

Paper Title*

*Note: Sub-titles are not captured in Xplore and should not be used

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

Abstract—Users face location-privacy risks when accessing Location-Based Services (LBSs) in an Opportunistic Mobile Social Networks (OMSNs). In order to protect the original requester’s identity and location, we propose a location privacy obfuscation protocols, called Appointment Card Protocol (ACP), utilizing social ties between users. To facilitate the obfuscation operations of queries, we introduce the concept called Appointment Card (AC). The original requesters can send their queries to the LBS directly using the information in the AC, ensuring that the original requester is not detected by the LBS. Also, a path for reply message is kept when the query is sent, to help reduce time for replying queries. Simulation results show that our protocol has a higher query success ratio than its counterparts.

Index Terms—obfuscation, encryption, location-privacy, opportunistic mobile social networks

I. 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] [21]. 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 requesters 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 requesters 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 requesters information with its own identity and location, which enables the LBS server to receive the query without any information

about the original requester. Also, the last friend records both the query and the identity of the original requester in his memory which enables him to forward replies back to 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 requesters 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.

II. 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 SLPD[4], Hybrid and Social-aware Location-Privacy in Opportunistic mobile social networks (HSLPO) [5] and Location Privacy-Aware Forwarding (LPAF) [6]. The obfuscation processes are performed by each separated node independently without any help from a third-party device. The SLPD is the basic version of the above three protocols, in which authors use two phases: the obfuscation phase and the free phase. A query from an original requester always starts in the obfuscation phase and passes through k friends. For example, the original requester sends the query to one of his friend in one hop, and then the friend forwards the query to another friend. We call these friends the agents. That process repeats for exactly k times. When the k th friend gets the query, he switches the query to the free phase and replaces the sender identity with his own identity, and then sends the query to the destination (e.g., LBS) using any DTN protocol. The disadvantage of the protocol is that it is hard to encounter a friend in the network, which can decrease the success ratio for the queries. 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. HSLPO tries to improve the delivery performance by using a stochastic model for location prediction. The dominant process is similar to that in SLPD, but an agent can forward the query to a user who is not a friend of the original requester if the user has more chance to deliver the query and a trust value is larger than a threshold. LPAF also attempts to improve SLPDs performance in a way that some ineligible users in SLPD are chosen as friends based on the additional requirements. The protocol Social-aware Location-privacy Protection (SLP) [20] also uses social tie to protect the privacy of the original requester and uses cloaking areas instead of the exact coordinates to enhance security, but the original requester still must take the risk that strangers might learn everything of the query.

III. SYSTEM MODEL

The network architecture consists of two main classes of entities: Users and Location-Based Service Providers (LBSPs). Users are mobile and communicate with others within a certain range, i.e., the communication range of their portable devices. For a given user, other users in the social network are either

strangers or his friends whom he can detect when they are in his communication range. Let $RS_{i,j}$ denote the relationship strength between user i and j . If $RS_{i,j}$ is larger than a specific threshold, FT_{min} , user j is considered a friend of i . LBSPs, which provide location-based services to the users are fixed and not part of the social network. We assume that the only information which is necessary for the LBSP is a location from the original requester, but the original requester should still give an identity to the LBSP so that the LBSP can reply to that identity.

We consider external attacker capable of eavesdropping on limited traffic in the network. We assume that the attacker can access the database of LBSPs, so that he can learn everything recorded in LBSP's memory, including user identities and locations. The attacker launches an inference attack on each user who uses the LBS in an attempt to learn user's private information based on location and context in the queries. Therefore, the key to protecting location-privacy is degrading the relationship between the user identity and the location provided by him so that the attacker can hardly infer the identity of the original requester by the known information.

We propose a protocol, called Appointment Card Protocol (ACP), to protect the identity and location-privacy of the original requester by providing other users' identity (agents), which can be any user in the network so that ACP can have a large anonymity set. The friends of the original requester separate the agents and the original requester so that the agents have no knowledge about the original requester.

IV. APPOINTMENT CARD PROTOCOL OVERVIEW

Our proposed ACP protects original requesters when they are served by LBSPs. A user (Agt_1) generates his own Appointment Cards (ACs) containing his own identity called Cid and a unique number called $Capt$ (a number generated by the creator of AC). The ACs are exchanged when two users encounter each other. When the original requester sends a query, he chooses an AC and sends the query using the identity Agt_1 of the first agent which is in the AC. The LBSP replies to Agt_1 when it receives the query. Agt_1 is the one who has generated the AC and the first agent of the AC. Agt_1 then forwards the reply to the next agent (Agt_2) who already has received AC from him, and so on until the reply reaches the last agent. The last agent is responsible for forwarding it to the original requester.

Figure 1 is an example of the execution of the ACP protocol. Explanations of the symbols in the figure are shown in Table I. These symbols and the figure are used throughout the chapter to help us describe the protocol. For simplicity, we will omit some superscripts and subscripts from these symbols in the following sections when there is no ambiguity. The whole process can be considered as the following parts: 1) exchanging cards among all users who are called agents (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 agents (i.e., 5, 6 and 7), and 5) relaying to the original requester (i.e., 8).

TABLE I
ACP SYMBOLS

Parameter	Meanings
U_ε	A user whose identity is ε .
AC_α^β	The β^{th} appointment card generated by a user α .
$AC_{\alpha,\gamma}^\beta$	AC_α^β is being forwarded by an agent γ .
$Query_\delta^{\alpha(\beta)}$	A query whose original requester is δ and using AC_α^β .
$Reply^{\alpha(\beta)}$	The reply of a query which uses AC_α^β .
$Reply_\gamma^{\alpha(\beta)}$	$Reply^{\alpha(\beta)}$ is being forwarded by an agent γ .
$Agt_i^{\alpha(\beta)}$	The i^{th} ($i \geq 1$) agent of AC_α^β .
$capt_\alpha^\beta$	The parameter $Capt$ in AC_α^β . see Table II
$aapt_\gamma^{\alpha(\beta)}$	The parameter $Aapt$ in AC_α^β , which is given by an agent γ . see Table II
AN	Both the $Capt$ and the $Aapt$ in an AC are called the Appointment Number.
NR_ε	The number of <i>ready</i> -ACs carried by U_ε

V. APPOINTMENT CARD

Since the original requester cannot use his own identity to communicate with the LBSP, he must use others' identity (Agt_1) to send queries, so that the LBSP can reply to the original requester through Agt_1 . Appointment cards make it possible for the agents to forward the reply to the original requester one by one. In other words, the appointment card indicates a path through which the original requestor can get its reply.

In Figure 1, the users a , b and c are the agents of the appointment card AC_a^β (i.e., $Agt_1^{\alpha(\beta)}$, $Agt_2^{\alpha(\beta)}$ and $Agt_3^{\alpha(\beta)}$). These agents are strangers, so the attackers can hardly infer c from the identity of a . At the same time, c is in the original requester d 's social tie (i.e., c is d 's friend or his friends' friend, so on), and he is the only one who knows how to reach d . Therefore, it is hard for attackers to infer the identity of d from the identity of a .

Notice that c receives a 's appointment card (i.e. AC_a^β) from a stranger b who knows the information of AC_a^β and the identity of the next agent c , so that it is unsafe for c to use that appointment card. In other words, the appointment card cannot be used until c exchanges it with another user (e.g., the user d) who trusts c . The appointment card is called a *ready appointment card* (*ready-AC*) after it leaves the last agent (i.e., the user c), or it is called the *distributing appointment card* (simply *distributing-AC*). It is obvious that *distributing-ACs* are transmitted among agents who can be strangers, while a user can only get *ready-ACs* from one of his friends.

To make users carry a similar number of *ready-ACs*, *ready-ACs* are also exchanged between friends, so that a user can get *ready-ACs* from his friends who have more *ready-ACs* than him. As a result, the last agent is not sure whether the user who gets the *ready-AC* from him is the original requester.

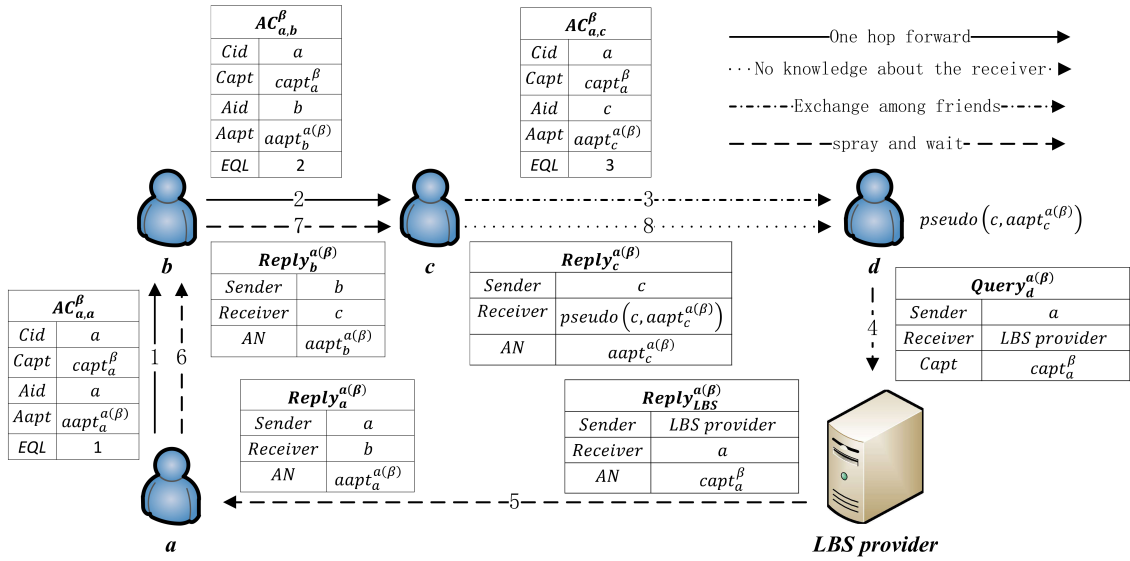


Fig. 1. Example of ACP Message Exchange

We introduce a pseudonym mechanism, which enables the last agent to forward the reply to an unknown original requester.

All users in the network are responsible for generating their respective ACs, and they are called the creators of their own ACs. The creator records his own identity (*Cid*) and a unique number (*Capt*) on his AC. When users exchange ACs, they modify *Aid* (the agent ID) and *Aapt* (the agent's appointment number) in the AC to enable the next agent to identify who is the predecessor. Entries of AC are shown in Table II.

TABLE II
APPOINTMENT CARD

Parameter	Meanings
<i>Cid</i>	The identity of the creator who generates the AC.
<i>Capt</i>	A unique number that distinguishes an AC from other ones generated by the same creator.
<i>Aid</i>	The identity of an agent who gives the AC to the recent holder. (The previous hop of the AC)
<i>Aapt</i>	A unique number that distinguishes an AC from other ones transmitted by the same agent.
<i>Timeout</i>	The time when the AC expires.
<i>EQ</i>	A queue (Exchange Queue) which records users who exchange the AC in order. Its length is <i>EQL</i> .

VI. AC LIFE CYCLE

The life cycle of an AC starts when it is generated by its creator. The first k (see Table III) agents add their identities into its Exchange Queue (*EQ*) (see Table II) before exchanging it, which increases the length (*EQL*) of the *EQ*. When the AC's *EQL* reaches k , it is eligible to be used in a query and is called a *ready* AC. When an AC is used in a query, it is marked as an *used* AC by the original requester who uses the AC. No matter what state (*distributing*, *ready* or *used*) an AC is in, it can expire, as shown in the Figure 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 once, it is switched to the *used* state. A *used* AC can also be used in other queries but cannot be given to anyone.

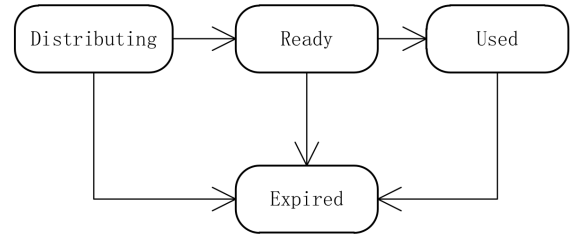


Fig. 2. AC's Life Cycle

VII. SYSTEM PARAMETERS

All the system parameters are shown in Table III.

TABLE III
IMPORTANT SYSTEM PARAMETERS

Parameter	Meanings
k	The obfuscation distance
m	The friend obfuscation distance
<i>Seg</i>	The distributing segment
<i>GP</i>	The generating period of appointment cards
<i>AT</i>	The timeout for appointment cards
τ	Avoiding time

A. Obfuscation Distance

The obfuscation distance k is the number of exchange before an AC is switched to the *ready* state. In other words, an AC

must be exchanged k times before it becomes a *ready-AC* which can be used by a user in queries.

As shown in Figure 3, an AC is exchanged along $Agt_1, Agt_2, \dots, Agt_k$. Since those agents are strangers, the only relationship between two adjacent agents is that they encounter each other somewhere. The relationship between Agt_1 and Agt_k becomes weaker when we increase k . In other words, attackers can hardly infer the identity of Agt_k when they only know the identity of Agt_1 , and their difficulty increases with the increase of parameter k . As a result, it is hard for attackers to infer the original requester, even though Agt_k is in the social tie of the original requester. Since the reply message must go through a series of agents, a long obfuscation distance also lengthens the path of the reply message. Therefore, a large k makes the original requester safer, while making it harder and longer for the reply to be delivered.

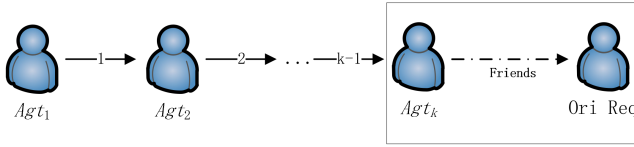


Fig. 3. Obfuscation Distance

B. Friends Obfuscation Distance

Since Agt_{k-1} is a stranger for Agt_k , it is possible that Agt_{k-1} is exactly the attacker. We assume that the attacker knows that Agt_k is in the social tie of the original requester. The attacker can assume that Agt_k is a close friend of the original requester, so the identity of Agt_k gives the attacker a good tip to infer the original requester. A solution to prevent the agent Agt_{k-1} from learning the original requester easily is that the last m ($1 \leq m \leq k$) agents are friends, as shown in Figure 4. Here, m is called *friends obfuscation distance*, and the last m agents are trusted agents.

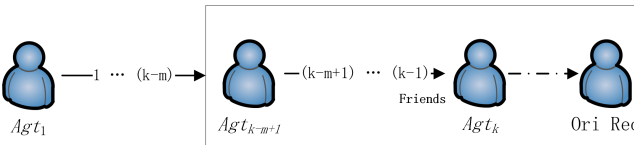


Fig. 4. Friends-Obfuscation Distance

It is true that Agt_i is a friend of Agt_{i+1} , $i > k - m$, but two nonadjacent trusted agents (e.g., Agt_i and Agt_{i+2}) might have a weak relationship. Since there are at least m trusted agents between the stranger Agt_{k-m} and the original requester, Agt_{k-m} can hardly infer the identity of the original requester based on information he learns. When $m = k$, all ACs must be exchanged between friends only. When $m = 1$, the last agent is the only one who is in social tie of the original requester, which is the case in our example.

However, friends encounter each other rarely, so having a large m increases the difficulty of distributing ACs. In this work, we assign m to 1.

C. Distributing Segment

We again use the example in Figure 1. U_a generates 3 ACs (i.e., AC_a^1, AC_a^2 and AC_a^3). It exchanges these ACs with U_b , then U_b exchanges them with U_c and so on. At last, all of them reach the original requester U_d . In this case, U_d has several ACs whose paths (the list of agents) are the same, so U_d has no choice but to choose one. In other words, since using these ACs results in the same reply path, U_d can simply choose an AC based on specific requirement it might have (e.g. the number of possible encounters of an agent).

If agents exchange ACs with different users, the above problem will not happen. We use the system parameter distributing segment, *Seg*, to avoid giving all ACs to the same user. Each user maintains *Seg* number of *Distributing AC Lists (DLs)*. It puts received *distributing-ACs* in one of those *DLs* randomly. If Agt_i exchanges ACs with another user, Agt_i selects one of his *DLs*, and only ACs in that *DL* will be exchanged with that user.

For example, we assign *Seg* = 2 so that each user has two *DLs* (i.e., *DL1* and *DL2*). When U_b receives $AC_{a,a}^1, AC_{a,a}^2$ and $AC_{a,a}^3$ from U_a , U_b may put $AC_{a,a}^1$ and $AC_{a,a}^2$ in its *DL1*, while $AC_{a,a}^3$ in its *DL2*. If U_b encounters U_c , U_b can give either $AC_{a,b}^1$ and $AC_{a,b}^2$ or only $AC_{a,b}^3$ to U_c . In this way, we separate ACs generated by the same creator.

D. Generating Period

Since ACs could expire, users must generate new ACs continuously. We use *GP* to denote the speed of generating new ACs per user, that is, each user generates a new AC every *GP* seconds.

E. AC Timeout

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

F. Avoiding Time

If an agent gets an AC at time t_i , he cannot get that AC again before $t_i + \tau$, where τ is parameter avoiding time. The reason for this is described in Section VIII-B.

VIII. PROTOCOL DETAILS

A. Generating Appointment Cards

Maintaining a certain number of ACs in the network is a prerequisite for users for sending queries. Considering that ACs can expire, users must generate ACs continuously. The user who generates an AC is called the creator of the AC, at the same time, he is also the first agent of the AC. ACs are created based on two principles: fairness and continuity.

ACs impose a burden on agents. In fact, agents are unlikely to benefit from relaying messages to the original requester because they need to allocate memory to save ACs' information,

and they consume energy to forward replies. To prevent users who generate more ACs than others from being overloaded, some fairness mechanism must be in place to ensure that all users generate a relatively equal number of ACs.

We assume that the AC timeout (i.e., AT) is 30 minutes, and every user generates 100 ACs at the beginning and generates no AC until the 30th minute. Since all agents remove the information of expired ACs from their memory, they can hardly deliver replies which use expired ACs. As a result, it is hard for a user to select an appropriate AC for his query at the 29th minute when all ACs only have 1 ($= 30 - 29$) more minute, because agents may not be able to forward the reply of his query even though the LBS receives the query successfully. Therefore, the generating strategy should be steady and sustainable.

In our protocol, each user generates a new AC every GP second. Since an AC has an AT -seconds timeout, there are $\delta = \frac{AT}{GP}$ appointment cards held by each user in the network. In other words, each user maintains about δ ACs which is generated by himself, and we meet the fairness criteria. Since users create ACs continuously, ACs have various timeouts so that it is likely for a user to pick an AC which does not expire in a long time (at least longer than his queries and replies).

B. Exchange Distributing Appointment Cards

Users exchange their *distributing*-ACs as frequently as possible, so that a *distributing*-AC can be switched to a *ready* one quickly. Still, there are some other conditions which should be satisfied when exchanging a *distributing*-AC.

We assume that two users, e.g., U_A and U_B , are walking together for some time. Also assume that U_A generates an AC (e.g., AC_A) at t_0 and exchanges it to U_B , then AC_A is given back to A and so on. The distributing phase of the AC only costs a few seconds, because U_A and U_B exchange AC_A time and again, but they are the only agents involved. The above scene defeats the purpose of exchanging ACs because it is easy for an attacker to infer the last agent who may be U_A or U_B from the first agent U_A .

We use a parameter *avoiding time*, τ , to optimize the agent selection strategy. If a user gets an *distributing*-AC at time t_i , he cannot get that *distributing*-AC again before $t_i + \tau$. For example, U_B receives a certain AC AC_A from U_A at t_0 , then U_B sends it to a user C (U_C) who can also send it to others. If a user carrying AC_A encounters U_B before $t_0 + \tau$, he cannot send AC_A to U_B . Therefore, if the parameter τ is larger than or equal to the parameter AT , there is no repeated agent in an AC's *EQ*. In other words, an AC never reaches an agent twice when τ is equal to AT . In this thesis, we assign τ to AT .

Besides, we should also avoid ACs having the same sequence of agents, as we mention in the distributing segment subsection. Now we propose the strategy of exchanging *distributing*-ACs.

Let us take a pair of users Alice and Bob as an example. If Alice encounters another user Bob, Bob tells Alice whether he trusts her (Bob view her as a friend). Alice picks one of her *distributing*-AC lists (e.g., $DL1$). Alice traverses all ACs

in $DL1$ and *distributing*-ACs which satisfy the following two conditions before exchanging them with Bob.

- 1) If the length of the AC's *EQ* (i.e., EQL) is not shorter than $k - m$, then Alice must be a friend of Bob.
- 2) Bob was not carrying the AC in the recent τ time interval.

When Alice sends a *distributing*-AC to Bob, she adds her identity and the current time to the AC's *EQ*. Besides, Alice must modify the AC's *Aid* to her own identity and its *Aapt* to a new one. She also records the information in Table IV as *relay-table* in her memory. We should notice that the first agent (i.e., the creator) does not have a $Aapt_{old}$, because he is precisely the one who has generated the AC, in which case the $Aapt_{old}$ indicates his own identity. When Bob gets those ACs, he puts each one of the received *distributing*-ACs to his DLs respectively and randomly. If an AC whose *EQ* is already equal to k , the AC must be switched to a *ready* one when Bob gets it. Bob puts *ready*-ACs on his *ready*-AC list instead of *distributing*-AC lists.

TABLE IV
RELAY TABLE ENTRIES

Parameter	Meanings
$Aapt_{old}$	The $Aapt$ generated by the previous agent
Aid_{old}	The identity of the previous agent
$Aapt_{new}$	The new $Aapt$ (generated by himself)
ID_{next}	The identity of the next agent
EQL	The length of the AC's <i>EQ</i> (should larger than 0)
AT	The time when the AC timeout.

C. Exchange Ready Appointment Cards

Users ask for *ready*-ACs only from their friends, which prevent strangers from learning the information of the users who are holding the *ready*-ACs. The strategy of exchanging *ready*-ACs should also meet some fairness criteria. That is, the number of each user's *ready*-ACs should be more or less equal.

Let us consider two friends Alice and Bob as an example. Alice has 20 ACs, while Bob has 10 ACs. When they encounter each other, it is reasonable that they both give half of their *ready*-ACs to the other. As a result, they both will have half of the total (15) ACs. However, that strategy does not work all the time as explained below.

The problem becomes more complex in the following condition. Alice is a friend of Bob, while Bob is not a friend of Alice. In other words, Bob trusts Alice, but Alice does not trust Bob. We assume that Alice is a trusted but suspicious girl so that many users trust her while she trusts few users while Bob is opposite. When users encounter Alice, they ask for *ready*-ACs from her, but Alice rarely asks for *ready*-ACs. Therefore, Alice carries few *ready*-ACs, while Bob carries many *ready*-ACs. When Alice and Bob encounter each other, it is Bob who is asking for *ready*-ACs instead of Alice. To make the strategy fair and efficient, users must compare the number of

their own *ready*-ACs and the number of *ready*-ACs carried by the other user when two users are exchanging *ready*-ACs.

When Bob encounters Alice, Bob tells Alice the number of his *ready*-ACs (NR_{Bob}) and whether he wants Alice's *ready*-ACs (if he trusts Alice). If Alice learns that Bob needs her *ready*-ACs, she compares the number of her *ready*-ACs (NR_{Alice}) and NR_{Bob} . If $NR_{Bob} \geq NR_{Alice}$, Alice gives no *ready*-AC to Bob; otherwise, she gives $\frac{NR_{Alice} - NR_{Bob}}{2}$ to Bob. In this way, *ready*-ACs do not concentrate in a group of users who trust many users.

The process of exchanging *ready*-ACs is much more straightforward than that for *distributing*-ACs. Users do not modify any information in the *ready*-ACs including *Aapt* and *Aid* so that *ready*-ACs do not change after they leave the last (k^{th}) agent.

The algorithm of exchanging ACs when a user U_i encounters another user U_j is shown in Algorithm 1. The two users tell the other whether they are friends at the very beginning of their encounter. After they both know their relationship, they start to exchange their *distributing*-ACs. When they both finish receiving the *distributing*-ACs from the other, they continue to exchange their *ready*-ACs at the same time. The process ends when they finish exchanging their *ready*-ACs (or they do not need to exchange *ready*-ACs).

D. Sending Queries

An original requester must send his queries to the LBSP while avoiding the LBSP from learning the identity of him. Besides, the original requester can still get the reply message from the LBSP. The basic idea is that the original requester sends his query using another user's identity to the LBSP so that LBSP can reply to that user who is responsible for forwarding the reply to the original requester. That user is the first agent (Agt_1) of the AC which is used by the queries of the original requester.

To enable LBSP to reply to Agt_1 , the original requester's query includes a sender identity Agt_1 , which is equal to the *Cid* in the AC. Since Agt_1 needs a *Capt* to identify the AC used by the query (and the reply), the *Capt* in the AC is also in the query. The network can deliver that query to the destination LBSP efficiently with any DTN protocols.

We use the example in Figure 1. The original requester U_d has an AC (i.e., $AC_{a,c}^\beta$) whose creator is U_a . When U_d uses it to send his query $Query_d^{a(\beta)}$, the sender identity is a , and the *Capt* of the query is the *Capt* of $AC_{a,c}^\beta$, as shown in Figure 5.

$AC_{a,c}^\beta$		$Query_d^{a(\beta)}$	
<i>Cid</i>	a	<i>Sender</i>	a
<i>Capt</i>	$capt_a^\beta$	<i>Receiver</i>	LBS provider
<i>Aid</i>	c	<i>Capt</i>	$capt_a^\beta$
<i>Aapt</i>	$aapt_c^{a(\beta)}$		
<i>EQL</i>	3		

Fig. 5. Constitute Query

Algorithm 1 Algorithm for exchanging ACs

```

1: procedure ENCOUNTER( $U_j$ )
2:   if  $U_i$  trusts  $U_j$  then
3:      $U_i$  tells  $U_j$  that  $U_j$  is viewed as a friend.
4:   else
5:      $U_i$  tells  $U_j$  that  $U_j$  is not viewed as a friend.
6:   end if
7:   Wait for  $U_j$  to tell  $U_i$  whether  $U_i$  is viewed as a friend.
8:    $U_i$  chooses a distributing list ( $DL_\omega$ ) randomly.
9:   for each  $AC_u$  in  $DL_\omega$  do
10:    if  $U_j$  was carrying  $AC_u$  in the recent  $\tau$  time then
11:       $AC_u$  cannot be sent to  $U_j$ 
12:      continue
13:    end if
14:    if the EQL of  $AC_u$  is equal to or larger than  $k - m$ 
15:      then
16:        if  $U_i$  is not trusted by  $U_j$  then
17:           $AC_u$  cannot be sent to  $U_j$ 
18:          continue
19:        end if
20:        Send  $AC_u$  to  $U_j$ 
21:      end for
22:      Tell  $U_j$  that all distributing-ACs are sent.
23:      Receive all distributing-ACs from  $U_j$ 
24:      if  $U_i$  trusts  $U_j$  then
25:         $U_i$  tells  $U_j$   $NR_i$ .
26:      end if
27:      if  $U_i$  is trusted by  $U_j$  then
28:        Wait for  $U_j$  to tell  $U_i$   $NR_j$ .
29:        if  $NR_j \geq NR_i$  then
30:          send no ready-AC to  $U_j$ 
31:        else
32:          send  $\frac{NR_i - NR_j}{2}$  number of ready-ACs to  $U_j$ 
33:        end if
34:        Tell  $U_j$  all ready-ACs are sent.
35:      end if
36:      if  $U_i$  trusts  $U_j$  then
37:        Wait for ready-ACs from  $U_j$ .
38:      end if
39: end procedure

```

The AC is marked as used when the query is ready to be sent so that the AC cannot be exchanged to other users. The original requester also uses a pseudonym to receive the reply, which is described later. The algorithm that a user U_i sends a query is shown in Algorithm 2.

E. Sending Replies

1) *The LBSP part:* When the LBSP received the query, it learns that the sender's identity is the first agent instead of the original requester, which protects the location privacy of the original requester. The LBSP reply to the first agent using the information in the query.

Algorithm 2 Algorithm for Sending Queries

- 1: **procedure** SENDQUERY
- 2: Choose an *ready*-AC, say $AC_{\alpha,f}^\beta$.
- 3: Assign the sender identity of the query to U_α
- 4: Assign the receiver identity to the LBS
- 5: Assign the *AN* to the $capt_\alpha^\beta$
- 6: Get a pseudonym $pseudo(U_f, aapt_f^{\alpha(\beta)})$
- 7: Use the pseudonym as U_i 's identity.
- 8: Send the query to the LBS.
- 9: **end procedure**

In Figure 6, the LBS provider reply to the sender U_a , when it receives $Query_d^{a(\beta)}$. The reply also includes the *Capt* of the query, which enables U_a to identify the AC used in the query and the reply.

$Query_d^{a(\beta)}$		$Reply_{LBS}^{a(\beta)}$	
Sender	a	Sender	LBS provider
Receiver	LBS provider	Receiver	a
Capt	$capt_a^\beta$	AN	$capt_a^\beta$

Fig. 6. Constitute Replies

F. The First Agent

When the first agent (i.e., Ag_{t1}) gets the reply from the LBSP, he learns the *AN* in the reply. He searches his *reply-table* to get the information of the AC used in the query and reply. If the AC does not expire, there must be a corresponding entry in his *reply-table*, as shown in Table V. Ag_{t1} gets an entry where the Aid_{old} and $Aapt_{old}$ are equal to his identity and the *AN* of the reply. As a result, he learns the identity of the next agent (i.e., Ag_{t2}) from the ID_{next} of the entry, so Ag_{t1} forwards the reply to Ag_{t2} . The *AN* of the reply is replaced with the $Aapt_{new}$ in the entry by Ag_{t1} , which enables Ag_{t2} to identify the AC in Ag_{t2} 's *reply-table*.

TABLE V
REPLY TABLE ENTRIES OF THE FIRST AGENT

Name of Entries	Value
Aid_{old}	The user's own identity
$Aapt_{old}$	The <i>Capt</i> of the AC used in the reply (query)
ID_{next}	The identity of the second agent
$Aapt_{new}$	The <i>Aapt</i> given to the second agent by the user.
<i>EQL</i>	1
<i>AT</i>	The time when the AC timeout.

For the example in Figure 1, the first agent is U_a . When he receives $Reply_{LBS}^{a(\beta)}$, he learns that it is a reply from the LBS and the *Capt* of the AC is $capt_a^\beta$. He searches his *reply-table* for an entry whose Aid_{old} is equal to a and $Aapt_{old}$ is equal to $capt_a^\beta$. As shown in Figure 7, the ID_{next} in the entry is equal to b so he modifies the receiver of the reply message to

U_b . U_a also modifies the *AN* in the reply message to $aapt_a^{a(\beta)}$ which is exactly equal to the $Aapt_{new}$ in his *reply-table* entry, which enables U_b identifies the AC in his *reply-table*.

$Reply_{LBS}^{a(\beta)}$		Reply table entry		$Reply_a^{a(\beta)}$	
Sender	LBS provider	Aid_{old}	a	Sender	a
Receiver	a	$Aapt_{old}$	$capt_a^\beta$	Receiver	b
AN	$capt_a^\beta$	ID_{next}	b	AN	$aapt_a^{a(\beta)}$
		$Aapt_{new}$	$aapt_a^{a(\beta)}$		
			

Fig. 7. The Reply of The First Agent

G. Intermediate Agents

The process of forwarding replies in the intermediate agents (the second to the $(k-1)^{th}$ one) is similar to that of the first agent. We take the second agent as an example. When the second agent receives the reply forwarded by the first one, he learns the sender's identity (i.e., the first agent) and the *AN* from the reply. In his *reply-table* shown in Table VI, there must be an entry which is related to the AC which is used in the reply (query), because he gets the AC from the previous agent. More specifically, his *reply-table* must include an entry whose Aid_{old} is equal to the previous agent's identity and the $Aapt_{old}$ is the value of *AN* in the gotten reply.

TABLE VI
REPLY TABLE ENTRIES OF THE SECOND AGENT

Name of Entries	Value
Aid_{old}	The identity of the previous agent (i.e. the first agent)
$Aapt_{old}$	The <i>Aapt</i> given by the previous agent.
ID_{next}	The identity of the next agent (e.g. the third agent)
$Aapt_{new}$	The <i>Aapt</i> given to the next agent by the user.
<i>EQL</i>	2
<i>AT</i>	The time when the AC timeout.

For the example in Figure 1, the second agent is U_b . When he receives $Reply_a^{a(\beta)}$, he learns that it is U_a who forwards the reply message $Reply_a^{a(\beta)}$ and the *Aapt* of the AC is $aapt_a^{a(\beta)}$. He searches his *reply-table* for an entry whose Aid_{old} is equal to a and $Aapt_{old}$ is equal to $aapt_a^{a(\beta)}$. As shown in Figure 8, since the ID_{next} in the entry is equal to c , he modifies the receiver of the reply message to U_c . U_b also modifies the *AN* in the reply message to $aapt_b^{a(\beta)}$ which enables U_c identifies the AC in his *reply-table*, because the $Aapt_{new}$ in the entry is equal to $aapt_b^{a(\beta)}$.

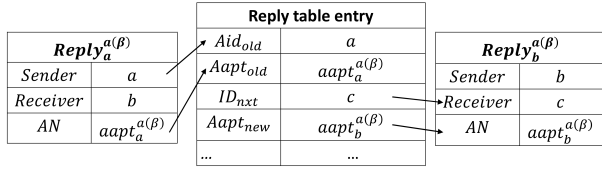


Fig. 8. The Reply of The Second Agent

For each agent Agt_i , where $2 \leq i \leq k-1$, he searches his *reply-table* for a correlative entry, when he receives a reply. The Aid_{old} and $Aapt_{old}$ in the entry should be equal to the sender's identity and the AN in the reply. The identity of the next agent Agt_{i+1} is ID_{next} . Agt_i also assign $Aapt_{new}$ to the AN in the reply to help Agt_{i+1} search Agt_{i+1} 's *reply-table*.

H. The Last Agent

The last agent also searches for a *reply-table* entry based on the reply, while he cannot get the identity of the next agent. The *reply-table* entry of the last agent is shown in Table VII.

TABLE VII
REPLY TABLE ENTRIES OF THE LAST AGENT

Name of Entries	Value
Aid_{old}	The identity of the previous agent (i.e., the second agent)
$Aapt_{old}$	The $Aapt$ given by the previous agent.
ID_{next}	VOID
$Aapt_{new}$	The $Aapt$ given to the original requester.
EQL	k
AT	The time when the AC timeout.

The last agent uses the same way to look up an entry in his *reply-table* as the previous agents. When he finds that the EQL is equal to k , he notices that he is the last agent. Then he is responsible for forwarding the reply to the original requester instead of another agent. He uses his identity and $Aapt_{new}$ in his *reply-table* entry to generates a pseudonym

$$psd_{Agt_k}^{a(\beta)} = pseudo(Agt_k, Aapt_{new}) \quad (1)$$

, where the function $pseudo(id, Aapt)$ is a public pseudonym generating function which everyone in the network knows, including the original requester.

When the original requester sends his query, he also gets the same pseudonym $psd_{Agt_k}^{a(\beta)}$ using the pseudonym generating function. Note that he can get parameters from the AC. Agt_k and $Aapt_{new}$ are equal to the Aid and $Aapt$ in the AC. He uses that pseudonym as his identity before he gets the reply.

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

For the example in Figure 1, the last agent is U_c . When he receives $Reply_b^{a(\beta)}$, he learns that it is U_b who forwards the reply message $Reply_b^{a(\beta)}$ and the $Aapt$ of the AC is $aapt_b^{a(\beta)}$.

He searches his *reply-table* for an entry whose Aid_{old} is equal to b and $Aapt_{old}$ is equal to $aapt_b^{a(\beta)}$. Since the EQL in the entry is equal to 3 (i.e. k), he recognizes that he is the last agent. U_c calculates the pseudonym $psd_c^{a(\beta)} = pseudo(c, aapt_c^{a(\beta)})$ then forward the reply to $psd_c^{a(\beta)}$. The original requester d gets the identity c and the $aapt_c^{a(\beta)}$ from the AC he uses, so that he uses the pseudonym $psd_c^{a(\beta)}$ as his identity. As a result, U_d get the reply from U_c . The AN of the reply is also assigned to $aapt_c^{a(\beta)}$ to avoid identical pseudonyms. The process is shown in Figure 9.

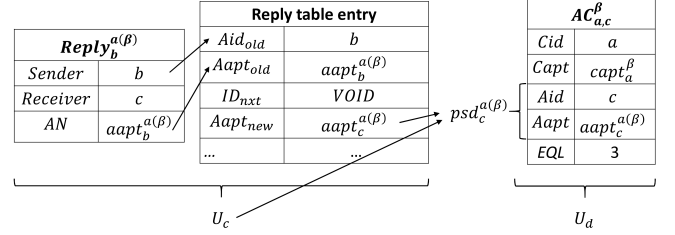


Fig. 9. The Reply of the Last Agent

The algorithm of forwarding replies when an agent U_i gets a reply message from U_{prev} is shown in Algorithm 3. U_i learns the AN and the identity of the sender from the reply message $Reply_{U_{prev}}^{a(\beta)}$. Based on this information, U_i checks his *reply-table* to find out where the reply should be forwarded to, which is ID_{next} . After U_i updates the AN of the reply message, he forwards the reply message to ID_{next} .

Algorithm 3 Algorithm For Forwarding Replies

```

1: procedure RECEIVED( $Reply_{U_{prev}}^{a(\beta)}$ )
2:   Get the  $AN$  from  $Reply_{U_{prev}}^{a(\beta)}$ 
3:   if  $U_{prev}$  is the LBS then
4:     search  $U_i$ 's reply-table to find an entry  $E_i^{a(\beta)}$ 
5:     whose  $Aapt_{old} = AN$  and  $Aid_{old} = U_i$ 
6:   else
7:     search  $U_i$ 's reply-table to find an entry  $E_i^{a(\beta)}$ 
8:     whose  $Aapt_{old} = AN$  and  $Aid_{old} = U_{prev}$ 
9:   end if
10:  Get the  $EQL$ ,  $Aapt_{new}$  and  $ID_{next}$  from  $E_i^{a(\beta)}$ 
11:  Assign the  $AN$  of  $Reply_{U_{prev}}^{a(\beta)}$  to  $Aapt_{new}$ .
12:  Assign the sender identity to  $U_i$ .
13:  if  $EQL = k$  then
14:     $ID_{next} \leftarrow pseudo(U_i, Aapt_{new})$ 
15:  end if
16:  Forward the reply to  $ID_{next}$ 
17: end procedure

```

IX. APPOINTMENT NUMBER

The Appointment Number (AN) including $Capt$ and $Aapt$ is significant information in the AC. We explain the rules of generating them in detail and talk about the effect of ACs' timeout mechanism in this section.

A. Creator Appointment Number

The Creator Appointment Number ($Capt$) is a number which is used to identify ACs generated by the same creator. In other words, if two ACs are generated by the same creator, and they do not expire, their $Capt$ must be different. Therefore, the first agent (i.e., the creator) cannot find two entries which have the same $Aapt_{old}$ in his *reply-table*, if their Aid_{old} are both his own identity.

B. Agent Appointment Number

The Agent appointment number ($Aapt$) is a number used to identify appointment cards which have the same agent. Agents generate a new $Aapt$ s for appointment cards before they exchange those cards to others. In other words, an agent gives any appointment card passing on by him a unique $Aapt$, which helps the next agent identify the appointment card in his *reply-table*. Since the $Aapt$ is unique, an agent who is not the first one cannot find two entries which have the same pair of Aid_{old} and $Aapt_{old}$, neither.

C. Timeout

Agents delete the entries, which contains the information of expired ACs, from his *reply-table*. Consequently, agents cannot forward replies using ACs which expire. That is the reason why an original requester must choose an AC which expires after his query and reply timeout. However, the timeout mechanism is still necessary for the protocol.

Since users are moving, the distance between agents and the original requester might be too large after a long time. As a result, it is hard for agents to forward the reply messages back to the original requester so that the original requester takes a higher risk when he uses an AC which exists for a long time. Then this kind of ACs might not be used after a period, while it cost agents a few memories to save the ACs' information in their *reply-table*. Therefore, all users remove the information of expired ACs to save their memories.

We should also notice that an unexpired AC and an expired one might have the same $Capt$ or $Aapt$. Because no agent keeps the record of the expired AC, the duplication cannot confuse agents.

X. EXPERIMENT

We used the map of Helsinki in our simulator to evaluate the performance of the proposed ACP. We also compare it to Binary Spray and Wait (BSW) protocol [?], distributed social based location privacy protocol (SLPD) [?] and our Multi-Hop Location-Privacy Protection (MHLPP). We simulated the continuous movement of users along streets on the map with one LBSP, fixed at a random location on the map.

For each user, we associate a random social value between 0 - 100%, each corresponding to other users. Since each social value is assigned with equal probability, we can compute the expected number of friends of a user. If a user whose social value is larger than 85% is called a friend and there are n users in the network, there are $n \times (1 - 85\%)$ friends.

The Shortest Path Map-Based Movement (SPMBM) [?] is used in our experiment. For each experiment, we give the simulator a random seed so that it can generate a pseudo-random number based on the seed. Therefore, all the factors including users' speed and locations are the same if two experiments have the same random seed. All four protocols are tested using the same set of random seeds.

Before each experiment, the simulator runs for 800 seconds (simulator time). Then we pick 100 users out of 126 users randomly, and each of them sends a query to the LBSP. Tests last for about 20 minutes (simulator time).

A. Average Query Success Ratio

The query success ratio is the percentage of delivered queries among some attempts. Since users sending 100 queries in each experiment, if s queries are delivered to the LBSP at time t , the query success ratio of time t is $s\%$.

As shown in Figure 10, we compare the average query success ratio of the four protocols with 5 kinds of communication radius (10, 30, 50, 70 and 90 meters). We observe that the ACP and the BSW get a high query success ratio, while the MHLPP and the SLPD are lower than the previous two protocols. The BSW is the highest one because it is a no-privacy protocol. The ACP is just a little lower than BSW, because the query delivery process of the ACP is almost the same as that of BSW. Since users of ACP must wait for available appointment cards, they cost more time to initial their queries. However, the ACP and the BSW are at the same level, comparing to the other two protocols. The MHLPP and the SLPD need to find friends to obfuscate their queries, which baffles their delivery process.

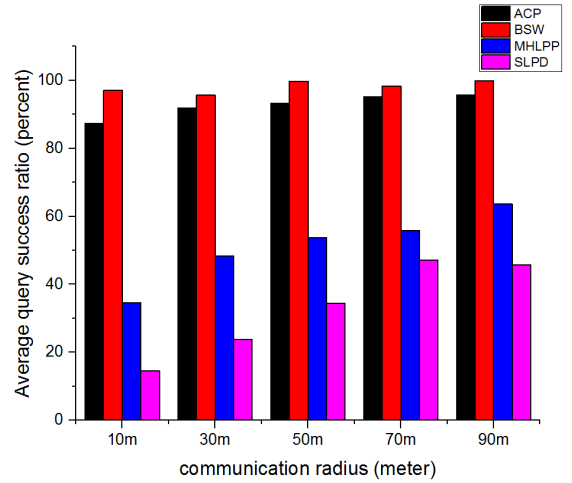


Fig. 10. Average Query Success Ratio (10 minutes)

The experiment results when we test 20 minutes is shown in Figure 11. Comparing to the average query success ratio at 10 minutes, the MHLPP and the SLPD achieve much higher success ratio after 20 minutes than at 10 minutes. That is because it cost them so much time in their obfuscation phases when they need to find friends. In fact, some of the queries of

the MHLPP and the SLPD still do not finish their obfuscation phase at 20 minutes.

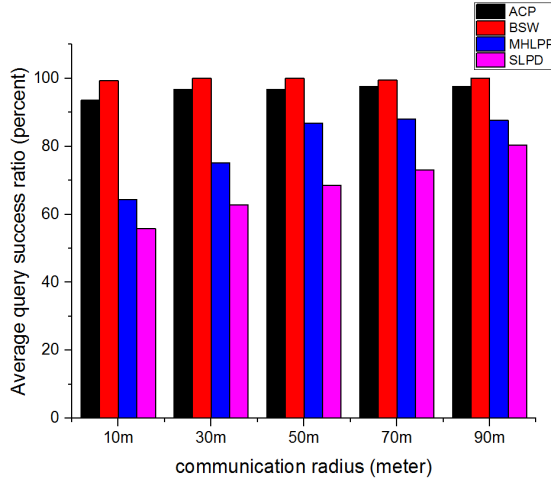


Fig. 11. Average Query Success Ratio (20 minutes)

The communication radius can influence the success ratio. In most of the cases, the success ratio rises when we increase the communication radius, and its influence is especially evident under 50 meters. A large communication radius makes it easy for users to encounter others, which is good for them to forward queries. However, a user who is so far away from the destination does not want the intermediates of his query encounters many users nearby. Because all users who carry copies of that query are near the sender instead of the destination, which decreases their query success ratio. Therefore, when the communication radius reaches 70 meters, the success ratios almost stay at the same level.

In Figure 12, we observe that the ACP and the BSW have better convergence speed than the MHLPP and the SLPD. In other words, the former two protocols have a faster speed to approach the 100% query success ratio than the latter two. At the very beginning, the ACP even has a little higher success ratio than the BSW. Because the ACP users need ready appointment card, and most of the users get their first ready appointment cards at places where there are many users. Therefore, users rarely generate query near the edges of the map at the beginning, which facilitates their queries delivery process. For example, if a user generates his query at the edge of the map, the copies of this query might be sent to users who are also at the edge, and it takes more time for them to deliver the query. If the user does not generate his query until he arrives a place nearer the center, the copies of this query will be carried by the users in the center with higher probability.

B. Average Reply Success Ratio

When the LBSP receives a query, it sends a reply to the requester. If the reply arrives the original requester before the test ends, we view it as a success; otherwise, the reply is failed. There are two reasons for the failure of replies: 1) the query

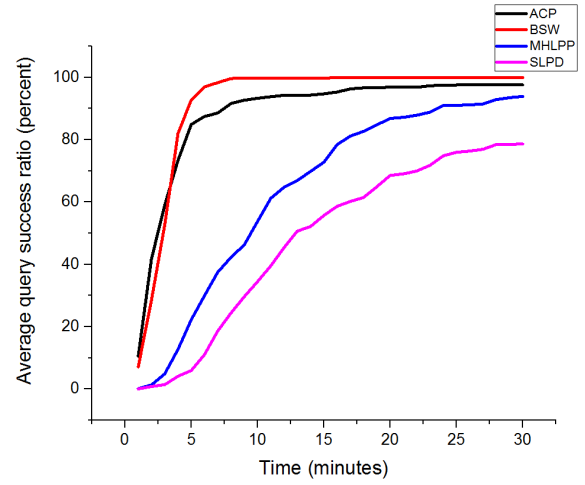


Fig. 12. Average Query Success Ratio With 50-Meters Communication Ratio

is not delivered to the LBSP successfully; 2) the query costs too much time so that the reply has no time to be delivered; 3) The route of the reply is too long. Since there are 100 queries in each experiment, the number of replies should be equal to 100.

In Figure 13, the BSW has a significant and reasonable higher success ratio than all other protocols, because it is a no-privacy protocol. The ACP is higher than the MHLPP and the SLPD, but its advantage is not as large as that in the query process. In fact, the reply process of the MHLPP and the SLPD are simpler than that of the ACP, but the ACP saves so much time in its query process that it earns a better reply success ratio than the other two.

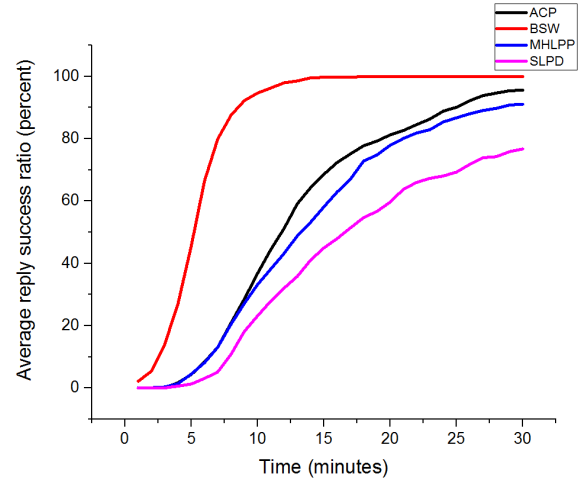


Fig. 13. Average Reply Success Ratio With 50-Meters Communication Ratio

C. Total Number of Query Relays

The query delivery processes of all the four protocols use the BSW protocol, and the BSW makes copies for queries and gives half of the copies to any users it encounters. That is

a significant cost for the network, so we use the number of query relaying (QR) to evaluate that cost. The QR is initialized as zero at the beginning of the test. When a user relay a (or several) copies of a query, we increase QR by 1. For example, in the SLPD, there are two phases: the obfuscation phase and the free phase. In the obfuscation phase, a query is forwarded among one-hop friends for k times. After that, it is forwarded by the BSW protocol. The BSW protocol makes c copies of the query and gives half of the copies to any encountering users. Then QR should be about $k + c$. Since a user gives all its copies to the destination if he encounters the destination at once, the QR may be smaller. The smaller that number is, the smaller cost of the network is. Since there are 100 queries, we divide QR by 100 to get an average value.

In Figure 14, we compare QR with four protocols. We observe that all the four protocols are at a similar level, the BSW and the ACP is a little lower than the other two. For the ACP and the BSW, they deliver queries so fast that users who carry more than one copies give all their copies to the destination before they send these copies to different users separately. As a result, many copies have no chance to be forwarded, which decreases the cost. While the MHLPP and the SLPD have obfuscation phases, the queries start to be delivered freely (in a BSW way) at a random place where might be so far away from the destination, so that almost all copies can be forwarded respectively.

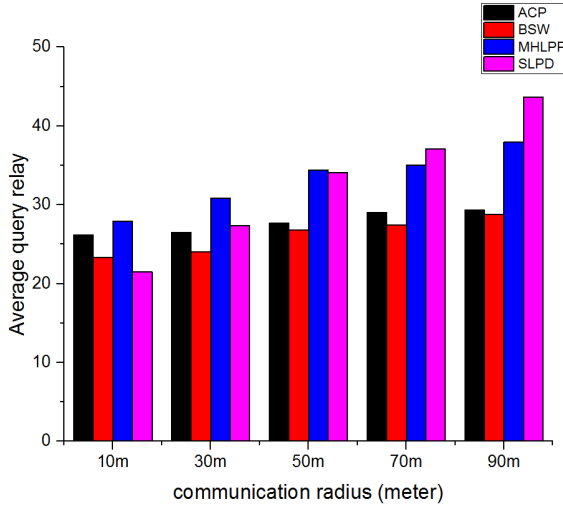


Fig. 14. Average Total Number of Forwarding Queries At 20 Minutes

The communication radius affects the total number of the forwarding queries, especially for the MHLPP and the SLPD. Those two protocols can finish their obfuscation phase more quickly which a larger communication radius, so that more queries can be forwarded freely (in the BSW way), which makes their QR larger.

D. Memory Cost

We count the number of queries carried by each user to evaluate the memory cost of the four protocols. Several copies of a query are counted for only once.

In the Figure 15, we compare the number of queries per user with the four protocols at 20 minutes. We observe that the BSW is the highest in most of the cases and the ACP always stays at a similar level as the BSW. The data of other two protocols (the MHLPP and the SLPD) increase as the communication radius. The MHLPP even exceeds the BSW when the communication radius is 90 meters. The reason is that quite a number of the BSW and the ACP users forward their copies to the destination so that there is no copy with them at 20 minutes, while the rest of them cannot forward their copies to the destination even given a large communication radius. At the same time, the number of the other two protocol's free phase queries is significantly influenced by the communication radius. The more queries are in the free phase; the more copies are in the network.

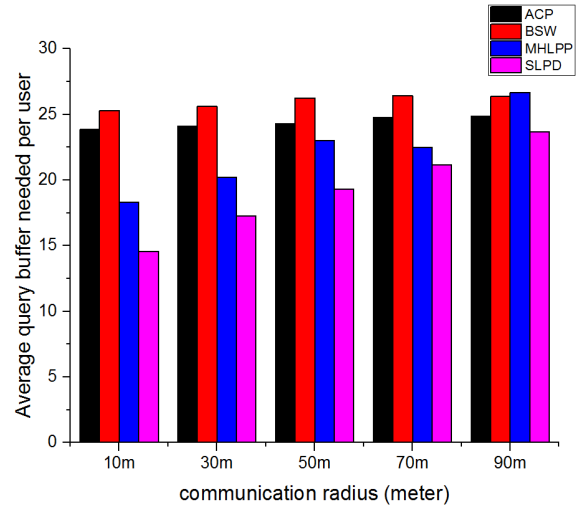


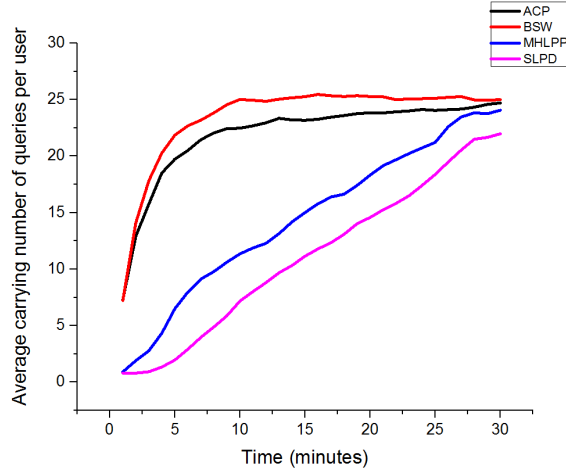
Fig. 15. Average Query Buffer Needed At 20 Minutes

The Figure 16 shows the average number of queries which are carrying by users when the communication radius is 10, 50 and 90 meters. The curves of the ACP and the BSW rise sharply at the beginning and then become flat, while those of the MHLPP and the SLPD rise smoothly and continuously.

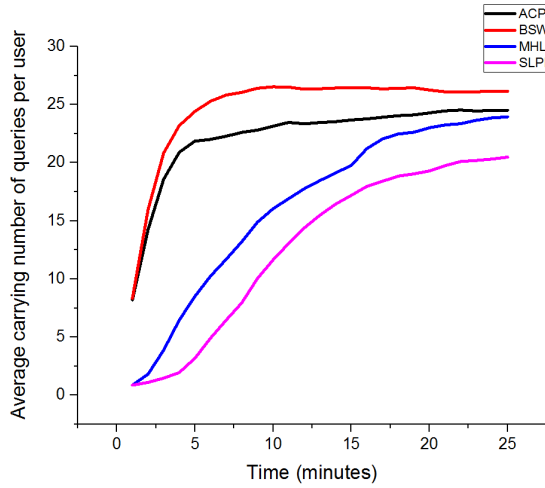
E. Distributing Appointment Cards

Exchanging appointment cards is a feature of the ACP, which imports burden into the network. We count the number of exchanging appointment cards per minute to evaluate the extra cost of the ACP.

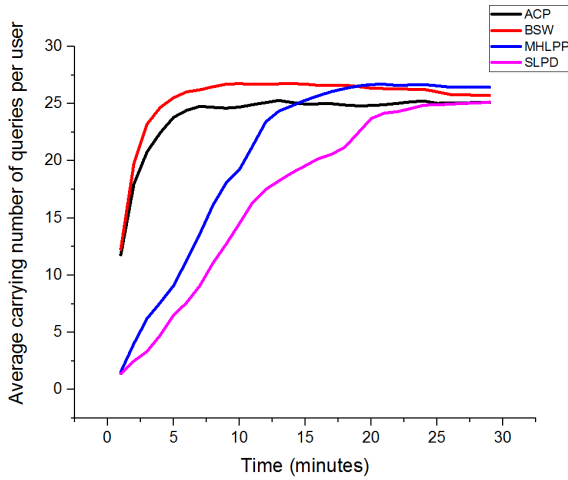
In Figure 17, we count the total number of exchanging ACs in the whole network. For example, if a user Alice encounters another user Bob, the total of exchanging ACs processes increases by one when Alice exchanges any ACs to Bob. We count the number of those exchanging processes occur per minute. As shown in the figure, the exchanging processes do not occur frequently, but about 2 times per minutes. Since the size of an appointment card is small, it does not cost the network many resources. At the same time, users can get many appointment cards to help them send queries, as shown



(a) communication radius is equal to 10 meters



(b) communication radius is equal to 50 meters



(c) communication radius is equal to 90 meters

Fig. 16. Average Number of Carried Queries Per User

in Figure 18. The number of ready ACs per user is raising smoothly and steadily.

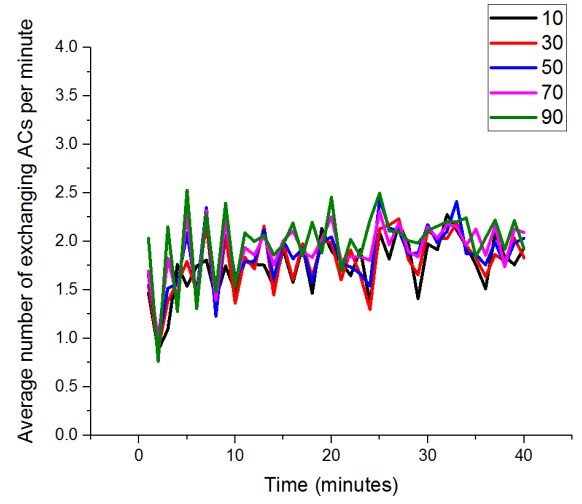


Fig. 17. Average Number of Exchanging ACs Per Minute

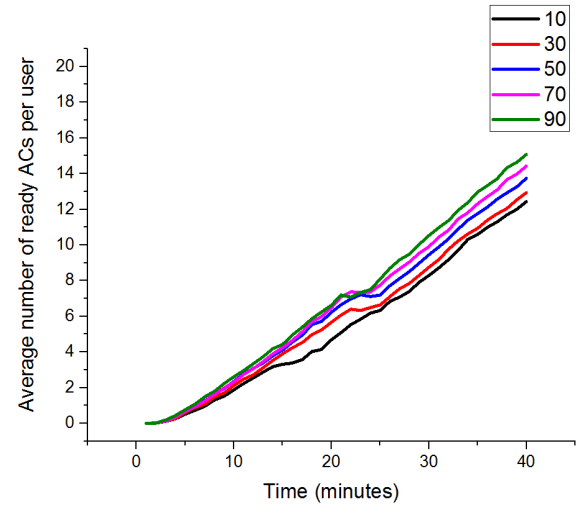


Fig. 18. Average Number of Ready ACs Per User

CONCLUSION

The location privacy-preserving protocol ACP uses social-relationship. It facilitates the obfuscation process by continuously exchanging appointment cards among users so that the original requester does not communicate with any of his friends when he sends a query. Simulation results show that it has a better performance with respect to the query-delivery success ratio and provides an acceptable obfuscation compared to its counterparts. Although ACP requires more time to forward the reply, the total time required for query delivery and reply delivery is still less than that for its counterparts SLPD and MHLPP. The major disadvantage is that the ACP must exchange ACs continuously, which can consume network

resources. However, that cost is low if users do not send too many queries.

REFERENCES

Please number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use “Ref. [3]” or “reference [3]” except at the beginning of a sentence: “Reference [3] was the first . . .”

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use “et al.”. Papers that have not been published, even if they have been submitted for publication, should be cited as “unpublished” [4]. Papers that have been accepted for publication should be cited as “in press” [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

REFERENCES

- [1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” *Phil. Trans. Roy. Soc. London*, vol. A247, pp. 529–551, April 1955.
- [2] J. Clerk Maxwell, *A Treatise on Electricity and Magnetism*, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
- [3] I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in *Magnetism*, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
- [4] K. Elissa, “Title of paper if known,” unpublished.
- [5] R. Nicole, “Title of paper with only first word capitalized,” *J. Name Stand. Abbrev.*, in press.
- [6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” *IEEE Transl. J. Magn. Japan*, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetism Japan, p. 301, 1982].
- [7] M. Young, *The Technical Writer's Handbook*. Mill Valley, CA: University Science, 1989.