and $16 \sim 20m/s$, corresponding to a walking human, a running human and a vehicle with moderate speed, respectively. The motion profile that specifies the complete user path is provided to MobiQuery at the beginning of each simulation.

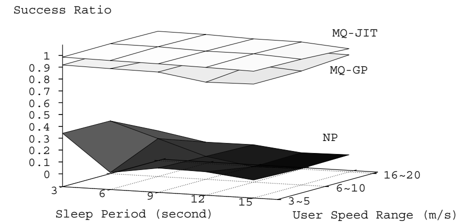

Figure 4. Performance comparison

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. Fig 4 shows the average success ratios MobiQuery and NP over 3 runs with different network topologies. The implementations of MobiQuery with greedy and just-in-time prefetching are denoted as MQ-GP and MQ-JIT, respectively. Fig. (4) shows that the success ratio of MQ-JIT achieves near 100% in all settings, even when the sleeping period is as long as $15s$, *i.e.*, 7.5 times the query period. In sharp contrast, the success ratio of NP remains below 35% and decreases when the sleep period or user speed increases. This result indicates that prefetching is crucial and highly effective to meet spatiotemporal constraints in sensor networks with low duty cycles. The success ratio of MQ-GP reaches about 90% when sleep period is shorter than $9s$ and decreases when sleep period becomes longer. MQ-GP performs consistently *worse* than MQ-JIT due to packet loss caused by high network congestion.

To examine the dynamic behavior of MobiQuery , we plot the data fidelity at each pickup point in Fig. 5. The sleep period is $15s$ and the user speed is $3 \sim 5m/s$. Both MQ-JIT and MQ-GP suffer from an initial warmup phase in which about 5 queries have relatively low data fidelity, which conforms to the prediction of (16) in Section 5.3. MQ-JIT achieves a data fidelity of 100% in most periods after the warmup phase. In contrast, the performance of MQ-GP incurs a significant variance due to packet loss caused by network congestion, which conforms to our analysis on network contention in Section 5.4. This behavior of MQ-GP makes it inappropriate for critical applications that require reliable performance. Our results also show that the latency of query results under MQ-GP also has a high variance, although all query deadlines are met (at the price of low data fidelity) due to the timeout scheme discussed in Section 4.4. These results are not shown due to space limitation.

6.3 Effects of Imperfect Motion Prediction

In this section, we evaluate the impact of motion profile settings including advance times, unexpected motion changes, and location errors. The results are the average of 5 runs (with 95% confidence interval) under different network topologies.

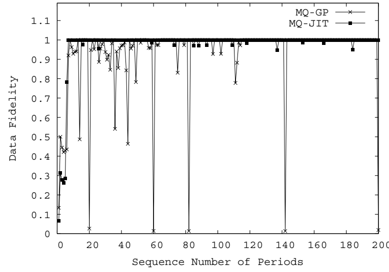

Figure 5. Dynamic behavior of MobiQuery

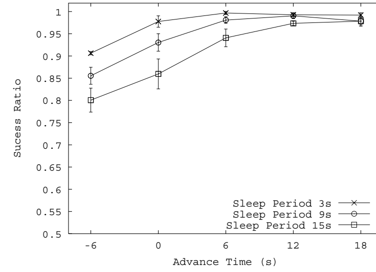

Figure 6. Success ratio vs. advance times

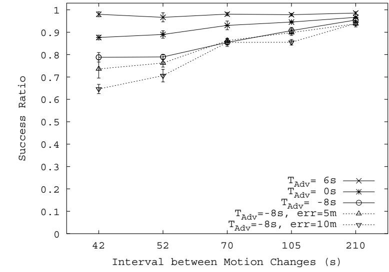

Figure 7. Success ratio with motion changes and location errors

Advance Time: We first test the effect of the advance time of motion profiles. The user changes the direction and speed every $70s$ in a simulation of $500s$. The range of the user's speed is $3 \sim 5m/s$. MobiQuery receives a new motion profile T_a seconds before a motion change occurs [3]. We vary T_a from $-6s$ to $18s$ in the simulations, modeling different motion profile generation scheme, *e.g.*, from a motion planner or a history-based motion predictor. Fig. 6 shows the success ratio of MQ-JIT under different sleep periods. When the sleep period decreases, the warmup interval becomes shorter and hence more queries succeed. For each sleep period, the success ratio increases with T_a. As predicted by (16) in Section 5.3, the warmup interval approaches zero when T_a reaches a threshold (*e.g.*, about 11 seconds for a sleep period of 9 seconds). In such a case, the success ratio converges to a level close to 100%. The difference with 100% is due to the occasional packet loss and initial warmup interval at the beginning of a simulation.

Unexpected Motion Changes: We now evaluate the effect of motion changes. The motion pattern of the user is the same as described earlier except that the interval between motion changes varies from $42s$ to $210s$. Fig. 7 shows the success ratio of MQ-JIT with sleep period of $9s$. As expected, motion changes have no impact on MQ-JIT when T_a is large. When T_a is negative, the performance drops due to longer warmup intervals, as the user changes the motion more frequently. However, MQ-JIT still achieves satisfactory performance even when the user changes motion very frequently. For example, MQ-JIT returns about 78.8% of the requested results even when the user changes his motion pattern every $42s$ (even though such a high frequency of motion changes is unlikely in practice).

Location Errors: We now evaluate the effect of location errors. Every time a motion change occurs, the motion predictor generates a new motion profile based on two GPS readings, as discussed in Section 4.1.1. The sampling period is $8s$. That is, a new motion profile is provided to MQ-JIT $8s$ after a motion change occurs. Each GPS reading has a random location error within $0 \sim \Delta$ meters. Δ takes $5m$ or $10m$, modeling the typical accuracy of GPS with/without differential correction, respectively [4]. As shown in Fig. 7 (dotted curves), MQ-JIT performs slightly worse when the location error increases. Furthermore, in both error settings, the success ratio increases as the user's motion changes less frequently and approaches the performance without location error when the interval between motion changes exceeds $70s$. For example, when the location error is $10m$, over 85% of queries succeed even when the user changes his motion pattern every $70s$. This result indicates that MobiQuery can work with practical motion prediction techniques and achieve satisfactory performance despite considerable delays and location errors in the motion profiles.

In summary, our results in this section demonstrate the robustness of MQ-JIT when operating with inaccurate/late motion profiles. MQ-JIT can take advantage of a small advance time of motion profiles to maintain perfect spatiotemporal services. Moreover, it can also tolerate motion prediction techniques with considerable delays, location errors, as well as high frequency of motion changes by the user.

6.4 Power Consumption

In MobiQuery, active nodes never turn off their radios before running out of energy and hence their power consumption is not effected by the settings of sleep schedules. In this section, we investigate the power consumption of sleeping nodes. The user changes its motion every $70s$ in a simulation of $400s$. For comparison, we also measured the power consumption of CCP without any query. In accordance with the measurement of Cabletron 802.11 network card in [3], the power consumption of transmitting, receiving, idle and sleeping modes of the radio are 1400mW, 1000mW, 830mW and 130mW, respectively.

[3]Recall that, when T_a is negative, the motion profile is provided to MobiQuery after a motion change occurs.

1063-6927/05 $20.00 © 2005 **IEEE**

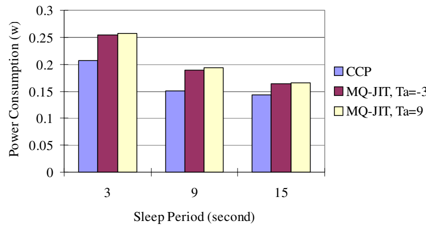

Figure 8. Power consumption per sleep node

Fig. 8 shows the average power consumption per sleeping node of 5 runs. As expected, both CCP and MQ-JIT consume less power as the sleep period increases. The increase in power consumption due to MobiQuery remains below $0.05w$ under all settings. MobiQuery consumes slightly less power when T_a decrease from $9s$ to $-3s$. In the latter case, as analyzed in section 5.3, MobiQuery incurs a warmup interval for each motion change and hence fewer nodes were woken up to participate in the query, resulting in lower energy consumption. The result of this section indicates that MobiQuery conserves the power consumption of the network by taking advantage of the sleep schedule.

7 Conclusions

We presented a novel spatiotemporal query service called MobiQuery in this paper. A key feature of MobiQuery is its analytical performance guarantees under spatiotemporal constraints despite a set of unique challenges in sensor networks including 1) extremely low duty cycles for conserving energy, 2) high storage cost and network contention due to continuous queries from mobile users, and 3) erroneous/late knowledge about user motion. Our results demonstrate that just-in-time prefetching enables MobiQuery to maintain desired spatiotemporal performance under various network and motion settings. The results also show that MobiQuery can tolerate imperfect motion profiles with considerable delays and location errors.

Acknowledgement

This work is funded in part by the NSF under an ITR grant CCR-0325529 and the ONR under MURI research contract N00014-02-1-0715. We thank the reviewers for their valuable feedback.

References

[1] A. R. Aljadhai and T. Znati. Predictive mobility support for qos provisioning in mobile wireless environments. *JSAC*, 19(10), October 2001.

[2] S. Bhattacharya, O. Chipara, B. Harris, C. Lu, G. Xing, and C.-L. Fok. Demo abstract: Mobiquery - a spatiotemporal data service for sensor networks. In *SenSys*, 2004.

[3] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. In *MobiCom*, 2001.

[4] D. Cooksey. Understanding the global positioning system (gps). Online Report, Montana State University-Bozeman.

[5] Crossbow. Mica2 wireless measurement system datasheet. 2003.

[6] Firebug-Project. http://firebug.sourceforge.net/gps_tests.htm.

[7] T. He, J. A. Stankovic, C. Lu, and T. Abdelzaher. Speed: A stateless protocol for real-time communications in sensor networks. In *ICDCS*, 2003.

[8] Q. Huang, C. Lu, and G.-C. Roman. Spatiotemporal multicast in sensor networks. In *Sensys*, 2003.

[9] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom*, 2000.

[10] H. S. Kim, T. F. Abdelzaher, and W. H. Kwon. Minimum-energy asynchronous dissemination to mobile sinks in wireless sensor networks. In *Sensys*, 2003.

[11] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[12] J. Li, J. Jannotti, D. S. J. D. Couto, D. R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *MobiCom*, 2000.

[13] Q. Li, M. D. Rosa, and D. Rus. Distributed algorithms for guiding navigation across a sensor network. In *MobiCom*, 2003.

[14] S. Madden, M. J. Franklin, , J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[15] Papyrus-Computer-Technologies-Ltd. Gps specification 2004.

[16] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler. The mote revolution: Low power wireless sensor network devices. In *Hot Chips 16*, 2004.

[17] X. Wang, G. Xing, Y. Zhang, C. Lu, R. Pless, and C. D. Gill. Integrated coverage and connectivity configuration in wireless sensor networks. In *Sensys*, 2003.

[18] Y. Xu, J. Heidemann, and D. Estrin. Geography-informed energy conservation for ad hoc routing. In *MobiCom*, 2001.

[19] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(2):9–18, 2002.

[20] F. Ye, H. Luo, J. Cheng, S. Lu, and L. Zhang. A two-tier data dissemination model for large-scale wireless sensor networks. In *MobiCom*, pages 148–159, 2002.

[21] W. Ye, J. Heidemann, and D. Estrin. Medium access control with coordinated, adaptive sleeping for wireless sensor networks. *IEEE/ACM Transactions on Networking*, June 2004.

[22] W. Zhang and G. Cao. Optimizing tree reconfiguration for mobile target tracking in sensor networks. In *INFOCOM*, March 2004.