Mobiscope: A Scalable Spatial Discovery Service for Mobile Network Resources*

Matthew Denny¹, Michael J. Franklin¹, Paul Castro², and Apratim Purakayastha²

¹ U.C. Berkeley, Berkeley, CA, USA
{mdenny,franklin}@cs.berkeley.edu
² IBM T.J. Watson Research Center, Hawthorne, NY, USA
{castrop,apu}@us.ibm.com

Abstract. Mobiscope is a discovery service where clients submit long-running queries to continually find sets of moving network resources within a specified area. As in moving object databases (MODBMSs), moving resources advertise their positions as functions over time. When Mobiscope receives a query, it runs the query statically against advertisements (ads) that are currently cached by Mobiscope, and then continuously over ads that Mobiscope receives subsequently. For scalability, Mobiscope distributes the workload to multiple nodes by spatial coordinates. Application-level routing protocols based on geography ensure that all queries find all matching resources on any node without significant processing overhead. Simulation results indicate that Mobiscope scales well, even under workloads that stress Mobiscope’s distribution model.

1 Introduction

With the growth in wireless connectivity, we are increasingly surrounded by objects that are connected to the network. Many of these items are highly mobile, such as cars, cell phones, PDAs, and even mobile sensor stations floating in rivers used to forecast weather (e.g., [19]). Many new applications require the discovery of mobile network resources keyed on position. Location-based service providers may want to find cars within certain geographic regions, and monitoring applications for mobile sensor stations may want to know sensor station density within some area in order to adjust sampling rates.

Mobiscope is a service for discovering moving network resources. Mobiscope accepts advertisements (ads) from network resources containing position information, and queries for network resources within a specified spatial region. For each query, Mobiscope returns a set of resources that are inside the region, and keeps this set current as resources move over time. Ads represent position as functions over time. Functions allow a system to interpolate the position of a resource between successive ads. A resource does not have to update its advertised position function until the function no longer reflects the resource’s position.

* Franklin and Denny have been partially supported by the NSF under grants IIS-0086057 and SI-0122599, by an IBM Faculty Partnership Award, and by research funds from Intel, Microsoft, and the UC Micro program.
Position functions originate in the moving objects database (MODBMS) literature [17, 15, 21, 12]. MODBMSs can use position functions to predict which objects will be within a given spatial region for some time interval. Although this previous work is relevant to Mobiscope, current MODBMSs have two major limitations when position functions are rapidly changing. First, MODBMSs do not provide the correct semantics for Mobiscope queries because MODBMSs derive future positions based only on advertisements that are currently stored by the MODBMS. If resources re-advertise new position functions after the MODBMS processes a query, the answer may become incorrect. Second, MODBMSs store position functions in a static, centralized database. These databases may not scale to a large number of moving objects if the objects rapidly update their position functions. Mobiscope must not suffer from these limitations if it is to be deployed in environments where many resources change position functions rapidly (e.g. a service tracking cars that start, stop, and change direction often in a large city).

Mobiscope addresses both the semantics problem and the scalability problem of MODBMSs. In Mobiscope semantics, the service first runs each new query statically over cached ads, and then continuously over new incoming ads. To ensure that clients maintain correct result sets for each query, Mobiscope employs a unique model that combines results from the static and continuous portion of each query.

To achieve scalability, ad and query processing in Mobiscope is distributed to a set of directories in a network. Each directory has a spatial service region, which dictates the ads and queries for which the directory is responsible. The Mobiscope network design accommodates large numbers of moving resources that communicate using wide-area wireless infrastructure. Mobiscope directories can be deployed alongside wireless infrastructure, such as basestations, basestation controllers, or gateways. The service regions of each directory can be defined by the range of the corresponding piece of infrastructure. If we assume that the density of wireless infrastructure per spatial region reflects the density of resources advertising in the region, such a deployment is likely to partition the workload efficiently. Mobiscope directories function at the application level, and can work over any underlying wireless infrastructure. In our environment, we assume that directories do not come online, go offline, or change service regions unless a) the total area covered by a Mobiscope network is changed, or b) a directory fails.

Mobiscope ensures full reachability such that, in the absence of failures, the response set for each query returns every matching ad in the system. Mobiscope directories route ads and queries among themselves to maintain full reachability. The routing protocols are designed to conserve processing overhead, which can become significant if the protocols are poorly designed.

To cope with failures, Mobiscope stores both ads and queries as soft-state. That is, Mobiscope directories do not require any expensive logging and recovery mechanisms because resources and clients periodically re-transmit ads and queries, respectively. If a directory fails and restarts or becomes temporarily disconnected from the network, it will eventually be repopulated with soft-state ads and queries, and every response set eventually contains every matching ad. Previous service discovery work (e.g. [11, 1, 2, 22, 9]), stores ads as soft-state. However, Mobiscope also stores continuous queries as soft-state.
To evaluate Mobiscope, we implemented a simulator and workload generator. We generate and run workloads that model a large number of rapidly updating resources and stress Mobiscope's distribution model. In these experiments, Mobiscope performs quite well, even under workloads that are less than ideal for its distribution model.

The rest of the paper is structured as follows. Section 2 describes the components of Mobiscope, as well as the query semantics. Section 3 presents the distributed processing algorithms used by Mobiscope. Section 4 presents the performance study, Section 5 discusses related work, and Section 6 concludes the paper.

2 Mobiscope Overview

In this section, we present the structure and semantics of the Mobiscope system. In Section 2.1, we introduce the major components of the system, as well as the ad and query formats. In Section 2.2, we describe the query semantics.

2.1 Mobiscope Components

In Mobiscope, there are three types of components: directories, resources that submit ads, and clients that submit queries.

Directories Mobiscope directories are located throughout the network and accept both ads and queries. For a given network of Mobiscope directories, we define $R_{total\ Area}$ to be the 2-dimensional rectangle that represents the area for which the network provides service $^3$.

Each directory has a service region (SR), which is a rectangle that covers some subset of $R_{total\ Area}$. Mobiscope is optimized for the case where the entire area of $R_{total\ Area}$ is completely covered by directory service regions. If $R_{total\ Area}$ is not totally covered, however, Mobiscope will still function. If a Mobiscope network is deployed alongside of wireless infrastructure, coverage holes in the infrastructure can be subsumed by directories with adjacent service regions. The Mobiscope network tolerates overlap between service regions. Overlap may occur, for example, if service regions are determined by the coverage area of wireless infrastructure components, and these coverage areas overlap.

Mobile Resources Resources notify Mobiscope of their position functions by sending advertisements to a directory in the network. Ads contain a unique identifier for a resource ($UID$), an expiration time ($tExp$), the resource's network address, and a position function ($pos$). Resources send ads to a directory such that the resource resides in the directory's service region. In Section 3, we explain how resources find the correct directory as they move throughout $R_{total\ Area}$.

The function $pos$ takes time as a parameter and returns an upper and lower bound for each dimension. Readers can find a detailed explanation of position function representations in [20]. Figure 1 shows an example of position functions for 3 mobile resources (mr1-3). For simplicity, this diagram shows functions in only one dimension

$^3$ Although we limit our discussion to the discovery of mobile resources in 2 dimensions, Mobiscope could be easily modified to discover resources in 3 dimensions.
plotted against time. These functions are shown as parallelograms, which represent the value range that each resource may take at a given time. For a resource \( mr \) that sends an ad \( a \) at time \( tXmit_a \), \( mr \) sends a successive ad with a new position function when \( mr \) detects that it has moved outside of \( a, \overline{pos}(t) \) at any time \( t : tXmit_a \leq t \leq a.tExp \). Discontinuities in the parallelograms in Figure 1 represent changes in the position functions. If the position function does not change, \( mr \) will send a new ad at time \( a.tExp - \Delta \), where \( \Delta \) is enough time for the ad to arrive at a directory. At least one directory in Mobiscope will cache each ad until it either expires or is replaced by an ad with the same UID. In the example in Figure 1, we assume that each ad \( a \) transmitted at time \( tXmit_a \) has \( a.tExp = tXmit_a + 2 \text{ min.} + \Delta \). In this case, resources must transmit new ads every other minute.

**Clients** To query Mobiscope, clients send queries to any directory in the Mobiscope network. Each query \( Q \) has a rectangle, \( rect \), the client’s network address, an expiration time \( tExp \), and an \textit{isRefresh} flag. Clients periodically send query \textit{refreshes} for each query with a new expiration time to maintain the query’s soft-state. Figure 1 shows a query \( Q1 \) in one dimension, where the rectangle is only a range of values for a time interval. \( Q1 \) expires at \( t = 14 \text{ min.} \). The queries will cease running after \( t = 14 \text{ min.} \) if the \( Q1 \) client does not send a query refresh. In the following section, we describe the semantics of the query output.

### 2.2 Query Semantics

For a client that submits \( Q \) to a directory at time \( tSubmitQ \), the directories work together to find a set of UID/network address pairs that indicate the resources currently in \( Qrect \). This set continually changes as resources move in and out of \( Qrect \). For example, consider \( Q1 \) in Figure 1, where \( tSubmitQ1 = 4 \text{ min.} \). Initially, \( Q1 \)'s result set should contain query responses for \( mr1 \) and \( mr2 \). At time \( t = 6 \text{ min.} \), it contains responses for \( mr1 \) and \( mr3 \). At time \( t = 12 \text{ min.} \), the result set again contains responses for \( mr1 \) and \( mr2 \).

When a directory receives a new query \( Q \) at time \( tSubmitQ \), directories find all cached ads \( a \) such that \( a, \overline{pos}(t) \) intersects \( Q, rect \) for some time \( t : tSubmitQ \leq t \leq a.tExp \). The query and ad routing policies we present in Section 3 ensure that \( Q \) finds all matching ads.

For each ad \( a \) found, a query response \( (qr_a) \) is generated for the client. A query response \( qr_a \) contains the UID and network address from \( a \), as well as a time interval \( (tStart, tExp) \). For query response \( qr_a \), \( qr_a.tStart \) is the smallest time \( t \geq tSubmitQ \) such that \( a, \overline{pos}(t) \) intersects \( Q, rect \), and \( qr_a.tExp = \min(a.tExp, t LearRegion) \), where \( t LearRegion \) is the smallest time \( t \geq tSubmitQ \) such that \( a, \overline{pos}(t) \) no longer intersects \( Q, rect \).

The directories send initial query responses to the client and run the query continuously over subsequent incoming ads. Consider an ad \( a_{new} \) that arrives at a directory where \( Q \) is running continuously. If \( a_{new} \)'s position function intersects \( Q, rect \) within the lifetime of the ad, the directory creates a query response \( qr_{a_{new}} \) for \( Q \)'s client. \( tStart \) and \( tExp \) for continuous query responses are defined in a similar manner to the initial query responses. Again, routing ensures that all continuous queries find all matching ads.
To demonstrate these semantics, we again refer to our example in Figure 1. To keep our example simple, we again assume that any ad a transmitted at time \( tXmit_a \) arriving at a directory at time \( t \) has an expiration time of \( tXmit_a + 2 \text{ min.} + \Delta \), where \( tXmit_a + \Delta \approx t \) (i.e., \( \Delta \) is approximately the network transit time for \( a \)). We also assume that all ads are transmitted on even numbered minutes. Consider query \( Q_1 \), with \( tSubmit_{Q_1} = 4 \text{ min.} \) and \( Q_1.tExp = 14 \text{ min.} \). Table 1 shows the query responses generated for \( Q_1 \) at different times, along with their (\( tStart, tExp \)) intervals. If the client does not send a refresh for \( Q_1 \) before 14 min., the query expires and ceases to run continuously.

<table>
<thead>
<tr>
<th>Time</th>
<th>Responses (( tStart, tExp ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>{mr1[4, 6 + \Delta], mr2[4, 6 + \Delta], mr3[5, 6 + \Delta]}</td>
</tr>
<tr>
<td>6</td>
<td>{mr1[6, 8 + \Delta], mr3[6, 8 + \Delta]}</td>
</tr>
<tr>
<td>8</td>
<td>{mr1[8, 10 + \Delta], mr2[8, 10 + \Delta], mr3[8, 10 + \Delta]}</td>
</tr>
<tr>
<td>10</td>
<td>{mr1[10, 12 + \Delta], mr2[10, 12 + \Delta]}</td>
</tr>
<tr>
<td>12</td>
<td>{mr1[12, 14 + \Delta], mr2[12, 14 + \Delta]}</td>
</tr>
</tbody>
</table>

Table 1. Query responses returned to client for \( Q_1 \).

If a directory receives a refresh for a query that is not currently running, the directory processes the query as a new query to ensure that the client has current query responses. This situation can occur if a directory fails and restarts. Although a client may receive duplicate query responses, duplicates will not affect the correctness because all query responses contain UIDs.

The clients cache query responses that they receive from the directories. For each active query \( Q \) running at a client, the client maintains two data structures: a cache and a future response queue (FRQ). When a client receives a query response \( qr_a \) at
time \( t \), the client places \( qr_a \) in the cache if \( t \geq qr_a.t_{Start} \). Otherwise, it places \( qr_a \) in the FRQ. When \( Q_1 \) is issued in our example, the client puts responses from \( mr1 \) and \( mr2 \) in the cache, and the response from \( mr3 \) in the FRQ. If there is a query response in the FRQ or cache for the query with the same UID, the client deletes the old response. The FRQ keeps an index on the \( t_{Start} \) field. When the current time equals the \( t_{Start} \) field of the query response at the head of the queue, the client moves the query response to the cache. If a query response expires in the cache, the client deletes it. Thus, the cache for a query represents the continuously updated result set of the query.

3 Mobiscope Distributed Processing

In this section, we show how the Mobiscope architecture supports the above semantics in a distributed environment. As described above, a directory processes an ad by caching it, replacing any ad with the same UID, and finding any continuous queries that intersect the ad's position function for some portion of the ad's lifetime. Similarly, a directory processes a query by finding any cached ads that match the query, and then running the query continuously over any new incoming ads. Within a single directory, this processing is similar to the symmetric treatment of queries and ads in PSoup [3]; however, Mobiscope must perform this processing in a distributed environment. A directory processes an ad it receives if and only if the ad's position function intersects the directory's service region for some time in the ad's lifetime. Similarly, a directory only processes queries that intersect its service region. Directories route ads and queries among themselves to ensure full reachability. The routing protocols must be efficient in their use of processing, or they could introduce unneeded overhead into the directories.

The ad and query routing protocols ensure that a) all ads are routed to directories that can process them, and b) each query is routed to a set of Mobiscope directories \( mdset \) such that \( \bigcup_{md \in mdset} md.SR \) contains the query rectangle. An example Mobiscope network in Figure 2 demonstrates why ad and query routing is needed. In this example, we assume that a resource sends ads to a directory such that the resource resides in the directory's service region at the time it sends its ad. We also assume that clients can submit queries to any directory in the network. Without routing, query \( Q1 \) will not find ad \( a2 \), and \( Q1 \) and \( Q2 \) will not find \( a3 \) unless their clients submit them directly to \( md4 \). The following two sections explain the query and ad routing algorithms, respectively.

Query Routing Consider a new query \( Q \) that comes into a directory \( md \) directly from the client. The minimal set of directories with service regions that contain \( Q.rect \) can be approximated by the directories with service regions that intersect \( Q.rect \). To find these directories, \( md \) uses a spatial routing table. A spatial routing table provides a mapping between directories and service regions in a Mobiscope network. A spatial routing table entry has the format (directory address, service region, directory ID, service region version number). The directory ID uniquely identifies the directory in the network, and the version number increases each time a directory the service region changes.

Each directory in a Mobiscope network has a spatial routing table entry for every directory in the network. One or more coordinators keep routing tables consistent.
When a directory goes offline, comes online, or changes its service region, it increments its version number and notifies a coordinator of the routing table entry change \(^4\). The coordinator propagates these changes to all other directories. In this paper, we assume that routing table entry propagation will scale well because we assume directories do not come online, go offline, or change service regions frequently. Thus, neither the coordinator nor the directories should experience scalability problems in processing the changes. For scenarios where these assumptions do not hold, our technical report describes a hierarchical organization for directories where routing table information is disseminated more efficiently \([6]\). For fault tolerance, the coordinators should be

---

\(^4\) Directory failures can be detected by heartbeat messages from neighboring directories, which update the coordinator accordingly.
replicated to several nodes and use a distributed consistency protocol to maintain the routing table (e.g., [13]).

A directory can find the matching directories for a new query by checking each spatial routing table entry. Since Mobiscope is designed to handle a large number of long-running queries, however, the network is optimized for the scenario where refreshes are much more frequent than new queries. If there are many directories, checking all routing table entries for frequent refreshes could result in significant overhead. For each query received directly from a client that is routed to other directories, a directory creates an entry in a query hashtable, keyed on a concatenation of the client address and rectangle \(^5\). It attaches to this entry a list of directories to which it sent the query, along with the version numbers from their routing table entries. When a directory receives a refresh for the query directly from the client, it checks that each of the version numbers listed in the query entry is equal to the version number currently in the corresponding routing table entry. If all of these checks succeed, the directory routes the refresh to the directories listed in query entry. Only if a version check fails does the directory need to consult the routing table. Since we assume a client will generally send the initial query request and all refreshes for a query to the same directory, directories normally avoid checking the routing table to process a refresh, which can significantly reduce overhead.

**Ad Routing** Consider a resource that submits an ad \(a\) to a directory \(md\). To route \(a\), \(md\) could check the entire spatial routing table for matching service regions as it does for queries. This solution could create significant overhead if \(md\) needs to process ads rapidly and the number of directories is large. Instead, directories usually use use **intersection** and **adjacency** entries to more scalably route ads through the network. For a given directory, these are the entries with service regions that are either adjacent to or intersect the directory’s own service region. In our example in Figure 2, \(md1\) has intersection entries for directories \(md2\), \(md4\), and \(md5\), but has no adjacency entries. Each directory finds its intersection and adjacency entries by consulting its routing table.

Using intersection and adjacency entries will provide more scalable routing than checking each spatial routing table entry. Given reasonable service region overlap, there are usually few intersection and adjacency entries per directory. Also, the number of intersection and adjacency entries is not usually dependent on the size of the routing table. In Figure 2, any directory in the network has a maximum of 8 intersecting entries and no adjacency entries. As long as service region overlap does not grow, this number does not change if more directories are added in a similar configuration.

Directories can route most ads using only intersection and adjacency entries. Only under exceptional conditions, which we explain at the end of this section, does a directory default to checking the entire spatial routing table. Mobiscope ensures that each resource sends ads to a directory \(md\) such that the resource resides in \(md.SR\). When a resource comes online, it sends its position to a coordinator \(^6\) or a well-known directory. The coordinator or well-known directory finds a directory with a service

---

\(^5\) Although a client can issue multiple queries with the same rectangle, Mobiscope can effectively treat them as the same query

\(^6\) We assume the coordinator addresses are well known to all resources in the system
region that currently contains the resource, and sends the resource the corresponding routing table entry. The resource then transmits an ad to the directory associated with this entry.

Unlike queries, a directory attempts to route any ad it receives to other directories, regardless of whether the ad was sent directly from a resource or from another directory. A directory usually checks only adjacency and intersection entries to find directories that should process the ad. Before forwarding an ad to other directories, it adds itself to the ad’s visitedDirList (VDL). The VDL is a list of directories that have already processed the ad. A directory only sends an ad to a remote directory if a) the directory has an adjacency or intersection entry for the remote directory, b) the directory’s service region intersects the ad’s position function for some time in the ad’s lifetime, and c) the remote directory is not already on the ad’s VDL.

To illustrate ad routing, we refer to our example in Figure 2. In this example, the ad a1 is received by MD1 because the ad’s position function intersects MD1’s service region at time tArr_{a1}. MD1 puts its ID on a1’s VDL, and sends a1 to MD2. Similarly, MD2 puts its ID on a1’s VDL, and sends a1 to MD3. MD1 is in a1’s VDL, so MD2 does not send a1 back to MD1. Because MD2 is on a1’s VDL when it reaches MD3, MD3 does not send the ad to any other directories. Using this routing algorithm, a is routed to all directories that process a, given that a’s position function predicts that the resource will be in some area covered by a service region until a expires.

When a directory is processing an ad, it sends its own routing table entry to the resource if it did not receive the ad directly from the resource. This way, the resource will have the routing table entries for all directories that process the ad. If the ad’s position function predicts that the resource will always be in some area covered by a service region, then the resource always knows the appropriate directory to send a subsequent ad to. Thus, a coordinator or a directory only has to check all routing table entries when a resource comes online. In the above example, the resource that sends a1 will have routing table entries for directories MD1, MD2, and MD3. If the resource changes position functions at any time during the lifetime of a1, the resource will send a new ad to MD1, MD2, or MD3, depending on which service region the resource is at the time it sends the ad.

In some cases, an ad’s position function may predict that the resource will be in an area that is not covered by any service region. This condition may occur, for instance, when a directory fails. A directory routing a can recognize this case if i) the directory finds no other directories to route a to from its adjacency and intersection entries, and ii) a’s position function does not intersect the directory’s service region at time a.tExp. If this situation occurs, the resource exits the service region before its ad’s expiration time, but does not enter another service region. In our example, either MD1 or MD2 would find this situation with a1 if either MD2 or MD3 fails, respectively. In these cases, the directory defaults to checking the entire routing table.

4 Performance

In this section, we report on a performance study of the Mobiscope distributed processing algorithms. For this study, we implemented a workload generator and simulator to test the scalability of Mobiscope and Mobiscope’s behavior under workloads
that stress Mobiscope’s distribution model. In Section 4.1, we describe the workload generator and simulator. In Section 4.2, we describe the parameters varied in the study, and we present results in Section 4.3.

4.1 Simulation Environment

Our simulation environment consists of a workload generator and Mobiscope simulator, both written in C++. Our workload generator creates workloads that emulate queries over resources moving in a congested area with rapid position function updates. Although other moving object data generators are available (e.g. [17]), we wrote our own so that we could introduce parameters to stress our distribution model. The ad workload reflects resources that move in a 10 x 10 mi. area, traveling along paths in a grid that are .1 mi. apart. Resources traveling in such a grid could reflect cars moving around a city with square blocks. By default, the resources are initially evenly distributed. However, the initial placement of resources can be skewed to a “hot” region using a simulation parameter.

Resources are either stopped or moving along one of the paths. Every minute, a resource chooses a new direction or stops with equal probability. Resources move at a speeds between 35 and 55 mi./hr, chosen randomly. A resource transmits an ad after choosing a new trajectory. The ad represents the resource either stopped or moving along a grid line, with the upper and lower bound functions providing a 1/5 mi. error bound on both dimensions. All ads have the same lifetime, determined by a parameter.

The query workloads consist of long running queries with square query regions that cover a subset of the simulation area. Since queries are long running, most queries are already running when the simulation starts, but the workload generator also generates 10 new queries per minute. The query rectangle centers are distributed evenly throughout the area by default, but can also be skewed to the hot region with a simulation parameter. The length of a square side is also a parameter. All queries are refreshed once a minute.

All the workloads presented here have 1000 continuous queries initially running and 100000 mobile resources. For each set of parameters, the workload generator generates a set of ads and queries initially in the Mobiscope network, along with new ads, queries, and refreshes for a given period of simulated time. For each simulation, we generate ads, queries, and refreshes for 2 minutes of simulated time7.

The simulator runs the workloads over different Mobiscope network configurations. Each configuration consists of n directories, each with a unique service region that covers 1/nth of the total area plus overlap with other service regions. The directories are placed in a √n by √n matrix, and overlap is controlled by a parameter. Figure 2 shows an example configuration where n = 9.

The simulator determines the ads and queries that are processed by each directory, and runs the workload against each directory in sequence. The directory implementation functions as described in Section 3, but it receives ads and queries via procedure

---

7 We ran additional experiments where we varied the number of moving resources from 10000 to 200000, the queries from 1000 to 10000, and the simulated time from 1 to 5 min. Although the magnitudes were different, the overall trends were essentially the same as the reported 100000 resources /1000 initial continuous query case
Table 2. Simulation Parameters

<table>
<thead>
<tr>
<th>Parameter Description</th>
<th>Values Range (default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of directories</td>
<td>1-250</td>
</tr>
<tr>
<td>% of each service region that overlaps other service regions</td>
<td>64-100  [5]</td>
</tr>
<tr>
<td>Ad expiration time [min.]</td>
<td>1-7.5  [1]</td>
</tr>
<tr>
<td>Length of the side of each query [m:]</td>
<td>20-50  [25]</td>
</tr>
<tr>
<td>Percentage of resources initially in hot region</td>
<td>60-250-100  [60-250]</td>
</tr>
<tr>
<td>Percentage of query rectangle centers in hot region</td>
<td>60-250-100  [60-250]</td>
</tr>
</tbody>
</table>

calls instead of from real resources and clients. The implementation processes ads and queries by checking them against each continuous query and cached ad, respectively. Directory processing might be improved by indexing the queries and ads with spatial indexes (see [8] for a survey) and spatio-temporal indexes (e.g. [17, 12]), respectively. Unfortunately, finding a spatial index that will improve performance is highly dependent on the workload. As stated in [10] , there is currently no thorough, unbiased experimental analysis to guide systems designers in choosing a spatial index for a given workload. The design space of spatio-temporal indexes is even less well understood. Although a performance study of various indexes under Mobiscope workloads provides interesting future work, this paper concentrates on studying the performance of the distribution model.

For each run, the initial queries and ads are loaded into the directory before the simulation begins. For each configuration and workload, we report the “wall clock” time for one directory to process all ads and queries in its entire workload for the given simulated time period. The directory is usually chosen from the middle of the directory matrix. For the experiments varying ad and query skew, however, we report the time for the directory covering the hot region. Experiments report a mean time for each set of parameters. Workloads for each set of parameters was generated and run repeatedly until the 95% confidence interval of the times was within 10% of the mean.

4.2 Experiment Parameters

Table 2 shows the parameters that we vary in our experiments, along with the ranges used and the default values. We tried to vary parameters that either a) test the scalability of the Mobiscope network, or b) reduce the effectiveness of Mobiscope’s distribution model.

To test scalability, we vary the number of directories in a configuration while keeping the workload fixed. These tests allow us to measure the incremental performance benefit of extra directories. To test the routing protocol performance, we also ran experiments with “naive” routing. In naive routing, a directory that receives an ad, query, or refresh directly from a resource or a client routes it by checking the entire routing table. To isolate the routing overhead, we ran experiments both with the ad and query processing turned off as well as normal processing.

We vary the rest of the parameters in Table 2 to increase the overhead due to the Mobiscope distribution model. In our study, we run several experiments to observe the
effects of increasing the proportion of ads and queries that are processed by more than one directory. A larger number of ads and queries processed by multiple directories creates more ad and query routing, as well as redundant processing costs at multiple directories.

We can vary several parameters to increase the number of these ads and queries. One such parameter is the amount of area a Moboscope directory service region overlaps with neighboring service regions. An increase in overlap increases both the number of queries and ads processed by multiple directories. Increasing ad expiration time increases the average number of directories with intersecting service regions. Similarly, an increase in query side length increases the average number of directories that processes each query.

In addition to increasing the number of ads and queries processed by multiple directories, we also introduce ad and query skew into our workloads to stress the distribution model. As opposed to increasing the overhead at each directory by increasing the number of ads and queries processed by multiple directories, we reduce the parallelization in Moboscope by increasing the amount of ad and query skew. As ads and queries are increasingly skewed toward a hot region, only one or a few directories will process most of the ads and queries. In our skew experiments, the hot region for both ads and queries is 1/16th of the total simulation area. The hot region is covered by one directory's service region, and we report this directory's time in the skew experiments.

4.3 Results

**Experiment 1: Vary Number of Directories** Table 3 shows the processing and routing times for different numbers of directories, using both naïve and Moboscope routing. The one directory time is given only for comparison. In this and all other experiments, all parameters except for those varied are set to the default values as listed in Table 2.

We expect to initially see large benefits to increasing the number of directories due to parallelization. At high numbers of directories, however, ads and queries match several service regions, and the performance increases should be diminished. At high numbers of directories, we expect to see much of the processing time from ad and query routing.

These experiments confirm the expected trends. The processing performance increase from using 4 directories instead of 1 is more than tenfold, but the performance increase from using 256 instead of 144 directories is less than double. At 16 directories and above, the routing time is more than 1/5th of the total processing time. In addition, we can see a large benefit to the Moboscope routing policy in comparison to the naïve routing policy when the number of directories is large. At 256 directories, for instance, the naïve processing time is more than 3 times that of the Moboscope processing time, and almost all of the time difference is due to routing costs.

**Experiment 2: Vary Service Region Overlap** In this experiment, we vary the percent of service region overlap and report the processing times. We define percent
of overlap to be the percent of area that a given service region overlaps with any other service region. Table 4 shows the results. For the remaining experiments, we only run configurations with 1 (1dir), 4 (4dir), and 16 (16dir) directories. In this experiment, we increase service region all the way to 100% to create a workload where all ads and queries match multiple service regions. The processing times for 16 and 4 directories increase significantly with overlap. Even at 100% overlap, however, multiple directories still process their workloads much faster than a single directory.

<table>
<thead>
<tr>
<th>Number of Directories</th>
<th>Total Processing Time (sec.)</th>
<th>Ad and Query Routing Time Only (sec.)</th>
<th>Total Processing Time (Naive, sec.)</th>
<th>Ad and Query Routing Time Only (Naive, sec.)</th>
<th>% SR Overlap</th>
<th>1dir (sec.)</th>
<th>4dir (sec.)</th>
<th>16dir (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>41.331</td>
<td>N/A</td>
<td>41.331</td>
<td>N/A</td>
<td>20</td>
<td>41.331</td>
<td>41.331</td>
<td>41.331</td>
</tr>
<tr>
<td>4</td>
<td>0.541</td>
<td>0.060</td>
<td>0.482</td>
<td>0.060</td>
<td>40</td>
<td>41.331</td>
<td>4.531</td>
<td>0.818</td>
</tr>
<tr>
<td>16</td>
<td>0.466</td>
<td>0.032</td>
<td>0.473</td>
<td>0.037</td>
<td>60</td>
<td>41.331</td>
<td>5.446</td>
<td>0.996</td>
</tr>
<tr>
<td>64</td>
<td>0.165</td>
<td>0.018</td>
<td>0.185</td>
<td>0.035</td>
<td>80</td>
<td>41.331</td>
<td>6.183</td>
<td>1.118</td>
</tr>
<tr>
<td>144</td>
<td>0.089</td>
<td>0.013</td>
<td>0.117</td>
<td>0.039</td>
<td>100</td>
<td>41.331</td>
<td>6.766</td>
<td>1.359</td>
</tr>
<tr>
<td>256</td>
<td>0.030</td>
<td>0.006</td>
<td>0.092</td>
<td>0.055</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Vary Number of Directories

Experiment 3: Vary Query Side Length In Figure 3, we graph the processing times for simulations with different query side lengths. Note that the y-axis in Figure 3 is on a logarithmic scale. At higher values of query side length, each incoming ad matches more queries on average, creating a higher cost for generating a query response. Thus, the single directory processing time rises with query side length and expiration time. However, ads and queries that are processed by multiple directories still add significant overhead, as the increase in 16 and 4 directory processing times is more than proportional to the single directory increase. 16 directories takes less than 1/50th the processing time of 1 directory when query side length = .25 mi. At query side length = 5 mi., this ratio is more than 1/6th.

Again, we use extreme parameter values to stress the Mobiscope distribution. At their maximum values shown here, query sides are half the length of the total simulation region. Even with these large query areas, 16 and 4 directories still significantly outperform the single directory. In addition to query side length, we ran similar experiments where we varied ad expiration time from 1 to 7.5 minutes. These experiments showed similar trends to the experiments varying query side length, and thus we do not show this graph here in the interest of conserving space.

Experiment 4: Vary Percent of Ads and Queries in the Hot Region To measure the effects of ad and query skew, we vary the percentage of queries and ads in the hot region. Figure 4 shows processing times for various percentages of query centers in the hot region when the resources are either initially evenly distributed or all placed in the hot region. Unlike Figure 3, the y-axis in Figure 4 is not on a
logarithmic scale. First, consider the processing time growth as the queries in the hot region increase and resources are kept evenly distributed. As expected, the 16 and 4 directory processing times rise with query skew. However, 1 directory processing time decreases slightly. This decrease is caused by the many ads that generate no query responses when most queries are in the hot region and ads are evenly distributed. When all mobile resources are initially in the hot region, the 1 directory processing time also increases with query skew. Because all ads are now all in the hot region, an increase in query skew increases the average number of matching queries per ad.

Note that when either ads or queries are evenly distributed, total skew in the other workload does not cause the 16 or 4 directory processing times to approach that of 1 directory. Not until all ads and most queries are in the hot region do the 16 and 4 directory processing times come close to the corresponding 1 directory processing time.

![Fig. 3. Vary Query Side Length](image3)

![Fig. 4. Vary Percent of Queries and Moving Resources In Hot Region](image4)

5 Related Work

Besides the MODBMS work mentioned earlier, much of the service discovery literature is also related to Mobiscope, such as [11, 1, 2, 22, 9]. None of these services efficiently
support discovery of moving network resources based on location. To our knowledge, none of these systems process advertisements with position functions. These services would face serious problems dealing with moving resources that advertise a continuously changing position.

Like Mobiscope, most service discovery services are distributed to multiple nodes. Service standards such as LDAP [22] and SLP [9] use very elementary distribution models that are not apropos to Mobiscope. Berkeley SDS [11] and VIA [2] attempt to route queries to only the relevant nodes by filtering queries as they travel through the system. Berkeley SDS filters queries through an administratively defined hierarchy of nodes to increase scalability. VIA nodes adaptively form clusters to determine query filtering policies based on query workloads. Berkeley SDS and VIA are both primarily built for queries with equality predicates, although [11] describes a design that allows Berkeley SDS to support range queries. None of these services have protocols that route queries and ads to nodes based on geography.

Outside of service discovery, two projects use similar routing protocols to Mobiscope: the CAN distributed hashtable [16] and Geocast [14]. CAN hashes a lookup key to a d-dimensional point, and each node is responsible for the points within a d-dimensional bounding rectangle in a d-dimensional torus. CAN uses routing very similar to Mobiscope to route both data items and requests to the correct node. Like other distributed hash tables, CAN relies on a hash function to distribute the keys uniformly; thus, CAN only routes on equality searches and maintains no locality within a node. The Geocast protocol uses routing protocols similar to Mobiscope to route messages between hosts based on their position. Although Geocast does allow a form of addressing that specifies a region as a destination, it does not allow hosts to express positions as functions.

Most recently, the continuous query work in the database community shows some similarity to Mobiscope. Like Mobiscope, PSOUP presents a system where a query response can consist of data that arrives either before or after a query arrives at the system [3]. Although PSOUP is built on a single node, the work on Flux shows how continuous query systems can be parallelized across nodes in a cluster [18]. However, neither of these systems deal with the problems of queries over moving objects.

Both PSOUP and Flux were developed as part of U.C. Berkeley’s TelegraphCQ project [7]. The Telegraph project, along with the IBM iQueue system for processing streaming data from pervasive data sources [4, 5], was the inspiration for the Mobiscope service. In the future, we hope to integrate Mobiscope with both these projects.

6 Conclusion

Mobiscope is a service where clients submit long running queries to continuously find the set of network resources currently in a given region. Like moving object DBMSs (MODBMSs), Mobiscope represents position with functions over time. Unlike MODBMSs, Mobiscope uses a new query model to first run queries over cached position functions, and then continuously over subsequent functions received after the query. For scalability, Mobiscope distributes the ad and query workload to multiple directories by geography. Directories use application-level routing protocols based on geography so that all queries find all matching resources. Our performance study of
Mobiscope shows that the Mobiscope service scales well, even under workloads which stress Mobiscope’s distribution model.

References