Processing Range-Monitoring Queries on Heterogeneous Mobile Objects

Ying Cai
Department of Computer Science
Iowa State University
Ames, IA 50011, U.S.A
E-mail: yingcai@cs.iastate.edu

Kien A. Hua
Computer Science Program, SEECS
Univeristy of Central Florida
Orlando, FL 32816, U.S.A
E-mail: kienhua@cs.ucf.edu

Guohong Cao
Department. of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802-6106, U.S.A
E-mail: gcao@cse.psu.edu

Abstract

We consider in this paper how to leverage heterogeneous mobile computing capability for efficient processing of real-time range-monitoring queries. In our environment, each mobile object is associated with a resident domain and when an object moves, it monitors its spatial relationship with its resident domain and the monitoring areas inside it. An object reports its location to server whenever its movement affects any query results (i.e., crossing any query boundaries) or it moves out of its resident domain. In the first case, the server updates the affected query results accordingly while in the second case, the server determines a new resident domain for the object. This distributive approach is able to provide accurate query results and real-time monitoring updates with minimal location update and server processing costs. In addition, the new scheme allows a mobile object to negotiate a resident domain based on its computing capability. Thus, a more capable object can have a larger resident domain reducing its chance of having to request a new resident domain because of moving out of it. This feature makes the new approach highly adaptive to the heterogeneity of mobile objects. In our performance study, we compare it with an existing approach using simulation. The study shows that the new technique is many times better in reducing mobile communication and server processing costs.

1. Introduction

Consider a computing environment with a large number of location-aware mobile objects. We want to retrieve the mobile objects inside a set of user-defined spatial regions and continuously monitor the population of these windows over a time period. In this paper, we refer to such continuous queries as *range-monitoring queries*. Efficient processing of range-monitoring queries could enable many useful applications. For instances, a teacher, on a field trip, can monitor several groups of children in different rooms of a museum; a pilot, as part of the preparation for landing, might want to be informed of other airplanes in the airport area for the next several minutes; a restaurant might want to know about people in its vicinity during lunch hours in order to send advertising messages; in digital battlefields, a commander might want to continuously monitor the imminent bombing areas and alert the friendly vehicles within those regions; similarly, we might want to track traffic condition in some area and dispatch more police to the region if the number of vehicles inside exceeds a certain threshold. In such applications, it is highly desirable and sometime critical to provide accurate results and update them in real time whenever mobile objects enter or exit the regions of interest.

Unlike conventional range queries, a range-monitoring query is a continuous query. It stays active until it is terminated explicitly by the user. As objects continue to move, the query results change accordingly and require continuous updates. A simple strategy for computing range-monitoring queries is to have each object report its position as it moves. The server uses this information to identify the affected queries, and updates their results accordingly. This simple approach requires excessive location updates, and obviously is not scalable. Each location update consists of two expenses - mobile communication cost and server processing cost. If a battery-powered object has to constantly report its location, the battery would be exhausted very quickly. It is well-known that sending a wireless message consumes substantially more energy than running simple procedures [1] [2] [3]. Furthermore, when the number of mobile devices is large, the server would experience a severe communication bottleneck and an overwhelming workload of determining the affected queries and updating their results.

To avoid excessive location updates, Prabhakar *et al* proposed in [4] and [5] a *Q-index* approach, in which each mobile object is assigned a *safe region*. A safe region is either a circular or a rectangular region that does not overlap with the boundaries of any query. These two types of safe regions are illustrated in Figure 1. It shows a mobile object A with its maximum rectangular and circular safe regions. Since there is no query boundary inside a safe region for the mobile object to cross over, its movement within this region cannot affect any query result. This property allows a mobile object to report its location only when it exits its current safe region. Unfortunately, determining a safe region requires intensive computation. For example, computing a rectangular safe region takes from $O(n)$ to $O(nlog^3n)$, where n is the number of queries [4]. Under this scheme, whenever a mobile object moves out of its current safe region, the server has to determine it a new safe region. The problem is even worse considering that adding a new query requires to re-compute safe regions for all mobile objects because the new query rectangle could affect all existing safe regions. With these limitations, it is unlikely that this approach can be used in a large scale real-time mobile system.

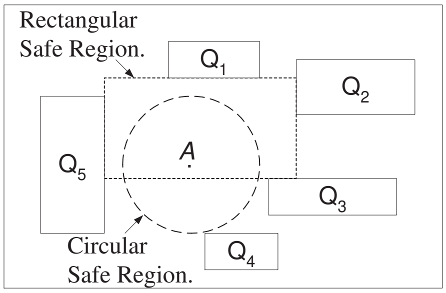

Figure 1. Examples of Safe Regions

In this paper, we address the aforementioned problem by proposing a scalable and adaptive technique for real-time processing of range-monitoring queries. Our goal is to support a location-based service capable of handling a large number of mobile objects and monitoring queries. Our strategy is to make each object aware of its nearby monitoring queries. When an object moves, it monitors its spatial relationships with these query regions, instead of its safe region as proposed in [4] and [5]; and informs server to update the affected query results when it enters or exits a query region. This distributive approach minimizes the expensive location updates and at the same time, it is able to provide accurate query results and update them in real time. In addition, allowing mobile objects to monitor the query regions directly relieves the server from the overwhelming workload of query evaluation. To make this strategy work, however, there are two challenges:

1. While an object moves, its nearby monitoring queries change continuously. The object must be updated with the correct information, or the query results will be inaccurate.
2. Mobile devices typically have vastly different computing capability in terms of CPU speed and memory capacity. Thus, the number of queries that they can carry and monitor at one time can be quite different. Assuming all of them have only the same processing power is unfair to the more capable ones because caching more queries allows them to contact the server less frequently for the updates of their nearby query regions.

To address these problems, we partition the database map into many disjoint subdomains and each range-monitoring query is handled as a number of adjacent monitoring regions, each contained by one subdomain. Each subdomain is allowed to contain no more than a certain number of monitoring regions, which is dictated by the least capable mobile objects, or it is recursively partitioned. Each mobile object is dynamically assigned a rectangular area, which consists of one or more subdomains, as its *resident domain*. A resident domain is selected such that the number of monitoring regions inside it does not exceed the processing capability of its host object. We have developed a new spatial access method, called *BP-tree* (Binary Partitioning tree), to manage the domain decomposition hierarchy and the monitoring regions. This index scheme allows us to efficiently determine a suitable resident domain for an object, according to its computing capability and its current location. At any one time, each mobile object caches the monitoring regions inside its own resident domain and evaluates them while it moves. When an object moves out of its current resident domain, the server searches the BP-tree and assigns it a new resident domain. Our simulation results indicate that the proposed technique is many times better than an improved version of Q-index in terms of communication and server processing costs.

This paper is a major extension of our previous work reported in [6], in which mobile objects are assumed to have the same processing capability. The technique proposed in this paper aims at effectively leveraging the heterogeneity of mobile objects for more efficient processing of range-monitoring queries. We call the proposed method *Monitoring Query Management* (MQM) technique. We note that conventional database management systems are designed to manage data, not queries. Since range-monitoring queries are continuous queries, many can be active simultaneously. Existing database management systems need to be extended with real-time query management capability in order to support range-monitoring queries. The research reported in this paper could be viewed as a step toward enhancing databases with such functionality in order to support mobile applications.

The remainder of this paper proceeds as follows. We discuss some related techniques in more detail in Section 2. Our technique, MQM, for real-time monitoring queries is presented in Section 3. In Section 4, we introduce the BP-tree indexing technique. The performance results are examined in Section 5. Finally, we give our concluding remarks in Section 6.

2. Related Works

Range queries have been studied extensively in geographic information systems (GIS), computer aided design (CAD), and other conventional database systems. In these systems, updates are infrequent and the database is relatively stable. In contrast, data in a mobile objects database change continuously. Maintaining such information in real time is a major challenge in designing location management systems. To capture the movement of mobile objects, three different strategies for initiating location updates were studied in [7]:

- *Time-based* Strategy: The stored location for each mobile object is updated periodically.
- *Movement-based* Strategy: The stored location is updated after the object has performed a predefined number of movements.
- *Distance-based* Strategy: The stored location is updated when the distance of the stored location from the actual location of the mobile object exceeds a predefined threshold.

In the above strategies, the location database is fixed until it is explicitly updated by the moving objects. To better estimate the actual location of mobile objects, the velocity of objects can be modeled using a linear function of time $f(t)$. Then an object does not have to report its position unless the deviation between its actual location and the *computed* location exceeds some threshold. Obviously, there is an inherent trade-off between the update cost and imprecision. A cost model was presented in [8] [9] to determine the update threshold for a desired precision. In addition, [9] proposed a set of *dead-reckoning policies* for dynamic adjustment of the update threshold. Using variable threshold was also considered in [10], in which an object can report its location less frequently in some time period if its movement is unlikely to affect any query result during this time interval.

At the server side, the continuous movements of mobile objects are usually approximated as many linear segments. To support range query operation, many approaches have been proposed to index the trajectory lines of each object. For examples, [11] proposes to transform line trajectories into points in a higher-dimensional space, and then index these points using regular spatial indices. In [12], the trajectories are indexed through their bounding rectangles that are time-parameterized. This indexing method was called TPR-tree. Another indexing scheme using external range trees [13] was presented in [14]. Performing range queries over historical data has also been discussed in [15] and [16]. Other recent works on supporting spatio-temporal data can be found in [17], [18], [19], and [20].

Although the aforementioned techniques can efficiently determine mobile objects inside a query window at some time point or within a certain time interval, they cannot be carried over to process range-monitoring queries. These schemes demand the server to do essentially all the work. It has to update the location database at a very high rate; and for each update, determine the affected queries and update their result. Such a scheme incurs excessive server load and can unlikely satisfy the real-time requirement. A server centric approach is also very demanding on client communications since clients need to report their location frequently. Although location estimation can be used to reduce the update frequency, this solution is only effective when the behavior of mobile objects does not change frequently. For instance, when a traffic light turns green, cars at the intersection can suddenly accelerates in any one of three possible directions. No estimation technique can reliably approximate the new locations in this case. It is also difficult to estimate locations when cars have to make unpredictable frequent stops in a downtown area. In such applications, frequent updates are inevitable to correct the estimation errors. In addition, using location estimation can provide only approximate query results because the location estimated at the server side using some velocity model typically has some deviation from the true position of the object[1]. Obviously, such an approach is not suitable for mission critical applications. The technique we present in this paper is more robust in the sense that it can be used for all location management applications.

Another related work is the location management in cellular communication systems [21] [22] [23] [24]. A cellular network divides the whole service area into a collection of *cells*. Each cell has a *base station* which communicates with the mobile objects in the cell. These base stations are interconnected with high-speed backbone networks that serve as the bridges for the mobile users to communicate with each other. As a mobile device moves from one cell to another, its point of attachment to the fixed network changes. Therefore, a central issue of the cellular networks is how to store, query, and update the information about the cells in which mobile objects reside. At one extreme, the location information of all users is replicated and maintained at all network sites. In this case, any one of these databases can be queried to locate a user. However, each time an object moves into a different cell, the location databases at all network sites must be updated. At the other extreme, location information is not maintained, and therefore no update cost. Locating a mobile user, however, requires paging the entire service area. To balance the lookup and update costs, many efforts have been done on various aspects of such location databases, such as architecture, placement and optimization, cache and replication, and so on. A comprehensive survey on this topic can be found in [25]. Although monitoring cellular-phone users within a fixed cell can be viewed as a *static range-monitoring query*, the above techniques cannot handle the ad hoc range-monitoring queries discussed in this paper.

3. Proposed Technique: MQM

In this section, we present our *Monitoring Query Management* (MQM) technique for scalable processing of range-monitoring queries. In our discussion, each range query is represented by a rectangular region. We will call a range query as a region provided there is no risk of confusion. Without loss of generality, we assume that there is only one server. As in many mobile environments, we also assume that each mobile device has very limited computing resources in terms of CPU speed and memory capacity, and each is able to exchange information with a stationary server such as reporting its current position and so on. The communication between server and mobile devices are through regular wireless broadcast. In practice, more efficient protocols such as *GeoCast* [26] [27] can be used for sending messages to mobile devices within a specific geographic region.

In MQM, the entire domain space is dynamically partitioned into a set of disjoint subdomains. Figure 2 shows an example of such partitioning. When a query overlaps with a subdomain, the overlapping area is called a *monitoring region* inside the subdomain and the query is a *relevant query* to the monitoring region. A query could create more than one monitoring region if it spans over more than one subdomain. For example, Q_1 makes only one monitoring region, R_1, while Q_2 has two monitoring regions, R_{21} and R_{22}. On the other hand, a monitoring region can have multiple relevant queries if these queries overlap the same area in a subdomain. For example, both Q_3 and Q_4 are relevant to the monitoring region R_{32}.

At any one time, each mobile object is associated with a *resident domain*, a rectangular region containing the object's current position. A resident domain consists of one or more subdomains and contains no more than the maximum number of monitoring regions that is acceptable to its hosting object. In this paper, we measure the computing capability of a mobile object by the maximum number of monitoring regions it can load and process at one time. When an object exits its resident domain, the object reports its new location to the server. In response, the server determines a new resident domain and informs the object of the monitoring regions inside it. We note that by caching as many monitoring regions as possible, the object can have a maximum size of resident domain and that minimizes its chance of having to request for a new resident domain. As an object moves about its resident domain, it monitors its spatial relationships with the monitoring regions inside it. When it enters or exits these regions, the object reports the server, which will then update the affected query results accordingly.

3.1. Server Design

At the server side, the subdomains and the monitoring regions are maintained using an index structure called BP-tree (*Binary Partitioning tree*). The details of BP-tree will be presented in the next section. In addition, we use a binary relation, called *Relevance Table*, to track the queries and their monitoring regions. We recall that a query is considered *relevant* to a monitoring region if the query contains the monitoring region. Each row of the Relevance Table is a tuple of (r, q), where r is a monitoring region and q is a query relevant to r. Many access structures can be used to retrieve the relevant queries efficiently given a monitoring region. For example, we can hash or build a B$^+$-tree index on the monitoring-region field. Alternatively, we can also store the entire information in an adjacency matrix in-

1 In this paper, we ignore the small inaccuracy caused by positioning systems. Thus, each object knows its $true$ position.

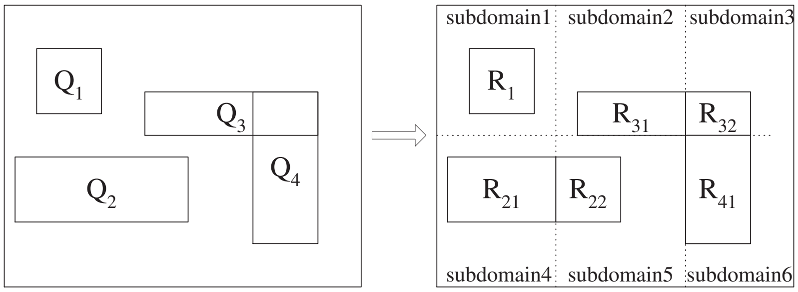

Figure 2. An Example of Domain and Query Decomposition

stead of a relational table. In the remainder of this paper, we will refer to this structure as a table and will not concern ourselves with its implementation details.

When a new range query q is submitted, the server searches the BP-tree for the subdomains it overlaps. For each of such subdomains, it determines the overlapping area, i.e., the monitoring region of this query inside this subdomain. The server then inserts a new tuple, (r, q), to the relevance table, where r is the monitoring region. If the monitoring region does not already exist, it is also inserted to the BP-tree and the server broadcasts a message $AddMonitoringRegion(r)$ to inform the mobile units that a new monitoring region is created. We will discuss how mobile units respond to server messages shortly. We allow each subdomain to contain only a limited number of monitoring regions, determined by the minimum computing capability of mobile devices. When the number of monitoring regions in a subdomain exceeds a predetermined *split threshold*, the subdomain, say d, is further partitioned into two subdomains d_1 and d_2. When this happens, the server broadcasts a $SplitDomain(d, d_1, d_2)$ message to update the affected mobile objects.

When a query q is terminated, the server searches the relevance table and deletes all tuples containing q as the relevant query. If a tuple, say (r, q), is deleted, and no other tuples in the table contain monitoring region r, then r is also deleted from the BP-tree. In this case, the server broadcasts a message $DeleteMonitoringRegion(r)$. Deleting a monitoring region might cause a subdomain to underflow. To prevent sparse subdomains, we merge a subdomain with its split counterpart if the aggregate number of their monitoring regions drops below a predetermined *merge threshold*. In this case, the server broadcasts the message $MergeDomain(d_1, d_2, l)$, where d_1 and d_2 are the two merging subdomains, and l is the combined list of monitoring regions inside d_1 and d_2.

We assume that each mobile object is identified by a unique identifier. The server expects two types of messages from the mobile units, and processes them as follows:

- When an object oid enters or exits a monitoring region r, it sends server a message $UpdateQueryResult(r, oid, p)$, where p is the current position of the object. In response, the server searches the table for all queries that are relevant to this monitoring region. If a relevant query contains position p, then the object is inside the query region and oid should be in the query result. Otherwise, delete oid from the query result.
- When an object oid initializes itself or exits its current resident domain, it sends a message $RequestResidentDomain(oid, p, n)$ to inquire its new resident domain, where p is the current position of the mobile object and n is the maximum number of monitoring regions it can accept. In response to this inquiry, the server searches the BP-tree to determine a resident domain for the mobile object. The server then broadcasts the message $SetResidentDomain(oid, d, l)$, where d and l denote the new resident domain of the object oid and the list of monitoring regions inside d, respectively.

3.2. Mobile Unit Design

The design of a mobile device consists of three main components: *Initialization*, *MessageListener*, and *RegionMonitor*. The following notations are used in the discussion of these components:

- *myID* : the unique identifier of the mobile unit.
- *myPos* : the current position of the mobile unit.
- *myDomain* : the current resident domain of the mobile unit.
- *myMRs* : the list of monitoring regions inside $myDomain$.
- *myCapacity* : the maximum number of monitoring regions acceptable to the mobile unit.

<u>**Initialization**</u>: This procedure is called when the mobile unit is powered on:

1. Set both $myDomain$ and $myMRs$ to $null$.
2. Spawn thread $MessageListener$.
3. Send message $RequestResidentDomain$ $(myID, myPos, myCapacity)$ to the server.
4. Spawn thread $RegionMonitor$.

MessageListener: The mobile unit listens to these messages and processes them as follows:

- $SetResidentDomain(oid, d, l)$: If $oid == myID$, then do the following:
  - Set $OldDomain = myDomain$.
  - Set $myDomain = d$.
  - Set $myMRs = l$.
  - If $OldDomain == null$ (i.e., the object is in the initialization stage), then for each monitoring region, say r, in $myMRs$, if r contains $myPos$, send server a message $UpdateQueryResult(r, myID, myPos)$.
- $AddMonitoringRegion(r)$:
  - Add monitoring region r to $myMRs$ if r is inside $myDomain$.
  - If r contains $myPos$, send server a message $UpdateQueryResult(r, myID, myPos)$.
- $DeleteMonitoringRegion(r)$: Delete monitoring region r from $myMRs$ if r is inside $myDomain$.
- $SplitDomain(d, d_1, d_2)$: If $myDomain == d$, then do the following:
  - If d_1 contains $myPos$, set $myDomain = d_1$; otherwise, set $myDomain = d_2$.
  - For each monitoring region in $myMRs$, say r, do the following:
    - Delete r if it does not overlap with the new $myDomain$.
    - Otherwise, replace r with the portion of the rectangle that is inside $myDomain$.
- $MergeDomain(d_1, d_2, l)$: If $myDomain$ overlaps with d_1 or d_2, do the following steps:
  - Set $myDomain$ to be the merged domain of d_1 and d_2.
  - Set $myMRs = l$.

RegionMonitor : When the mobile unit moves, it monitors its spatial relationships with its resident domain and the monitoring regions it knows as follows:

- If the object moves out of $myDomain$, then request for a new resident domain by sending the server a message $RequestResidentDomain$ $(myID, myPos, myCapacity)$.
- For each monitoring region r in $myMRs$, it checks if it enters or exits r and when this happens, it sends a message $UpdateQueryResult(r, myID, myPos)$ to update the server.

Under our technique, the cost of server processing is minimal because the queries are actually processed distributively by the mobile objects near the query areas. The reduction in the server load allows the same server to support many more mobile objects. In addition, by having mobile objects to monitor their nearby queries directly, the query results can be updated accurately in real time. At first, it might seem that this strategy will cause the mobile devices to consume more power. On the contrary, significant energy is saved due to substantial reduction in the number of location updates - an object does not need to report its location unless its movement affects some query result or it moves out of its resident domain. We note that in practice, power required by CPU is minimal compared to sending data over the wireless radio. For example, the energy cost of transmitting 1Kb over a distance of 100 meters is approximately 3 joules. By contrast, a general-purpose processor with 100 MIPS/W power could efficiently execute 3 million instructions for the same amount of energy [2]. In our case, it takes only 4 simple numerical comparisons to determine if a mobile object is inside a query rectangle.

We note that $myCapacity$ of a mobile device can be adjusted dynamically to reflect its processing capability at different times. When the device requires more CPU cycles and/or memory for other tasks with higher priorities, it can negotiate with the server for a smaller resident domain using a smaller $myCapacity$. Alternatively, we can consider allowing a mobile object to unilaterally reduce its resident domain to achieve the same effect. Although this option makes our technique even more flexible, we will not investigate it further in this paper. In the above discussion, we also intentionally left out the users of the location-based service. These users could connect to the server through conventional wired networks. They could also be the mobile devices mentioned in the above discussion. For completeness, the server also needs to provide the interface for submitting queries and viewing query results. This issue is beyond the scope of this paper.

3.3. Advantages of the MQM Approach

The performance benefits of the proposed technique are as follows:

- *Low Communication Cost* : A mobile object does not need to report to the server unless the object enters or exits a monitoring region, or moves out of its current resident domain. This strategy significantly reduces its power consumption for wireless communication.

- *Scalability* : The range queries are evaluated distributively relieving the server from becoming a bottleneck. This feature makes MQM highly scalable allowing it to support a very large number of mobile objects and range-monitoring queries.
- *Reliability* : MQM is more reliable than techniques based on location estimation. These techniques typically require server to estimate the object locations according to some velocity model in order to save the update communication costs. For techniques on balancing the update cost and imprecision, interested readers are referred to [28], [8], [9], and [29]. The query results based on the estimated locations are approximated and could be inaccurate. As a contrast, MQM provides accurate and real-time query results.
- *Simplicity* : Another advantage of MQM is its simplicity. It does not use any velocity models or dead-reckoning algorithms [29].

4. BP-Tree: Binary Partitioning Tree

In MQM, whenever an object moves out of its resident domain, the server needs to determine a new resident domain and retrieve its enclosed monitoring regions for the object. The resident domain should be as large as possible but contain no more than a certain number of monitoring regions, depending on the processing capability of the object. In our research, this operation is supported efficiently by modifying Quad-tree [30] to index the decomposed domain and monitoring queries. We call the new index structure as BP-tree (*Binary Partitioning tree*) in this paper. In this section, we present its data structure in details and discuss how it can be used to determine a suitable resident domain for an object. A formal description of BP-tree operations can be found in Appendix.

A BP-tree consists of two types of node: *domain node* and *data node*. All internal nodes are domain nodes and all external nodes are data nodes. The data structure for a domain node consists of two entries, each has the form (R, P), where R holds the upper-left and lower-right coordinates of a rectangular subdomain, and P links another domain node or a data node. Each domain node represents a decomposition of a parent subdomain. As illustrated in Figure 3, the decomposition of subdomain d_2 consists of two subdomains, d_{21} and d_{22}, each stored in one entry of the domain node representing d_2. In addition to the two entries, each domain node also uses a variable, $size$, to record the total number of monitoring regions indexed under this domain node. A data node stores the monitoring regions that are inside its parent subdomain. A data node contains an array of rectangles, each holding a monitoring region, and a variable $size$, recording the total number of monitoring regions. As an example, the data node linked by the domain entry of d_{11}, as shown in Figure 3, is used to store all monitoring regions inside d_{11}. We note that the size of data nodes is limited by the minimum processing capability assumed for mobile objects. This parameter is used to determine the split threshold for data nodes. Thus, a mobile object can load at least one data node.

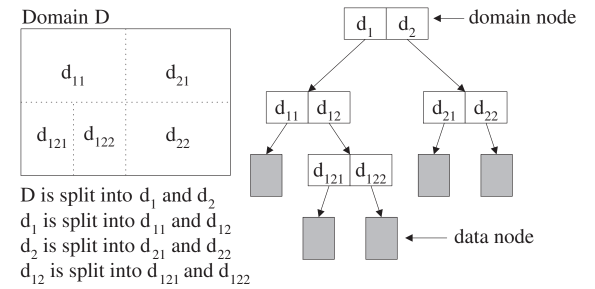

Figure 3. A BP-tree Example

With this data structure, the operation of searching for a suitable resident domain for a mobile object is quite simple. Given a mobile object at position p and its processing capability n, we can determine its resident domain by searching from the root of BP-tree. If the number of monitoring regions inside the root domain is acceptable to the object (i.e., no larger than n), the root domain becomes the object's resident domain. Otherwise, we descend the tree to check the subdomain that contains position p. The subdomain is the object's resident domain if the object can load all monitoring regions inside it; otherwise, we check the child domains of the subdomain and this operation is done recursively until we find a subdomain in which the number of monitoring regions is acceptable to the object. We note that in the worst case, a mobile object takes a leaf subdomain as its resident domain. When a resident domain is determined, we then retrieve all monitoring regions inside it and send them to the requesting object.

5. Performance Study

To evaluate the performance of our Monitoring Query Management approach, we compare it with the Q-index technique [4] [5]. As we are also interested in how the overall system performance is improved by taking advantage of the heterogeneous mobile computing capability, we implemented two versions of MQM:

- *Plain MQM* : This version allows each mobile object to cache and process only the monitoring regions stored in one data node. That is, it exploits only the minimum mobile computing capability, which is dictated by the least capable mobile device.

- *Adaptive MQM* : In this implementation, each mobile object is allowed to load monitoring regions as many as possible, according to their true computing capability. This is the proposed scheme in this paper.

5.1. Performance Metrics

The performance metrics selected for this study are the *server processing cost* and *communication costs*. We discuss them as follows:

- *Server Processing Cost* : This cost is measured as the total number of index-tree nodes accessed in order to process requests from the mobile objects. This is a good measure of the server processing cost because server operations involve mainly navigating the BP-tree and processing the data stored in the leave nodes. The cost of searching the Relevance Table is ignored because it can be implemented as a hash file, and takes only $O(1)$ to retrieve the relevant queries for a given monitoring region.
- *Server Communication Cost* : This cost is measured as the total number of messages transmitted from the server (to the mobile units). This is a good measure of the relative server communication cost because our messages are very short. For example, if we use 16 bytes to identify a monitoring rectangle, then it takes only 800 bytes to represent a full data node with 50 entries.
- *Mobile Communication Cost* : This cost is measured as the total number of messages sent by the mobile objects to the server. Again, this measure reflects the relative mobile communication cost quite accurately because our messages are very short.

We differentiate the two types of communication costs because sending a message requires a mobile device substantially more power than listening and receiving a message. The rationale for these three performance metrics is as follows:

- *Scalability Measure*: The server processing cost and server communication cost are good indicators of whether the server can become a bottleneck. They are good measures of system scalability.
- *Power Conservation Measure*: The mobile communication cost alone is a good measure of power conservation. We do not take into account the mobile computation cost because it is negligible compared to the power required for transmitting wireless data at a relatively high rate [2] [3].

We note that although the above model cannot predict the exact costs of these techniques, the intention of this performance study is to predict their relative performance under varying circumstances. Thus, the better technique, as measured by the above metrics, is able to scale up better to support a larger user community, and requires less battery power from the mobile devices.

5.2. Simulation Model

We implemented the BP-tree using a center-split strategy as discussed in Appendix - splitting at the middle of the longer dimension. The same BP-tree is used to index the monitoring regions for both MQM techniques and the Q-index approach. This allows us to compare the server computation costs of these techniques fairly. The concept of *safe region* of Q-index scheme is implemented as follows. Given a mobile object, we first search the BP-tree for the subdomain that contains its current location. We then compute the largest circular region based on the monitoring regions within the subdomain as the safe region for this object, such that the safe region does not overlap with the boundaries of any query. We choose not to use rectangular safe region because its implementation is much more complicated while the achieved performance is quite similar to that of using circular safe region, as indicated in [4]. We note that we actually compare our technique with an improved version of Q-index in this study. Our BP-tree approach avoids the excessive workload of computing a safe region by limiting the consideration to only the monitoring regions stored in one data node. It, however, can achieve a near-optimal safe region. This is due to the fact that the size of a safe region is mainly determined by the query regions surrounding its host object. With the original Q-index technique, it would have to examine all the queries for the safe region of each mobile object. Obviously, this is not feasible for a real-time system. In fact, the technique discussed in [4] determined the safe region only once at system startup due to the high cost. The algorithm has a complexity of $O(n^2)$, where n is the total number of queries. It is not clear how they handle the situation when an object exits its current safe region. With the new strategy, we can compute a new safe region easily.

Other simulation parameters are as follows. We generate 10,000 mobile objects and place them in a uniform distribution over a rectangular domain of [0...10K, 0...10K]. The performance data under other distributions such as Gauss and Poison were also collected, but omitted here as they are quite similar. The computing capability of these objects follows a zipf distribution with a deviation of 0.7, ranging from 100 to 1000 monitoring regions. Thus, the size of BP-tree data node is set to be 100. We assume each object has constant processing capability throughout the simulations. Similar to [4], the velocities of these mobile objects follow a zipf distribution with a deviation of 0.7, and fall in between 0 and 100 per time unit. Again, the velocity of each object is constant throughout each simulation run. Their initial moving directions are set randomly. Each object moves linearly until it reaches any one of the four boundaries of the database domain, in which case it reflects its direction and continues to move at the same speed. This process is repeated until the simulation is ended.

For each simulation run, we generate a certain number of square range-monitoring queries. The sizes of these monitoring squares range from 1×1 to 100×100 and they are placed in the domain space following a uniform distribution. Each simulation run consists of two phases, and lasts 10,000 simulated time units. During the first phase, a new query is inserted every time unit until we have inserted the desired number of queries. During the second phase, the objects continue to move around, but we do not add any more query. The reason for separating these two phases is due to the fact that Q-index performs poorly whenever the system experiences new queries, because it must examine the location of all mobile objects for every new query. It is not interesting to compare our technique with Q-index under such circumstances. In this paper, we will focus on comparing the two techniques based on their phase-2 performance. In other words, the performance comparisons reported in this paper are quite conservative. In practice, the performance improvement due to MQM should be even more significant.

5.3. Simulation Results

The performance data collected in the second phase are plotted in Figure 4, Figure 5, and Figure 6. All three figures indicate that Q-index performs significantly worse than both MQMs. The various costs of Q-index are very high, and increase rapidly with the number of queries. Figure 4 and Figure 5 show that this approach is not scalable since the server experiences a very heavy load of computation and communication tasks. A small increase in the number of queries can make the server become a bottleneck. We have similar observations about the mobile objects in Figure 6. Apparently, Q-index requires substantially more battery power as the mobile units need to communicate with the server much more frequently. Again, this cost increases rapidly as we increase the number of queries. This is another factor that limits the scalability of the Q-index approach. On the contrary, we observe no such problems with both MQM techniques. Compared to that of Q-index, the curves for both of them are quite low and flat. Briefly, we explain the performance difference between the MQM strategy and the Q-index scheme as follows:

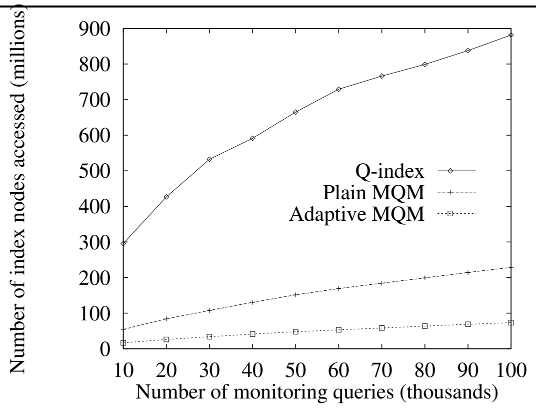

Figure 4. Server processing cost under various workloads

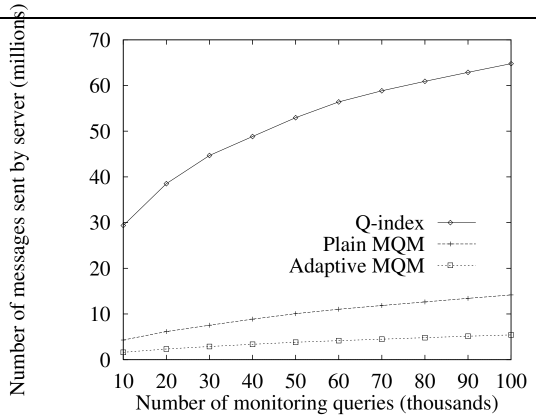

Figure 5. Server communication cost under various workloads

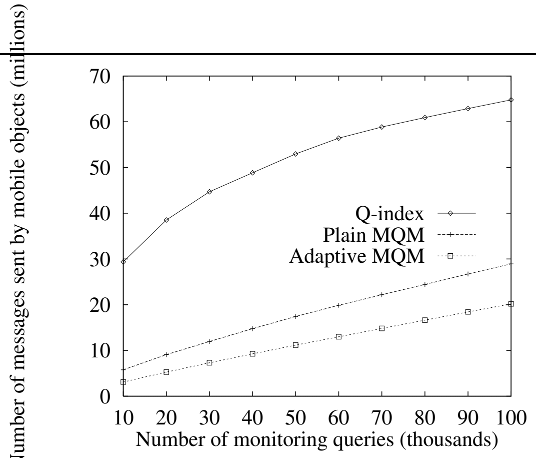

Figure 6. Mobile communication cost under various workloads

- Server Processing Cost: The server processing cost is higher for Q-index due to the following reasons. (1) Under Q-index, the server has to examine all the queries overlapping with the current resident domain for potential updates each time an object exits its safe region. This computation is often wasted since exiting a safe region does not imply that the object is entering or exiting any query region. As the number of queries increases, this cost also increases, as showed in Figure 4. (2) Under Q-index, when a mobile object indeed enters or exits a monitoring region, it simply reports that it exits its safe region, because the object is not aware of its nearby monitoring regions. As a result, the server has to search the BP-tree to determine if the object enters or exits any monitoring region for potential update of query results. In contrast, a mobile object in MQM provides the affected monitoring region directly, allowing the server to retrieve the relevant queries without looking up the BP-tree.
- Communication Costs: The communication costs are higher for Q-index due to the following reason. A mobile object, with MQM, does not need to report its position until it exits its current resident domain, which consists of one or more subdomains. In contrast, an object, under Q-index, must report its position as soon as it exits its current safe region. Since a safe region is not allowed to overlap with any query boundaries, it is generally many times smaller than the containing subdomain. Thus, a mobile object under Q-index needs to communicate with the server much more frequently. We observe in Figure 5 and Figure 6 that the communication costs increase with the number of queries. This is due to the fact that increasing the number of queries reduces the average size of the safe regions, and therefore the chance of requesting a new safe region becomes higher.

These performance figures confirm that the proposed MQM technique is much more efficient than the Q-index scheme in supporting range-monitoring queries in a large scale location-based service. In addition, the performance difference between the two versions of MQM also shows the effectiveness of exploiting the heterogeneous mobile computing environment to minimize the mobile communication cost and server processing need. All three figures indicate that the adaptive version of MQM constantly outperforms its plain version. In particular, the performance gap widens when the number of queries increases. Without using the maximum computing capability of mobile devices, the plain MQM incurs at least 100% more server communication and processing cost than the adaptive MQM, as showed in Figure 4 and Figure 5. Figure 6 also shows that the plain MQM consumes significantly more battery power. The performance difference between the two versions of MQM can be explained as follows. By loading as many monitoring regions as its capacity allows, a mobile object can have a maximum size of resident domain, which reduces the likelihood that it will have to request a new resident domain. Thus, the mobile device consumes less battery power. Since the server workload mainly involves navigating the BP-tree to determine resident domains, a smaller number of resident domain requests will free more server resources allowing the server to serve a larger mobile community.

6. Concluding Remarks

The advances in GPS, GIS, and wireless technologies will enable billions of online wireless appliances that are location-aware in the coming years [31]. Exploiting this enormous computing environment will bring about many new important applications. In this paper, we addressed the challenge of providing region monitoring services that require continuous updates of the query results in real time. Unlike traditional database management systems which are designed to manage data, we introduced a query management technique called MQM (*Monitoring Query Management*). Our technique is able to provide accurate and real-time query results without requiring the constant location updates from mobile objects. While it minimizes the mobile communication overhead, the new approach also relieves the server from having to monitor the movement of mobile objects and tracking the query results. Thus, MQM is highly scalable. In addition, the new technique allows a more capable mobile objects to trade its processing capability for reducing communication costs. Therefore, it is also adaptive to the heterogeneity of mobile objects. To the best of our knowledge, MQM is the only technique available today that is able to process real-time range-monitoring queries in large-scale mobile applications.

References

[1] M. Stemm and R. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Trans. on Communications, vol.E80-B, no.8, p. 1125-31*, E80-B(8):1125–31, 1997.

[2] G.Pottie and W. Kaiser. Wireless sensor networks. *Communications of the ACM*, 43(5):51–58, May 2000.

[3] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of Intl. Conf. Data Engineering*, 2002. to appear.

[4] S. Prabhakar, Y. Xia, D. Kalashnikov, W. G. Aref, and S. hambrusch. Queries as data and expanding indexes: techniques for continuous queries on moving objects. In *TR., Dept. of Computer Science, Purdue University*, 2000.

[5] S. Prabhakar, Y. Xia, D. Kalashnikov, W. G. Aref, and S. hambrusch. Query indexing and velocity constrained

indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 15(10):1124–1140, October 2002.

[6] Ying Cai and Kien A. Hua. An adaptive query management technique for efficient real-time monitoring of spatial regions in mobile environments. In *Proc. of the 21st IEEE Int'l Performance, Computing, and Communication Conference (IPCCC)*, pages 259–266, April 2002.

[7] A. Bar-Noy, I. Kessler, and M. Sidi. Mobile users: To update or not to update? *ACM/Baltzer Wireless Networks*, 1(2):175–185, 1995.

[8] O. Wolfson, S. Chamberlain, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *Proc. of Intl. Conf. Data Engineering*, pages 588–596, 1998.

[9] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: issues and solutions. In *Proc. of the 10th Int'l Conference on Scientific and Statistical Database Management*, pages 111–122, July 1998.

[10] K. Lam, O. Ulusoy, T. S. H. Lee, E. Chan, and G. Li. An efficient method for generating location updates for processing of location-dependent continuous queries. In *DASFAA'01*, pages 218–225, Hong Kong, China, April 2001.

[11] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. of ACM PODS'99*, pages 261–272, 1999.

[12] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *ACM Proc. of SIGMOD'00*, pages 331–342, 2000.

[13] L. Arge, V. Samoladas, and J.S. Vitter. On two-dimensional indexability and optimal range searching indexing. In *Proc. of ACM PODS'99*, pages 346–357, 1999.

[14] P. K. Argarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. of ACM PODS'00*, pages 175–186, 2000.

[15] D. Pfoser, Y. Theodoidis, and C. S. Jensen. Indexing trajectories of moving point objects. In *Chorochronos Technical Report, CH-99-3*, 1999.

[16] D. Pfoser, C. S. Jensen, and Y. Theodoidis. Novel approaches in query processing for moving objects. In *Proc. of VLDB'00*, 2000.

[17] L. Forlizzi, R. H. Guting, E. Nardelli, and M. Scheider. A data model and data structures for moving objects databases. In *Proc. of ACM SIGMOD'00*, pages 319–330, 2000.

[18] J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

[19] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE TKDE*, 10(1):1–20, 1998.

[20] B. Salzberg and V. J. Tsotras. A comparison of access methods for temporal data. In *TimeCenter TR-13*, 1997.

[21] M. Sidi and I. Cidon. A multi-station packet-radio network. *Performance Evaluation*, 35(2):65–72, February 1988.

[22] R. Steele. The cellular environment of lightweight hand held portables. *IEEE Communications Magazine*, pages 20–29, July 1988.

[23] R. Steele. Deploying personal communication networks. *IEEE Communications Magazine*, pages 12–15, September 1990.

[24] D. J. Goodman. Cellular packet communications. *IEEE Transaction on Communications*, 38:1272–1280, August 1990.

[25] E. Pitoura and G. Samaras. Locating objects in mobile computing. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):571–592, July/August 2001.

[26] J. C. Navas and T. Imielinski. Geographic addressing and routing. In *Proc. of ACM/IEEE MobiCom'97*, Budapest, Hungary, September 1997.

[27] T. Imielinski and J. C. Navas. Gps-based geographic addressing, routing, and resource discovery. *Communications of the ACM*, pages 86–92, April 1999.

[28] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proc. of Intl. Conf. Data Engineering*, pages 422–432, 1997.

[29] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[30] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley, Reading, MA, 1990.

[31] C. S. Jensen, A. Friis-Christensen, T. B. Pedersen, D. Pfoser, S. Saltenis, and N. Tryfona. Location-based services – a database perspective. In *Proc. of the Eighth Scandinavian Research Conference on Geographical Information Science*, pages 59–68, Norway, June 2001.

Appendix: BP-tree Operations

With BP-tree, the monitoring regions are grouped according to their containing subdomains and each group is stored in one data node. Meanwhile, the domain decomposition hierarchy is captured by the organization of BP-tree domain nodes. Before we discuss the detailed operations of BP-tree, we define the following notations:

- Given an entry (R, P) in a domain node, $R.child\_node$ denotes the child node pointed at by P.
- Given a BP-tree node D, $D.domain$ is the domain represented by this node, $D.parent$ refers to the parent node who has an entry pointing to D ($D.parent$ is $null$ if D is the BP-tree root), and $D.size$ is the total number of monitoring regions stored in the data nodes descending from D.
- Given two rectangles, R_1 and R_2, $R_1 \cap R_2$ represents their overlapping area.

6.1. Search

When the server receives message $RequestResidentDomain(oid, p, n)$, it needs to determine a resident domain for the object oid, given its current position p and computing capability n. The resident domain should contain as many monitoring regions as possible, but no more than n. With BP-tree, this can be done efficiently by calling $Search(root, p, n)$, where $root$ is the BP-tree root:

Search(D, p, n)

1. If $D.size <= n$, then return $D.domain$ and all monitoring regions indexed under D. $D.domain$ is the new resident domain for the requesting mobile object.
2. Otherwise, search for the entry, say (R, P), in D such that rectangle R contains position p.
3. Recursively call $Search(R.child\_node, p, n)$.

6.2. Insert

A BP-tree is initialized as a root domain node with one empty data node. That is, the first entry of the root is set to (R, P), where R is the entire domain and P points at an empty data node. The variable $size$ of the node is set to 0. When a new query arrives, the server descends the BP-tree to look for the data nodes whose domains overlap with the query rectangle. For each overlapping area, i.e., a monitoring region, say r, a new tuple (r, q) is added to the relevance table. The monitoring region is also inserted into the BP-tree if it does not already exist in BP-tree. The fact that only distinct monitoring regions are stored in the BP-tree allows our technique to deal with overlapping queries. When a new monitoring region is inserted to a data node, the variable $size$ of each domain node in the searching path, from the root to the data node, is increased by 1. Thus, given a domain node, we can know easily the total number of monitoring regions it contains.

An insert might cause a data node to overflow. When this happens, its domain is split. A number of decomposition schemes can be used to split a domain. A simple approach is *center split*, i.e., split the domain vertically or horizontally into two equal-sized subdomains. The direction of the split can be determined by comparing the dimensions of the domain. For example, we can split on the longer dimension to avoid having long and narrow subdomains. The monitoring regions spanning over the split line are also split and the relevance table is updated accordingly. Each time a data node is split, the server broadcasts a message $SplitDomain(d, d_1, d_2)$ to notify mobile units that some domain d has been decomposed into d_1 and d_2. We have discussed how a mobile unit reacts to such a message.

When a new query q arrives, we call BP-tree operation $Insert(D, q)$, where D is set to be the BP-tree root:

Insert(D, q)

1. If D is a domain node, then for each entry, say (R, P), in D, call $Insert(R.child\_node, q)$ if R overlaps with q.
2. If D is a data node, then do the following:
   - Set r to be equal to $q \cap D.domain$.
   - Insert a new tuple, (r, q), to the relevance table.
   - If no monitoring region in D is equal to r, then do the following:
     - Add monitoring region r to D.
     - Broadcast $AddMonitoringRegion(r)$ message.
     - Set D' equal to $D.parent$ and repeat the following until D' is $null$:
       * Increase $D'.size$ by 1.
       * Set D' equal to $D'.parent$.
     - If D is full, call $SplitDataNode(D)$.

SplitDataNode(D): Split BP-tree data node D

1. Look for the entry, say (R, P), in $D.parent$ such that P points at D.
2. Split domain R into two subdomains, R_l and R_r.
3. Broadcast message $SplitDomain(R, R_l, R_r)$.
4. Create two new data nodes, $left$ and $right$.
5. Create a new domain node, D', and set its two entries to be (R_l, P_l) and (R_r, P_r), where P_l and P_r point to data nodes $left$ and $right$, respectively.
6. Redirect P in the entry (R, P) of $D.parent$ to point at D'.
7. For each monitoring region R_i stored in D, do the following:
   - If R_i overlaps with R_l and monitoring region $R_i \cap R_l$ does not exist in $left$, then do the following:
     - Insert monitoring region $R_i \cap R_l$ into $left$.
     - Increase $left.size$ by 1.
   - If R_i overlaps with R_r and monitoring region $R_i \cap R_r$ does not exist in $right$, then do the following:
     - Insert monitoring region $R_i \cap R_r$ into $right$.
     - Increase $right.size$ by 1.
   - If R_i overlaps with both R_l and R_r, then for each tuple, say (r, q), in the relevance table, do the following if r is equal to R:
     - Replace tuple (r, q) with $(R_i \cap R_l, q)$.
     - Add new tuple $(R_i \cap R_r, q)$ to the table.
8. Set $D'.size$ equal to $D.size$ and repeat the following until D' is $null$:
   - Increase $D'.size$ by $left.size + right.size - D.size$.
   - Set D equal to $D'.parent$.
9. Discard D.
10. Call $SplitDataNode(left)$ if $left$ is full.
11. Call $SplitDataNode(right)$ if $right$ is full.

6.3. Delete

The *Delete* operation is used when a query, say q, needs to be terminated. The server first checks the relevance table and deletes all tuples containing q as the relevant query. If a tuple, say (r, q), is deleted and there are no more tuples in the table containing monitoring region r, then r is also deleted from the BP-tree by calling $Delete(D, r)$, where D is the BP-tree root. A message $DeleteMonitoringRegion(r)$ is then broadcast. Deleting a monitoring region may cause a subdomain to underflow. When this happens, the subdomain and its split counterpart are merged. When two subdomains, say d_1 and d_2, are merged, the server broadcasts a message $MergeDomain(d_1, d_2, l)$, where l is the combined list of monitoring regions inside d_1 and d_2. The formal description of the $Delete$ operation is given below.

Delete(D, r): Delete monitoring region r from the BP-tree rooted at node D

1. Decrease $D.size$ by 1.
2. If D is a domain node, then search for the entry, say (R, P), in D such that R contains r, and call $Delete(R.child\_node, r)$.
3. Otherwise, D is a data node and do the following:
   - Remove monitoring region r from D.
   - Call $MergeUnderflows(D.parent)$.

MergeUnderflows(D): Merge the children nodes of D if necessary

1. If both children of D are data nodes and they can be merged into one, then do the following:
   - Create a new data node, D'.
   - Move all monitoring regions stored in both children nodes of D into D'.
   - Search for the entry, say (R, P), in $D.parent$ such that P points at D, and redirect P to point at D'.
   - Discard D and its children nodes.
   - Call $MergeUnderflows(D'.parent)$.