Efficient Proximity Detection among Mobile Users via Self-Tuning Policies

Man Lung Yiu [#] Leong Hou U [†] Simonas Šaltenis [‡] Kostas Tzoumas [‡]
[#]Hong Kong Polytechnic University [†]University of Hong Kong [‡]Aalborg University
[#]csmlyiu@comp.polyu.edu.hk [†]hleongu@cs.hku.hk [‡]{simas,kostas}@cs.aau.dk

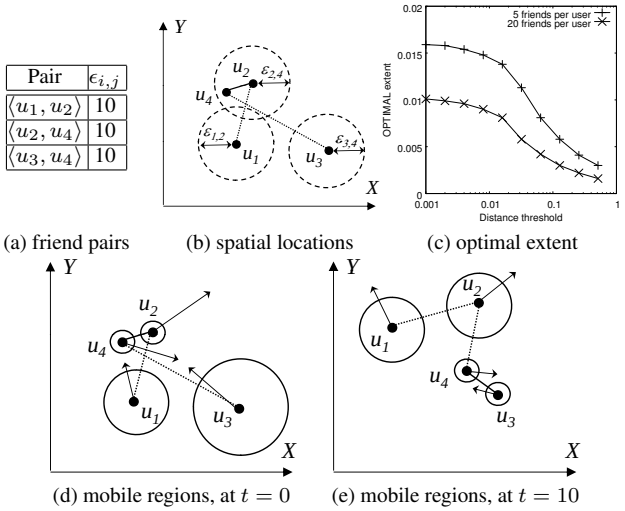
ABSTRACT

Given a set of users, their friend relationships, and a distance threshold per friend pair, the *proximity detection* problem is to find each pair of friends such that the Euclidean distance between them is within the given threshold. This problem plays an essential role in friend-locator applications and massively multiplayer online games. Existing proximity detection solutions either incur substantial location update costs or their performance does not scale well to a large number of users. Motivated by this, we present a centralized proximity detection solution that assigns each mobile client with a mobile region. We then design a *self-tuning* policy to adjust the radius of the region automatically, in order to minimize communication cost. In addition, we analyze the communication cost of our solutions, and provide valuable insights on their behaviors. Extensive experiments suggest that our proposed solution is efficient and robust with respect to various parameters.

1. INTRODUCTION

Given a set U of mobile users, the social network G among them, and a spatial distance threshold $\epsilon_{i,j}$ per friend pair, the *proximity detection* problem [1] reports each pair $\langle u_i, u_j \rangle$ that satisfies two conditions: (i) the users u_i and u_j are adjacent in G, and (ii) the Euclidean distance $dist(u_i, u_j)$ between them is at most $\epsilon_{i,j}$. Figure 1a illustrates the friend pairs among 4 users $u_1, u_2, \cdots, u_4$. Locations of these users are shown in Figure 1b. For instance, the pair $\langle u_2, u_4 \rangle$ belongs to the result because they are friends located within the distance $\epsilon_{2,4}$ from each other (see their solid line). In contrast, u_4 is not within the distance $\epsilon_{3,4}$ from u_3, so the pair $\langle u_3, u_4 \rangle$ is not in the result (see their dotted line).

The proximity detection problem finds important applications for moving users, in both the real and the virtual worlds. The high availability of location positioning technologies (e.g., GPS) integrated with powerful mobile communication devices enables a new class of location-based online services. In particular, emerging friend-locator services[1] such as Google Latitude, Fire Eagle, and Ipoki, allow a user to share his location with friends and find the ones close to him. AdSocial [11] is yet another prototype that allows users to share their locations with friends via mobile devices. The proximity detection functionality is a natural extension of such services, where a reasonable ϵ value could be the distance of 10 street blocks, for instance. Similarly, for massively multiplayer online games, users wish to receive notifications when allied members come sufficiently close.



(a) friend pairs (b) spatial locations (c) optimal extent

(d) mobile regions, at $t = 0$ (e) mobile regions, at $t = 10$

Figure 1: Example of proximity detection

For mobile applications, communication cost is the most important optimization goal, due to the limited bandwidth and battery power on the users' mobile devices [4, 8, 9, 21]. In addition, the optimization of communication cost also helps alleviating the server's computational load as the clients contact the server less often. A straightforward proximity detection solution forces each client (i.e., user) to report his location to the server periodically (e.g., every second). Such a brute force solution requires that the service provider must invest large amounts of resources on communication bandwidth at the server side. Thus, it is essential to develop an efficient proximity detection technique that reduces the communication cost between the server and the users.

The importance of the above applications calls for the design of communication-efficient proximity detection methods. The proximity detection problem was first introduced by Efrat and Amir [1]. Recently, Xu and Jacobsen investigated a more general variant of the problem [20]. Both solutions suffer from certain drawbacks. The first is a distributed solution and requires each user to maintain information for every friend, placing heavy burden at the clients. The second solution is centralized and incurs a substantial commu-

[1]http://www.google.com/latitude/
http://fireeagle.yahoo.net/
http://www.ipoki.com/

Proceedings of the VLDB Endowment, Vol. 3, No. 1

nication cost caused by frequent location updates of clients.

Motivated by this, we design a centralized solution that leverages the wealth of research on *location-tracking policies* [17, 19]. Instead of maintaining the user's exact position, the server represents the user by a *mobile region* that is guaranteed to contain the user's exact position. A user updates his position, sending a message to the server, only when he has moved out of his mobile region. The server searches for pairs of users that satisfy the proximity detection constraints using the mobile regions, rather than the exact user locations. This search process can result in false positives, which need to be further filtered by probing a few users for their exact locations.

The extent of a mobile region only affects the communication cost of; it does not influence the correctness of our solution. Ideally, the client should use an optimal extent value for his mobile region such that the total communication cost is minimized. However, even for uniform data and workload, our experimental result shows that the optimal extent varies with respect to the distance threshold ϵ and the number of friends per user, as shown in Figure 1c. Our challenge resides in automatically and dynamically *tuning* the right extent for each mobile region, in order to reduce the total communication cost. This problem is difficult because: (i) the right extent varies among different objects, and (ii) the right extent changes as time passes. We illustrate property (i) by Figure 1d, where the mobile region of each user is represented by a moving circle centered at the user's location. Since u_3 is far from u_4, we should assign a large extent to the mobile region of u_3, in order to reduce the frequency of updates sent from u_3 to the server. As u_2 and u_4 are close together, we should assign small extents to both the mobile regions of u_2 and u_4, such that the server has more accurate information and reduces the frequency of probing them. The transition from Figure 1d to Figure 1e illustrates property (ii). As time passes, the distances among the users change, so their optimal extents also change.

The contributions of this paper are summarized as follows.

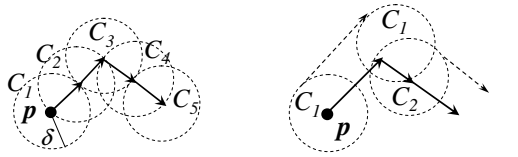
- We present a basic client-server solution, called FMD, that employs a fixed-radius policy for mobile regions and detects the proximity among users by utilizing their mobile regions.
- We develop a communication cost model of FMD, which enables us to demonstrate that the fixed-radius policy is not powerful enough for minimizing the communication cost.
- We design two methods that dynamically tune the mobile regions of the users, and analyze their dynamic behavior.

The rest of the paper is organized as follows. Section 2 surveys the relevant related work. Section 3 discusses the assumed architecture of the system. Section 4 presents a tracking-based solution for proximity detection and analyzes its communication cost, then develops self-tuning solutions for the problem and analyzes their dynamic runtime behavior. Section 5 presents the results of the performance experiments, followed by conclusions in Section 6. Table 1 (in Appendix A) summarizes the notation that will be used throughout the paper. The Appendix also contains proofs of the lemmas in this paper and supplementary experimental details.

2. RELATED WORK

2.1 Location Tracking and Adaptive Caching

A location tracking policy enables the server to track the location of a mobile user with a given distance bound δ. The objective is to minimize the communication cost between the server and its clients. The *point-based update policy* [19] represents a user p by a static circle with its radius as δ and its center as the last reported location of p. Instead of knowing the user's exact position, the server only knows that the user is located inside the circle (with the radius δ). In Figure 2a, the user p is moving along the arrows. While staying within the circle C_1, the user does not need to send any location update to the server. When p leaves C_1, it defines a new circle C_2 and issues an update to the server. Subsequent circles are created as p continues moving. Instead of using a static circle, the *vector-based update policy* [17] uses a moving circle whose center is predicted by using the last-reported location and velocity of the object (see Figure 2b). The vector-based policy incurs lower communication cost than the point-based policy in real world data sets [17].



(a) point-based update policy (b) vector-based update policy

Figure 2: Example of location tracking

In the approximate caching problem [10], a remote data source maintains the exact value v of a data item, whereas the server caches an approximate copy $\mathcal{V}$ (i.e., an interval) of the item such that $\mathcal{V}$ contains v. When the value v changes such that $v \notin \mathcal{V}$, the data source performs a *value refresh* by enlarging the interval $\mathcal{V}$ and updating its copy at the server. At the server, incoming queries are evaluated by using $\mathcal{V}$ whenever possible. In case $\mathcal{V}$ is not precise enough for answering a query q, the server performs a *query refresh* by requesting the data source to shrink the interval $\mathcal{V}$ such that it satisfies the precision requirement of q. Tzoumas et al. [16] apply approximate caching to the two-dimensional domain for maintaining an efficient index on moving objects at a centralized server. Unlike our problem, their goal is to optimize the server-side computation cost, but not the communication cost between the server and the users.

2.2 Continuous Spatial Query Processing

Server-side continuous spatial query processing methods can be divided into two categories: instant monitoring and predictive evaluation. *Instant monitoring* [6, 7] aims at maintaining the query result up-to-date only, based on the rationale that the future locations of objects and queries are unpredictable. The server typically refreshes the result periodically (every ΔT time units), according to location updates received from the moving objects. In this category, SINA [6] and CPM [7] are representative solutions that employ a spatial partitioning grid at the server side for efficiently maintaining the result of range queries and k nearest neighbor queries, respectively. *Predictive evaluation* [5, 22] models the future locations of objects by linear motion functions, and predicts the query result from now to future. When the motion function of an object changes, it issues an update to the server, which recomputes the (future) query result associated with the object. Iwerks et al. [5] examine temporal events that lead to future update of result, and develop algorithms to maintain the result for k nearest neighbor queries and spatial join queries on moving points. Zhang et al. [22] study the continuous processing of intersection join between two sets of moving rectangles, assuming that each rectangle issues an update to the server within at most T_M timestamps. Their solution exploits T_M to reduce the effort of server-side processing. All work above focuses on computational efficiency at the server but

ignores communication cost between users and the server. In addition, techniques [5] and [22] ignore friend pairs and disallow personalized thresholds $\epsilon_{i,j}$.

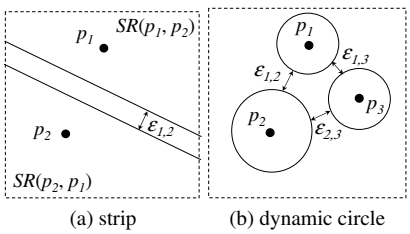
The *safe region* concept has been extensively studied for saving the communication cost on processing continuous queries. For the scenario of static (range or kNN) queries on moving objects [4, 8], the safe region $SR(p)$ of an object p is defined as a region such that the result of any query is guaranteed to be unchanged by p while p stays within $SR(p)$. For the scenario of moving queries on static objects [9, 21], the safe region $SR(q)$ of a query point is defined as a region such that its query result remains unchanged in the region. Regarding a kNN query, Zhang et al. [21] computes an order-k Voronoi cell as the safe region of q. Nutanong et al. [9] formulates a larger safe region for q by using the $(k + \Delta k)$ nearest neighbors of q, where Δk is a tunable parameter that decides the trade-offs between communication cost and computational cost. None of the above methods are directly applicable to proximity detection, in which both the users and their friends are moving objects.

The proximity detection problem has been studied in [1, 14, 15, 20]. Efrat and Amir [1] propose a distributed solution where mobile users communicate with each other. Specifically, for each pair of friends p_1 and p_2, a strip of width ϵ is defined between their last-reported locations. Since this solution does not employ a server, the safe region of an object is defined per object pair. In Figure 3a, the safe region $SR(p_1, p_2)$ of p_1 relative to p_2 is the region above the strip. While p_1 travels within $SR(p_1, p_2)$ it does not need to send its location to p_2. Otherwise, p_1 must send its location to p_2, and p_2 determines whether they are within proximity. In addition, p_2 informs p_1 of the new strip between them. This approach requires each user to maintain a strip for each of his friends, thus its performance does not scale well to a large number of friends. In the worst case, when a user travels across the strips of all friends, he needs to send a message to every friend. This problem can be avoided in client-server solutions where the clients only need to communicate with the server but not with all friends.

Treu et al. [14, 15] develop client-server solutions that focus on reducing the communication cost of proximity detection. The *dynamic centered circle* method [14] assigns each user p_i a circle such that the minimum distance between any two circles is above ϵ (see the example of Figure 3b). However, the circle is static, causing the user to move beyond it soon, triggering a location update to the server. In a related work, Treu et al. [15] employ moving sector regions for tracking users' locations and detecting the proximity among them at the server. The moving sector region of a user is described by three parameters with predefined values: an angular threshold θ, the minimum speed V_{min} and the maximum speed V_{max}. Our analysis in Section 4.2 shows that such predefined parameter values fail to achieve low communication costs in all cases. Motivated by this, our solutions aim at optimizing the communication cost by automatically tuning their parameters.

Xu and Jacobsen [20] generalize the proximity detection problem to the *constraint detection* problem. A constraint is satisfied when a specified set of k objects can be enclosed by a circle with a diameter of at most ϵ. They propose a centralized solution, which tracks objects in a space-partitioning grid. An object (i.e., client) does not issue any location updates to the server until it enters another cell of the grid. Based on the locations of the objects in the grid, their solution identifies the objects that definitely satisfy the constraints and the objects that definitely dissatisfy the constraints. The remaining objects are probed to get their precise locations and the constraints are checked for them. Their solution consists of an adaptive procedure [20] for splitting and merging cells, in order to reduce the computational cost at the sever. In contrast, our main objective is to optimize the communication cost of our solutions.

Table 2 of Appendix A summarizes the characteristics of the above proximity detection methods. Most of existing work adopt the client-server architecture. The adaptive multi-layer grid [20] is the only work that employs automatic tuning techniques but it focuses on the server CPU cost. Our paper is the first to study automatic tuning techniques for optimizing the communication cost.



(a) strip (b) dynamic circle

Figure 3: Safe regions for different queries

3. PROBLEM SETTING

We first present the system architecture for proximity detection, and then introduce preliminary concepts.

3.1 System Architecture

Following the setting in continuous spatial query processing [6–8], we adopt a client-server architecture (see Figure 4). Each client (i.e., user) maintains a set of motion parameters that (conservatively) describe its current location and the predicted future movement. The server stores the set U of users, their motion parameters, the set G of friend pairs, and the proximity detection threshold $\epsilon_{i,j}$ for each friend pair.

We adopt the *instant monitoring* framework [6–8] as described in Section 2.2. Each user measures his location periodically every ΔT time units (e.g., every second), and the server checks the proximity among users periodically every ΔT time units. The value ΔT is termed as the *epoch*. In fact, periodic location measurement is common for both real-world positioning (e.g., with GPS) and virtual-world positioning in online games. Note that the value ΔT also captures the minimum latency in the communication network [8].

Users communicate with the server via messages. A user may send an *update message* to the server, updating its location and motion parameters stored at the server. The server can send a *probing message* to a user, requesting him to issue an update message. When the server detects that two friends u_i and u_j are within proximity (i.e., $dist(u_i, u_j) \leq \epsilon_{i,j}$), the *proximity-notification message* will be sent to those two users. Our main goal is to reduce the overall communication cost which is measured as the sum of all messages (update, probing, notification) per epoch.

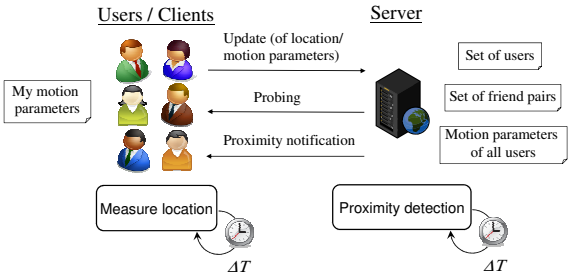


Figure 4: System architecture

3.2 Preliminary Concepts

We first define the concept of a mobile region, which will be used extensively in this paper. The movement of the user u can be described by three *motion parameters*:

- $u.T_{ref}$, the time at the user's last update to the server
- $u.P_{ref} = (u.X_{ref}, u.Y_{ref})$, the location of u at $u.T_{ref}$
- $u.V_{ref} = (u.V_X, u.V_Y)$, the velocity of u at $u.T_{ref}$

Given any time t such that $t \geq u.T_{ref}$, the predicted location $(u.X(t), u.Y(t))$ of u at time t is modeled by two linear motion functions [18] as shown in Equations 1 and 2. The type of motion function used is independent of the correctness of our proposed solutions (to be discussed in Lemma 3 later). The correctness is still preserved even if the users have unpredictable motion patterns.

Let $u.\lambda$ be the radius parameter of a user u. The *mobile region* $R_t(u)$ of u is defined as the time-dependent circle with its radius $u.\lambda$ and its center $(u.X(t), u.Y(t))$ (shown in Equation 3). Here, $u.\lambda$ serves a different purpose than the user-defined bound δ used in location tracking (described in Section 2.1). Our $u.\lambda$ is an *internal system parameter* to be set by the application but not the user.

$$u.X(t) = u.X_{ref} + u.V_X \cdot (t - u.T_{ref}) \quad (1)$$

$$u.Y(t) = u.Y_{ref} + u.V_Y \cdot (t - u.T_{ref}) \quad (2)$$

$$R_t(u) = \bigodot ((u.X(t), u.Y(t)),\ u.\lambda) \quad (3)$$

The value of $u.\lambda$ affects the communication cost, but not the correctness of our proposed solutions. The server only knows that a user is located in his mobile region $R_t(u)$, but not the user's exact location. When a user moves outside his mobile region, he must update his mobile region so that the server's information is always kept valid.

Given two circles R and R', we use $mindist(R, R')$ to denote the minimum distance between R and R', and $maxdist(R, R')$ to denote their maximum distance [20].

$$mindist(R, R') = \max\{dist(R.c, R'.c) - R.\lambda - R'.\lambda, 0\} \quad (4)$$

$$maxdist(R, R') = dist(R.c, R'.c) + R.\lambda + R'.\lambda \quad (5)$$

where $R.c$ denotes the center point of R, and $R.\lambda$ represents the radius of R. Based on these distance bounds, we obtain the following lemmas, which enable the server to determine the proximity between the users u_i and u_j without knowing their exact locations. The server needs to probe the users u_i and u_j for their exact locations only when both condtions below do not apply.

LEMMA 1. **Unqualified-pair pruning.**
At time t, if $mindist(R_t(u_i), R_t(u_j)) > \epsilon_{i,j}$, then the exact distance between u_i and u_j is greater than $\epsilon_{i,j}$.

LEMMA 2. **Qualified-pair detection.**
At time t, if $maxdist(R_t(u_i), R_t(u_j)) \leq \epsilon_{i,j}$, then the exact distance between u_i and u_j is at most $\epsilon_{i,j}$.

4. PROXIMITY DETECTION METHODS

Section 4.1 presents a basic solution for proximity detection, by setting the radius $u.\lambda$ of each user's mobile region to a fixed value. Section 4.2 studies the communication cost model of the basic solution, and reveals the drawback of using a fixed radius for the users' mobile regions. Section 4.3, presents solutions that automatically tune $u.\lambda$ for each user, in order to optimize the communication cost. Finally, Section 4.4 conducts an analysis on the dynamic behavior of our self-tuning solutions.

4.1 Fixed-Radius Mobile Detection (FMD)

Based on the mobile region concept, we proceed to develop our *Fixed-Radius Mobile Detection Method* (FMD) for continuous proximity detection among the users. It consists of a client-side algorithm and a server-side algorithm. Recall from Section 3 that both the client algorithm and the server algorithm are invoked every ΔT time units, via a timer event. At the end of this section we discuss how to eliminate this restriction.

The client-side algorithm for user u proceeds as follows. First, the user's mobile region $u.\lambda$ is set to a pre-defined, system-wide value λ. The client waits for an event E, which can be either (i) a timer event triggered by the timer, or (ii) a probing event sent by the server. In case E is a timer event, the client checks whether its current location falls outside its mobile region $R_t(u)$. If so, then it issues an update to the server (with updated motion parameters $u.T_{ref}$, $u.P_{ref}$, and $u.V_{ref}$). In case E is a probing event, the client needs to issue an update to the server as well. Algorithm 1 (in Appendix C) is the pseudocode for the FMD client algorithm.

The server maintains a set Γ that stores the result pairs per epoch. The server algorithm examines each pair $\langle u_i, u_j \rangle$ in the set G of friend pairs and employs the standard filter-refinement processing, which covers the following three cases.

- The first case is to detect a pair that cannot belong to the result, by checking whether the minimum distance between the mobile regions $R_t(u_i)$ and $R_t(u_j)$ is greater than the proximity threshold $\epsilon_{i,j}$ (see Lemma 1). If so, the server needs not further process the pair.
- The second case is to detect a pair that must belong to the result, by checking whether the maximum distance between $R_t(u_i)$ and $R_t(u_j)$ is within $\epsilon_{i,j}$ (see Lemma 2). If so, the pair is inserted into the result set Γ.
- The last case is the *refinement step* which probes the exact locations of u_i, u_j, for deciding whether they are actually within proximity. If so, the pair is inserted into the result set Γ. A set U' stores the users who issued updates in this epoch, so that they cannot be probed multiple times.

Finally, the server notifies the user pair to be within proximity if it belongs to the result set Γ. Algorithm 2 (in Appendix C) is the pseudocode for the server algorithm. The correctness of the FMD algorithm is proved in this lemma.

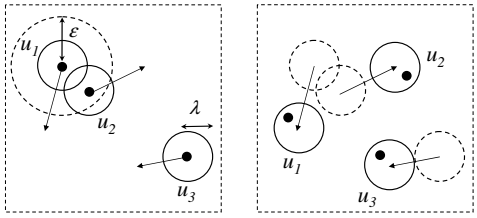
LEMMA 3. **Correctness of FMD.**
The result set Γ computed by the FMD algorithm contains no false positives and no false negatives, regardless of the prediction function $(u.X(t), u.Y(t))$ being used.

Example of FMD. We proceed to illustrate how the algorithms work with an example. For simplicity, we assume that all friend pairs use the same ϵ value, and each user is a friend of every other user. Figure 5a shows the locations of users u_1, u_2, and u_3 at time $t = 0$. Each user u_i is enclosed by a mobile region $R_t(u_i)$, whose radius equals to λ (a system parameter). The velocity of each mobile region is indicated by an arrow. Since $mindist(R_t(u_3), R_t(u_1)) > \epsilon$, the server determines that the pair $\langle u_3, u_1 \rangle$ cannot be within proximity. Similarly, the pair $\langle u_3, u_2 \rangle$ is found to be not in the result set. As $mindist(R_t(u_1), R_t(u_2)) \leq \epsilon$, it is possible for the users u_1 and u_2 to be within proximity. Thus, the server probes the exact locations of u_1 and u_2, then detects their proximity, and sends them a proximity message. Figure 5b depicts the locations of the users at the next epoch (i.e., $t = \Delta T$). Even though the users' movements deviate slightly from their motion functions (i.e., the arrows defined in the last epoch), the users

still travel within their mobile regions so they do not send any updates to the server. Note that the minimum distance between the mobile regions of each pair of users is greater than ϵ. Thus, the server does not send proximity message to any user.

Optimizations. In Appendix D.1, we study how to reduce the communication cost of proximity-notification messages in FMD by using the incremental notification technique [6].

The time complexity of the server algorithm is $O(|G|)$ per epoch because all friend pairs in the set G are examined in every epoch. In Appendix D.2, we optimize the computational cost of the server algorithm by adapting event time triggers [5]. Furthermore, this technique makes the server algorithm a truly continuous algorithm, which avoids using the epoch parameter ΔT.



(a) at time $t = 0$ (b) at time $t = \Delta T$

Figure 5: Example of mobile regions

4.2 Communication Cost Model of FMD

In this section, we develop a model for the communication cost of the FMD method, which will reveal that the fixed-radius policy is not suitable for minimizing the communication cost.

Similar to previous research [13], we make certain uniformity assumptions in order to keep the cost model tractable. In addition, the uniformity assumptions represent the worst case for our approach, which benefits from skew. The goal of the cost model is to reveal the influences of problem parameters and guide us in the development of advanced techniques in the next section.

We first discuss the parameters in our cost model. We use ΔT to denote the length of an epoch, and λ to denote the radius of a mobile region. Let N be the number of users and m be the average number of friends per user. For simplicity, all friend pairs use the same ϵ for proximity detection, and locations of objects are uniformly distributed in the spatial domain $[0, 1]^2$.

In the following, we focus on deriving the total communication cost (i.e., the number of messages) of our solution per epoch. Specifically, the total communication cost CT_{total} can be decomposed into three components: (i) The proximity notification cost, CT_{notify}, measures the notification messages from the server to users that satisfy the proximity detection conditions. (ii) The update cost, CT_{update}, is caused by a client moving beyond its mobile region. (iii) The probe cost, CT_{probe}, is caused by probing messages from the server to users that are processed by the "refinement step" of the server algorithm described above, as well as the associated invoked location updates by these users.

We ignore CT_{notify} in our analysis because it is independent of the mobile region radius λ. Also, the incremental proximity notification technique only affects only CT_{notify} but not CT_{update} nor CT_{probe}, so it is not analyzed here.

Cost of update. To estimate the cost of updates caused by a client moving beyond its mobile region, we need to estimate the average interval between two updates of a user $U(\lambda)$. The function $U(\lambda)$ is highly dependent on the movement type and the user's velocity. For simplicity, we assume the following movement model. First, observe that the user u_i issues an update when it reaches the border of the (moving) circle C_i. At the time when C_i is built, the user u_i is located at the center of C_i and moves with some velocity vector V. If the user would continue to move with this velocity, no updates would be necessary. Instead, we assume that, in-between the updates, the user moves with a different average velocity vector V'. Let $\Delta V = |V' - V|$. Then, $U(\lambda) = \lambda/\Delta V$ and the expected number of epochs for the user to perform an update is $U(\lambda)/\Delta T$. Thus, the probability for a user to update (per epoch) is shown in Equation 6. By multiplying this probability with the number N of users, we obtain the total update cost CT_{update} in Equation 7.

$$Pr_{update}(\lambda) = \min\{\frac{(\Delta V)(\Delta T)}{\lambda}, 1\} \quad (6)$$

$$CT_{update} = N \cdot \min\{\frac{(\Delta V)(\Delta T)}{\lambda}, 1\} \quad (7)$$

Cost of probing. When the minimum distance between two circles C_i and C_j is within ϵ, the distance between the centers of C_i and C_j must be between 0 and $\epsilon + 2\lambda$. Thus, the probability that $mindist(C_i, C_j) \leq \epsilon$ is given by Equation 8. When the maximum distance between two circles C_i and C_j is within ϵ, the distance between the centers of C_j and C_j must be between 0 and $\max\{\epsilon - 2\lambda, 0\}$. Thus, the probability that $maxdist(C_i, C_j) \leq \epsilon$ is given by Equation 9. The refinement step of the server side algorithm is executed when (i) the minimum distance between the mobile regions of u_i and u_j is at most ϵ, and (ii) their maximum distance is greater than ϵ. Therefore, the probability of invoking the refinement step for the pair $\langle u_i, u_j \rangle$ is shown in Equation 10.

$$Pr(mindist(C_i, C_j) \leq \epsilon) = \pi(\epsilon + 2\lambda)^2 \quad (8)$$

$$Pr(maxdist(C_i, C_j) \leq \epsilon) = \pi(\max\{\epsilon - 2\lambda, 0\})^2 \quad (9)$$

$$Pr_{refine}(\lambda, \epsilon) = \pi(\epsilon + 2\lambda)^2 - \pi(\max\{\epsilon - 2\lambda, 0\})^2 \quad (10)$$

Since we assume that each user has m friends on average, the probing message is sent to the user u_i if he participates in the refinement step of any of his friends. Thus, the probability of sending a probing message to u_i is: $(1 - (1 - Pr_{refine})^m)$. Furthermore, the probing message is only needed when u_i has not issued any update in the current epoch, which happens with probability $(1 - Pr_{update}(\lambda))$. When the probing message is sent to u_i, an update message will be sent back from u_i to the server, i.e., two messages in total. Thus, the total probing cost for all users is:

$$\begin{aligned} & CT_{probe} \quad (11) \\ = \; & N \cdot (1 - Pr_{update}(\lambda)) \cdot \left(1 - \left(1 - Pr_{refine}(\lambda, \epsilon)\right)^m\right) \cdot 2 \\ = \; & 2N \cdot \left(1 - \min\{\frac{(\Delta V)(\Delta T)}{\lambda}, 1\}\right) \\ & \cdot \left(1 - \left(1 - \pi(\epsilon + 2\lambda)^2 + \pi(\max\{\epsilon - 2\lambda, 0\})^2\right)^m\right). \end{aligned}$$

Discussion. From the equations of CT_{update} and CT_{probe}, we observe that a small λ value leads to high update cost, whereas a large λ value incurs high probing cost. In fact, there exists an optimal value of λ that minimizes the total communication cost of updating and probing. The optimal λ value also depends on other factors such as ΔV, m, and ϵ. For instance, a large ΔV leads to high update cost so the optimal λ should have a high value. A high m leads to high probing cost so the optimal λ should be small.

In Appendix E.1, we apply our cost model to estimate the total communication cost of the FMD method. From the estimation results, we observe two important properties regarding the optimal value of λ: (i) Given the values of ϵ and m, there exists an optimal λ such that the overall communication cost is minimized, and (ii) The optimal λ varies for different values of m and ϵ.

4.3 Self-Tuning Mobile-Region Algorithms

In real-life scenarios, the spatial distribution of users' locations can be skewed, and it can dynamically change over time. In addition, different users could use different proximity distance thresholds. These factors make the cost model in Section 4.2 inaccurate for estimating the optimal λ value for the FMD method. To tackle this problem, we present self-tuning solutions for automatically tuning the extent of each user's mobile region.

Instead of fixing the value of λ, we now associate each user u with an individual, variable radius value $u.\lambda$. Equation 3 is modified by replacing λ with $u.\lambda$. Initially, the value $u.\lambda$ is initialized to a predefined radius value, λ_0. In addition, we employ a tuning parameter α, which will be described shortly. Our tuning methods should be designed in such a way that:

- the value of $u.\lambda$ can be automatically tuned to its optimal value fast, regardless of the initial value λ_0
- the automatic tuning process should be robust for a wide range of values for α

The rationale is that, the client needs not be worried about choosing the values of λ_0 and α, and yet the achieved performance is comparable to that of using the optimal values of λ_0 and α.

Expansion and contraction of mobile regions. We start by introducing two operations, called contraction and expansion, for adjusting the extents of mobile regions.

Let α be the *scale-factor* parameter, where $\alpha > 1$. The *contraction* operation recomputes the mobile region $R_t(u)$ of the user u, by multiplying $u.\lambda$ with $\frac{1}{\alpha}$ and using the updated motion functions $u.X(t)$ and $u.Y(t)$ of u. This operation is used to reduce the refinement probability in Equation 10, in order to save probing cost in future. The *expansion* operation recomputes the mobile region $R_t(u)$, by multiplying $u.\lambda$ with α and using the updated motion functions of u. This operation is applied to reduce the update probability in Equation 6, in order to save update cost in future.

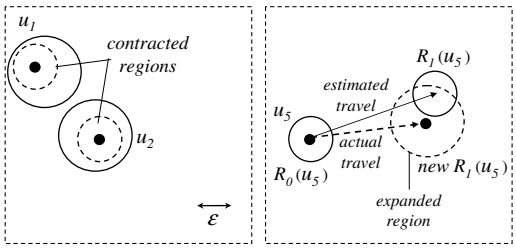
From the analysis in Section 4.2, we discover that the update cost is high at a small $u.\lambda$ whereas the probing cost is high at a large $u.\lambda$. We thus propose the following intuitive principle to reduce the update cost and the probing cost.

PRINCIPLE 1. **Tuning Principle**
If the update probability is too high (i.e., too small $u.\lambda$), the expansion operation should be applied.
If the probing probability is too high (i.e., too large $u.\lambda$), the contraction operation should be applied.

Based on the above principle, we present two systematic techniques for self-tuning $u.\lambda$ for each client. They do not necessarily tune $u.\lambda$ to its optimal value; but they definitely safeguard against undesirable cases (e.g., against too small or too large values of $u.\lambda$).

Reactive Mobile Detection (RMD). Our *Reactive Mobile Detection* method (RMD) is an extension of FMD with the following modifications. There are two cases for the client to issue an update to the server. The first case is triggered when the client moves outside its mobile region; the client should then perform the expansion operation in order to save update cost in the future. The second case is caused by probing from the server; the client should then perform the contraction operation in order to save probing cost in the future.

Let's consider the friend pair $\langle u_1, u_2 \rangle$ in Figure 6a, with the value $\alpha = 2$. The pair executes the refinement step as the minimum distance between their mobile regions is within ϵ. The server probes the locations of users u_1 and u_2, and then they apply the contraction operation on their mobile regions. In Figure 6b, the actual movement of the user u_5 (dotted arrow) deviates from its estimated movement (solid arrow, known at time $t = 0$). At time $t = 1$, the location of u_5 falls outside the mobile region $R_1(u_5)$. The expansion operation is used to update the mobile region of u_5.



(a) region contraction (b) region expansion

Figure 6: Self-tuning of mobile regions, $\alpha = 2$

Cost-Based Mobile Detection (CMD). Our *Cost-Based Mobile Detection* method (CMD) is an extension of FMD with the following modifications. Each user u maintains two local counters: (i) $\mathcal{CT}_{update}(u)$, for keeping the number of messages caused by updates, and (ii) $\mathcal{CT}_{probe}(u)$, for keeping the number of messages caused by probing. Initially, the user sets both counters to zero.

When the user encounters an update, the value of $\mathcal{CT}_{update}(u)$ is incremented by one. When the user receives a probing message from the server, the value of $\mathcal{CT}_{probe}(u)$ is incremented by two (because the user needs to send a message back to the server). Each time the user is about to do an update or receives a probing message, the user compares $\mathcal{CT}_{update}(u)$ and $\mathcal{CT}_{probe}(u)$. If $\mathcal{CT}_{update}(u) > \mathcal{CT}_{probe}(u)$, the user aims to reduce the future update cost by performing the expansion operation on his mobile region. If $\mathcal{CT}_{probe}(u) > \mathcal{CT}_{update}(u)$, the user aims to reduce the future probing cost by performing the contraction operation.

The responsiveness of both self-tuning methods can be adjusted by changing α. Intuitively, a large α value improves the self-tuning ability of the algorithms but they can become over-sensitive to small changes in the workload. We explore the effects of the different values of α in our performance study.

4.4 Dynamic Behavior Analysis

In this section, we provide insights into the dynamic behavior of the RMD method, from the perspective of a single user u. Figure 7 depicts the state diagram of a single user. The state of a user consists of its λ value. Assume that the particular value of the user's λ is currently λ_i. The state diagram captures the transitions between states which are of three kinds. First, with probability ps_i the user remains in the same state λ_i if its mobile region did not contract or expand. Second, with probability pc_i the user can move to a state $\lambda_{i-1} = \frac{1}{\alpha} \cdot \lambda_i$ if its mobile region contracted. This incurs a communication cost of 2. Finally, with probability pe_i the user can move to a state $\lambda_{i+1} = \alpha \cdot \lambda_i$ if its mobile region expanded. This incurs a communication cost of 1. Note that pe_i and pc_i are mutually exclusive. Based on the equations in Section 4.2, we derive:

$$\begin{aligned} pe_i &= Pr_{update}(\lambda) \\ pc_i &= (1 - Pr_{update}(\lambda)) \cdot \left(1 - \left(1 - Pr_{refine}(\lambda, \epsilon)\right)^m\right) \\ ps_i &= 1 - pe_i - pc_i \end{aligned}$$

Using these probabilities, we apply the Monte Carlo method to traverse the state diagram of Figure 7, for estimating a user's communication cost. Appendix E.2 demonstrates that RMD converges fast to the optimal state, regardless of the initial value of λ used.

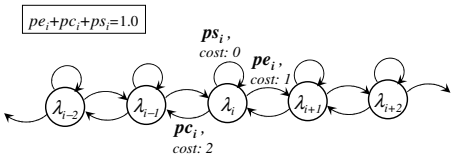


Figure 7: State diagram of the RMD method

5. EXPERIMENTAL STUDY

In this section, we experimentally evaluate the communication cost of the proposed algorithms. The communication cost is our primary optimization goal, as elaborated in the introduction. It is also a platform-independent measurement. We implemented our proposed solutions and the solutions developed in related work [1, 14, 20]. Table 3 (in Appendix A) summarizes the used algorithms. For fairness to the AMLG algorithm [20], we set its parameters according to those used in [20], i.e., the partition split/merge parameter w is set to $0.0025 \cdot N$, where N is the number of users.

Experimental setting. Table 4 (in Appendix A) summarizes the default values and ranges of the parameters used in our experimental study. The proximity distance threshold ϵ is used for all algorithms, whereas the parameters λ and α are only applicable to our algorithms.

The network-based moving object data generator [3] is used to generate the movement of N users on the Oldenburg road network during 100 timestamps. We normalize the spatial domain size to $[0, 1000]^2$. A distance unit represents 1 meter and the interval time between two adjacent timestamps is 1 second. The data generator allows us to control the speed limit V_{limit} (i.e., the maximum possible speed) of all users.

The default social network is a randomly generated social graph called SYN. The graph contains N nodes and each node has an average of m adjacent edges (i.e., corresponding to m friends). We also obtained a social network corresponding to the co-authorship relation in the DBLP Bibliography[2], where each node represents an author and each edge indicates that the corresponding authors are co-authors in some paper. The graph is obtained from the DBLP records during the years 1998–2007; it has $N = 330354$ nodes and each node has an average degree of $m = 6.261$. We generate the users' locations for this DBLP network by using the data generator [3] described above.

Sensitivity of internal system parameters. We first study the effect of the parameters λ and α on our proposed methods (FMD, RMD, and CMD).

Figure 8a shows the cost of the methods with respect to the initial mobile-region radius λ. The cost trend of FMD agrees with our analysis in Section 4.2, even though the moving objects generated by [3] do not satisfy the uniformity assumption used in our analysis. At very small λ value, the cost of FMD is high due to frequent location updates. At very large λ value, FMD also has high cost because it suffers from frequent probing by the server. On the other hand, the cost of our proposed self-tuning methods RMD and CMD is relatively insensitive to the value of λ. The reason is that they adjust each user's mobile region radius ($u_i.\lambda$) dynamically based on the location-update events and probing events they encounter. Due to the high communication cost, the method FMD is removed from the subsequent experiments.

Figure 8b plots the cost of our self-tuning methods RMD and CMD as a function of the scale factor α. Observe that both methods achieve low cost for a reasonable range of α values, namely from 1.25 to 8. At a very small α value (e.g., 1.01), the mobile regions of users are adjusted slowly so the cost remains high. At a large α value (e.g., 16), the methods over-adjust the mobile regions frequently so the cost becomes high. Note that RMD incurs a lower cost than CMD for all α values.

Additional experiments on studying the dynamic behavior of RMD and CMD can be found in Appendix F.1.

Variants of RMD and CMD. We then study different variants of RMD and CMD. By default, RMD and CMD are employing: (i) the incremental notification technique for reporting proximity status, and (ii) the vector-based update policy for location tracking.

A variant method is indicated with the suffix DN if it uses direct notification (i.e., notifying the users for each epoch whenever their proximity is detected). The suffix PT is shown with a variant method that applies the point-based policy for location tracking.

Figure 8c plots the communication cost breakdown of the variants, at the default experimental setting. Note that the notification method (i.e., with DN vs. the default) only affects the number of proximity messages to be sent to the users, but not the number of messages sent from the users. The location-tracking method (i.e., with PT vs. the default) influences both types of message cost. A less effective tracking method (e.g., with PT) leads to a high amount of update messages generated from clients.

In the above experiment, we also measure the number of proximity events (per epoch) occurred at the client side. The number of proximity events is the same for all the variants, even though they employ different notification techniques and location tracking techniques.

Scalability experiments. In the subsequent experiments, we compare the communication cost of our self-tuning solutions (RMD and CMD) with the competitors (DS, DCC, and AMLG), with respect to various parameters.

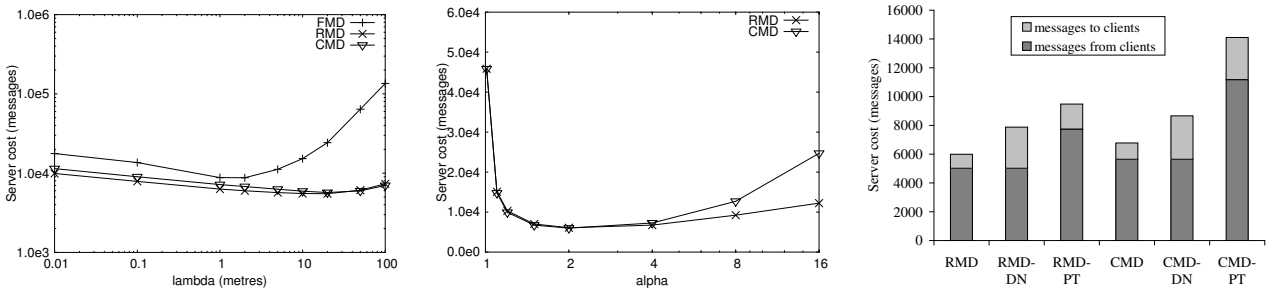
Figure 9a shows the cost of the methods as a function of the number of users N. The cost of RMD/CMD is the lowest and it scales linearly as N increases. This suggests that our proposed methods are practical for realistic applications involving a huge number of users, like the friend-locator applications and massively multiplayer online games mentioned in the introduction.

Appendix F.2 presents additional experiments on studying the scalability of the methods in terms ϵ, m, and the speed limit.

Results on real data. Finally, we study the performance of the methods for the scenario using a real social network — the DBLP co-authorship graph. This network possess some inherent properties of real network that may not be easily reflected by the random graph SYN we used earlier. Thus, the DBLP graph would provide us indicative performance of the methods on realistic social networks. Figures 9b,c show the cost of the methods on the DBLP graph, as a function of the proximity detection distance ϵ and the speed limit, respectively. DS incurs very high cost at large ϵ values. AMLG and DCC can have high cost as they do not utilize the velocity information of the users to reduce the update cost. Observe that RMD/CMD outperforms others in the figures.

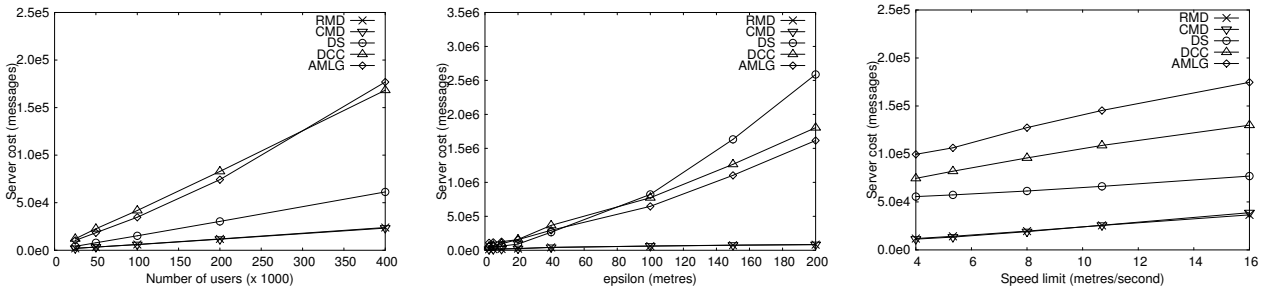
Summary. The experimental study shows that our proposed self-tuning proximity detection methods (RMD and CMD) have robust performance with respect to various parameters and their communication costs are scalable to a large number N of users and to a large number m of friends per user. Furthermore, the sensitivity experiments show that RMD and CMD incur low communication cost for a wide range of internal system parameter values λ and α. The dynamic behavior experiments demonstrate that, for extreme

[2]http://www.informatik.uni-trier.de/ ley/db/



(a) cost vs. λ (b) cost vs. α (c) cost breakdown, at the default setting

Figure 8: Sensitivity experiments, SYN graph



(a) varying N, SYN graph (b) varying ϵ, DBLP graph (c) varying speed limit, DBLP graph

Figure 9: Scalability experiments

values of λ and α, the cost of RMD converges faster than CMD to a steady state.

6. CONCLUSIONS

Motivated by applications like friend-locator location-based services and massively multiplayer online games, we develop communication efficient client-server algorithms for continuous proximity detection among mobile users. The algorithms build on the previous research on update policies in location tracking in order to issue updates only when a user exits a moving region associated with its predicted position. In addition, the number of location probes from the server to the users is reduced by controlling the sizes of the moving regions of the users. We develop self-tuning mechanisms for providing continuous adjustment of the extents of moving regions. Performance experiments show that our algorithms are robust with respect to various parameters and they have substantially lower communication cost than existing solutions.

7. REFERENCES

[1] A. Amir, A. Efrat, J. Myllymaki, L. Palaniappan, and K. Wampler. Buddy Tracking - Efficient Proximity Detection Among Mobile Friends. In *INFOCOM*, 2004.

[2] R. Benetis, C. S. Jensen, G. Karčiauskas, and S. Šaltenis. Nearest and Reverse Nearest Neighbor Queries for Moving Objects. *VLDBJ*, 15(3):229–249, 2006.

[3] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2):153–180, 2002.

[4] H. Hu, J. Xu, and D. L. Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *SIGMOD*, 2005.

[5] G. S. Iwerks, H. Samet, and K. P. Smith. Maintenance of K-nn and Spatial Join Queries on Continuously Moving Points. *ACM TODS*, 31(2):485–536, 2006.

[6] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, 2004.

[7] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *SIGMOD*, 2005.

[8] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao. A Threshold-Based Algorithm for Continuous Monitoring of k Nearest Neighbors. *IEEE TKDE*, 17(11):1451–1464, 2005.

[9] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The V*-Diagram: A Query-dependent Approach to Moving KNN Queries. *PVLDB*, 1(1):1095–1106, 2008.

[10] C. Olston, B. T. Loo, and J. Widom. Adaptive Precision Setting for Cached Approximate Values. In *SIGMOD*, 2001.

[11] E. Sarigöl, O. Riva, P. Stuedi, and G. Alonso. Enabling Social Networking in Ad Hoc Networks of Mobile Phones. *PVLDB*, 2(2):1634–1637, 2009.

[12] Y. Tao and D. Papadias. Spatial Queries in Dynamic Environments. *ACM TODS*, 28(2):101–139, 2003.

[13] Y. Theodoridis and T. K. Sellis. A Model for the Prediction of R-tree Performance. In *PODS*, 1996.

[14] G. Treu and A. Küpper. Efficient Proximity Detection for Location Based Services. In *Workshop on Positioning, Navigation and Communication (WPNC)*, 2005.

[15] G. Treu, T. Wilder, and A. Küpper. Efficient Proximity Detection among Mobile Targets with Dead Reckoning. In *MOBIWAC*, 2006.

[16] K. Tzoumas, M. L. Yiu, and C. S. Jensen. Workload-Aware Indexing of Continuously Moving Objects. *PVLDB*, 2(1):1186–1197, 2009.

[17] A. Čivilis, C. S. Jensen, and S. Pakalnis. Techniques for Efficient Road-Network-Based Tracking of Moving Objects. *IEEE TKDE*, 17(5):698–712, 2005.

[18] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.

[19] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Distrib. Parallel Databases*, 7(3):257–387, 1999.

[20] Z. Xu and H.-A. Jacobsen. Adaptive Location Constraint Processing. In *SIGMOD*, 2007.

[21] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based Spatial Queries. In *SIGMOD*, 2003.

[22] R. Zhang, D. Lin, K. Ramamohanarao, and E. Bertino. Continuous Intersection Joins Over Moving Objects. In *ICDE*, 2008.

APPENDIX

A. LIST OF SYMBOLS, ALGORITHMS, AND EXPERIMENTAL PARAMETERS

Table 1: Summary of Notation

Symbol	Meaning
U	the set of users
u_i	a user in U
G	the set of friend pairs (u_i, u_j): $u_i, u_j \in U$
$\epsilon_{i,j}$	proximity distance threshold for (u_i, u_j)
$u_i.X(t), u_i.Y(t)$	(predicted) motion functions of u_i
$R_t(u_i)$	mobile region of u_i at time t
$u_i.\lambda$	the radius of the mobile region $R_t(u_i)$
$dist(p, p')$	Euclidean distance between points p and p'
$mindist(R, R')$	minimum dist. between regions R and R'
$maxdist(R, R')$	maximum dist. between regions R and R'
α	the scale factor (tuning parameter)

Table 2: Summary of Proximity Detection Methods

Method	Architecture	Goal	Auto-tune	Personal $\epsilon_{i,j}$
Distributed Strip [1]	distributed	comm.	×	✓
Dynamic Centered Circle [14]	client-server	comm.	×	✓
Dynamic Sector [15]	client-server	comm.	×	✓
Adaptive Multi-Layer Grid [20]	client-server	server CPU	✓	✓
Iwerks et al. [5]	client-server	server CPU	×	×
Zhang et al. [22]	client-server	server CPU	×	×
RMD / CMD (this paper)	client-server	comm.	✓	✓

Table 3: Summary of Algorithms

Alg.	Name	Presented in
FMD	Fixed-Radius Mobile Detection	Section 4.1
RMD	Reactive Mobile Detection	Section 4.3
CMD	Cost-Based Mobile Detection	Section 4.3
DS	Distributed Strip	Ref. [1]
DCC	Dynamic Centered Circle	Ref. [14]
AMLG	Adaptive Multi-Layer Grid	Ref. [20]

Table 4: Summary of Parameters

Parameter	Default	Range
Number of users N	100K	25K – 400K
Speed limit V_{limit} (metres/s)	8.00	4.00 – 16.00
Number of friends per user m	10	3 – 500
Proximity distance ϵ (metres)	10	2.5 – 50
Initial mobile region radius λ	20	2 – 100
Scale factor α	2	1.01 – 16
Length of an epoch ΔT	1 second	

B. PROOFS OF LEMMAS

PROOF. **Proof of Lemma 1.**
The region $R_t(u_i)$ contains the actual location of u_i at the time t. Also, the region $R_t(u_j)$ contains the actual location of u_j at the time t. Therefore, $mindist(R_t(u_i), R_t(u_j))$ is smaller than or equal to the exact distance between u_i and u_j. Since we are given that $mindist(R_t(u_i), R_t(u_j)) > \epsilon_{i,j}$, the exact distance between u_i and u_j is greater than $\epsilon_{i,j}$. □

PROOF. **Proof of Lemma 2.**
The region $R_t(u_i)$ contains the actual location of u_i at the time t. Also, the region $R_t(u_j)$ contains the actual location of u_j at the time t. Therefore, $maxdist(R_t(u_i), R_t(u_j))$ is greater than or equal to the exact distance between u_i and u_j. Since we are given that $maxdist(R_t(u_i), R_t(u_j)) \leq \epsilon_{i,j}$, the exact distance between u_i and u_j is at most $\epsilon_{i,j}$. □

PROOF. **Proof of Lemma 3.**
Lines 4–6 of Algorithm 1 (the client algorithm) guarantees that the mobile region $R_t(u)$ of the user contains the current location of the user u at current time t, regardless of the prediction function being used. Note that this property is the pre-condition of Lemmas 1 and 2.

In Algorithm 2 (the server algorithm), a friend pair $\langle u_i, u_j \rangle$ is inserted into Γ either at Line 7 or at Line 15. In the former case, it is guaranteed to be an actual result, due to Lemma 2. In the latter case, the exact locations of the users u_i and u_j are probed at Lines 9–12, before computing their exact distance. Thus, the result set Γ contains no false positives.

A friend pair $\langle u_i, u_j \rangle$ is not included into Γ if it does not satisfy the condition at Line 6 or Line 14. In the former case, the pair is guaranteed not to be a result, due to Lemma 1. In the latter case, the exact distance between the users is computed for the checking at Line 14. Therefore, Γ does not have any false negatives. □

C. PSEUDO-CODES OF ALGORITHMS

Algorithm 1 Mobile Region Detection Client

algorithm MobileRegion-Detect-Client(User u, Event E, Radius λ)
1: let the current time be t;
2: measure the current location of u as (x_{cur}, y_{cur}) and velocity as (V_X, V_Y);
3: **if** E is a timer event **then**
4: **if** $R_t(u)$ does not contain (x_{cur}, y_{cur}) **then**
▷ *the client moved outside its mobile region*
5: $u.T_{ref}$:=t; $u.P_{ref}$:=(x_{cur}, y_{cur}); $u.V_{ref}$:=(V_X, V_Y);
▷ *location update message (contributes to cost CT_{update})*
6: **send** $R_t(u)$ (with its parameters) to the server;
7: **else if** E is a probing event (from the server) **then**
▷ *the server probed for our exact location*
8: $u.T_{ref}$:=t; $u.P_{ref}$:=(x_{cur}, y_{cur}); $u.V_{ref}$:=(V_X, V_Y);
▷ *location update message (contributes to cost CT_{probe})*
9: **send** $R_t(u)$ (with its parameters) to the server;

Algorithm 2 Mobile Region Detection Server

algorithm MobileRegion-Detect-Server(Distances $\epsilon_{i,j}$, Set of users U, Set of friend pairs G)
1: let the current time be t;
2: **receive** updates from the subset of users of $U' \subseteq U$ issuing them;
3: Γ:= an empty set; ▷ *the result set in this epoch*
4: **for each** friend pair $\langle u_i, u_j \rangle \in G$ **do**
5: **if** $mindist(R_t(u_i), R_t(u_j)) \leq \epsilon_{i,j}$ **then** ▷ *Lemma 1*
6: **if** $maxdist(R_t(u_i), R_t(u_j)) \leq \epsilon_{i,j}$ **then** ▷ *Lemma 2*
7: insert the pair $\langle u_i, u_j \rangle$ into Γ;
8: **else** ▷ *refinement step*
▷ *probe messages (contribute to cost CT_{probe})*
9: **if** $u_i \notin U'$ **then**
10: **send** a probing event to u_i for its update;
11: **if** $u_j \notin U'$ **then**
12: **send** a probing event to u_j for its update;
13: U':=$U' \cup \{u_i, u_j\}$;
14: **if** $dist(u_i.P_{ref}, u_j.P_{ref}) \leq \epsilon_{i,j}$ **then**
15: insert the pair $\langle u_i, u_j \rangle$ into Γ;
16: **if** $\langle u_i, u_j \rangle \in \Gamma$ **then**
▷ *notification messages (contribute to cost CT_{notify})*
17: **notify** users u_i and u_j with a proximity message;

D. OPTIMIZATIONS

D.1 Optimization on Proximity Notification

The original server algorithm (Algorithm 2) sends a proximity message to the pair $\langle u_i, u_j \rangle$ if it belongs to the current result set Γ (see Lines 16–17). For example, the pair $\langle u_i, u_j \rangle$ belongs to Γ from epoch 1 to epoch 4, in Figure 10. This incurs expensive communication cost as the server needs to send four notification messages to those pairs.

We apply the incremental notification technique [6] to reduce the number of proximity messages to be sent to the users. The idea is to report the *status change* of proximity rather than the proximity itself. Let Γ be the result set in the current epoch and Γ' be the result set in the previous epoch (see Figure 10).

- If the pair $\langle u_i, u_j \rangle$ belongs to Γ but not Γ', then the current epoch is the initial epoch for u_i and u_j to be within proximity. Thus, a plus-status message is sent to that pair (e.g., the status change in epoch 1 in the figure).
- At the client side, each user assumes that his proximity status with his friends is identical to that of the previous epoch, unless he receives status message from the server. For instance, the users u_i and u_j assume themselves to be within proximity from epoch 2 to epoch 4 (see Figure 10), without needing the server to send them any message.
- If the pair $\langle u_i, u_j \rangle$ belongs to Γ' but not Γ, then the current epoch is the initial epoch for u_i and u_j not to be within proximity. Thus, a minus-status message is sent to that pair (e.g., the status change in epoch 5 in the figure).

This technique reduces the message cost to 2, when compared to the message cost (4) paid in the original method discussed before.

To implement the above technique, it suffices to replace a few lines of Algorithm 2 by the corresponding lines of Algorithm 3. This optimization leads to significant savings in communication cost, especially for the scenario where a pair of friends are staying or traveling together.

Epoch	0	1	2	3	4	5	6	7
$\langle u_i, u_j \rangle \in \Gamma$	×	✓	✓	✓	✓	×	×	×
$\langle u_i, u_j \rangle \in \Gamma'$	×	×	✓	✓	✓	✓	×	×
Status change		+				−		

Figure 10: Proximity status of a friend pair $\langle u_i, u_j \rangle$

Algorithm 3 Server Algorithm with Notification Optimization

```
      ......
 3:  Γ:= an empty set; Γ':= the result set in the last epoch;
 4:  for each friend pair ⟨u_i, u_j⟩ ∈ G do
      ......
16:      if ⟨u_i, u_j⟩ ∈ Γ ∧ ⟨u_i, u_j⟩ ∉ Γ' then
17:          notify users u_i and u_j with a plus-status message;
18:      else if ⟨u_i, u_j⟩ ∈ Γ' ∧ ⟨u_i, u_j⟩ ∉ Γ then
19:          notify users u_i and u_j with a minus-status message;
```

D.2 Server Computational Cost Optimization

Algorithm 2 (at the server-side) needs to examine each friend pair $\langle u_i, u_j \rangle$ in every epoch. This incurs substantial computational cost at the server. In this section, we present an effective technique for filtering out the friend pairs that cannot contribute to the result at the current epoch. Furthermore, this technique enables the server algorithm to become a truly continuous algorithm, which avoids using the time epoch parameter ΔT.

Trigger time concept. The idea of trigger time originates from [5]. Given two users u_i and u_j, the *trigger time* $\omega(u_i, u_j)$ is defined as the earliest time t (after the current time t_{cur}) such that the minimum distance between their mobile regions $R_t(u_i)$ and $R_t(u_j)$ at time t is within ϵ.

$$\omega(u_i, u_j) = \min\{t \mid t \geq t_{cur} \wedge mindist(R_t(u_i), R_t(u_j)) \leq \epsilon\}$$

Unless u_i or u_j sends an update, the pair $\langle u_i, u_j \rangle$ is guaranteed to be not within proximity until the time $\omega(u_i, u_j)$, so it needs not be processed until that time.

The distance between two moving points have been studied in [2,5,12]; we extend their work to computing the minimum distance between two mobile regions (i.e., moving circles).

Computation of trigger time. To compute the trigger time $\omega(u_i, u_j)$ between two moving circles $R_t(u_i)$ and $R_t(u_j)$, we first compute the possible time interval such that the distance between two circles is within ϵ. Let λ_i and λ_j be the radii of the moving circles $R_t(u_i)$ and $R_t(u_j)$ respectively. Without the loss of generality, let us assume that $u_i.T_{ref} = u_j.T_{ref} = 0$. Equations 1 and 2 then become simpler. The motion functions of a user u_i along X and Y dimensions are:

$$\begin{aligned} u_i.X(t) &= S_{i,X} + V_{i,X} \cdot t \\ u_i.Y(t) &= S_{i,Y} + V_{i,Y} \cdot t \end{aligned}$$

where $S_{i,X} = u_i.X_{ref}$, $S_{i,Y} = u_i.Y_{ref}$, $V_{i,X} = u_i.V_X$, and $V_{i,Y} = u_i.V_Y$.

The following inequality is used to find the possible time interval such that the minimum distance between $R_t(u_i)$ and $R_t(u_j)$ is within ϵ.

$$\begin{aligned} mindist(\ &\odot((u_i.X(t), u_i.Y(t)), \lambda_i)\ , \\ &\odot((u_j.X(t), u_j.Y(t)), \lambda_j)\ ) \quad \leq \quad \epsilon \end{aligned}$$

By applying Equation 4, the above inequality between two circles can be expressed as another inequality between their centers, as shown below:

$$(u_i.X(t) - u_j.X(t))^2 + (u_i.Y(t) - u_j.Y(t))^2 \leq (\epsilon + \lambda_i + \lambda_j)^2$$

After expanding the terms $u_i.X(t)$, $u_j.X(t)$, $u_i.Y(t)$, $u_j.Y(t)$, we then obtain:

$$\begin{aligned} &((S_{i,X} - S_{j,X}) + (V_{i,X} - V_{j,X}) \cdot t)^2 + \\ &\quad ((S_{i,Y} - S_{j,Y}) + (V_{i,Y} - V_{j,Y}) \cdot t)^2 \quad \leq \quad (\epsilon + \lambda_i + \lambda_j)^2 \end{aligned}$$

The above inequality can be rearranged to a quadratic inequality: $A \cdot t^2 + B \cdot t + C \leq 0$ where A, B, and C are constants based on the values $S_{i,X}$, $S_{i,Y}$, $V_{i,X}$, $V_{i,Y}$, $S_{j,X}$, $S_{j,Y}$, $V_{j,X}$, $V_{j,Y}$, and ϵ. We solve the above inequality to find the possible time interval for t, and then set $\omega(u_i, u_j)$ to the earlier time $t \geq t_{cur}$ in such an interval (if any).

Efficient indexing of trigger times. Next, we investigate how to index the trigger times of friend pairs in an efficient manner. Specifically, we propose to use a main-memory heap for organizing the friend pairs $\langle u_i, u_j \rangle$ based on their trigger time $\omega(u_i, u_j)$. A disk-based B^+-tree can be used instead, if the main memory is not large enough to store all friend pairs. At the current time, the pairs satisfying $\omega(u_i, u_j) = t_{cur}$ are retrieved from the top of the heap $\mathcal{H}$ and then processed by Lines 5–17 of Algorithm 2. When the server receives an update from a user u_i, its trigger time $\omega(u_i, u_j)$ for each friend u_j is recomputed and the heap is updated accordingly.

Now, we show how this approach can be further improved using a two level heap structure. For every user u_i, the trigger time $\omega(u_i, u_j)$ of his friend pairs are stored in a local heap $\mathcal{H}_i^{loc}$. Only the top pair on $\mathcal{H}_i^{loc}$ is inserted into the global heap $\mathcal{H}$. When the trigger time $\omega(u_i, u_j)$ of a pair is changed, first its local heap $\mathcal{H}_i^{loc}$ is updated. If the top pair of $\mathcal{H}_i^{loc}$ is changed, then $\mathcal{H}$ is updated accordingly. This two level heap structure reduces the update time complexity to $O(\log(N) + \log(m))$, where N is the number of users and m is the number of friends per user.

Note that this approach not only saves the computational cost but also expands the applicability to the system. It enables the system to support real-time event handling based on the trigger times of objects, instead of the fixed epoch.

Experiments of computational cost saving techniques. We proceed to examine the computational cost saving offered by the above trigger time technique. All methods were implemented in C++ and the experiments were performed on an Intel Core2Duo 2.66GHz CPU machine with 2 GBytes memory, running on Ubuntu 8.04. We use the default experimental setting as discussed in Section 5.

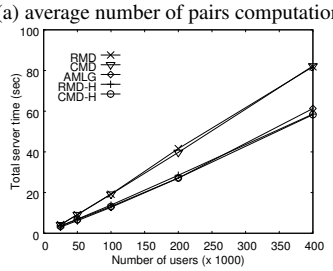
Let RMD-H and CMD-H represent the heap-based models of RMD and CMD respectively. For every timestamp (server's algorithm iteration), RMD, CMD, and AMLG need to process a fixed number of pairs computation, which is equal to the number of pairs in the system. For the heap-based models, the pairs computation is invoked by two cases, (i) the pairs satisfying $\omega(u_i, u_j) = t_{cur}$ are retrieved from the top of the heap $\mathcal{H}$ and then *processed* by Lines 5–17 of Algorithm 2 and (ii) when u_i issues an location update message to the server, its *trigger time* for each friend is recomputed.

For RMD-H and CMD-H (both with the trigger time technique), Figure 11a shows the effect of the number of users on the ratio of the average number of pairs processed and the number of trigger time updates, as a percentage to the total number of pairs, per timestamp. Note that both of them are machine-independent measurements. In the default case, the sum of pairs computation is around 20% which helps reducing the response time significantly as shown in next experiment.

Figure 11b demonstrates the total server CPU time of our basic approaches (RMD/CMD) and heap-based approaches (RMD-H/CMD-H), for the entire workload (with 100 timestamps). Observe that the time of RMD-H/CMD-H ranges from 59% to 82% of the time of RMD/CMD. Due to the overhead of heap maintenance, the improvement in CPU time is not as dramatic as the improvement in the number of comparisons (in Figure 11a). Besides, RMD-H/CMD-H performs almost the same to AMLG (± 3 s in this experiment). In summary, our RMD/CMD methods have better communication cost than AMLG at client side while they are able to achieve similar computational cost to AMLG at server side.

N	RMD-H		CMD-H	
	pair processing ratio	trigger update ratio	pair processing ratio	trigger update ratio
25	8.43%	18.82%	7.55%	20.45%
50	7.80%	15.68%	7.02%	16.44%
100	7.26%	13.63%	6.61%	14.19%
200	7.22%	13.29%	6.59%	13.77%
400	7.18%	13.23%	6.57%	13.70%

(a) average number of pairs computation



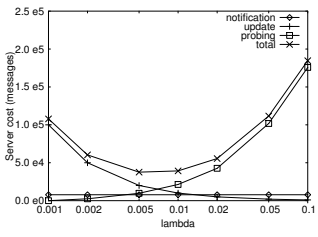
(b) total server time (sec)

Figure 11: Effect of N on the heap-based trigger time technique

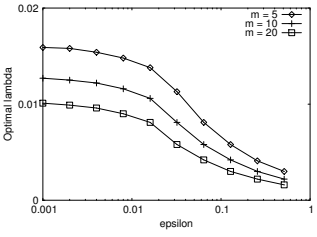
E. ADDITIONAL ANALYTICAL RESULTS

E.1 Estimation Results of FMD Cost Model

We study a typical scenario where there are $N = 100000$ users, each user having $m = 10$ friends, and the other parameters are fixed to $\Delta T = 1$ and $\Delta V = 0.001$. Recall that the spatial domain is $[0, 1]^2$ in our analysis. Figure 12a shows the decomposition of the communication cost with respect to λ, at a fixed $\epsilon = 0.05$ value. Clearly, the notification cost is independent of λ. The update cost becomes high at small λ values and the probing cost is high at large λ values. Observe that there exists an optimal λ value that minimizes the total communication cost. In the next study, we find the optimal λ that minimizes the total communication cost for each test case. Figure 12b depicts the optimal λ value as a function of ϵ, for three different number of friends $m = 5, 10, 20$. Observe that no single λ value can minimize the communication cost in all cases. The optimal λ value depends on the specific values of ϵ and m being used.



(a) cost decomposition, $\epsilon = 0.05$



(b) optimal λ vs. ϵ

Figure 12: Estimation results, fixing $N = 100000$, $\Delta T = 1$, $\Delta V = 0.001$

E.2 State Transition Analysis of RMD

We apply the Monte Carlo method to traverse the state diagram of Figure 7, in order to estimate the user's communication cost.

We run the simulation for 1 million timestamps, with the other parameters set to their default values: $m = 10$, $\epsilon = 0.05$, $\Delta T = 1$, $\Delta V = 0.001$, $\alpha = 2$. We test with three different values for the initial λ and measure the relative frequencies of each state (i.e., current λ value) in each test. The simulation result is shown in Figure 13. Regardless of the initial λ value used, the three runs have almost identical frequency distribution of the states. This implies that the RMD method is robust with respect to the initial λ setting. It is worth noticing that RMD resides in three of the states for more than 90% of time, and the optimal state (0.01, see Figure 12b) has the highest relative frequency. This suggests that RMD tends to stay at the optimal state for a number of consecutive timestamps, leading to low amortized cost per timestamp.

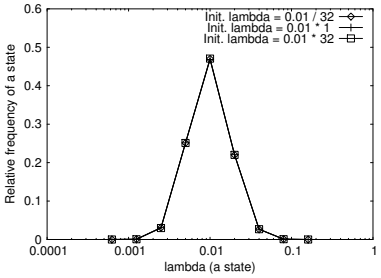


Figure 13: Relative frequency of a state

F. ADDITIONAL EXPERIMENTS

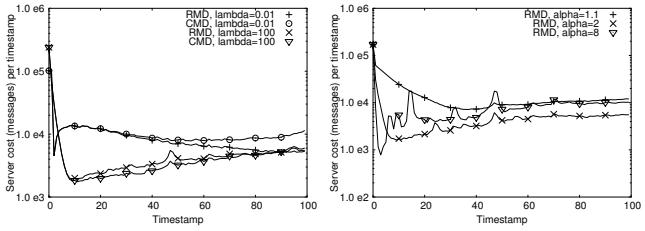
In the following, we present the results of additional experiments, which have not been included into the main experimental section due to space limit.

F.1 Dynamic Behavior Experiments

We then investigate in detail the dynamic behavior of our self-tuning methods (RMD and CMD) over time, for some extreme λ and α values. We measure the communication cost for each method at each timestamp.

Figures 14a shows the cost of the methods at each timestamp, using a small (0.01) and a large (100) initial value of λ. The average mobile region radius is continuously adjusted and the communication cost enters into a steady state as time elapses.

Figures 14b shows the cost of the RMD method, using a small (1.1), a medium (2), and a high (8) value for α. At a low α value, RMD adjusts the mobile regions slowly and it takes 30 timestamps to enter into a steady state. At a high α value, RMD over-adjusts the mobile regions so the cost has oscillations in the first few timestamps. Nevertheless, the oscillations diminish gradually and the cost stabilizes. At a medium α value, RMD reaches the steady state very fast and its cost does not oscillate.



(a) cost along the time, various λ (b) cost along the time, various α

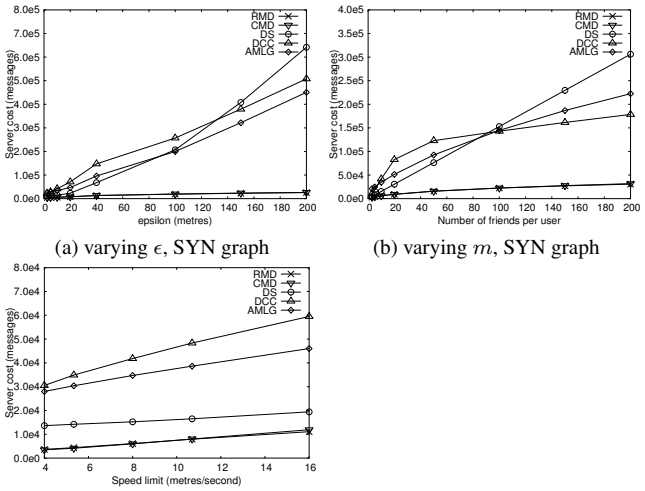
Figure 14: Dynamic behavior experiments, SYN graph

F.2 Scalability Experiments

Figure 15a plots the cost of the methods as a function of the proximity detection distance ϵ. Our methods RMD and CMD have similar cost and they outperform the competitors for all ϵ values. Since AMLG has an adaptive procedure for spliting/merging the partitions of users' locations, it performs better than DCC. However, both AMLG and DCC do not exploit the velocities of the users, so they incur higher update cost than RMD/CMD. At a large ϵ value, DS forces the users to exchange send messages with most of their friends. Obviously, this is much more expensive than client-server methods (RMD, CMD, DCC, AMLG) that require a user to send his location to the server at most once per epoch.

We then study the effect of the number of friends m on the communication cost of the methods, as shown in Figure 15b. Again, at a large m value, DS becomes much more expensive than the client-server methods. DCC and AMLG scale better than DS with respect to m. Observe that the performance of RMD/CMD scales very well with m, and outperforms the competitors by a wide margin at high m values.

Figure 15c plots the cost of the methods with respect to the speed limit. At a low speed limit, the users in DS seldom travel outside their assigned safe regions defined by strips, thus DS has a low cost compared to DCC and AMLG. Note that RMD/CMD has the lowest cost for a wide range of speed limit values.

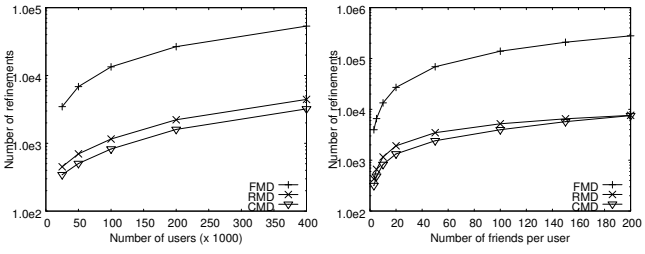


(a) varying ϵ, SYN graph (b) varying m, SYN graph

(c) varying speed limit

Figure 15: Additional experiments

Figure 16a,b show the number of refinements of our methods as a function of the number of users N and the number of friends per user m, respectively. Since RMD and CMD are able to tune the extents of users' mobile regions automatically, they incur much fewer refinements than FMD.



(a) varying N, SYN graph (b) varying m, SYN graph

Figure 16: Effect of the number of users and friends on the number of refinements