Providing Location-privacy in Opportunity Mobile Social Network

by

Rui Huang

A Thesis submitted to the Faculty of Graduate and Postdoctoral Studies

in partial fulfillment of the requirements for the degree of

**MASTER OF APPLIED SCIENCE**

in Electrical and Computer Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering

University of Ottawa

Ottawa, Canada

September 3, 2017

© Rui Huang, Ottawa, Canada, 2017

Abstract

Todo

Acknowledgements

Todo

Table of Contents

[Abstract i](#_Toc498870348)

[Acknowledgements ii](#_Toc498870349)

[Table of Contents iii](#_Toc498870350)

[List of Figures v](#_Toc498870351)

[List of Tables vi](#_Toc498870352)

[Chapter 1 Introduction 1](#_Toc498870353)

[Chapter 2 Background and Related Work 3](#_Toc498870354)

[Chapter 3 Multi-Hop Location-Privacy Protection 4](#_Toc498870355)

[3.1. System Model 4](#_Toc498870356)

[3.2. Details of MHLPP 5](#_Toc498870357)

[3.3. Requirement parameters 9](#_Toc498870358)

[3.4. Privacy Analysis 10](#_Toc498870359)

[3.5. Complexity discussion 11](#_Toc498870360)

[3.6. Performance Analysis 12](#_Toc498870361)

[3.7. Query success ratio 14](#_Toc498870362)

[Chapter 4 Appointment Card Protocol 21](#_Toc498870363)

[4.1. System Model 21](#_Toc498870364)

[4.2. Appointment Card Protocol Overview 21](#_Toc498870365)

[4.3. Appointment Card 22](#_Toc498870366)

[4.4. AC life cycle 24](#_Toc498870367)

[4.5. System parameters 24](#_Toc498870368)

[4.5.1. Obfuscation distance 25](#_Toc498870369)

[4.5.2. Friends obfuscation distance 25](#_Toc498870370)

[4.5.3. Distributing segment 26](#_Toc498870371)

[4.5.4. Generating Speed 27](#_Toc498870372)

[4.5.5. AC timeout 27](#_Toc498870373)

[4.6. Generating appointment cards 27](#_Toc498870374)

[4.7. Exchange distributing appointment cards 28](#_Toc498870375)

[4.8. Exchange ready appointment cards 30](#_Toc498870376)

[4.9. Sending queries 31](#_Toc498870377)

[4.10. Sending replies 31](#_Toc498870378)

[4.10.1. The LBSP part 31](#_Toc498870379)

[4.10.2. The first agency 32](#_Toc498870380)

[4.10.3. Intermediate agencies 33](#_Toc498870381)

[4.10.4. The last agency 34](#_Toc498870382)

[4.11. Exchange sending process 36](#_Toc498870383)

[4.11.1. Generate new ACs 37](#_Toc498870384)

[4.11.2. Pick dispensed ACs 38](#_Toc498870385)

[4.11.3. Pick ready ACs 38](#_Toc498870386)

[4.11.4. Send ACs 38](#_Toc498870387)

[4.11.4.1. Get the number of ACs 39](#_Toc498870388)

[4.11.4.2. Check duplicated appointment numbers 39](#_Toc498870389)

[4.12. Exchange receiving process 40](#_Toc498870390)

[4.13. Transmit replies 40](#_Toc498870391)

[4.14. Example 42](#_Toc498870392)

[Chapter 5 Vertex reduce algorithm 46](#_Toc498870393)

[5.1. Ignorable vertex and reserved vertex 46](#_Toc498870394)

[5.2. Vertex reducing 48](#_Toc498870395)

[5.2.1. Remove ignorable vertices 48](#_Toc498870396)

[5.2.2. Tidy reserved vertices connections 48](#_Toc498870397)

[5.2.3. Iterations 49](#_Toc498870398)

[5.3. Vertex assembling 50](#_Toc498870399)

[Chapter 6 Finding nodes in a range 52](#_Toc498870400)

[6.1. Static nodes 52](#_Toc498870401)

[6.2. Mobile nodes 53](#_Toc498870402)

[6.3. Complexities 54](#_Toc498870403)

[REFERENCES: 55](#_Toc498870404)

List of Figures

[Figure 4.1 AC exchange process (send) 7](#_Toc492582640)

[Figure 4.2 AC exchange process (receive) 10](#_Toc492582641)

[Figure 4.3 Transmit reply 11](#_Toc492582642)

List of Tables

[Table 1 Appointment Card 3](#_Toc492582645)

[Table 2 AC system parameters 4](#_Toc492582646)

[Table 3 AC states 5](#_Toc492582647)

[Table 4 friend-encounter list **错误!未定义书签。**](#_Toc492582648)

[Table 5 query-send list **错误!未定义书签。**](#_Toc492582649)

[Table 6 coordinate-query-send list **错误!未定义书签。**](#_Toc492582650)

[Table 5 AC storages 5](#_Toc492582651)

# 

# Introduction

Location privacy is becoming a major concern in OMSNs which can be viewed as Delay Tolerant Networks (DTNs) [1], exhibiting lack of continuous connectivity. More specifically, in OMSNs, it is not necessary for senders to have an end-to-end routing path to their destinations. Nodes make contacts when they encounter each other. Location-Based Services (LBSs) are common applications in OMSNs and are widely used in military, government organizations, emergency services and many commercial sectors [2], especially after the proliferation of localization technologies, like GPS. LBS users often send their location to LBS providers/servers. Many people access LBSs with their portable devices, as a result, their locations are also bound to their devices. In this case, LBS users face a continuous risk that their location may be leaked from the LBS applications. This makes people unwilling to use LBSs. Thus, location privacy protection has become a critical issue in LBS applications.

Early location privacy protection methods, such as obfuscation algorithms, generate anonymized areas for the original LBS requesters so that the requesters are mixed with a group of other LBS users [3]. Users send an anonymized area instead of their exact coordinate to the LBS providers when launching a LBS request. After that, social ties are incorporated into obfuscation algorithms to improve or protect location privacy. For example, the authors in [4] use social ties to determine trustable friends who could be chosen as intermediaries to forward obfuscation queries. The authors in [5] and [6] present algorithms which aim at improving delivery performance. However, compared to [4], the query success ratios of [5] and [6] have been shown to increase only by 5% and 11% respectively, which are not significant.

These papers assume that attackers can access the LBS servers, which enables them to learn LBS users’ identities and locations. If a user sends a query to a LBS server with his real identity and location (e.g., a query asking for a path from his current location to a certain place), attackers can locate the user quite easily. Therefore, hiding the original requester’s real identity and the location is the focus of our work.

Inspired by [5] and [7] which use the social network for messages forwarding in mobile ad hoc networks, we propose a distributed location-privacy algorithm, called MHLPP, to guarantee location-privacy and achieve a higher query success ratio. The introduction of social networks enables us to hide the original requester’s information behind his friends. When a user wants to send a query, he starts to look for friends based on information in his social network. He sends his query to the first encountered friend who is then responsible for forwarding the query to the intended location. This friend can also pass the query to one of his friends when they encounter. When the distance between the user carrying this query and the original requester exceeds a specified threshold, the user sends the query to the LBS server directly without having to find a friend to pass on. At that time, he also replaces the original requester’s information with its own identity and location, which enables the LBS server to receive the query without any information about the original requester. After receiving the query, the LBS server replies to the last friend (the user sending the query to the LBS) who then transmits it to the original requester.

MHLPP contains an *obfuscation phase* and a *free phase*. The process of finding friends for forwarding before the last friend sends the query out is called the obfuscation phase. In the free phase, the friend holding the query simply sends it to the LBS server, replacing the original requester’s identity with its own. The authors in [5] take a similar approach but differ from our approach as follows. Instead of finding *k* friends to finish the obfuscation phase as in case of [5], MHLPP takes the query to a place a specified distance away from the original requester. Moreover, [5] only selects friends among its neighbors while MHLPP selects friends among one-hop and multi-hop neighbors. This improvement enables MHLPP to gain a higher query success ratio when there are fewer friends in the network. In order to provide a secure communication among requesters and friends, especially multi-hop neighbor friends, encryption algorithms are used in MHLPP, unlike [5]. Our simulation results show that both one-hop and multi-hop connection between friends are acceptable in MHLPP while preventing un-trusted intermediate users from knowing the content of the queries.

The rest of this paper is organized as follows. Section II presents some related protocols in privacy-protection. The process of MHLPP is described in Section III. Experimental results are provided Section IV followed by some concluding remarks.

# Background and Related Work

Users face risks of information breach when they access a semi-trusted LBS provider, because anyone who has access to data in LBSs is able to steal and misuse LBS users’ location-privacy. Considering that LBSs rely on location-aware computing, it is unavoidable to leak users' location from LBSs. Therefore, balancing “these two competing aims of location privacy and location awareness” [8] is always a challenge.

Some early solutions, like [9] and [3], generate a specific area based on *k*-anonymity [10] for each user who needs to send queries. For example, [9] gives a rectangle as an anonymized area, in which all nodes form a group to hide the original requester. But, it requires at least *k* connected agents to complete its obfuscation process. [3] uses a central anonymity server as a mix router. As a result, it is necessary for each node to have a continuous connection to the server. That is hard to achieve in a sparse DTN. With a similar problem as in [3], [11] employs a matchmaker which is used to match users and advertisements, then users can achieve anonymization of their identities and locations from the matchmaker. However, the matchmaker is a high-risk in the network, because it collects so much private information. In the work in [12], exact locations and requests from clients are replaced by a location anonymization engine before they arrive at LBS providers. Since the anonymization engine learns all exact locations and requests, it becomes a better target for malicious attack.

There are protocols with more servers. The servers are settled in the network and each one of them takes charge of a certain area. [13] uses roadside units (RSUs) as mix servers in a vehicular DTN, and the destination is encrypted during forwarding, so eavesdropping queries cannot help attackers to locate users. But deploying the RSUs is not always feasible. In [14], sensor nodes which are scattered throughout the network provide anonymized locations for users. Since the sensor nodes’ coverage should possess a non-overlapping characteristic, it is difficult to deploy them in real-world. Besides, the mix servers and sensor nodes might be more prominent targets than LBS providers.

However, the system could also be a distributed one, like protocols [4], [5] and [6]. The obfuscation processes are performed by each separated node independently without any help from a third-party device. The utilization of label as in [15] makes it easier for nodes to mix themselves into a group, which is a significant difference from these protocols and the previous ones. Algorithms above use groups instead of an area to protect users. [5] imports 2 concepts: the obfuscation phase and the free phase. In the obfuscation phase, queries must be transmitted between friends for *k* times. When there are only a few friends in the network, it is hard for a node to find an available next hop in obfuscation phase. [5] and [6] attempt to improve [4]’s performance in the obfuscation phase, which might be a safety tradeoff, because some ineligible users in [4] are chosen as friends based on the additional standards imported by [5] and [6].

# Multi-Hop Location-Privacy Protection

## System Model

Our network architecture consists of two main entities: Users and LBS Providers (LBSPs). Due to the introduction of the social network, the users’ social information can be used in obfuscation forwarding process. Based on available information in the social network, the relationship between two users can be considered as friends or strangers. The user who makes a query to an LBSP will be called the original requester while the others are called intermediate users. LBSPs are located in fixed locations and their coordinates are known by all users when users join the network. Attackers are assumed to be able to access LBSPs, and attempt to locate original requesters. We assume that the LBSPs are semi-trusted and the strangers are un-trusted. We also assume that both entities have sufficient resources, like computational capability, storage and battery power.

Since two friends could be a pair of multi-hop neighbors, users can leverage Optimized Link State Routing Protocol [17] to seek friends continuously after entering the network, so that they can recognize each other and make contact in time. When a user carries an obfuscation phase query, he might send the query to a multi-hop friend through several strangers. In this case, a secure communication is necessary for between them, so that they must send the query encrypted to prevent strangers from learning anything about the query. Each user obtains a pair of asymmetric keys (public and secret key) before he joins the network from a certificate authority using well-regarded techniques, like in [16]. Whenever a user detects a new friend, he sends a request to the friend asking for his public key. In this way, a user can get his friends’ public key when they encounter each other. Even though several strangers can be active in the obfuscation phase, the queries can still be securely sent to the user’s friend.

The relationship strength is often “a hidden effect of nodal profile similarities” [18]. Let *SVi,j* denote a value of relationship strength which user *i* determines whether user *j* is an acceptable friend based on the relationship strength. For every pair of users (*i* and *j*), we assume that there is an *SVi,j* . If *SVi,j* is bigger than a specific friend threshold *Tmin*, set by the original requester, user *j* is considered as a friend of user *i*; otherwise, it will be treated as a stranger. The notations used in this paper and their meanings are shown in Table I.

## Details of MHLPP

MHLPP aims to protect the original requester’s (*N*0’s) location-privacy using an obfuscation path. In other words, a query *q* which needs to be obfuscated must go through a series of friends after it leaves *N*0. The whole process includes two parts: the obfuscation phase and the free phase. In the former phase, *q* is only transmitted among friends, until it is sent to an area called “*obfuscation area*”. At the end of that phase the last friend *Nf* replaces all *N*0’s information by its own and forwards *q* with an arbitrary DTN forwarding protocol, like the one suggested in [19]. In this case, what attackers can learn from the database in LBSP is *Nf*’s information, so they can hardly infer the original requester’s identity and location based on that information. The free phase starts when the query *q* is forwarded by *Nf* and ends when it reaches the LBSP.

|  |  |
| --- | --- |
| Parameter | Meanings |
| *N0* | the original requester |
| *Ni* | if *i* > 0, it denotes the friend chosen by *Ni*-1.  If *i* = 0, it is *N0*. |
| *Nf* | the last friend who handles the obfuscation query |
| *Nd* | the destination or the LBSP |
| *Ki* | the public key of *Ni* |
| *Si* | the secret key of *Ni* |
| *q* | a query of *N0* |
| *rq* | the requirement for the query *q* |
| *msg* | a message contains *q* and *rq* |
| *Emsgi* | the encrypted *msg* using *Ki* |
| *Sid* | the original requester’s identity |
| *Did* | the destination’s identity |
| *Ls* | the location of *N*0 when it sends the query to *N*1 |
| *Rp* | the inner radius of the obfuscation area |
| *Rs* | the external radius of the obfuscation area |
| *Tmin* | the social value bound for friends |
| *Cmax* | the extra path limit in each obfuscation forward |

Because *Nf* is the only identity the LBSP knows, the LBSP has no choice other than replying to the last friend *Nf* when it receives the obfuscated query. The reply can be delivered with an arbitrary DTN routing protocol as the free phase query does. The friend *Nf* should remember who is the real destination (*N0*) of this reply, then he transmits it to *N*0. In this way, *N*0 is able to send a query *q* to an LBSP while not exposing his own information.

The obfuscation phase of a query *q* starts when the query leaves the original requester *N*0. When a user is holding an obfuscation phase query, he starts sensing connected friends continuously, which enables it to communicate with one-hop or multi-hop neighbor friends. Even though users in the mobile network use OLSR protocol [17] to detect others automatically, they do not communicate with their friend unless they have a requirement to send an obfuscation query. Also, they do not ask their friends for public keys. Therefore, carrying an obfuscation phase query requires a user to execute MHLPP algorithm.

When *N*0 finds the first available friend *N*1, he asks *N*1 for a public key *K*1, which will enable him to encrypt his query using *K*1. That prevent others, e.g. strangers, from learning information in the message *msg* that that contains both *q* and *rq*. *rq* is *N*0’s requirement for *q*, which is always sent with the query *q* and remains constant until the end of the obfuscation phase (we discuss this in part C). Friends who get the query can infer *q*’s obfuscation area based on parameters *Rp*, *Rs* and *Ls* in *rq*, which is a ring with inner radius *Rp*, external radius *Rs* and center *Ls*. Before *N*0 sends the query *q* to his friend, he initializes parameter *Ls* to his current location and encrypts *msg* using *K*1 to get *Emsg*1 which is what *N*0 sends to *N*1.

The destination of *Emsg*1 is *N*1, which is a plaintext in *Emsg*1, so that other intermediate users (strangers) can help *N*0 forward *Emsg*1 to *N*1. In this step, strangers transmit *Emsg*1 using the OLSR protocol if *N*1 is a multi-hop neighbor of *N*0. Strangers learn nothing other than the identity of *N*1, because both *q* and *rq* are encrypted. They cannot help attackers locate *N*0 because they do not know *Sid* (the identity of *N*0) and *Did* (the LBSP) included in *rq*.

When *N*1 receives *Emsg*1, he decrypts it with its secret key *S*1 to get *q* and *rq*. If *q* is already in the obfuscation area defined in *rq* (i.e., it is already in the ring), the query *q* finishes its the obfuscation phase. Then *N*1 replaces all information of *N*0 with his own. For example, the *Sid* is replaced with *N*1. If a location is necessary for the LBS, *N*1 uses his own current location and records this change in his memory before initiating the free phase. A free phase query can be then be forwarded to the destination (i.e. LBSP).

If *q* is still in the obfuscation phase (not in the ring), *N*1 performs similar actions just as *N*0 expect modifying *rq*. Another difference is that instead of finding friend randomly, *N*1 would seek for a friend who is nearer in the obfuscation area, and so will the following friend.

The detailed algorithm is explained in Algorithm 1 where *Nx* is a neighbor of *Ni* who is carrying the query *q*. Function“*DealWithQuery*” is responsible for dealing with a query. For *N*0, he generates the query *q* and its requirement *rq*. If *Ni* receives *Emsgi*, he decrypts it with his own secret key *Si*. If *q* finished its obfuscation phase at *Ni*, *q* will be required to be forwarded in free phase immediately. Otherwise, *q* needs to be processed in the obfuscation process. Both *q* and *rq* are stored in *Ni* until they are sent to the next friend. *Ni* starts detecting friends continuously if and only if *Ni* carries one or more obfuscation phase queries.

When *Ni* detects a new neighbor *Nx* (one-hop or multi-hop neighbor), he follows steps in “*WhenEncounterUser*”. For an expired query, *Ni* simply drops it. If the query *q* is already inside its obfuscation area, *Ni* switches it to the free phase. If *q* stays in the obfuscation phase and *Nx* is an available friend, *Ni* encrypts both *q* and *rq* using *Nx*’s public key *Kx* to get an encrypted message *Emsgx*. Then *Ni* forwards *Emsgx* to *Nx* and stops sensing friends after *Emsgx* departs from it.

When we mention that *Ni* switches a query *q* to the free phase, *Ni* actually replaces *N*0’s information with its own one in *q* to get *q\** and records this replacement in its storage, then *Ni* uses the Spray and Wait protocol [19] to forward *q\** in plaintext. That allows *Ni* to hide *N*0’s identity and forward a reply from LBSP to *N*0.

## Requirement parameters

In the obfuscation phase, *rq* is always in *msg* so that friends who get *msg* can make decisions (e.g. selections of friends) based on it.

All parameters (i.e., *Sid*, *Did*, *Rp*, *Rs*, *Ls* ,*Tmin* and *Cmax*) in *rq* are given by *N*0 before *Emsg*1 leaves *N*0. Parameters *Sid* and *Did* record the identities of *N*0 and the destination (LBSP) *Nd*, based on which last friend *Nf* is able to send the query freely to *Did* (*Nd*) and forward the reply to *Sid* (*N*0).

The obfuscation area is a ring with an inner radius *Rp* and an external radius *Rs*. As shown in Fig. 1a, the obfuscation area is actually the grey area “*a*”. Obfuscation area must guarantee both the original requester’s location-privacy and location awareness. In other words, the value of *Rp* should be big enough, so that there are sufficient users in the inner circle (with a radius *Rp*). At the same time, *Rs* should be small enough so that the LBSP can provide a service, acceptable to *N*0.



A user *Nx* can be chosen by *Ni* as a friend for *q* if and only if *SVi,x* is bigger than the threshold *Tmin*. The original requester *N*0 can set various values for his queries based on their importance. If *Tmin* is large, there would be fewer friends for any users in the network which reduces the query success rate to a certain extent, as a result. We assume that the original requester can balance the level of privacy and the success ratio.

Most DTN routing protocols aim to deliver queries through the shortest path, while MHLPP pays more attention to security in its obfuscation phase. Consequently, the obfuscation process in MHLPP results in a longer path from the original requester *N*0 to the destination *Nd*. To limit the length of the path, we introduce parameter *Cmax* which is the maximum extra path (e.g., the difference of the length and the distance between the *Ni* and LBSP) we can tolerate. For any friend *Ni* who gets a *Cmax* from *rq*, if he selects *Nx* as the next friend, the extra path should not be longer than *Cmax*. Let’s denote the optimal path from user *m* to *n* by *Dis*(*m*,*n*), and the extra path from *Ni* to *Nd* through *Nx* by *Ci,x,d*. Then *Ci,x,d*  can be defined as follow.

 (1)

*Ci,x,d* must be a value smaller than *Cmax*. If *Dis*(*m*,*n*) is the straight-line distance between point *m* and *n*, then next friend *Nx* should be in an ellipse *EC* with focus points *Ni* and *Nd*. Let’s denote the coordinate of *Ni* by  and the coordinate of *Nd* by . Then, the equation of the ellipse *EC* is

 (2)

As shown in Fig. 1a, *Ls* is the center of the ring while *Rp* and *Rs* are the inner and external radii, respectively. The query *q* switches to the free phase when it enters the ring area.

As shown in Fig. 1b, *Ni* who is carrying obfuscation queries should choose his next friend *Nx* in the ellipse, which avoids the query *q* going through an unacceptably long path.

In conclusion, a query *q* starts at the center and moves inside the ring, until it reaches the obfuscation area. The point *Ni* in Fig. 1b should be inside the ring, and the point *Nd* might be anywhere. As a result, a user *Ni* who is carrying an obfuscation query detects a friend continuously who has a larger distance from *LS* and inside an ellipse. If there is a friend like that, *Ni* sends the query to that friend *Nx*.

## Privacy Analysis

We assume that attackers can achieve all information in LBSPs know. Obviously they can know the identity of the last friend *Nf* who replaces *N*0’s information with his own. It is possible for the attackers to locate *Nf* with little cost. For example, if *N*0 stops moving after sending the query *q*, it is reasonable for *Nf* to believe that *N*0 is in a ring centered at the location of itself with radii *Rp* and *Rs*. In other words, the distance between *Nf* and *N*0 should be in a range between *Rp* and *Rs*. If attackers find all users who satisfy this condition, the original requester might be among these users with high probability. Then, a success ratio  to locate *N*0 can be measured by a conditional probability

 (3)

where  is the probability that *N*0 is in the ring (i.e., the distance between *N*0 and *Nf* is larger than *rp* and smaller than *rs*). Here,  is the number of users who are in the ring. Attackers locate *N*0 successfully if and only if *N*0 is in the ring, at the same time, attackers pick the correct one from all  users at that area.

In the worst case, attackers know exact values *Rp* and *Rs*. Then, the Eqn. (3) becomes

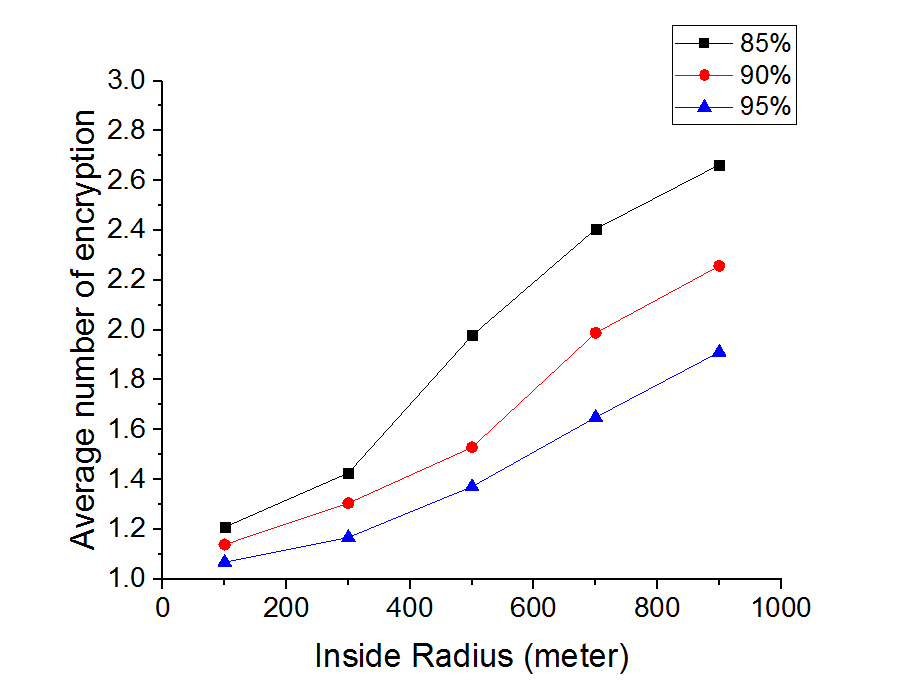
 (4)

where  is the probability that *N*0 is on the ring (i.e., the distance between *N*0 and *Nf* is larger than *Rp* and smaller than *Rs*).  is the number of users who are in the ring. Since parameters (e.g., *Rp* and *Rs*) in *rq* are kept secret among trusted friends in our system model, attackers can hardly get the actual values of those parameters.

## Complexity discussion

In order to guarantee secure communications among friends, encryption is introduced in our protocol. In the obfuscation phase, the query is transmitted along friends, i.e. *N*0, *N*1, *N*2, ..., *Nf*. When the query is sent from *Ni* to *Ni*+1, a pair of encryption and decryption is needed, so the number of such pairs *Ten* should be equal to *f*.

*Ten* grows with both *Tmin* (threshold used to decide friend relationship) and inner radius *Rp*. Essentially, it is the number of friends participating in transmitting a query *q* in its obfuscation phase that influences *Ten*. Given a smaller *Tmin*, a user carrying the obfuscation phase query has more chances to encounter more friends in a certain area. A larger *Rp* also leads to a bigger area inside the ring, so that there are more friends in this area. We evaluate the number of encryptions and decryptions in our simulation. Fig. 2 shows the average number of encryptions (*Ten*) with different friend thresholds *Tmin* and inner radius *Rp*. We observe that *Ten* increases steadily as we increase *Tmin* and *Rp*.



It is evident that encryption and decryption process in MHLPP results in an extra cost in both energy and computational resources. However, the number of encryptions and decryptions is quite low (below 3) which reasonable based on the experiment.

## Performance Analysis

We use the map of Helsinki in our simulator to evaluate MHLPP. It is also compared against the known protocol, Hybrid and Social-aware Location-Privacy in Opportunistic mobile social networks (HSLPO) [5]. The simulation parameters are shown in Table II. All pedestrians and cars are users in MHLPP. These users are moving on the map along streets continuously. There is an LBSP fixed at a random location on the map. For each user, we give him random social values between 0% and 100%, each corresponding to all other users. Each value has the same probability, so we can compute the expected number of friends of an user. For example, if we are given a privacy threshold (*Tmin*) of 85%, then there might be 15% (100%-85%) users who are friends of a certain user.

As shown in Table II, there are 126 users in the map. For each of them, say user *i*, we give him 126 random *SV* values, which denotes the relationship strength between him and other users, so that there are 1262 *SVs* in our simulation. The *SV*s are between 0 and 100. As a result, if the *Tmin* is equal to 85, the average number of friends of each user should be 18.9 (=).

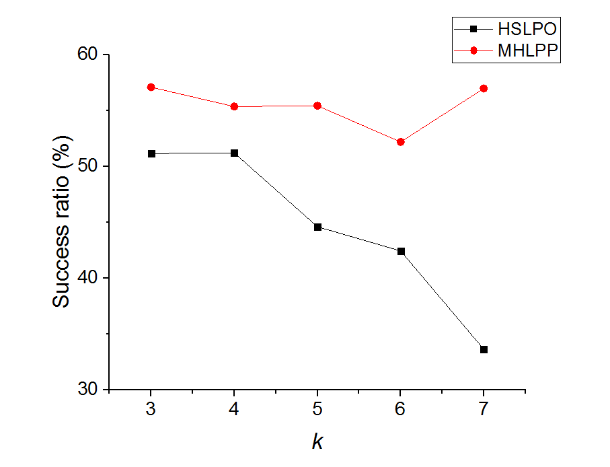
|  |  |
| --- | --- |
| Parameter | Value |
| Simulation Time | 10 minutes |
| Map Size (W x H) | 4500 m x 3400 m |
| Total number of users | 126 |
| Pedestrians/ Cars | 84/42 |
| Communication Area Radius | 10m – 90 m |
| Pedestrian Speed | 1.8-5.4 Km/h |
| Car Speed | 10-50 Km/h |

Users are placed at random locations at the beginning of each experiment. We choose another random point for each user, so that he can move back and forth along streets between that point and the point his starts at. The user speed depends on its type (pedestrians or cars) and set randomly. All queries have a 10-minute timeout. The queries which are expired before they reach the LBSP (the destination) are considered to be failed in our success ratio statistics.

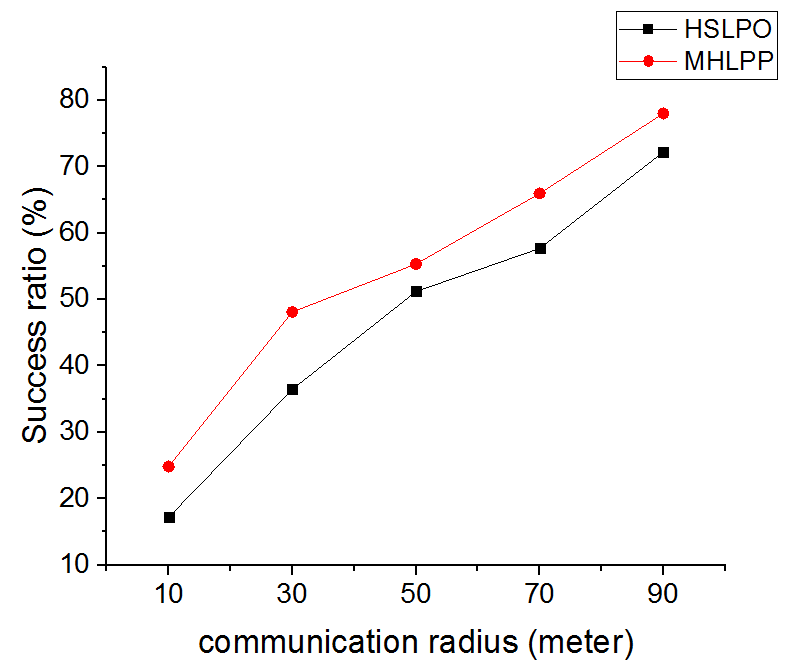
Figs. 3-5 compare performances between HSLPO and MHLPP for different values of *k*, communication radius and privacy threshold (*Tmin* in MHLPP). The *k* is the privacy-level requirement in HSLPO. Both HSLPO and MHLPP have different criteria in which a query can switch to the free phase. To make them comparable, we create a new parameter, called *obfuscation distance*. If a query leaves *N*0 at location *La* and switches to the free phase at *Nf* whose location is *Lb*, then the obfuscation distance is the straight-line distance between *La* and *Lb*. We test the obfuscation distances of HSLPO with different parameters, and then we set the inner radius of MHLPP to those values. The query success ratio is the ratio of delivered queries to the total number of queries. The number of hops (*h*) is the number of intermediate users between *N*0 and the destination (LBSP). We count the number of users surrounding the last friend in a specific range, which is *k* times the communication radius. We calculate the entropy using the reciprocal of the number of surrounding users.

## Query success ratio

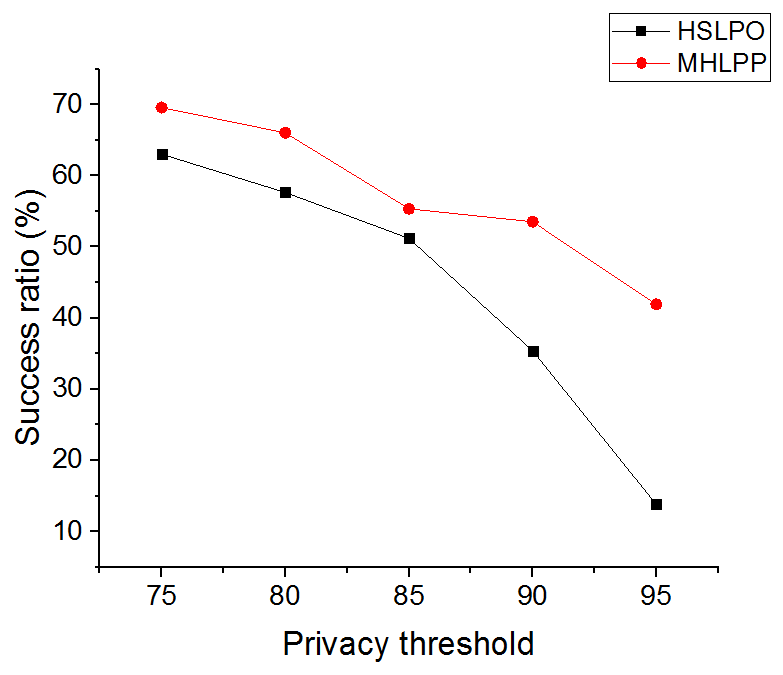
The query success ratio is the percentage of delivered queries among a number of attempts. Based on the timeout value in Table II, a query is delivered successfully, if it arrives at the LBSP (the destination) before the timeout; otherwise it fails. We use the query success ratio to evaluate the delivery performance of MHLPP.



(a)



(b)



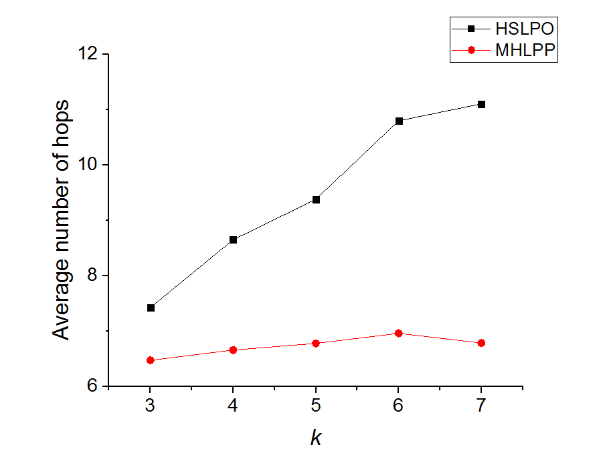
(c)

Fig. 3. The query success ratio comparison between HSLPO and MHLPP. (a) Query success ratio with various K. (b) Query success ratio with various communication radiuses. (c) Query success ratio with various privacy thresholds.

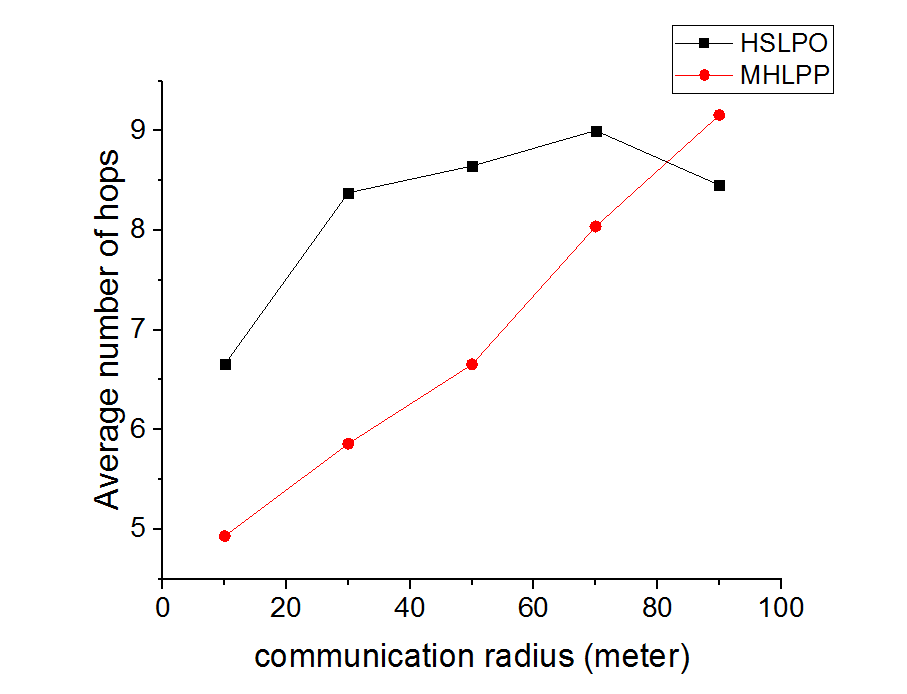
As shown in Fig. 3a, the success ratio in MHLPP is always higher than that in HSLPO. As the value of *k* increases, HSLPO success ratio drops sharply while MHLPP remains stable. This is because the larger *k* is, the harder it is for HSLPO to find enough friends in a limited time. The lack of friends has less impact on MHLPP. We observe that the success ratio of MHLPP rises when *k* = 7. That is because it depends on the inner radius which is equal to the obfuscation distance of HSLPO. The obfuscation distance decreases when *k* = 7, because most of the queries which complete their obfuscation phase have a short obfuscation distance. In Fig. 3b, both HSLPO and MSLPP values increase and have the same trend when given a larger communication radius. The reason is that the communication radius effects the free phase more than the obfuscation phase for both. As shown in Fig. 3c, higher privacy threshold leads to lower success ratio in two algorithms. Its impact on HSLPO is more intense than that on MHLPP, which is the most important characteristic of MHLPP. MHLPP has a better performance than HSLPO especially when there are fewer friends in the network. MHLPP can transmit messages with the help from strangers in its obfuscation phase while HSLPO cannot.

Number of Hops

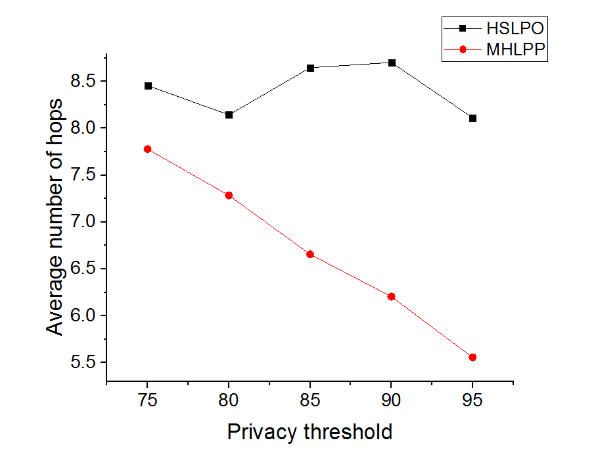
We count the number of hops it takes for queries to be delivered successfully and calculate the average. Every user who takes part in the delivery process is considered in the hop count. We introduce this criterion to measure the routing path length of the algorithm. MHLPP is more sensitive to the probability that a user encounters a friend than HSLPO is. The reason is that MHLPP aims to reach a certain distance rather than taking certain number of hops. In other words, MHLPP continues sending queries to other friends until queries enter their obfuscation area. In this process, MHLPP takes every chance to forward queries. If it is hard for MHLPP to find friends, it can also take the queries to obfuscation areas with fewer friends. Therefore, the probability that users encounter friends has less impact on the performance of MHLPP. The result of this experiment is shown in Fig. 4.



(a)



(b)



(c)

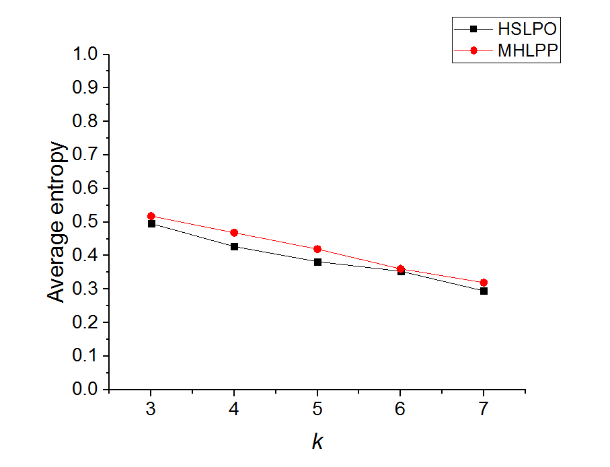
Fig. 4. The number of hops comparison between HSLPO and MHLPP. (a) number of hops with various *k*. (b) number of hops with various communication radii. (c) number of hops with various privacy thresholds

In Fig. 4a, the number of hops in HSLPO is affected by parameter *k* obviously. Especially, the first *k* hops forwarders must be friends, which makes it hard for HSLPO to have queries to be forwarded successfully. Both protocols have similar number of hops in their free phases, but HSLPO has exactly *k* hops in its obfuscation phase, while MHLPP can have fewer than *k* hops. In figure 4b, the number of hops in MHLPP grows with the communication radius obviously, while it does not change a lot in HSLPO, because only successful delivery queries are counted in the statistics. Given a large communication radius, MHLPP has a much higher success ratio for delivered queries as it can connect more friends. In Fig. 4c, the value of MHLPP drops for higher privacy thresholds. The reason is the same as Fig. 4b. For HSLPO, no matter how hard it is to find a friend, it attempts to find exactly *k* friends. However, if it is too hard to find a friend, MHLPP’s friend can carry the query while moving and complete the obfuscation process. For higher privacy thresholds (i.e. resulting in fewer friends), MHLPP chooses to carry queries other than finding friends. That results in a drop in the number of hops.

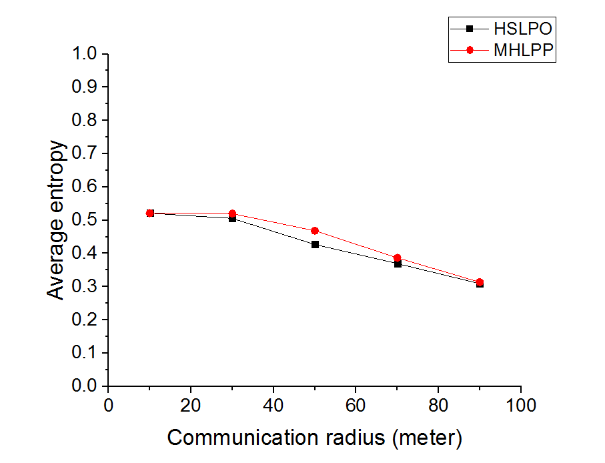
Security

Since the principles with that two protocols protect original requesters are different, we evaluate the probability that attackers locate the original requester if the distance between him and the last friend is smaller than a some value *r*, which is equal to *k* times the communication radius. Since the last friend reveals himself to the LBSP, attackers might locate him accurately. We assume that attackers know the privacy parameter *k*. In the worst case, the distance between the original requester and the last friend is smaller than *r* when attackers start to locate the original requester. That gives attackers a chance to locate the original requester. We count all users who are inside the radius *r* of the last friend, and the original requester is one of them. For example, if there are *m* users in the area, the probability should be *1/m*. Figure 5 compares the entropy *E* of both HSLPO and MHLPP. We use the following formula for computing entropy:

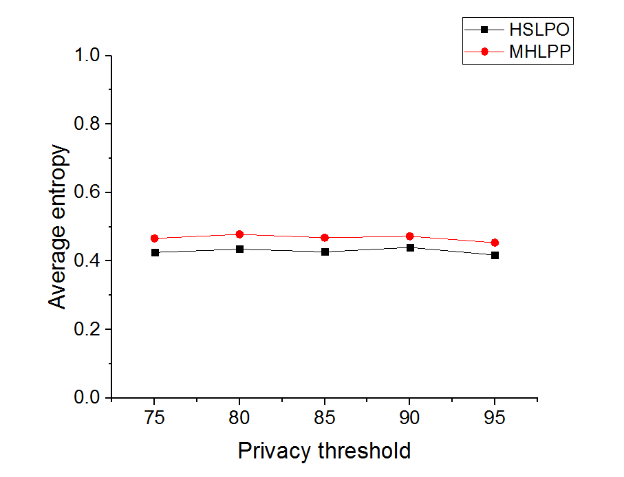
From Fig. 5a, we observe that MHLPP has very small (about 0.04) increase in entropy compared to HSLPO. When the original requester is in the circle centered at the last friend, HSLPO is a little more secure than MHLPP but not by very much. That is because HSLPO always switches to the free phase when the last friend encounters the previous friend, so the previous friend must in the circle. MHLPP does not have this condition. Both graphs in Fig. 5a and Fig. 5b have the same trend. The curve of MHLPP is also a little higher than that of HSLPO, while two curves almost meet when the communication radius is small or large. Given a small radius, the entropy of both protocols are small. When the radius is large, we can ignore the effect of the previous friend mentioned about for Fig. 5a as there are so many users in the circle. From Fig. 5c, since the circle neither expands or shrinks, we observe that the two curves exhibit similar behavior. The values of HSLPO are always lower than the correlated values of MHLPP. However, as we observed in the experiment, the last friend is always hundreds of meters away from the requester.



(a)



(b)



(c)

Fig. 5. Locating probability entropy comparison between HSLPO and MHLPSP. (a) Locating probability entropy with various K. (b) Locating probability entropy with various communication radiuses. (c) Locating probability entropy with various privacy thresholds.

# Appointment Card Protocol

## System Model

The network architecture consists of two main entities: Users and Location-Based Service Providers (LBSPs). Users are mobile and communicate with others in a certain range, e.g., the communication range of their portable devices. For a user, other ones in the social network are either strangers or his friends whom he can detect when they are in his communication range. Let *RSi,j* denotes the relationship strength between user *i* and *j*, if *RSi,j* is larger than a specific friend threshold *FTmin*, user *i* and *j* (*Ui* and *Uj*) are friends. LBSPs are fixed and not connected in a social network, which provides Location-Based Services (LBSs) for users. We assume that the only information which is necessary for the LBS is the location of the original requester (*U*0), but he should still give an identity to the LBSP so that the LBSP can reply to that identity.

Attackers can access the database of LBSPs, so that they can learn everything recorded in LBSPs, including users’ identities and locations. If *U*0 sends queries with his own identity and accurate location, his will be located by attackers easily, so the key is to hide his identity and location. We propose Appointment Card Protocol (ACP) to protect the identity and location-privacy of the original requester by providing a fake identity which disenable attackers to locate the original requester with records in the LBSP database.

## Appointment Card Protocol Overview

We propose an Appointment Card Protocol (ACP) to protect original requesters when they are served by LBSPs. A user (*Agc*1) generates his own Appointment Cards (ACs) containing his identity, a unique Appointment Number (AN) and a location which are be exchanged when he encounters another user. When *U*0 sends a query, he chooses an AC and sends the query with the identity (*Agc*1), AN and location in the AC. A LBSP replies to *Agc*0 when it receives the query. Since *Agc*1 is actually the one who generates the AC, he re-transmits it to the user (*Agc*2) whom he exchanged the AC to, and so do the following users until the reply reach *U*0.



Figure 4.1 whole process example

As shown in Figure 4.1, the whole process can be considered as the following parts: 1). exchanging cards among all users who are called agencies (i.e., 1 and 2); 2). exchanging cards among friends (i.e., 3); 3). sending query using information of appointment cards (i.e., 4); 4). forwarding the reply among agencies (i.e., 5, 6 and 7); 5). relay to the original requester (i.e., 8). This figure will help us to describe the protocol in the whole charpter.

## Appointment Card

Since the original requester cannot use his own identity to communicate with the LBS provider (LBSP), he must use others’ identity () to send queries, so that the LBSP must reply to the original requester using . Appointment cards make it possible that agencies forward the reply to one by one, while they know nothing about . In fact, the appointment card indicates a path, through which can get its reply. In the Figure 4.1, we can see that , and are strangers, so that attackers can hardly infer from the identity . is a friend in ’s social tie (’s friend or his friends’ friend) and is also the only one who knows how to reach . In other words, it is hard for attackers to infer from .

We should notice that received the appointment card from who might be a stranger, so that it is not safe for to use the appointment card, because a stranger knows the information of that appointment card and the identity of the next agency . The appointment card cannot be used until exchanges it to a friend (e.g., ). can use it to send queries or exchange it to other friends. Then the appointment card is called **ready appointment card**, or it is called **distributing appointment card**.

Any user in the network can generate his own Appointment Cards (ACs) and he is called the creator of those ACs. The creator writes down his own identity (*Cid*), a unique Creator Appointment Number (*Capt*) on his AC. When users exchange appointment cards, they use parameters and to identify whom the appointment cards are received from. Entries of AC are shown in Table 1.

Table 1 Appointment Card

|  |  |  |
| --- | --- | --- |
| ***entry*** | ***explanation*** | |
| *Cid* | The identity of the creator who generates the AC. |
| *Capt* | A unique number that distinguishes an AC from other ones generated by the same creator. |
| *Aid* | The identity of an agency who gives the AC to the recent holder. (The previous hop of the AC) |
| *Aapt* | A unique number that distinguishes an AC from other ones transmitted by the same agency. |
| *time-out* | The time when the AC expires. |
| *EQ* | A queue (Exchange Queue) which records users who exchange the AC in order. Its length is *EQL*. |

## AC life cycle

The life cycle of an AC starts when it is generated by its creator. The creator initializes the *Cid* and *Capt* when he generates it. These two entries remain constant in the whole life cycle. In the first *k* (see Table 2) exchanges, the agency who is sending it modify its *Aid* to his own identity, replace the *Aapt* by a new appointment number generated by him and adds his identity into its *EQ* before sending it out. When the AC’s *EQ* length reaches *k*, it is eligible to be used in a query and called a ready AC. A ready AC cannot be exchanged to strangers. If a user exchanges a ready AC to his friends, he never modifies any entries in the appointment card (i.e., *Aid*, *Aapt* and *EQ*). Each of ACs can be used for only one time. In other words, an AC expires, when reaching its time-out or it is used in a query.



Figure 4.2 AC's life cycle

As shown in the Figure 4.2, an AC starts at the distributing state. If the length of its *EQ* reaches *k*, it is switched to the ready state. It can timeout in all states. If an AC is used for once, it is switched to the disabled state. A disabled appointment card can do nothing but waiting for timeout.

## System parameters

All system parameters are shown in Table 2.

Table 2 AC system parameters

|  |  |  |
| --- | --- | --- |
| **Parameters** | **explanation** | |
| *k* | The minimum length of the exchange record for an AC eligible to be used in a query. |
| *Seg* | dispensed ACs are exchanged when two friends encounter each other. |
| *GS* | The generating speed of appointment cards |
| *AT* | The timeout for appointment cards |

### Obfuscation distance

The obfuscation distance is the number of exchange before an AC is switched to the ready state. Let *k* denote the obfuscation distance, then an AC must be exchanged for *k* times before it becomes a ready AC.

As shown in Figure 4.3, an AC is exchanged by , , ..., . Since those agencies are strangers, the only relationship between two adjacent agencies is that they encounter each other somewhere. The relationship between and becomes weaker when we increase *k*. In other words, attackers can hardly infer the identity of when he only knows the identity of , and his difficulty increases with the parameter *k*.



Figure 4.3 obfuscation distance

### Friends obfuscation distance

Since and are strangers, it is possible that is exactly the attacker. We assume that the attacker knows the original requester’s () friends, then the identity of gives the attacker a good tip to infer . A solution to prevent the agency from learning is that the last *m* agencies are friends, as shown in Figure 4.4. The *m* is called friends obfuscation distance. The last *m* agencies are called trusted agencies.

For each pair of trusted agencies (i.e., and , ), they are friends. But two trusted agencies who are not adjacent might have a weak relationship (e.g., a friend of friends). Since there are *m* trusted agencies between the stranger and at least, can hardly infer from all information he learns. When , all ACs can be exchanged between friends only.

However, friends encounter each other rarely so that a big *m* increases the difficulty of distributing ACs. In this paper, we assign *m* to 1.



Figure 4.4 friends obfuscation distance

### Distributing segment

We assume that generates 3 ACs (i.e. , and ). It exchanges these ACs to , then exchanges them to and so on. At last, all of them reach . In this case, has several ACs whose routes (the list of agencies) are the same, so that has no choice but using the only route to send queries.

If agencies exchange ACs to different users, the above problem will not happen. We use a system parameter distributing segment to avoid exchange all ACs which are received together to the same user. Each user has distributing AC lists. It put received distributing ACs in one of those lists randomly. If exchanges ACs with another user, selects one of the lists randomly. Only ACs in that list will be exchanged to the other user.

For example, we assign . When receives , and from , may put and in its list1, while in its list2. If encounters , can exchange either and or only to . In this way, we separate ACs generated by the same creator.

### Generating Speed

Since ACs could be use and expire, users must generate new ACs continuously. We use to denote the speed of generating new ACs per user. Each user generates a new AC every second.

### AC timeout

Let denote the timeout of ACs. An AC expires after it is generated for seconds. When an AC expires, all agencies delete its information from their memory.

## Generating appointment cards

Maintaining is a certain number of ACs is a prerequisite for user to sending queries, so that users must generate ACs continuously on 2 principles, which are fairness and continuity.

Since creators and agencies act as obfuscation servers for the original requesters, ACs impose burdens on these users. In fact, agencies (a creator must be an agency) are unlikely to benefit from relaying messages to the original requester. They need to allocate memories to save ACs’ information so that they can forward replies correctly. To avoid users going into overload, fairness is an essential part of the generating strategy. In other words, users should generate a similar amount of ACs.

We assume that the timeout (, see 4.5.5) is 30 minutes, and every user generates 100 ACs at the beginning (the 0th minute) and generates no ACs until the 30th minute. It is hard for a user to select an appropriate AC for his query at the 29th minute, because all agencies of a selected AC will remove it information from their memories 1 () minute later, so that they cannot forward any replies, when the AC expires. Therefore, a steady and sustainable strategy is a prefer choice, instead of an intensive one.

In our protocol, each user generates a new AC every (see 4.5.4) second. Since an AC has a second timeout and we assume that no AC is used, there are about appointment cards in the network per creator after the system processes for a long time. In other words, each user maintains about ACs which is generated by himself, and we achieve the fairness part. Since users generate ACs continuously, ACs have various timeout so that it is likely for a user to pick an AC whose does not expire for a long time.

## Exchange distributing appointment cards

Since a ready appointment card must be exchanged at least k times, users exchange their distributing appointment card as frequently as possible, so that a distributing appointment card can be switched to a ready one quickly. As shown in 4.5.1 and 4.5.2, a distributing appointment card can be exchanged to any other users when its *EQL* is less than , while it must be exchanged to a friend after that (equal to or larger than ). However, there are also other conditions of whom a distributing appointment card can be exchanged to.

We assume that two friends (e.g., *A* and *B*) walk together for a long time, as shown in Figure 4.5. If *A* generates an AC (e.g., ) at and exchanges it to *B*, then is given back to *A* at , and so on. At , the *EQ* of is {}, which lose the meaning of exchanging, because it is easy for an attacker to infer *B* the last agency from *A* the creator. Sometimes, an *EQ* which is {A, C, B, A, D} is acceptable for users, but it is hard for the above one (only A and B).



Figure 4.5 walk together

We import an parameter to optimize the agency selection strategy. If a user gets an AC at time , he cannot get that AC again before . For example, the user *B* receives from the user *A* at , then *B* send it to a user *C*, and *C* can also send it to others. If a user who is carrying encounters *B* before , he cannot send to *B*. Therefore, if the parameter is infinite, there is no repeated agency in an AC’s *EQ*. In other words, an AC never reaches an agency twice.

Besides, we should also avoid ACs to having the same sequence of agencies, as we mention in 4.5.3. Then we get the strategy of exchanging distributing appointment cards.

Let us take a user *A* as an example. If user A encounters user B, user A pick one of his appointment card lists (see 4.5.3) (e.g., list1). Then user A checks whether he is a friend of the user B. The user A traverse all appointment cards in the list1, and appointment cards which match all the following conditions are exchanged to the user B.

1. If the length of the AC’s EQ (i.e. *EQL*) is not shorter than , then the user A must be a friend of the user B.
2. The user B does not carry the AC in the recent (time).

When the user *A* is sending a distributing AC to B, *A* adds its identity and recent time to the AC’s EQ. Besides, A must modify the AC’s to his own identity and its to a new one. The information in Table 3 are recorded in the user A’s memory. We should notice that the first agency (i.e., ) does not have a , because he is exactly the one who generates the AC, then the should be his own identity. When the user *B* gets those ACs, he puts each one of the received ACs to its appointment card lists respectively and randomly.

Table 3 relay table entries

|  |  |
| --- | --- |
| Name of entries | descriptions |
|  | The generated by the previous agency |
|  | The identity of the previous agency |
|  | The new (generated by himself) |
|  | The identity of the next agency |
|  | The length of the AC’s EQ (should larger than 0) |
|  | The time when the AC timeout. |

## Exchange ready appointment cards

Ready appointment cards can be only exchanged from a friend, because users do not want more strangers to know the information of ACs. The strategy of exchanging ready ACs face 2 main problems. First, some users send much more queries than others; second, the number of each user’s ready ACs should be similar.

We take users A and B as an example and assume that they are friends. Each of them has 10 ready ACs at the beginning, but the user A’s ACs exhausts quickly, because he sends 10 ACs. At the same time, the user B does not send any ACs. When they encounter each other, it is reasonable that the user B gives half of his ready ACs to his friend user A.

However, the problem becomes more complex in the following condition. Alice is a friend of Bob, while Bob is not a friend of Alice. So that Bob trusts Alice, but the user Alice does not trust the user Bob. We assume that the user Alice is a trustful but suspicious girl, so that many users trust her while she trusts few users. Bob is opposite. When users encounter Alice, they ask for ready appointment cards from her, but Alice asks for ready appointment cards rarely. Therefore, Alice carries few ACs, while Bob carries so many ACs. When Alice and Bob encounter each other, it is Bob who is asking for appointment cards instead of Alice. To make the strategy fair and efficient, users compare the number of their own ready ACs and the number of ready ACs carrying by the other user when two users are exchanging ready ACs.

When Bob encounters Alice, Bob tells Alice the number of his ready ACs (i.e., ) and whether he wants Alice’s ready ACs. If Alice learns that Bob needs her ready ACs, she compares the number of her ready ACs (i.e., ) and . If , Alice gives no ready AC to Bob; otherwise, she gives to Bob. In this way, ready ACs do not concentrate in a group of users who trust many users. Users do not modify any information in the ready ACs including and , so that ready ACs keep static after they leave the last () agency.

## Sending queries

For an original requester who want to send queries to the LBSP, his query must be delivered to the LBSP while the LBSP cannot learn the identity of him. Besides, the reply from the LBSP can be delivered to . The basic idea is that sends his query using another user ()’s identity to the LBSP, so that LBSP can reply to , then forward the reply to . As shown in the former Table 1, appointment card offers information to achieve that goal.

To enable LBSP reply to , ’s query includes an identity , which is the in the appointment card. also needs the to identify which appointment card the query is using. So that the query includes the and the in the appointment card, as shown in Figure 4.6. The network can deliver that query to the destination LBSP easily with any DTN protocols.



Figure 4.6 constitute query

The appointment card is marked as used when the query is ready to be sent, so that the appointment card cannot be used again in any other queries. Neither can it be exchanged to other users.

## Sending replies

### The LBSP part

When the LBSP received the query, it learns that the sender’s identity is instead of , which makes The LBSP reply to using the information in the query, as shown in Figure 4.7.



Figure 4.7 constitute replies

The receiver of the reply is , and the reply also includes the which is equal to the in the query.

### The first agency

When the first agency (i.e., ) gets the reply from the LBSP, he learns the in the reply. Since he is the first agency and the creator of the appointment card used in the query (and reply), the information of that appointment card is in his reply table (see Table 3). There must be an entry like Table 4 in his reply table, if that appointment card does not expire. He uses his identity () and the in the reply as the key to search his reply table, so that he knows the identity of the next agency (i.e. ), which enables the first agency to forward the reply to the correct agency . The unique number can help when he checks his reply table.

Table 4 reply table entries of the first agency

|  |  |
| --- | --- |
| Name of entries | value |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | The time when the AC timeout. |

The first agency modifies the reply’s sender ID (to ), receiver ID (to ) and *AApt* (to ), then forward the reply to .



Figure 4.8 the reply of the first agency

### Intermediate agencies

The process of forwarding replies for the intermediate agencies (the second to the one) is similar with that of the first agency. We take the second agency as an example. When the second agency receives the reply forwarded by the first one, he learns the which is equal to (see Figure 4.8) from the reply. There should be an entry in his reply table as shown in Table 5.

Table 5 reply table entries of the second agency

|  |  |
| --- | --- |
| Name of entries | value |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | The time when the AC timeout. |

He uses the sender ID () and the of the reply as a key to search his reply table, so that he knows the identity of the next agency (i.e. ), which enables him to forward the reply to . The unique number can help when checks his reply table.

For each agency , where , he searches his reply table for a correlative entry, when he receives a reply. The and in the entry should equal to the sender ID and the in the reply. The is the next agency whom is responsible for forwarding the reply to. also assign to the in the reply to help (must be ) search ’s reply table. The modified reply is shown in Figure 4.9.



Figure 4.9 the reply of the second agency

### The last agency

The last agency also searches for a reply table entry based on the reply, while he does not get the identity of the next agency, because he is the last one. The reply table entry of the last agency is shown in Table 6.

Table 6 reply table entries of the last agency

|  |  |
| --- | --- |
| Name of entries | value |
|  |  |
|  |  |
|  | void |
|  |  |
|  |  |
|  | The time when the AC timeout. |

The last agency finds that the is equal to , so that he notices that he is the last agency. He has no agency to be forwarded to, but forwards to the original requester. He uses his identity () and to generates a pseudonym . The function is a public pseudonym generating function that everyone in the network knows, including the original requesting .

When sends his query, he also gets the same pseudonym using the pseudonym generating function. He gets parameters from the appointment card (see Table 1). and are equal to the and in the appointment card. He uses that pseudonym as his identity before he gets the reply.

When users deliver the reply from the last agency, they are looking for a user whose identity is that pseudonym. Since the original requester is the only user who uses the pseudonym as his identity, gets the reply.

## Appointment number

The appointment number including and is significant information in the appointment card.

### Card appointment number

The card appointment number () is a number which is used to identify appointment cards generated by the same creator (see 4.10.1). In other words, if two appointment cards are generated by the same creator and they do not expire, their must be different. Therefore, when the first agency looks up his reply table (Table 4), he cannot find two entries which have the same pair of and .

### Agency appointment number

The agency appointment number () is a number used to identify appointment cards who have the same agency. In 4.7, agencies generate a new for appointment cards before they exchange those cards to others. In other words, an agency gives every appointment card passed on by him a unique , which helps the next agency search for the appointment card in his reply table. Since the is unique, an agency who is not the first one cannot find two entries which have the same pair of and , neither.

### Timeout

As mentioned in 4.5.5, agencies delete the entry of is deleted when an appointment card expires.

he picks an AC from his *ready-list*. He gives preference to the AC that has the earliest time-out. As a result, he sorts ACs in the *ready-list* in ascending order based on their remaining time and selects the first AC (e.g. *ac*) to send his query. That means the query is sent under the name of *ac*’s creator (i.e., *Cid*), and the location of the query is also *ac*’ location (i.e., *Clct*). Besides, *ac*’ *Capt* is also included in the query. The query can be transmitted to LBSPs by an arbitrary DTN routing protocol (e.g., Spray and Wait). When a LBSP receives the query, it provides services based on ac’s location and reply to ac’s creator (ac’s *Cid*). Users can also use a general DTN routing protocol to deliver the reply.

The (ready) AC is an authorization that anyone holds it can send a query to LBSPs using its creator’s identity (*Cid*) and location (*Clct*). When LBSPs receives the query, it replies to the creator (*Cid*) and attaches the *Capt* in reply, which enable the creator to determine who is the next one to transmit the reply.

Both *Capt* and *Aapt* are appointment numbers, which are unique random integers. When we mention “unique”, it means that there are not an identical (same value and type) existing appointment number generated by the same user in the network. Existing means that the CA which is relevant to the appointment number does not expires. For example, if a user (*Ui*) generates a *Aapt* “123456” and it does not expire, he cannot generate another *Aapt* “123456”, but a *Capt* “123456” is acceptable. He may also generate another *Aapt* “123456” after the previous one expires. It would not prevent another user (*Uj*, *j*≠*i*) generating an *Aapt* “123456” no matter whether *Ui* generates one or not.

The parameter *k* and the *EQL* also divides ACs into 2 states: dispensed and ready (see Table 3**错误!未找到引用源。**). Ready ACs can be used when their holder sends a query. Users still exchange ready ACs among their friends, but they do not add themselves in ACs’ *EQ* any more. Dispensed ones are not qualified to be used by current holder. Therefore, in order to make an AC useful, it should be exchanged for *k* times as fast as possible.

Table 7 AC states

|  |  |
| --- | --- |
| **State** | **Explanation** |
| *Dispensed* | *EQL* < *k* |
| *Ready* | *EQL* = *k* |

In order to manage ACs, each user maintains two lists: the *dispense-list* and the *ready-list*. It is obviously that *dispense-list* contains dispensed ACs and *ready-list* contains ready ones.

Table 8 AC storages

|  |  |
| --- | --- |
| *List Name* | *Explanation* |
| *ready-list* | It contains ready ACs. |
| *dispense-list* | It contains dispensed ACs. |

To avoid ACs generated by a creator simultaneously being dispensed to the same *k*th user, we employ the parameter *Seg* and *dc*. The *dc* is a counter which is initialized to a random integer in the interval [1, *Seg*] when a user received an AC. The AC should not be sent to the next friend until *dc* reaches 0, which enable users to split these ACs into about *Seg* batches.

## Exchange sending process

Each AC is unique and hold by only one user at one time. When a user exchanges an AC to a friend, he loses the control of that AC and cannot give it to another one. We discuss the process that users exchange ACs with their friends. The process that a user (*Ui*) exchanges ACs to his friend is shown in Figure 4.3 AC exchange process (send). When *Ui* encounters a friend (*Uj*), he would send 3 kinds of ACs to the friend, which include dispensed ACs, ready ACs and new ACs. Even though new ACs must be dispensed ACs, we discuss them separately. During the exchange process, *Ui* generates the specified number of ACs and selects a part of dispensed ACs and ready ACs, then he sends them to his friend. He also remembers whom he sends them to, which enables him to transmit replies to the friend.



Figure 4.10 AC exchange process (send)

### Generate new ACs

Each user (*Ui*) has an obligation to generate proper number of ACs for their friends, so that he guarantees that the number of ACs created by him in the network is larger than *DN* after he exchanges ACs with a friend (*Uj*). We assume that he must give *Uj* *Sg* new ACs to achieve that goal. *Ui* initializes entries for his new ACs, i.e., *Cid* (his identity), *Capt* (a new appointment number), *Clct* (his current location), *Aid* (A default value *Aiddefault*), *Aapt* (The same value as the *Cid*), *time-out*. The *EQ* in a new AC is initialized to empty, and *dc* is 0. We assume that the number of *Ui*’s ACs that are remained in the network is equal to *RNi*. The method of determining *RNi* is discussed in 0. Then .

### Pick dispensed ACs

*Ui* decreases *dc*s of all his dispensed ACs, so that he gets *Sd* ACs whose *dc* values are equal to zero. They should be sent to his friend. Since the parameter dc is initialed as an integer between 1 and *Seg* randomly, there are about  dispensed ACs eligible to be sent to the friend. That enables us to separate ACs which come from a common creator.

### Pick ready ACs

There are two ways for a user to provide ready ACs: 1. give his own ready ACs to friends directly; 2. dispensed ACs whose *EQL* values are equal to *k*-1. Since ready ACs are precious for every user, the strategy that a user shares ready ACs with his friends should save his own ready ACs. the number of ready ACs which would be sent to the friend is



, where *Sd*,*k*-1 denotes the number of dispensed ACs whose *EQL* values are equal to *k*-1. We assume that *Uj* has no ACs at all when they encounter. After their exchange, if *Uj* has more ready ACs than *Ui*, *Ui* must have all his old ready ACs with him; if *Ui* loses some of his ready ACs, *Uj* does not have more ready ACs than *Ui*. In another word, *Ui* should do his best to help his friend *Uj*, but *Ui* is not in a dilemma worse than his friends. That strategy makes users who sends more queries than friends supplement their AC consumption quickly, while protecting their friends’ profits. Then *Ui* picks *Sr* ready ACs randomly and sets their *dc*s to 0.

### Send ACs

An AC should be sent to *Uj*, if and only if its *dc* is equal to 0. Before it is sent to *Uj*, *Ui* adds its identity to the its *EQ*. Besides, *Ui* replaces its *Aid* and *Aapt* with his own identity and a new appointment number. The old *Aid* is recorded in his *agency-list*, so do both old and new *Aapt*s. The structure of the *agency-list* is shown in Table 5. When the AC comes from his *dispense-list*, he checks its *EQL*. If it is equal to k and the last entry of *EQ* is himself, he generates a unique pseudonym for the *AC* (*Psd*) and records the pseudonym in the *agency-list* (*Nxt*). Otherwise, he records the identity of his friend in *Nxt*. Since items which is relevant to an expired ACs would be deleted, each item in the *agency-list* is related to a distinct existing AC. Information in it can be used for statistics showing in the following sections.

Table 9 Agency\_list entries

|  |  |
| --- | --- |
| *List Name* | *Explanation* |
| *Aidold* | The previous agency identity. |
| *Aaptold* | The previous agency appointment number |
| *Aaptnew* | The current agency appointment number |
| *Nxt* | The identity of the next friend. |
| *time-out* | The time when the AC expires |

#### Get the number of ACs

*Ui* can get the number of existing ACs which is generated by him based on the information in his *agency-list*. He counts the items in the list whose *Aidold* value is equal to *Aiddefault*, that is the *RNi*.

#### Check duplicated appointment numbers

All existing appointment numbers generated by a user are recorded in his *agency-list*. If *Aidold* value is equal to *Aiddefault*, the *Aaptold* is a *Capt*; Otherwise, the *Aaptnew* is a *Aapt*. When the user generates a new appointment number, he must check the *agency-list* to guarantee it is unique.

## Exchange receiving process

The process that *Ui* receives ACs from his friend is shown in Figure 4.4 AC exchange process (receive). ACs are divided into two groups based on their *EQL* values, which are dispensed ACs and ready ACs. The *dc* value of a dispensed AC is assigned to a random integer in the interval [1, *Seg*], then the AC is moved into his *dispense-list*. Ready ACs are moved into his *ready-list*, and their dc values are set to -1.



Figure 4.11 AC exchange process (receive)

## Transmit replies

When a user receives a reply, he checks his *agency\_list*. If the sender identity is the LBSP, he searches for a record in his agency\_list, whose *Aidold* is *Aiddefault* and *Aaptold* is equal to the *Aapt* in the reply. If the sender identity is a user, he looks up for an item whose *Aidold* is equal to the sender identity and *Aaptold* is equal to the *Aapt* in the reply.

Based on the record he finds, he modifies information in the reply. The sender identity is set to his own identity; the destination is set to the user whose identity is *Nxt*; *Aapt* is assigned to *Aaptnew*. The item related to that AC can be deleted after the replacement, then he replies to the next friend (*Aidnew*).

After the original requester sends the query out, he claims himself as the one whose identity is *Psd*. That identity is equal to the *Nxt* in the *k*th user’s *agency\_list*. As a result, nobody knows the identity of the original requester. The process is shown in Figure 4.5.



Figure 4.12 Transmit reply

## Example

We assume there are four users in our social network: Alice, Bob, Charlie and David and there are 3 pairs of friends (i.e. Alice and Bob, Bob and Charlie, Charlie and David) in our example. We assign the system parameter *k* to 3, so that each Appointment Card must be exchanged for three times before it can be used. The system parameter *DN* is equal to 100, so that each user guarantees there are more than 100 ACs generated by him in the network.

When Alice encounters Bob, both of them generate 100 new ACs and give all the new ACs to the other one. Table 6 shows the number of ACs hold by Alice and Bob after the exchange where ACAlice,1 denotes the ACs generated by Alice and with *EQL* equal to 1.

Table 10 Alice and Bob exchange ACs

|  |  |  |
| --- | --- | --- |
| Users | Alice | Bob |
| Dispensed ACs | 100 ACBob, 1 | 100 ACAlice, 1 |

Then Bob encounters a friend Charlie who is a new user. Since Bob has already generated 100 ACs and none of them are used, he generates no AC for Charlie. But, Bob should give half of his dispensed ACs to Charlie while Charlie generates 100 new ACs and give them all to Bob.

Table 11 Bob and Charlie exchange ACs

|  |  |  |
| --- | --- | --- |
| Users | Bob | Charlie |
| Dispensed ACs | 100 ACCharlie, 1  50 ACAlice, 1 | 50 ACAlice, 2 |

When Charlie encounters David, Charlie gets 100 new ACs from David. David get 25 ACs which are generated by Alice from Charlie. The 25 ready-ACs hold by David can also be exchanged to other users, as shown in Table 8.

Table 12 Charlie and David exchange ACs

|  |  |  |
| --- | --- | --- |
| Users | Charlie | David |
| Dispensed ACs | 100 ACDavid, 1  25 ACAlice, 2 |  |
| Ready ACs |  | 25 ACAlice, 3 |

The route of one of the ready ACs (for example, the ACAlice, 3) after it is received by David is shown as follow. For convenience, we donate it by AC*x*. David gives a pseudonym (Say *p*) to AC*x* before he exchanges it to another user. We assume that David gives it to Elizabeth and Elizabeth gives it to Franklin, and Franklin would use the AC*x* to send the query. Franklin sends the query pretending he is Alice and receives the reply pretending he is *p.* More specifically, in the replying phase, LBSP replies to Alice and then the reply would be transmitted by Bob, Charlie and received by David. After that, David broadcast the reply to *p*, other users would ignore that reply while Franklin realized it is the reply of his query since he uses the AC with pseudonym *p.*

Table 13 The exchange history of ACx

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| User name | *Cid* | *Capt* | *Aid* | *Aapt* | *Pseudonym* |
| Alice | Alice | 233 | Alice | 557 | -- |
| Bob | Alice | 233 | Bob | 743 | -- |
| Charlie | Alice | 233 | Charlie | 691 | -- |
| David | Alice | 233 | David | 283 | *p* |
| Elizabeth | Alice | 233 | David | 283 | *p* |
| Franklin | Alice | 233 | -- | -- | -- |

When Franklin wants to send a query to a LBSP, he uses Alice’s identity and attaches the *Capt* (i.e. 233) in the query. Then he uses the pseudonym *p* as his identity when he receives a reply.

The LBSP replies to Alice, and the *Capt* (i.e. 233) is included in the message. When Alice receives the message, she checks her *agency\_list* and finds the entry whose *Aidold* is equal to *Aiddefault* and *Aaptold* is equal to 233. Based on the entry, she attaches the *Aaptnew* (i.e. 557) in the reply and sends it to *Nxt* (i.e. Bob). We collect entries which are related to the AC*x* in Table 10. Then Alice remove the item from her *agency\_list*. Also, the number of ACs generated by her decreases to 99 (=100-1), so that she should generate a new AC when she encounters a friend.

When Bob receives the reply from Alice, he also checks his *agency\_list* and learns that the reply is related to the AC*x* based on the sender identity (i.e. Alice) and the *Aapt* (i.e. 557). He modifies the *Aapt* to 743 in the reply message and relays it to Charlie. Charlie also follows the same process. Different from the previous 3 friends, David relays the message to a pseudonym *p*, because the *Nxt* value is equal to *p* in his *agency\_list*. Since Franklin is using the pseudonym *p* as his identity, he can get the reply message at last.

Table 14 Users' agency\_list

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| User Identity | *Aidold* | *Aaptold* | *Aaptnew* | *Nxt* |
| Alice | *Aiddefault* (LBSP) | 233 | 557 | Bob |
| Bob | Alice | 557 | 743 | Charlie |
| Charlie | Bob | 743 | 691 | David |
| David | Charlie | 691 | 283 | *p* |

Table 15 Important information in the reply messages

|  |  |  |  |
| --- | --- | --- | --- |
| Transmit | Sender | Receiver | *Aapt* |
| LBSP → Alice | LBSP | Alice | 233 |
| Alice → Bob | Alice | Bob | 557 |
| Bob → Charlie | Bob | Charlie | 743 |
| Charlie → David | Charlie | David | 691 |
| David → *p* | David | *p* | 283 |

In a nutshell, the *Aapt* in the reply is the key for each user (friend) to relay the reply message to the next one. After a user relays the reply message to the next one, he removes the entry from his *agency\_list*.

# Vertex reduce algorithm

Movement of network nodes is a significant factor for the performance of DTNs. To evaluate the performance of different DTN protocols, researchers presented different kinds of movement models, like in [2]. Since shortest path algorithm is an essential and time-cost part of creating many movement models, it is an important topic to optimize the time-cost for simulators calculating the all-pairs shortest path problem in the entire map.

The best known non-negative edge weight undirected map all-pairs shortest path algorithm [3] has a complexity , which is still expensive when *n* is huge. For most real-world city map, there are thousands of points in a single square kilometer. Since a tens square kilometers map is reasonable for a DTN protocol evaluation, a simulator must cut down the time-cost of the calculation of the all-pairs shortest path.

Real-world map developers often use short straight lines to present curves, like in [4], so that a several-meters curves may contain tens of points. It is obviously that we do not need to calculate shortest path for every one of these points. In this paper, we present an Vertex Reduce Algorithm (VRA) to reduce the number of points before the process of the shortest path algorithm. The basic idea is that VRA removes all points whose degrees are less than 3 from the map, while keeps the result of all-pairs shortest path algorithm correct.

The rest of this paper is organized as follows: 5.1 presents some basic lemmas and definitions. The process of VRA is described in 5.2 and 5.3.

## Ignorable vertex and reserved vertex

Vertices in the graph are considered as ignorable and reserved ones. In each iteration of VRA, we remove the ignorable vertex from the graph, while the reserved vertices compose a new graph.

Definition: the vertex whose degree is larger than 2 is the reserved vertex (R).

Definition: the vertex whose degree is smaller than or equal to 2 is called the ignorable vertex (G).

Definition: if two reserved vertices (e.g. ) are connected by a sequence of ignorable vertices (e.g. ), then the sequence from to (i.e., ) is a line-segment (LS).

A reserved vertex can belong to different line-segments, while an ignorable vertex belongs to a unique line-segment. Both the reserved vertex and the ignorable vertex are called vertices (V)

Definition: the shortest route between two vertices inside a line segment is called the inner shortest path (SPI), Let denote the inner shortest path between two vertices (i.e. ) inside a line segment.

Lemma1: If two vertices (i.e. , ; ) are in the same line-segment, the shortest path between them is

Since an all-pairs SPI has a complexity equal to or smaller than and the *n* here is much fewer than the number of points on the entire map, the time-cost for SPI is ignorable comparing to the time-cost of the entire map all-pairs shortest path calculation.

Lemma2: We assume that there are two different line-segments (i.e., and ). We pick a vertex from and a vertex from , then the shortest path between and is

where . Here and are in the same line-segment, so we can use lemma1 to calculate , so does . The is the only part we need to calculate using all-pair shortest path algorithms. We should notice that and could be the same vertex, but it does not make any difference to the lemma.

## Vertex reducing

VRA iteratively removes ignorable vertices from the graph until only reserved vertices remained. In each iteration, we remove all ignorable vertices from the graph but keep the edges. If there are more than one routes between a pair of reserved, we keep the shortest one and remove others.

### Remove ignorable vertices

Since the degree of ignorable vertices is no more than 2, an ignorable vertex has only 0, 1 or 2 neighbours. In the case of 0 or 1 neighbour, we simply delete the ignorable vertex; if it has two neighbours, we connect its two neighbours before it is removed, as shown in Figure 5.1. When we remove an ignorable vertex, the weight of the line which connects its two neighbours is equal to the sum of its recent two lines’ weights (i.e., ). After we remove all ignorable vertices in a line-segment, the two reserved vertices at the ends of the line-segment are connected with a line directly, whose weight is the sum of all the intermediate ones’ (i.e., ).



Figure 5.1 Remove ignorable vertices

### Tidy reserved vertices connections

After we remove all ignorable vertices, all the reserved vertices are connected directly. However, it is possible that there are several connections between a pair of reserved vertices. The shortest route between a pair of neighbour vertices makes other longer ones redundant obviously, so that we remove all routes except the shortest one, as shown in Figure 5.2. We assume that is the shortest line in their three connections, then and are removed.



Figure 5.2 Tidy reserved connections

### Iterations

The whole algorithm is shown in Algorithm 1. The input of the algorithm is the original entire map, which is called . In the *i*th iteration, VRA makes a copy of as . If , the is the output of the previous (i.e., ) iteration. VRA removes all ignorable vertices in the as described in 5.2.1 and remove all unnecessary routes between reserved vertices as described in 5.2.2. In other words, is smaller and smaller as *i* increases, because we always remove ignorable vertices from them. If is equal to , which means that the *i*th iteration makes no modification on the graph, the algorithm ends.

Algorithm 1 vertex reducing

**function** reduce(*Graphi*)

Copy *Graphi* to *Graphi*+1

**for** every ignorable vertex *Gu* in *Graphi*+1 **do** remove(*Gu*) (see 5.2.1)

**for** every pair of reserved vertices **do** remove redundant connections (see 5.2.2)

**end function**

**function** VRA(*Graph*0)

i=0

**while** i=0 or there are any differences between *Graph*i-1 and *Graph*i

**do**

**call** reduce(*Graph*i)

i=i+1

**end**

**end function**

## Vertex assembling

We assume that the vertex reducing process stops at the *i*th iteration. Then the is the input of an all-pairs shortest path algorithm. Let denote the shortest path from to in . We can infer the all-pairs shortest path of based on with lamme1 and lamme2. The algorithm ends when we get the result of . The algorithm is shown in Algorithm 52.

Algorithm 52

function

for all ignorable vertices in as

for all other vertices in as

if and are in the same line-segment then

Use lemma 1 to calculate their shortest path.

else

Use lemma 2 to calculate their shortest path.

endif

end

end

end function

function

while i > 0 do

call

end

end function

We start from the and assemble all intermediate graphs, until we get the result of . For any intermediate , we have already get the all-pairs shortest path result of its reserved vertices in the previous iteration (i.e. ). Then we calculate the shortest path from any ignorable vertices to all others. When the two vertices are in the same line-segment, we use lemma1 to calculate their shortest path. The lemma 1 includes 4 parts: , , and . Since we already get the result of in the previous iteration, we just need to calculate 3 parts, which is easy and has a small scale. If the two vertices are in different line-segments, we use lemma2 to calculate the shortest path. We also already get all the parts from the previous iteration so that we just need to deal with their parts.

# Finding nodes in a range

We assume that there are nodes in a planar map (size: ). Those nodes keep moving on the map slower than a speed threshold (). If the distance between two nodes and is smaller than a radius threshold (), they are connected. We want to calculate all-pairs connections for all nodes on the map periodically.

If we calculate these connections directly, the complexity is . We assume that and nodes are distributed on the entire map symmetrically, then the expected number of nodes which has a connection with a node () is . If we traverse all other nodes () for , it is a waste of time.

We propose an Nodes in the Range Algorithm (NRA) to optimize the calculation of all-pairs connections. we draw grids whose size is on the map. when we calculate connections of the node , only nodes in surrounding grids will be traversed.

## Static nodes

If nodes on the map are still, the radius threshold is the only factor of the selection of the grid size (). The length of the side of a grid should be equal to , as shown in Figure 6.1. For any node in the dark grid, all nodes that the distance between them is equal to or shorter than should be in the grey or dark grid. Therefore, if we want to get all connected nodes to node , we just simply need to traverse all nodes in the grey and dark grids instead of all nodes in the entire map. In this case, the expected number of nodes which we need to traverse is when we check connections for a node . If we calculate connections for all nodes, we simply need to get the distances of about pairs of nodes, instead of pairs.



Figure 6.1 grid size for still nodes

## Mobile nodes

The problem is a bit more complicated if nodes are moving. Assuming that we want to calculate all-pairs connections at , first, we draw grids at time , so that we know which grid is a node in at . Instead of , we want to get connections at . In this case, the length might not equal to the radius threshold .

Suppose that we have two nodes and , and their distance is at . Since the maximum speed of nodes is , their distance at satisfies

where . When , these two nodes are connected at , then must satisfy . If they are in the same or adjacent grids at , as shown in Figure 6.2. We have , so that : . In other words, there is no grid between two nodes at .



Figure 6.2 length of grid

## Complexities

The complexity of drawing grids is , while that of checking connections is , so the checking part is much more expensive than the drawing part. Since , the checking part achieve its best performance when , then its complexity is . That is equal to the complexity when nodes are still.

**REFERENCES**:

1. A. Lastname1, B. Lastname2, C. Lastname3 and D. Lastname4, “Article name,” in Publication name, vol. 13, issue 2, pp. 223–244, 2017.
2. K. Fall, "A Delay-Tolerant Network Architecture for Challenged Internets," in Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 27-34, 2003.
3. J. Schiller, A. Voisard, "Location-Based Services," Morgan Kaufmann, 2004.
4. M. Gruteser, D. Grunwald, “Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking,” in Proceedings of the 1st International Conference on Mobile Systems, Applications and Services, pp. 31-42, 2003
5. S. Zakhary, M. Radenkovic, "Utilizing social links for location privacy in opportunistic delay-tolerant networks," in Proceedings of International Conference on Communications (ICC), 2012.
6. S. Zakhary, M. Radenkovic, A. Benslimane, "The quest for location-privacy in opportunistic mobile social networks," in Proceedings of IWCMC, 2013.
7. S. Zakhary, M. Radenkovic, A. Benslimane, "Efficient Location Privacy-Aware Forwarding in Opportunistic Mobile Networks," IEEE Trans. on Vehicular Technology, Vol. 63, No. 2, pp. 893 – 906, 2014.
8. E. Daly, M. Haahr, “Social Network Analysis for Routing in Disconnected Delay-Tolerant MANETs,” in Proceedings of the 8th ACM International Symposium on Mobile Ad Hoc Networking and Computing, 2007.
9. M. Duckham, “Moving forward: Location privacy and location awareness,” in Proceedings of the 3rd ACM International Workshop on Security and Privacy in GIS and LBS, pp. 1-3, 2010.
10. T. Hashem, L. Kulik, “Safeguarding location privacy in wireless ad-hoc networks,” in Proceedings of UbiComp, pp. 372-390, 2007.
11. L. Sweeney, "k-Anonymity: A Model for Protecting Privacy," International Journal on Uncertainty,. Fuzziness and Knowledge-based Systems, Vol. 10, No. 5, pp. 557-570, 2002.
12. M. Mano, Y. Ishikawa, “Anonymizing User Location and Profile Information for Privacy-aware Mobile Services,” in Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Location Based Social Networks, pp 68-75, 2010.
13. T. Wang, L. Liu, “Privacy-aware mobile services over road networks,” in Proceedings of the VLDB Endowment, pp. 1042-1053, Vol. 2, 2009.
14. R. Lu, X. Lin, X. Shen, “SPRING: A Social-based Privacy-preserving Packet Forwarding Protocol for Vehicular Delay Tolerant Networks,” in Proceedings of IEEE INFOCOM, 2010.
15. J. Zhang, C. Chow, “REAL: A Reciprocal Protocol for Location Privacy in Wireless Sensor Networks,” IEEE Trans. on Dependable and Secure Computing, Vol. 12, No. 4, pp. 458 – 471, 2015.
16. P. Hui, J. Crowcroft, “How Small Labels create Big Improvements,” in Proceedings of PERCOM Workshop, pp. 65–70, 2007.
17. Diffie, W., Hellman, M., "New directions in cryptography," IEEE Trans. on Information Theory, Vol. 22, No. 6, pp. 644-654, 1976.
18. P. Jacquet, P. Muhlethaler, T. Clausen, “Optimized link state routing protocol for ad hoc networks,” in Multi Topic Conference, 2001.
19. R. Xiang, J. Neville, M. Rogati, “Modeling Relationship Strength in Online Social Networks,” in Proceedings of the 19th International Conference on World Wide Web, pp 981-990, April 26 - 30, 2010.
20. T. Spyropoulos, K. Psounis, C. S. Raghavendra, "Spray and Wait: An Efficient Routing Scheme for Intermittently Connected Mobile Networks," in Proceedings of the ACM SIGCOMM Workshop on Delay-tolerant Networking, pp. 252-259, 2005.
21. Working Day Movement Model
22. all pairs shortest path algorithm with expected running time O(n2logn)
23. https://www.openstreetmap.org/