Continuous Proximity Monitoring in Road Networks

Hans-Peter Kriegel
Ludwig-Maximilians-University
Munich
Oettingenstr. 67
München, Germany
kriegel@dbs.ifi.lmu.de

Peer Kröger
Ludwig-Maximilians-University
Munich
Oettingenstr. 67
München, Germany
kroegerp@dbs.ifi.lmu.de

Matthias Renz
Ludwig-Maximilians-University
Munich
Oettingenstr. 67
München, Germany
renz@dbs.ifi.lmu.de

ABSTRACT

In this paper, we consider the following scenario: a set of mobile objects continuously track their positions in a road network and are able to communicate with a central server. The server which gets position updates from the moving objects has to detect the event that two objects reach or exceed a specified proximity distance. This way, the server is permanently aware of all pairs of objects that are within a certain distance range. Obviously, the communication costs between the objects and the server quickly become the bottleneck if a position update is sent to the server at each tracking time slot. We propose update strategies in order to reduce the communication overhead by defining special regions for each object. These regions are defined such that no position updates at the server are required as long as the objects do not leave their corresponding regions. We present efficient algorithms for updating these regions and detecting proximity/separation when objects leave their corresponding regions. Furthermore, we empirically evaluate the different strategies in terms of communication overhead, i.e. the number of required position updates.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Search process

General Terms

Algorithms,Performance

Keywords

proximity monitoring, traffic networks

1. INTRODUCTION

With the spreading of modern mobile devices like PDAs or car navigation systems, location-based services (LBS) became more and more important. The detection of proximity among moving objects in traffic networks is a key issue in many LBS applications such as routing applications, traffic jam analysis, and prediction, as well as in traffic mining tasks e.g. traffic pattern analysis and continuous clustering of moving objects in a traffic network. Typically, an object that moves around in a network is able to track its position, e.g. via a GPS transmitter. In order to deploy location-based services provided by systems embedded in stationary networks, it is necessary that the mobile individuals are able to communicate with the network, e.g. via cell phone. This way, the traffic can be continuously monitored at a stationary central server which can communicate with the mobile individuals which we call clients in the remainder. An important information for the above mentioned LBS applications is to know at what time which clients are in proximity and which clients are in separation, i.e. to know for each client the set of all other clients that are within or outside a range of a critical distance ε. This information has to be computed by the central server which globally detects the set of clients that are in proximity, i.e. are within distance ε, and the set of clients that are in separation, i.e. are not within distance ε. The global detection of proximity and separation corresponds to a kind of proximity self-join and separation self-join of all clients moving within the network. The result of these joins is a set of client pairs that are in proximity and a set of client pairs that are in separation. The join results can be used to support continuous density-based clustering as proposed in [4] which in turn provides traffic flow analysis and information routing among the mobile individuals assuming car to car communication is available. If no privacy issues are infringed, the join results can even be sent to the corresponding clients for local analysis or further services.

Since the clients in the network are generally in motion, the results of the proximity join and separation join most likely change steadily over time. A naive approach for keeping the join results up-to-date is to recompute all proximity and separation information after each new position update of the clients at the server from scratch. However, this has two severe disadvantages: First, since not all clients are affected by a change in proximity/separation, a lot of already existing information will be redundantly re-computed. This is obviously rather inefficient. Second, even worse, each client must send a position update at each tracking time slot. This results in a very large communication traffic overhead. In real-world scenarios, where we have more than thousands of clients, the communication costs are the bottleneck and this solution is simply infeasible. While there have been some more sophisticated approaches proposed addressing the position update cost specialized for continuously moving clients in Euclidean space, no solution exists for clients moving in a road network.

In this paper, we present an efficient approach for clients moving in a road network that provides solutions for both afore mentioned problems. In particular, we propose to compute for each client a proximity region and a separation region at the server. The proximity region of a client c defines that part of the traffic network in which c can move around without causing any proximity violation

ACM GIS '08, November 5-7, 2008, Irvine, CA, USA

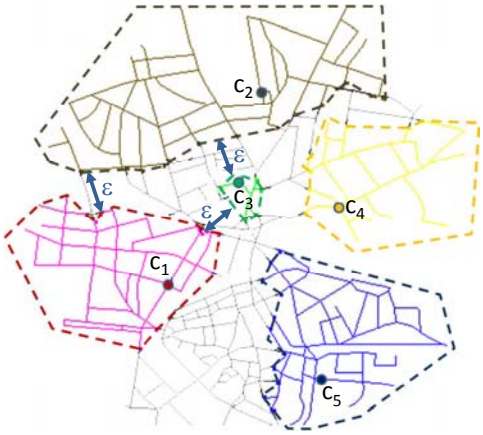


Figure 1: Illustration of proximity regions in a traffic network.

with other clients that are currently in separation with c. The separation region of a client c is defined analogously. The server sends these regions back to the corresponding client. As far as a client does not leave its corresponding regions, it does not need to send a position update due to the conservative definition of the regions, i.e. its movement does not affect the join results. On the other hand, if a client moves across the boundary of one of its regions, it must send a position update to the server. The server then must update the join results and must re-compute the corresponding regions to send it back to the affected clients. We present efficient algorithms for these re-computations that only update the affected regions. As a consequence, our approach provides the following advantages over the naive solution. First, it incrementally updates only those regions that are affected by the position change of a particular client rather than redundantly re-computing the complete proximity and separation information. Thus, also only those parts of the proximity and separation joins are updated that are affected by the position change and a complete re-computation of the entire join results is usually avoided. Second, it minimizes communication costs between clients and server because as long as the clients move within their regions, no position updates need to be sent from the clients to the server. The general idea of using proximity regions is visualized in Figure 1. The region of each client is defined in such a way, that as long as each client stays in its own proximity region, it cannot approach any other client more than the distance threshold ε. Consequently, the minimum distance between the boundaries of two regions is ε.

The rest of the paper is organized as follows. In Section 2, we discuss related work. Section 3 formalizes the problem of proximity and separation detection in traffic networks. Our novel concepts for efficient proximity detection and separation detection are described in Sections 4. Section 5 presents a comprehensive experimental evaluation of the proposed concepts and Section 6 concludes the paper.

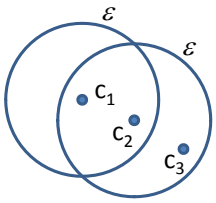
2. RELATED WORK

In recent years diverse approaches for spatial query processing on moving objects have been proposed. Some of them are based on adequate access methods that can cope with highly dynamic data [12, 20, 21, 22, 2]. They assume that the objects update their positions isochronously, if the current position exceeds a given distance from the assumed position [12, 20, 21] or after a fix number of certain motion patterns. Jensen et al. [12] proposed to use an update efficient B^+-tree that organizes time-stamped location vectors. The location vectors were linearized based on their time-stamp and based on a space-filling curve allowing to partition the data according to their time-stamp while preserving spatial proximity. Saltenis et al. proposed in [20, 21] a time-parameterized variant of the R-tree [7] called the TPR-tree. The TPR-tree efficiently organizes objects assuming that the objects move according to a linear function of time. Later Tao et al. presented in [22] the TPR*-tree which improves the original TPR-tree by taking unique features of moving objects into account. Argarwal et al. introduced in [2] three indexing approaches for objects moving along a linear trajectory.

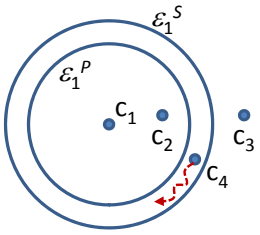
Generally, the approaches for spatial query processing on moving objects can be classified into two classes, snapshot-based methods and methods based on continuous query monitoring. Snapshot based methods [12, 20, 21, 22, 2, 26, 11, 18, 9, 13] aim at efficiently processing queries that are issued at certain points of time. Some of these methods exploit index structures that can cope with moving objects [3, 12, 20, 21]. Our approach falls into the category continuous query monitoring [6, 16, 24, 10, 19, 8, 5, 14, 25, 17]. In contrast to our approach, most existing approaches have focused exclusively on Euclidean space [8, 5, 1, 23, 15]. In our scope are techniques that are based on determining the validity of (previous) queries based on their current locations [26, 8, 5, 1, 23, 15]. In this context, the approaches [23, 15] are very close to our approach. Given a set of client objects moving on a plane in the Euclidean space, the goal is to detect when two objects approach each other within a given proximity. Furthermore, the events when two objects lose their proximity should also be detected. The detection is performed at a central server which can communicate with the clients. The proposed approaches aim at reducing the communication between the mobile clients and the central server. In order to keep the communication cost low, each moving object is assigned to a region forming of a circle. The regions are constructed in such a way, that the mobile clients do not need to send their actual positions to the central server until they leave their regions. This strategy is quite close to that of our approach, except for the fact that we detect the proximities within a spatial network. In a spatial network, approaches based on the Euclidean distance cannot be used. Rather, we need other types of "safe" regions.

3. PROXIMITY AND SEPARATION JOINS IN TRAFFIC NETWORKS

We assume that a set of clients $\mathcal{O}$ move around in a given traffic network and each client tracks its position on a globally fixed set of discrete time slots, e.g. every 10 seconds. A traffic network is a bidirectional graph $\mathcal{G} = (N, E)$ containing a set N of nodes and a set E of edges connecting the nodes. Each node $n \in N$ is assigned to a two-dimensional point $Pos(n)$ in the Euclidean space. Each edge $e \in E$ is assigned to a part $Pos(e)$ of the Euclidean space representing the possible locations on the corresponding road segment. Each client $c \in \mathcal{O}$ is assigned to a position $c.pos \in Pos(\mathcal{G})$, where $Pos(\mathcal{G})$ denotes all positions $Pos(\mathcal{G} = (N, E)) = \{p \in \{Pos(n) \cup Pos(e)\} : n \in N, e \in E\}$ in the Euclidean space. This way, a client c can be assigned to a node $n \in N$ or to an edge $e = (n_1, n_2) \in E$. A weight is assigned to each edge (n_1, n_2) between two nodes $n_1, n_2 \in N$ that reflects the cost that is required to traverse it, i.e. move from n_1 to n_2 or *vice versa*. In addition, we assume that we can also compute a weight representing the costs to move from a position $p \in Pos(e)$ of an edge $e \in E$ to another position $p' \in Pos(e)$ of that edge. This way, if a client is located on an edge $e = (n_1, n_2)$, the distance of



(a) Client c_1 and c_2 are in proximity to each other, i.e. $c_2 \in P(c_1)$ and $c_1 \in P(c_2)$. Client $c_3 \in P(c_2)$ and $c_3 \in S(c_2)$.



(b) Client $c_2 \in P(c_1)$ and $c_3 \in S(c_1)$, while either $c_4 \in P(c_1)$ or $c_4 \in S(c_1)$.

Figure 2: Examples of clients that are in proximity or in separation in the Euclidean space.

the client to n_1 is given in order to compute the exact traversal cost to both adjacent nodes n_1 and n_2. The distance $dist(.,.)$ between two clients in the network is measured by the network distance using these weights, e.g. computed by the Dijkstra algorithm.

The problem of proximity and separation detection is as follows. Given a user-defined distance threshold ε, the server should compute a *proximity join*, i.e. the set

$$\mathcal{O}_1 \bowtie_P \mathcal{O}_2 = \{(c_1, c_2) \in \mathcal{O}_1 \times \mathcal{O}_2 \mid dist(c_1, c_2) < \varepsilon\}$$

of pairs of clients that are in proximity to each other[1]. The set

$$\mathcal{O}_1 \bowtie_S \mathcal{O}_2 = \{(c_1, c_2) \in \mathcal{O}_1 \times \mathcal{O}_2 \mid dist(c_1, c_2) \geq \varepsilon\}$$

represents the *separation join*, i.e. the set of pairs of clients that are in separation to each other.

Both join sets need to be kept up-to-date, i.e. they need to be updated and adjusted when clients approach to or depart from each other.

For convenience, given a client $c_1 \in \mathcal{O}$ we call all clients $c_2 \in \mathcal{O}$ with $dist(c_1, c_2) < \varepsilon$ *in proximity of* c_1. The set of clients that are in proximity of c_1 is defined as $P(c_1) = \{c_2 \in \mathcal{O} \mid dist(c_1, c_2) < \varepsilon\}$. Analogously, the set of clients that are *in separation to* c_1 is defined as $S(c_1) = \{c_2 \in \mathcal{O} \mid dist(c_1, c_2) \geq \varepsilon\}$.

Let us note that proximity and separation is a symmetric relationship, i.e. $c_1 \in P(c_2) \Leftrightarrow c_2 \in P(c_1)$ and $c_1 \in S(c_2) \Leftrightarrow c_2 \in S(c_1)$ as shown in the example illustrated in Figure 2(a). In addition, obviously,

$$\mathcal{O}_1 \bowtie_S \mathcal{O}_2 = (\mathcal{O}_1 \times \mathcal{O}_2) - (\mathcal{O}_1 \bowtie_P \mathcal{O}_2)$$

holds, as well as $c_1 \in P(c_2) \Rightarrow c_1 \notin S(c_2)$ and, thus, $c_1 \in S(c_2) \Rightarrow c_1 \notin P(c_2)$.

Let us note that clients may cause a permanent change between proximity and separation to other clients because they steadily move from proximity to separation and *vice versa* during two tracking time slots. This in turn causes a steady update of the corresponding join sets with a huge communication and computation overhead. To avoid this, in many applications, it can be sensible to introduce a lazy update of proximity and separation detection by using two thresholds ε^P and ε^S for proximity detection and separation detection, respectively, where $\varepsilon^S > \varepsilon^P$. The sets $P(c)$ and $S(c)$ of all clients $c \in \mathcal{O}$ have to be defined accordingly, i.e. $\forall c_2 \in \mathcal{O} : dist(c_1, c_2) < \varepsilon^P \Rightarrow c_2 \in P(c_1)$ and $\forall c_2 \in \mathcal{O} : dist(c_1, c_2) > \varepsilon^S \Rightarrow c_2 \in S(c_1)$. When realizing lazy updates where we assume that $\varepsilon^S > \varepsilon^P$ clients which are between both proximity/separation borders to a client c can either be in proximity of c or in separation of c, i.e. $\forall c_2 \in \mathcal{O} : \varepsilon^P < dist(c_1, c_2) < \varepsilon^S \Rightarrow (c_2 \in P(c_1) \vee c_2 \in S(c_1))$, e.g. client c_4 in the example shown in Figure 2(b). If $\varepsilon^P = \varepsilon^S$, c_4 would permanently change its membership in $P(c_1)$ and $S(c_1)$. Consequently, the join results would also change permanently which can be avoided by the lazy update setup where $\varepsilon^P < \varepsilon^S$. The larger the difference between ε^S and ε^P is, the smaller is the probability of such a permanent change.

In the remainder of this paper we consider that clients moving within the traffic network and continuously tracking their own positions in the network can communicate with a central server. The join computation, i.e. the detection that an client is in proximity/separation to another client, has to be done at the server because we assume that the clients do not have much computational power (e.g. a PDA) while the server does. In order to perform this computation, the server needs some information of the current locations of all clients which has to be transmitted from the clients to the server. The main goal is to transmit only the location information which is absolutely necessary to continuously maintain the correct join result. This is important to minimize the communication traffic between the clients and the server which is usually the bottleneck in such scenarios. Furthermore, the costs required to maintain the proximity/separation sets $P(c)$/$S(c)$ of all clients can be reduced minimizing the location updates at the server side, since the proximity/separation properties have to be checked only if the location of a client is updated. Our proposed solution works in both a normal as well as in a lazy update scenario, i.e. using only one threshold ε or using two separate thresholds ε^P and ε^S to define the proximity and separation join. For clarity reasons, we focus on using only one threshold ε.

4. MONITORING PROXIMITY AND SEPARATION

The key idea of our approach is to store for each client $c \in \mathcal{O}$ a *proximity region* and a *separation region* in which c can move safely without causing any proximity or separation alert, i.e. as long as c does not reach the boundary of one of these regions, the final proximity and separation join results are up-to-date and need no revision. Thus, as long as an client does not reach the boundary of one of its regions, it does not need to send a position update to the server and the server does not need to verify the proximity/separation property of c w.r.t. any other client $\hat{c}$.

The concept of proximity regions is introduced in the following definition.

DEFINITION 1 (PROXIMITY REGION). *The* proximity region *of a client $c \in \mathcal{O}$, denoted by $PR(c)$, is defined as the set of positions in the network graph $\mathcal{G}$ that have a distance larger than ε to any position $\hat{p}$ of proximity regions of clients $\hat{c}$ that are in separation to c, formally*

$$PR(c) = \{p \in \text{Pos}(\mathcal{G}) \mid \forall \hat{c} \in S(c), \hat{p} \in PR(\hat{c}) \ : \ dist(p, \hat{p}) \geq \varepsilon\}$$

Note that if we want a lazy update strategy, we have to replace ε by ε^P (cf. Section 3).

Analogously, the concept of separation regions is defined below.

DEFINITION 2 (SEPARATION REGION). *The* separation region *of a client $c \in \mathcal{O}$ is defined as the set of positions in the network*

[1]Usually, $\mathcal{O}_1 = \mathcal{O}_2 = \mathcal{O}$ holds, i.e. we are interested in a self-join of $\mathcal{O}$

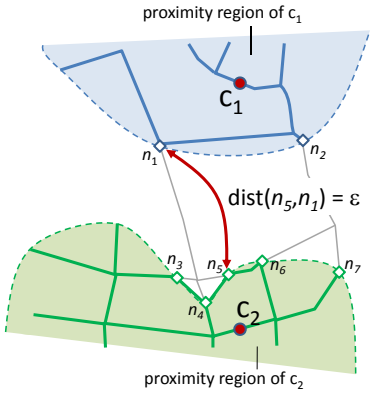


Figure 3: Visualization of the concept of proximity regions.

graph $\mathcal{G}$ that have a distance smaller or equal to ε to any position $\hat{p}$ of separation regions of clients $\hat{c}$ that are in proximity of c, formally

$$SR(c) = \{p \in \text{Pos}(\mathcal{G}) \mid \forall \hat{c} \in P(c), \hat{p} \in SR(\hat{c}) \ : \ dist(p, \hat{p}) \leq \varepsilon\}$$

Again, if we want a lazy update strategy, we have to replace ε by ε^S (cf. Section 3).

Figure 3 illustrates the concept of proximity regions. Here the distances between all possible positions $p \in PRc_1$ of the proximity region of client c_1 to all possible positions $\hat{p} \in PR(c_2)$ of client c_2 are at least ε. For example, all paths between the nodes n_1 and n_5 have at least a length of ε. Consequently, the distance between c_1 and c_2 is greater than ε, as long as they stay in their own regions. This condition can be violated only if at least one of the two clients reaches the border of its own proximity region. In this case, a proximity alert together with a position update is sent to the server where the proximities to the other clients that are affected by the proximity alert are tested (in order to possibly update the final join results) and all corresponding regions have to be re-computed. Analogously, the separation regions are defined such that clients can move safely within their regions without causing any update requirements. A procedure similar to the proximity alert has to be processed if an client approaches the boundary of its separation region, i.e. causing a separation alert. Again, the big advantage of this concept is that clients need only to send their actual position to the central server if their last move caused a proximity or separation alert, rather than sending their current position after each tracking time slot. Obviously, this usually saves a huge amount of communication cost.

Figure 4 presents an overview of the general process of monitoring proximity and separation among clients continuously moving within a traffic network using proximity and separation regions. Each client $c \in \mathcal{O}$ stores its actual proximity and separation regions $PR(c)$ and $SR(c)$. Each client c continuously tests if its current position is still within its proximity and separation region, i.e. whether $c \in PR(c)$ and $c \in SR(c)$. If at least one of these two conditions is violated, a proximity and/or separation alert is sent to the central server together with the current position of client c. At the server side, all candidates C_c that are affected by the alert of client c need to be determined. Depending on the type of alert, these candidates are acquired from $P(c)$ (proximity alert, $C_c \subseteq P(c)$) or from $S(c)$ (separation alert $C_c \subseteq S(c)$). This is necessary in order to avoid a recomputation of all proximity/separation regions. Usually, only the regions of a few clients are affected by the current alert. After having identified the affected candidates, the server

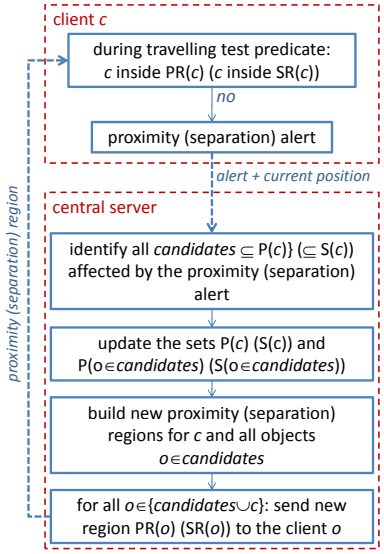


Figure 4: Proximity (separation) detection algorithm.

needs to update the set $P(c)$ and for all $\hat{c} \in C_c$ the sets $P(\hat{c})$ in case of a proximity alert and the set $S(c)$ and for all $\hat{c} \in C_c$ the sets $S(\hat{c})$ in case of a separation alert. For this purpose, the server has to poll the position of all candidates in C_c to get their actual positions. These updated sets are used in two ways. First, the updated join results can be computed from the updated proximity/separation sets at the server. Second, the server can also compute new proximity and/or separation regions for c and all candidates C_c. Finally, these new proximity/separation regions are sent to the corresponding clients.

In the following, we detail the most important steps of this general process. In particular, we discuss how proximity and separation regions are initialized. We will then show how these regions can be stored efficiently on the client and on the server side. Last but not least, we present efficient algorithms for processing a proximity alert, i.e. testing for proximity and updating all affected regions.

4.1 Initializing and Updating Proximity and Separation Regions

4.1.1 *Proximity Region*

The basic idea of a proximity region $PR(c)$ is to identify a section of the network in which a given client c can move around safely without causing a tentative proximity alert. Definition 1 provides only a description of properties $PR(c)$ must meet rather than a constructive blueprint for building such a region. In addition, the proximity region of any client $c \in \mathcal{O}$ obviously depends on the proximity regions of (possible all) clients that are in separation to c. Therefore, we can construct many different sets of positions that qualify as proximity regions according to Definition 1. If we assume that all clients move with the same characteristics such as velocity, we are interested in those regions that equally maximize the regions of all clients simultaneously with the additional constraint that each client has a maximum distance to its region boundaries. In this scenario, the optimal proximity region of a client c is a Voronoi cell around c w.r.t. all clients $\hat{c} \in S(c)$ with boundary width ε. If the characteristics of movement are different for some clients, the

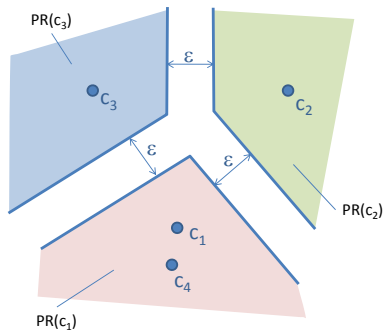


Figure 5: Initial proximity regions of clients c_1, c_2 and c_3. Here, $S(c_1) = \{c_2, c_3\}$, whereas $c_4 \in P(c_1)$.

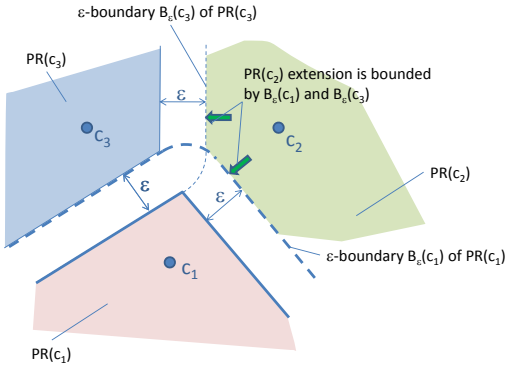


Figure 6: Proximity region of client c_1 with the corresponding ε-boundary.

Voronoi-like regions have to be adapted likewise. However, for clarity reasons, we focus on the simple case that all clients move with the same characteristics.

Due to these considerations, we initially compute Voronoi cells around a client c w.r.t. all clients $\hat{c} \in S(c)$ and a boundary width of ε. In the example illustrated in Figure 5 the initial proximity regions are generated according to all three clients c_1, c_2 and c_3 which are *in separation of* each other, i.e. $S(c_1) = \{c_2, c_3\}$ and $S(c_2) = \{c_1, c_3\}$. Here, the proximity region of client c_4 (not depicted in the example) does not affect the proximity region of c_1, as c_4 is in $P(c_1)$ but not in $S(c_1)$. For this purpose, we can launch a parallel Dijkstra expansion from each client $c \in \mathcal{O}$. The Dijkstra expansion for client c is bounded by graph nodes n that have already been reached by the expansion of a client $\hat{c} \in S(c)$. From the distance of c to n and $\hat{c}$ to n, the exact position of the center between c and $\hat{c}$ on the path through n which is mostly located on an edge can be determined. From this position, we can search backwards in the direction of c with a distance of $\varepsilon/2$ to determine the border to $PR(c)$ and also in the direction of $\hat{c}$ in order to determine the border of $PR(\hat{c})$. The backward search is simply done by backtracking those paths that have been found during the previous Dijkstra expansion.

The Voronoi cell based computation of the proximity regions is only appropriate for those clients for which we need to build a new proximity region. This includes the clients that are currently not assigned to a proximity region or which need an update of their proximity region. However, when constructing a proximity region of a client $c \in \mathcal{O}$, we must also take into account those clients $\hat{c} \in S(c)$ that are not affected by the proximity alert of c and, thus, do not require an update of their proximity region. In fact, we need a second concept called ε-boundary of the proximity region to manage the update of a proximity region efficiently.

DEFINITION 3 (ε-BOUNDARY). *The* ε-boundary *of the proximity region $PR(c)$ of a client $c \in \mathcal{O}$, denoted by $B_\varepsilon(c)$, includes all positions in the network graph $\mathcal{G}$ that are in ε distance from the proximity region $PR(c)$, formally*

$$B_\varepsilon(c) = \{p \in \text{Pos}(\mathcal{G}) \mid dist(p, PR(o)) = \varepsilon\},$$

where the distance of a position $p \in \text{Pos}(\mathcal{G})$ and a region $R \subseteq \text{Pos}(\mathcal{G})$ is the minimum distance of p to all positions $r \in R$, i.e.

$$dist(p, R) = \min_{r \in R} dist(p, r).$$

Figure 6 illustrates the ε-boundary of the proximity region of client c_1. The dotted positions define the ε-boundary corresponding to those positions in the network graph that have a distance of ε to $PR(c_1)$.

The ε-boundary of a proximity region can be used to update other proximity regions efficiently because obviously, the proximity regions are bounded by the ε-boundaries of neighboring clients. For example, in Figure 6, the proximity region of client c_2 is bounded by the ε-boundaries of client c_1 and client c_3.

4.1.2 Separation Region

In a similar way we construct the proximity regions, we have to generate the separation region $SR(c)$ assigned to each client $c \in \mathcal{O}$. The separation region $SR(c)$ of an client c identifies the part of the road network that can be traversed by c without causing a tentative separation alert, i.e. c is no longer guaranteed to be in proximity with another client $\hat{c} \in P(c)$. Consequently, the separation region of a client $c \in \mathcal{O}$ depends on the separation regions of all clients $\hat{c} \in P(c)$ that are in proximity to c (cf. Definition 2). Similar to the formation of the proximity regions, for the construction of the separation region we aim at maximizing the distance between the client and the region border.

A separation region of a client c is composed by the joint overlap of auxiliary regions called ε-regions guaranteeing that the clients travelling within them are definitely in proximity. The ε-regions are defined for pairs of clients as follows:

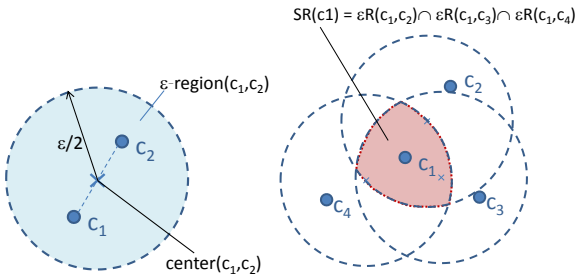
DEFINITION 4 (ε-REGION). *The* ε-region *defined for a pair of clients $(c, \hat{c}) \in \mathcal{O} \times \mathcal{O}$, denoted by $\varepsilon R(c, \hat{c})$, is a set of positions in the network graph $\mathcal{G}$ that includes the positions of c and $\hat{c}$ and guarantees that arbitrary two positions in $\varepsilon R(c, \hat{c})$ have a distance of at most ε to each other, formally*

$$\varepsilon R(c, \hat{c}) = \{p \in \text{Pos}(\mathcal{G}) \mid dist(p, center(c, \hat{c})) \leq \frac{\varepsilon}{2}\},$$

where $center(c, \hat{c})$ denotes the center point located on the shortest path between c and $\hat{c}$, i.e.

$$center(c, \hat{c}) = \min\{p \in \text{Pos}(\mathcal{G}) \mid dist(p, c) = dist(p, \hat{c})\}$$

For example, the separation region of client $c_1 \in \mathcal{O}$ which is in proximity with three other clients c_2, c_3 and c_4, i.e. $S(c_1) = \{c_2, c_3, c_4\}$, is depicted in Figure 7. Figure 7(a) shows the ε-region $\varepsilon R(c_1, c_2)$ of the client pair c_1 and c_2 which is centered at the center point between c_1 and c_2 and covers the part of the network which is within the $\frac{\varepsilon}{2}$ range around the center point. Figure 7(b) shows the separation region of c_1 which is specified by the part of the network that is covered by all three ε-regions $\varepsilon R(c_1, c_2)$, $\varepsilon R(c_1, c_3)$ and $\varepsilon R(c_1, c_4)$.



(a) ε-region for the client pair c_1 and c_2

(b) Separation region of client c_1 built by the joint overlap of all ε-regions $\{\varepsilon R(c_1, \hat{c})/; \mid \hat{c} \in P(c_1)\}$

Figure 7: Separation region of a client (in Euclidean space)

For the initial generation of the separation region of a client $c \in \mathcal{O}$, first the center point $center(c, \hat{c}) \in Pos(\mathcal{G})$ is computed according to each client $\hat{c} \in P(c)$. Then, the corresponding ε-regions $\{\varepsilon R(c_1, \hat{c}) \mid \hat{c} \in P(c)\}$ are generated and finally intersected to generate the separation region of c. Note, that an *eps*-region has to be constructed for a pair of clients only, if there does not already exist one which is still valid, i.e. fulfills the criterion specified in Definition 4. The separation region $SR(c)$ has to be updated as soon as there is an update of one of the *eps*-regions assigned to it, i.e. if an existing *eps*-regions is removed from or a new one is assigned to $SR(c)$.

4.2 Managing Proximity and Separation Regions

In the following, we describe the data structures used in order to efficiently manage and access the information about the proximity region and the separation region of each client $c \in \mathcal{O}$. Since client and server must have different views on that information, we distinguish between client-side and server-side data structures.

Client-side. Each client $c \in \mathcal{O}$ needs to know only the regions $PR(c)$ and $SR(c)$ in which it can safely move without the need to initiate a proximity/separation alert. Other information is not relevant for the client and may conflict with privacy issues, e.g. no client should have any hints on the exact position of other clients[2]. In fact, only the boundaries of the corresponding regions would suffice to detect when c leaves a region. As we aim at keeping the client-server communication traffic low and the clients obtain their region information from the server, only region boundaries, i.e. a list of edges and the exact region boundary positions are transmitted from the server to the clients. Consequently, for each client c we store

- the graph $\mathcal{G} = (N, E)$ representing the road network,
- a list of all positions $\{p \in PR(c) \mid \forall p' \in Pos(\mathcal{G}), p' \notin PR(c), dist(p, p') = dist(p' = PR(c))\}$, denoted by `c.PR`,
- a list of all positions $\{p \in SR(c) \mid \forall p' \in Pos(\mathcal{G}), p' \notin SR(c), dist(p, p') = dist(p' = SR(c))\}$, denoted by `c.SR`.

[2]Note that with the information on its boundaries an client may infer knowledge about the position of its neighbors that are not in proximity but it cannot identify those clients.

For efficient retrieval, these lists can be indexed by a simple B-tree or using a hashmap. Let us note that the road-network graph does not necessarily need to be stored at client-side, since the regions do not need to be specified by means of network components. For example, the region borders can be in the form of GPS coordinates. Obviously, the information which resides on the client side should be not very large in size. This is important since we can assume that each client has only limited resources for computation and storage.

Server-side. The server must store the following information. First of all, it has to store the graph $\mathcal{G} = (N, E)$ representing the traffic network. Secondly, for the proximity detection the server must manage the proximity regions and the corresponding ε-boundaries of all clients in $\mathcal{O}$. We can combine both pieces of information as follows.

For each edge $e = (n_1, n_2) \in E$ of the network, we store

- a list of all clients $c \in \mathcal{O}$ for which this edge intersects their proximity region, i.e. $\{Pos(e) \cap PR(c)\} \neq \varnothing$, denoted by `e.PRL`.
- a list of all clients $c \in \mathcal{O}$ for which their ε-boundary $B_\varepsilon(c)$ is located on this edge, i.e. $\{Pos(e) \cap B_\varepsilon(c)\} \neq \varnothing$, denoted by `e.EBL`.

The entries of `e.PRL` and `e.EBL` are associated with an exact position $p \in Pos(e)$ if necessary, e.g. if the edge partially intersects a region of client c the exact position of the region bound must be stored with c in `e.PRL`. Obviously, this also holds for the list `e.EBL`. Again, to ensure efficient retrieval, these lists can be indexed using a B-tree or a hashmap.

Analogously to the data structures necessary to store the various proximity regions, we need several lists to store the separation regions. For each edge $e \in E$ we need to store

- a list of all clients $c \in \mathcal{O}$ for which this edge intersects their separation region, i.e. $\{Pos(e) \cap SR(c)\} \neq \varnothing$, denoted by `e.SRL`.
- a list of all pairs of clients $(c, \hat{c}) \in \mathcal{O} \times P(c)$ for which this edge intersects their separation region, i.e. $\{Pos(e) \cap \varepsilon R(c, \hat{c})\} \neq \varnothing$, denoted by `e.EpsRL`.

Again the entries of `e.SRL` and `e.EpsRL` are associated with an exact position $p \in Pos(e)$ if necessary. Furthermore, the lists required to manage the separation regions can also be indexed using a B-tree or a hashmap for efficiency reasons.

Note that if $\{Pos(e) \cap SR(c)\} \neq \varnothing$ then there must be a client $\hat{c}$ such that $\{Pos(e) \cap \varepsilon R(c, \hat{c})\} \neq \varnothing$. On the other hand, if $\{Pos(e) \cap \varepsilon R(c, \hat{c})\} \neq \varnothing$ it may be that $\{Pos(e) \cap SR(c)\} = \varnothing$ because $SR(c)$ is the intersection of all ε-regions associated with c (see above).

In order to enable that updates of proximity regions and separation regions can be made in an efficient way, at server-side we additionally have to store for each client $c \in \mathcal{O}$ a list of edges or nodes that intersect the proximity region and separation region of c, denoted by `c.PR` and `c.SR`. Furthermore, we have to store for each client $c \in \mathcal{O}$ and each client $\hat{c} \in P(c)$ which is in proximity with c a list of edges or nodes that intersect the ε-region $\varepsilon R(c, \hat{c})$, denoted by $c(\hat{c}).\varepsilon R$. Once, a new region has to be generated or an existing region of a client $c \in \mathcal{O}$ has to be extended, the affected edges are inserted into the corresponding lists. Again, the clients are inserted in the lists `e.PRL`, `e.EBL`, `e.SRL` and `e.EpsRL`. If a region of a client $c \in \mathcal{O}$ has to be reduced or has to be deleted, we have to determine the set of affected edges $E' \subseteq E$ from the lists `c.PR`, `c.SR` or $c.\hat{c}.\varepsilon R$ and have to remove c from the lists

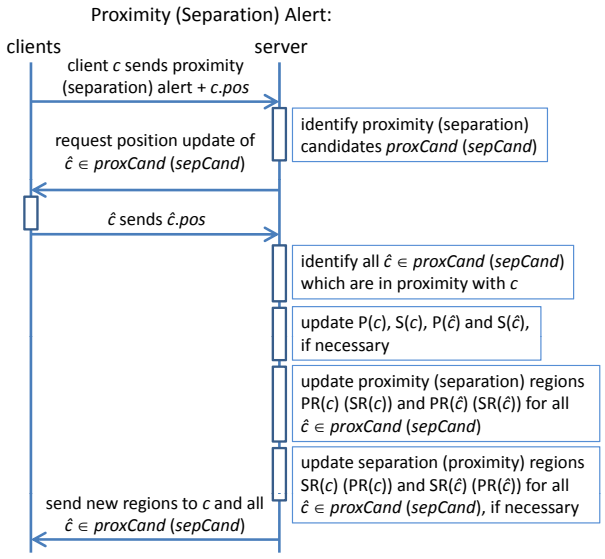


Figure 8: Message sequence of a proximity (separation) alert.

stored for each $e \in E'$. Finally, the lists associated with c have to be updated.

4.3 Proximity And Separation Alert Processing

If a client c crosses the border of $PR(c)$ or $SR(c)$, c has to send an alert to the server including its current position, because now c may have moved in proximity or separation to one or several other clients. In the case c is leaving $PR(c)$ a proximity alert has to be issued and if c is leaving $SR(c)$ it has to issue a separation alert.

Proximity Alert. After the server has received a proximity alert from a client $c \in \mathcal{O}$, the server must check if there is any other client $\hat{c} \in \mathcal{O}$ such that c and $\hat{c}$ have not been in proximity so far but possibly are in proximity. Note that this check is important in order to avoid that the server-side join of the clients produces incorrect results (cf. Section 3). In addition, the server has to update the proximity region of c and that of the other affected clients. The procedure and communication between the clients and the server associated with a proximity alert is illustrated in the message sequence chart depicted in Figure 8.

First, the server has to identify all clients that are affected by the proximity alert issued by c, i.e. those clients $\hat{c} \in S(c)$ for which client c moved beyond their ε-boundary $B_\varepsilon(\hat{c})$. We call this set of clients *proximity candidates* `proxCand` which can be efficiently identified by the server by searching the corresponding ε-boundaries stored in the lists $e.EBL$ assigned to the edges e of the network graph. Subsequently, the server has to request a position update to all proximity candidates $\hat{c} \in$ `proxCand`. After the server has received the actual positions from the clients $\hat{c} \in$ `proxCand`, it pair-wise computes the distances between c and all $\hat{c} \in$ `proxCand` to test if c and $\hat{c}$ are now in proximity, i.e. if $dist(c, \hat{c}) < \varepsilon$. If so, $\hat{c}$ is now in proximity with c, and thus, we have to update the proximity/separation sets by adding c to $P(\hat{c})$ and adding $\hat{c}$ to $P(c)$. Furthermore, c and $\hat{c}$ have to be removed from $S(\hat{c})$ and $S(c)$, respectively. Next, the proximity regions and potentially some separation regions of c and all other proximity candidates have to be updated.

For those clients $\hat{c} \in$ `proxCand` fulfilling the proximity criterion, i.e. $dist(c, \hat{c}) < \varepsilon$, we do not need to update their proximity regions. The rationale for this is, although the recent proximity region of $\hat{c}$ does not need to be bounded by the proximity region of c any longer, and thus, could be enlarged, it still conservatively guarantees that client $\hat{c}$ is separated from all other clients $c' \in S(\hat{c})$. Since we want to reduce the communication costs, we do not update the region of these clients because otherwise, we would be required to send this new information from the server to all those clients. However, we have to update their separation regions including the region of client c. Initially, we remove the corresponding entries from the affected edge lists $e.SRL$ associated with the old separation regions. Then, the ε-regions $\varepsilon R(c, \hat{c})$ are built pair-wise. Afterwards, the separations regions of client c and the other clients $\hat{c}$ are rebuilt by means of the new sets of ε-regions, as described in Section 4.1.2. Finally, we insert the new separation regions into the corresponding edge lists $e.SRL$ and transmit the new separation regions back to the clients.

Up to now, we have considered the update processing associated with the proximity candidates $\hat{c} \in$ `proxCand` fulfilling the proximity criterion. In the following, we treat the updates associated with the other proximity candidates $\hat{c} \in$ `proxCand` which are still not in proximity of c. For these clients we do not need to update their separation regions, since their proximity sets $P(\hat{c})$ and separation sets $S(\hat{c})$ have not modified. However, these clients including client c require new proximity regions. Analogous to the separation region update, first, the old proximity regions are removed from the corresponding edge lists $e.PRL$. Then, the proximity regions are built for c and all clients $\hat{c} \in$ `proxCand` through the parallel Voronoi cell based computation using Dijkstra as described in Section 4.1.1. Finally, the edge lists $e.PRL$ of the network graph are updated and the new proximity regions are transmitted back to the corresponding clients.

Separation Alert. Analogous to the update process induced by a proximity alert we have to process the updates if a client issues a separation alert to the server. Again, this process guarantees correct join results at server-side (cf. Section 3). Generally, the separation alert processing, simplified illustrated in Figure 8, is quite similar to the proximity alert processing. The main difference is that the set `sepCand` of clients $\hat{c} \in P(c)$ associated with the ε-regions $\varepsilon R(c, \hat{c})$ client c has left are affected. After the server has requested and received the exact positions from these clients, the server identifies those clients $\hat{c} \in P(c)$ which are now in separation, i.e. $dist(c, \hat{c}) > \varepsilon$. Consequently, the corresponding proximity and separation sets have to be updated, analogous to the proximity alert process. In this case, the separation regions of client c and all clients $\hat{c} \in$ `sepCand` have to be updated, definitely. While, the proximity regions have to be updated only for those clients $\hat{c} \in$ `sepCand` which are not in proximity to c any longer. The proximity region of c has to be updated if the latter case is true for at least one client $\hat{c}$. For an efficient region update we access the edge lists as described above for the proximity alert processing. Finally, the new regions (proximity regions and separation regions) are transmitted back to the corresponding clients.

5. EXPERIMENTAL EVALUATION

5.1 Test Bed

For the experimental evaluation of our approach we used a traffic simulator that is able to simulate multiple cars moving on a given road network. Initially, the cars are assigned to a starting point and a target. Both the starting and the target points are randomly dis-

(a) "OL" (b) "SG"

Figure 9: Traffic network graphs used as experimental test bed.

tributed over the network. After the cars are placed at their starting points they simultaneously move on the shortest path to their targets. As soon as a car reaches its destination, it is removed from the road network and a new car with a new starting position and a new destination is generated. The velocities of the cars are set to a realistic value.

For our experiments we used two different road network graphs of different size. The first network graph (OL) which is depicted in Figure 9(a) corresponds to the road network of the city of Oldenburg, Germany, which is a well-known benchmark road network frequently used by diverse query processing approaches. It contains about 7,036 road segments and 6,105 intersection nodes. The second network graph (SG) is depicted in Figure 9(b). It contains about 679 road segments and 533 intersection nodes.

We performed our experiments by running the simulation with the corresponding parameter settings and counted the following events during the simulation run that indicate the total communication cost.

- ProxUp corresponds to the number of position updates due to proximity alerts (client → server).
- SepUp equals to the number of position updates due to separation alerts (client → server).
- Polls represents the number of position polls inquired from the server, including the request (client ← server) and the answer of the client (client → server).
- MsgCnt corresponds to the total number of required messages between the clients and the server.

We measured the above events w.r.t. the number of motion steps per car which indicates the message traffic required when using the naive brute-force method where for each car the position updates are simply permanently sent to the server.

We compare our approach, in the following denoted by NMA, with the Dynamic Centered Circle approach (denoted by DCC in the following) proposed in [23, 15] which is the state-of-the-art approach designed for the Euclidean space. In order to adapt the DCC approach to be able to detect proximities and separations in road networks correctly, we simply need to use the network distance instead of the Euclidean distance to define the corresponding proximity and separation regions.

In all our experiments we set $\varepsilon^P = \varepsilon^S = \varepsilon$, since this is the most relevant environment in real-world applications and it is also the most challenging one as discussed at the end of Section 3.

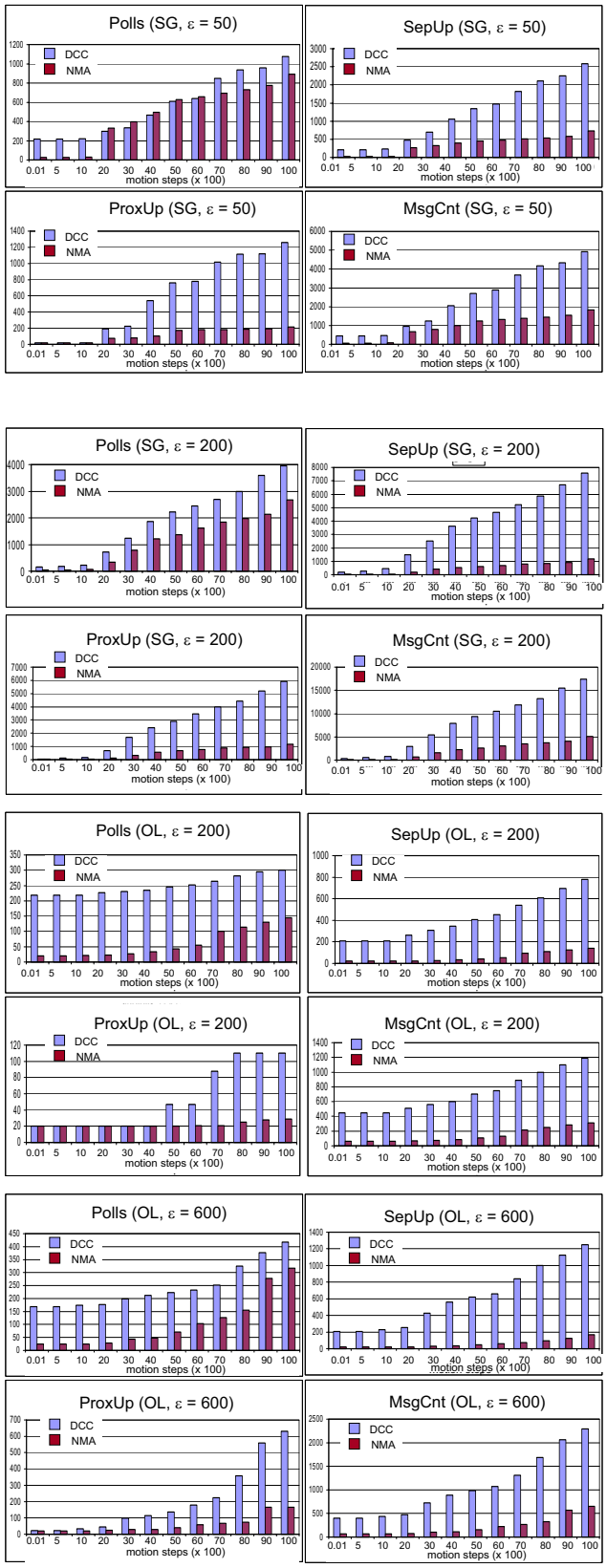
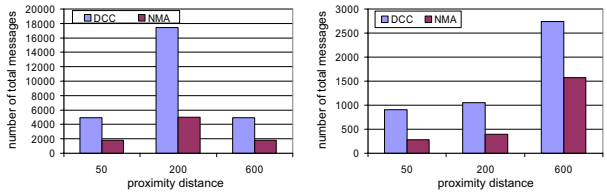
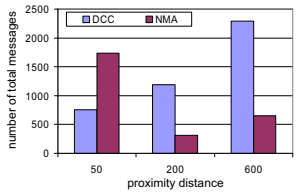


Figure 10: Performance of NMA and DCC on "SG" and "OL" with 20 clients and different values for ε.

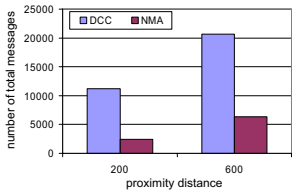


(a) SG, #cars = 20

(b) SG, #cars = 50



(c) OL, #cars = 20



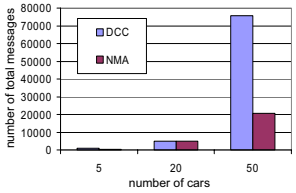
(d) OL, #cars = 50

Figure 11: Performance of NMA and DCC on "SG" and "OL" w.r.t. ε.

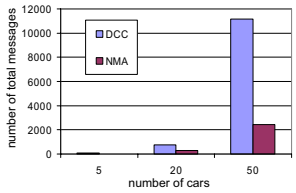
5.2 Performance Evaluation

In a first experiment we evaluated the impact of the proximity/seperation distance threshold ε on the parameters measured to evaluate the communication costs. For that purpose, the number of clients currently traveling around in the network was fixed to 20. We used a range of considerably larger values for ε for "OL" compared to "SG" in order to account for the different sizes of the network graphs. The results on the smaller "SG" network are depicted in Figure 10 (first eight charts) featuring a setting of $\varepsilon = 50$ and $\varepsilon = 200$, respectively. The figures display the evolution of the costs during simulation time. The total simulation lasts 10,000 tracking time slots. As it can be seen, for all measured parameters, our NMA approach outperforms the DCC approach significantly. A similar observation can be made on the larger "OL" graph (cf. Figure 10, last eight charts), featuring a setting of $\varepsilon = 200$ and $\varepsilon = 600$, respectively. Obviously, in all experiments, using NMA instead of DCC yields a considerable reduction of the communication costs. Both approaches scale linearly w.r.t. the proceeding of simulation time, but NMA with a clearly smaller slope. It is also interesting that the DCC approach performs considerably well at the beginning in terms of proximity updates. On both networks, the DCC approach seems to produce rather good proximity regions at first: after a short period of simulation, only few clients left their proximity regions so that these regions need to be updated. However, with proceeding simulation time, even the costs for the proximity updates clearly increase over the NMA approach. This suggests that the quality of the proximity regions of the DCC approach is steadily decreasing after each update. As a consequence, more clients reach their region boundaries after a shorter time causing in turn a higher number of region updates. We make a similar observation when using the NMA approach, but the increase of the cost is much lower compared to the DCC approach.

The total communication costs after 10,000 tracking time slots in terms of the number of total messages on the "SG" graph using 20 and 50 cars in motion w.r.t. varying settings of ε are shown in Figures 11(a) and 11(b), respectively. The results of the same experiment on the "OL" graph is shown in Figures 11(c) and 11(d) with 20 and 50 cars in motion, respectively. Again, the results confirm the observations made before that our approach outperforms the DCC algorithm and scales better to different values of ε.



(a) "SG", $\varepsilon = 200$



(b) "OL", $\varepsilon = 200$

Figure 12: Performance of NMA and DCC on both data sets w.r.t. the number of clients.

A second range of experiments evaluates the performance of the competitors w.r.t. the number of clients moving around in the network. For that experiments, we fixed the value of ε to a given value. Again, we only report the number of total messages after the entire simulation with 10,000 time steps. In Figure 12(a) the results are shown for the "SG" graph using $\varepsilon = 200$. As it can be observed, when varying the number of cars from 5 to 20 and to 50, the more cars are around in the network, the more our NMA approach outperforms the DCC approach. In case of 50 moving clients, the superiority of NMA over DCC is the highest. Again similar observations can be made on the "OL" data set when varying the number of cars from 5 to 20 and to 50. Figure 12(b) depicts the performance results of the competitors for $\varepsilon = 200$. The results confirm that the more clients are in motion in the network, the higher is the performance gain of NMA over DCC in terms of saved communication costs.

In summary, we can conclude that our NMA approach outperforms the DCC approach significantly in terms of client-server communication costs. The main reason for this result is that our NMA approach generates much more accurate proximity and separation regions than the DCC approach.

6. CONCLUSIONS

In this paper, we have proposed a method for efficiently monitoring proximity and separation joins among objects (clients) moving around in road networks in a client/server scenario. Usually, the computationally complex join determination is done at the server side, thus, the bottleneck of monitoring proximity and separation joins is usually the communication cost caused by position polls and position updates between the server and the clients. To reduce the communication costs, we propose a solution based on identifying regions for each client in which the corresponding client can move around without causing a proximity/separation alert, i.e. a possible change at the overall join results. We present efficient algorithms for the case when an client exceeds its proximity or separation region to test for changes at the join results and update the affected regions. Our experiments show a considerable superiority of our solution over a state-of-the-art approaches that was originally proposed for Euclidean spaces but can be adapted to road networks rather straightforward.

To extend this work in the future, we see the following points. First, it is very intersecting, yet challenging, to consider knowledge about the movement of the clients (direction, velocity, etc.) when building and updating the regions of each client. Integrating this knowledge may lead to optimized regions and, thus, to less prox-

imity and/or separation alerts. Second, the idea of lazy updates using different ε-thresholds for proximity and separation has been sketched. A more thorough evaluation of these ideas (which could not be done here due to space limitations) could bear new insights like "in which situations are lazy updates better than normal updates?" Such an information could be used to develop e.g. a clever combination of lazy and normal update strategies that adapts flexibly to the actual setup.

7. REFERENCES

[1] A. A. Afrat, J. Myllymaki, L. Palaniappan, and K. Wampler. "Buddy Tacking - Efficient Proximity Detection among Mobile Friends". In *Proceedings of IEEE Infocom 2004, Hongkong*, 2004.

[2] P. K. Argarwal, L. Arge, and J. Erickson. "Indexing moving points". In *Proceedings of the Conference on Principles of Database Systems (PODS), Chicago, IL*, pages 175–186, 2000.

[3] R. Benetis, C. C. Jensen, G. Karciauskas, and S. Saltenis. "Nearest neighbor and reverse nearest neighbor queries for moving objects". In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, pages 44–53, 2002.

[4] C. Böhm, B. Braunmüller, M. M. Breunig, and H.-P. Kriegel. "Fast Clustering Based on High-Dimensional Similarity Joins". In *Int. Conf. on Information Knowledge Management (CIKM)*, 2000.

[5] Y. Cai, K. A. Hua, and G. Cao. "Processing range-monitoring queries on heterogeneous mobile objects". In *Proceedings of the IEEE International Conference on Mobile Data Management (MDM)*, pages 27–38, 2004.

[6] B. Gedik and L. Liu. "MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System". In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT), Crete, Greece*, pages 67–87, 2004.

[7] A. Guttmann. "R-trees: A dynamic index structure for spatial searching". In *Proceedings of the SIGMOD Conference, Boston, MA*, 1984.

[8] H. Hu, J. Xu, and D. Lee. "A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects". In *Proceedings of the SIGMOD Conference, Paris, France*, pages 479–490, 2005.

[9] X. Huang, C. S. Jensen, and S. Saltenis. "Multiple k Nearest Neighbor Query Processing in Spatial Network Databases". In *Proceedings of the International Conference on Advances in Databases and Information Systems (ADBIS)*, pages 266–281, 2006.

[10] G. S. Iwerks, H. Samet, and K. Smith. "Continuous k-nearest neighbor queries for continuously moving points with updates". In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), Berlin, Germany*, pages 512–523, 2003.

[11] C. S. Jensen, J. Kolar, T. B. Pedersen, and I. Timko. "Nearest neighbor queries in road networks". In *Proceedings of the ACM International Symposium on Geographic Information Systems (ACMGIS)*, pages 1–8, 2003.

[12] C. S. Jensen, D. Lin, and B. C. Ooi. "Query and Update Efficient B+-Tree Based Indexing of Moving Objects". In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB), Toronto, Canada*, pages 768–779, 2004.

[13] M. R. Kolahdouzan and C. Shahabi. "Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases". In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB), Toronto, Canada*, pages 840–851, 2004.

[14] N. Koudas, B. C. Ooi, K.-L. Tan, and R. Zhang. "Approximate NN queries on streams with guaranteed error/performance bounds". In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB), Toronto, Canada*, pages 804–815, 2004.

[15] A. Küpper and G. Treu. "Efficient Proximity and Separation Detection among Mobile Targets for Supporting Location-based Community Services". In *ACM SIGMOBILE Mobile Computing and Communications Review, 10(3)*, 2006.

[16] M. F. Mokbel, X. Xiong, and W. G. Aref. "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases". In *Proceedings of the SIGMOD Conference, Paris, France*, pages 623–634, 2004.

[17] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. "Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring". In *Proceedings of the SIGMOD Conference, Baltimore, ML*, pages 634–645, 2005.

[18] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. "Query processing in spatial network databases". In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), Berlin, Germany*, pages 802–813, 2003.

[19] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. "Query indexing and velocity constrained indexing: Scalable techniques for conitnuous queries on moving objects". *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[20] S. Saltenis and C. S. Jensen. "Indexing of Moving Objects for Location-Based Services". In *Proceedings of the 18th International Conference on Data Engineering (ICDE), San Jose, CA*, pages 463–472, 2002.

[21] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. "Indexing the Positions of Continuously Moving Objects". In *Proceedings of the SIGMOD Conference, Dallas, TX*, pages 331–342, 2000.

[22] Y. Tao, D. Papadias, and J. Sun. "The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries". In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), Berlin, Germany*, pages 790–801, 2003.

[23] G. Treu and A. Küpper. "Efficient Proximity Detection for Location Based Services". In *In Proceedings of the Joint 2nd Workshop on Positioning, Navigation and Communication 2005 (WPNC05) and 1st Ultra-Wideband Expert Talk (UET05), Hannover, Germany*, 2005.

[24] X. Xiong, M. F. Mokbel, and W. G. Aref. "SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases". In *Proceedings of the 21st International Conference on Data Engineering (ICDE), Tokyo, Japan*, pages 643–654, 2005.

[25] X. Yu, K. Q. Pu, and N. Koudas. "Monitoring k-Nearest Neighbor Queries Over Moving Objects". In *Proceedings of the 21st International Conference on Data Engineering (ICDE), Tokyo, Japan*, pages 631–642, 2005.

[26] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. "Location-based spatial queries". In *Proceedings of the SIGMOD Conference, San Diego, CA*, pages 443–454, 2003.