IMPACT: International Journal of Research in Engineering & Technology (IMPACT: IJRET)
ISSN(E): 2321-8843; ISSN(P): 2347-4599
Vol. 2, Issue 8, Aug 2014, 11-22
© Impact Journals

A SCALABLE SERVER ARCHITECTURE FOR MOBILE PRESENCE SERVICE IN SOCIAL NETWORK APPLICATIONS

STELLA ANTONY K[1] & SHIJI C. G[2]

[1]Research Scholar, Department of Computer Science & Engineering, Marian Engineering College, Trivandrum, Kerala, India

[2]Assistant Professor, Department of Computer Science, Marian Engineering College, Trivandrum, Kerala, India

ABSTRACT

Social network applications are becoming increasingly popular on mobile devices. A mobile presence service is an essential component of a social network application because it maintains each mobile user's presence information, such as the current status (online/offline), GPS location and network address, and also updates the user's online friends with the information continually. If presence updates occur frequently, the enormous number of messages distributed by presence servers may lead to a scalability problem in a large-scale mobile presence service. To address the problem, we propose an efficient and scalable server architecture, called PresenceCloud, which enables mobile presence services to support large-scale social network applications. When a mobile user joins a network, PresenceCloud searches for the presence of his/her friends and notifies them of his/her arrival. PresenceCloud organizes presence servers into a quorum-based server-to-server architecture for efficient presence searching. It also leverages a directed search algorithm and a one-hop caching strategy to achieve small constant search latency. We analyze the performance of PresenceCloud in terms of the search cost and search satisfaction level. The search cost is defined as the total number of messages generated by the presence server when a user arrives; and search satisfaction level is defined as the time it takes to search for the arriving user's friend list. The results of simulations demonstrate that PresenceCloud achieves performance gains in the search cost without compromising search satisfaction**.**

KEYWORDS: Social Networks, Mobile Presence Services, Distributed Presence Servers, Cloud Computing

INTRODUCTION

Because of the ubiquity of the Internet, mobile devices and cloud computing environments can provide presence-enabled applications, *i.e.*, social network applications/services, worldwide. Facebook, Twitter, Foursquare, Google Latitude, buddycloud and Mobile Instant Messaging (MIM), are examples of presence-enabled applications that have grown rapidly in the last decade. Social network services are changing the ways in which participants engage with their friends on the Internet. They exploit the information about the status of participants including their appearances and activities to interact with their friends. Moreover, because of the wide availability of mobile devices (e.g., Smartphones) that utilize wireless mobile network technologies, social network services enable participants to share live experiences instantly across great distances. For example, Facebook receives more than 25 billion shared items every month and Twitter receives more than 55 million tweets each day. In the future, mobile devices will become more powerful, sensing and media capture devices. Hence, we believe it is inevitable that social network services will be the next generation of mobile Internet applications.

A mobile presence service is an essential component of social network services in cloud computing environments. The key function of a mobile presence service is to maintain an up-to-date list of presence information of all mobile users. The presence information includes details about a mobile user's location, availability, activity, device capability, and preferences. The service must also bind the user's ID to his/her current presence information, as well as retrieve and subscribe to changes in the presence information of the user's friends. In social network services, each mobile user has a friend list, typically called a buddy list, which contains the contact information of other users that he/she wants to communicate with. The mobile user's status is broadcast automatically to each person on the buddy list whenever he/she transits from one status to the other. For example, when a mobile user logs into a social network application, such as an IM system, through his/her mobile device, the mobile presence service searches for and notifies everyone on the user's buddy list. To maximize a mobile presence service's search speed and minimize the notification time, most presence services use server cluster technology. Currently, more than 500 million people use social network services on the Internet. Given the growth of social network applications and mobile network capacity, it is expected that the number of mobile presence service users will increase substantially in the near future. Thus, a scalable mobile presence service is deemed essential for future Internet applications.

In the last decade, many Internet services have been deployed in distributed paradigms as well as cloud computing applications. For example, the services developed by Google and Facebook are spread among as many distributed servers as possible to support the huge number of users worldwide. Thus, we explore the relationship between distributed presence servers and server network topologies on the Internet, and propose an efficient and scalable server-to-server overlay architecture called PresenceCloud to improve the efficiency of mobile presence services for large-scale social network services.

First, we examine the server architectures of existing presence services, and introduce the buddy-list search problem in distributed presence architectures in large-scale geographically data centers. The *buddy-list search problem* is a scalability problem that occurs when a distributed presence service is overloaded with buddy search messages.

Then, we discuss the design of PresenceCloud, a scalable server-to-server architecture that can be used as a building block for mobile presence services. The rationale behind the design of PresenceCloud is to distribute the information of millions of users among thousands of presence servers on the Internet. To avoid single point of failure, no single presence server is supposed to maintain service-wide global information about all users. PresenceCloud organizes presence servers into a quorum-based server-to-server architecture to facilitate efficient buddy list searching. It also leverages the server overlay and a directed buddy search algorithm to achieve small constant search latency; and employs an active caching strategy that substantially reduces the number of messages generated by each search for a list of buddies. We analyze the performance complexity of PresenceCloud and two other architectures, a Mesh-based scheme and a Distributed Hash Table (DHT)-based scheme. Through simulations, we also compare the performance of the three approaches in terms of the number of messages generated and the search satisfaction which we use to denote the search response time and the buddy notification time. The results demonstrate that PresenceCloud achieves major performance gains in terms of reducing the number of messages without sacrificing search satisfaction. Thus, PresenceCloud can support a large-scale social network service distributed among thousands of servers on the Internet.

The contribution of this paper is threefold. First, PresenceCloud is among the pioneering architecture for mobile presence services. To the best of our knowledge, this is the first work that explicitly designs a presence server architecture that significantly outperforms those based distributed hash tables. PresenceCloud can also be utilized by Internet social network applications and services that need to replicate or search for mutable and dynamic data among distributed presence servers. The second contribution is that we analyze the scalability problems of distributed presence server architectures, and define a new problem called the buddy-list search problem. Through our mathematical formulation, the scalability problem in the distributed server architectures of mobile presence services is analyzed. Finally, we analyze the performance complexity of Presence- Cloud and different designs of distributed architectures, and evaluate them empirically to demonstrate the advantages of PresenceCloud.

THE PROBLEM STATEMENT

We describe the system model, and the *buddy-list search problem*. Formally, we assume the geographically distributed presence servers to form a serverto-server overlay network, $G = (V,E)$, where V is the set of the Presence Server (PS) nodes, and E is a collection of ordered pairs of V. Each PS node $ni \, \varepsilon \, V$ represents a Presence Server and an element of E is a pair $(ni,nj) \, \varepsilon \, E$ with $ni,nj \, \varepsilon \, V$. Because the pair is ordered, $(nj, ni) \, \varepsilon \, E$ is not equivalent to $(ni, nj) \, \varepsilon \, E$.So, the edge $(ni; nj)$ is called an outgoing edge of ni, and an incoming edge of nj. The server overlay enables its PS nodes to communicate with one another by forwarding messages through other PS nodes in the server overlay. Also, we denote a set of the mobile users in a presence service as $U = \{u1,\ldots, ui,\ldots,um\}$, where $1 \leq i \leq m$ and m is the number of mobile users. A mobile user ui connects with one PS node for search other user's presence information, and to notify the other mobile users of his/her arrival. Moreover, we define a *buddy list* as following.

Definition 1. Buddy list, $Bi = \{b1, b2,\ldots,bk\}$ of user $ui \, \varepsilon \, U$, is defined as a subset of U, where $0< k \leq |U|$. Furthermore, B is a symmetric relation, *i.e*, $ui \, \varepsilon \, Bj$ implies $uj \, \varepsilon \, Bi$.

For example, given a mobile user up is in the buddy list of a mobile user uq, the mobile user uq also appear in the buddy list of the mobile user up. Note that to simplify the analysis of the Buddy-List Search Problem, we assume that buddy relation is a symmetric. However, in the design of PresenceCloud, the relation of buddies can be unilateral because the search operation of PresenceCloud can retrieve the presence of a mobile user by given the ID of the mobile user.

Problem Statement: Buddy-List Search Problem When a mobile user ui changes his/her presence status, the mobile presence service searches presence information of mobile users in buddy list Bi of ui and notifies each of them of the presence of ui and also notifies ui of these online buddies. The Buddy-List Search Problem is then defined as designing a server architecture of mobile presence service such that the costs of searching and notification in communication and storage are minimized.

Analysis of a Naive Architecture of Mobile Presence Service

In the following, we will give an analysis of the expected rate of messages generated to search for buddies of newly arrived user in a naive architecture of mobile presence services. We assume that each mobile user can join and leave the presence service arbitrarily, and each PS node only knows those mobile users directly attached to it. We also assume the probability for a mobile user to attach to a PS node to be uniform. Let's denote _ the average arriving rate of mobile users in a mobile presence service. In this paper, we focus on architecture design of mobile presence services and leave the problem of designing the capacity of presence servers as a separate research issue. Thus, we assume each PS node to have infinite service capacity. Hence, $\mu = \gimel/n$ is the average rate of mobile users attaching to a PS node, where n is denoted the number of PS nodes in a mobile presence service. Let h denote the probability of having all users in the buddy list of ui to be attaching to the same PS node as ui. It is the probability of having no need to send search messages when ui attaches to a PS node. Thus,

$h = \Pi\ 1/\mathrm{n} = n\ \hat{}|Bi|.$

$|Bi|$

The expected number of search messages generated by this PS node per unit time is then

$(n - 1) \times (1 - h) \times \mu.$

For a reasonable size of set Bi (e.g., $|Bi| \geq 3$) and $n \geq 100$, we consider the expected number Q of messages generated by the n PS nodes per unit time, then we have

$Q = n \times (n - 1) \times (1 - h) \times \mu$

$= n \times (n - 1) \times (1 - h) \times \gimel/n$

$\approx (n\text{-}1) \times \gimel.$

Thus, as the number of PS nodes increase, both the total communication and the total CPU processing overhead of presence servers also increase. It also shows that $\gimel$ is another important parameter having impact on the system the number of PS nodes increase, both the total communication and the total CPU processing overhead of presence servers also increase. It also shows that $\gimel$ is another important parameter having impact on the system overhead. When $\gimel$ increases substantially, it has a major impact on system overhead.

Collectively, we refer to the above phenomena as the *buddy-list search problem*, and any distributed presence server architectures could inevitably suffer from this scalability problem. The analysis shows that the buddy list searching operation of mobile presence services is costly and should be designed with caution.

DESIGN OF PRESENCECLOUD

The past few years has seen a veritable frenzy of research activity in Internet-scale object searching field, with many designed protocols and proposed algorithms. Most of the previous algorithms are used to address the fixed object searching problem in distributed systems for different intentions. However, people are nomadic, the mobile presence information is more mutable and dynamic; a new design of mobile presence services is needed to address the buddylist search problem, especially for the demand of mobile social network applications.

PresenceCloud is used to construct and maintain a distributed server architecture and can be used to efficiently query the system for buddy list searches. PresenceCloud consists of three main components that are run across a set of presence servers.

In the design of PresenceCloud, we refine the ideas of P2P systems and present a particular design for mobile presence services. The three key components of PresenceCloud are summarized below:

- PresenceCloud server overlay: organizes presence servers based on the concept of *grid quorum system*. So, the server overlay of PresenceCloud has a balanced load property and a two-hop diameter with $O(\sqrt{n})$ node degrees, where n is the number of presence servers.
- One-hop caching strategy: is used to reduce the number of transmitted messages and accelerate query speed. All presence servers maintain caches for the buddies offered by their immediate neighbors.
- Directed buddy search: is based on the directed search strategy. PresenceCloud ensures an one-hop search, it yields a small constant search latency on average.

PresenceCloud Overview

The primary abstraction exported by our PresenceCloud is used to construct a scalable server architecture for mobile presence services, and can be used to efficiently search the desired buddy lists. We illustrated a simple overview of PresenceCloud in Figure 1. In the mobile Internet, a mobile user can access the Internet and make a data connection to PresenceCloud via 3G or Wifi services. After the mobile user joins and authenticates himself/herself to the mobile presence service, the mobile user is determinately directed to one of Presence Servers in the PresenceCloud by using the Secure Hash Algorithm, such as SHA-1. The mobile user opens a TCP connection to the Presence Server (PS node) or control message transmission, particularly for the presence information. After the control channel is established, the mobile user sends a request to the connected PS node for his/her buddy list searching.

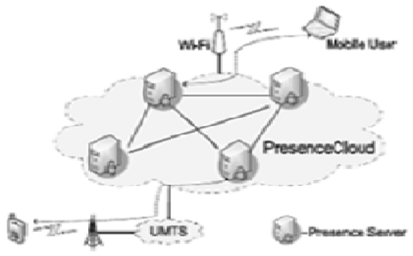

Figure 1: An Overview of PresenceCloud

Our PresenceCloud shall do an efficient searching operation and return the presence information of the desired buddies to the mobile user. Now, we discuss the three components of PresenceCloud in detail below.

PresenceCloud Server Overlay

The PresenceCloud server overlay construction algorithm organizes the PS nodes into a server-to-server overlay, which provides a good low-diameter overlay property. The low-diameter property ensures that a PS node only needs two hops to reach any other PS nodes. The detailed description is as follows.

Our PresenceCloud is based on the concept of *grid quorum system*], where a PS node only maintains a set of PS nodes of size $O(\sqrt{n})$, where n is the number of PS nodes in mobile presence services. In a PresenceCloud system, each PS node has a set of PS nodes, called *PS list*, that constructed by using a grid quorum system, shown in Figure 2 for n=9. The size of a grid quorum is $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$. When a PS node joins the server overlay of PresenceCloud, it gets an ID in the grid, locates its position in the grid and obtains its PS list by contacting a root server1. On the $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ grid, a PS node with a grid ID can pick one column and one row of entries and these entries will become its PS list in a

PresenceCloud server overlay. Figure 3 illustrates an example of PresenceCloud, in which the grid quorum is set to $\lceil\sqrt{9}\rceil \times \lceil\sqrt{9}\rceil$. In the Figure 3, the PS node 8 has a PS list *{*2,5,7,9*}* and the PS node 3 has a PS list *{*1,2,6,9*}*. Thus, the PS node 3 and 8 can construct their overly networks according to their PS lists respectively.

We now show that each PS node in a PresenceCloud system only maintains the PS list of size O($\sqrt{n}$), and the construction of PresenceCloud using the grid quorum results in each PS node can reach any PS node at most two hops.

Lemma 1: *The server overlay construction algorithm of PresenceCloud guarantees that each presence server only maintains a presence server list of size O($\sqrt{n}$), where n is the number of presence servers.*

Proof: We consider a grid quorum system of $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ and arrange elements of the global set $G = \{1, 2,\ldots,n\}$ as a $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ array. For each grid point *i* has a quorum set, *Si* that contains a full column plus a full row of elements in the array. Then it is clear that for any grid point *i* in a grid quorum system, the size of |*Si*| is equal to $2\lceil\sqrt{n}\rceil - 1$. Therefore, for any PS node in a PresenceCloud, the size of a PS list maintained at a PS node is O($\sqrt{n}$).

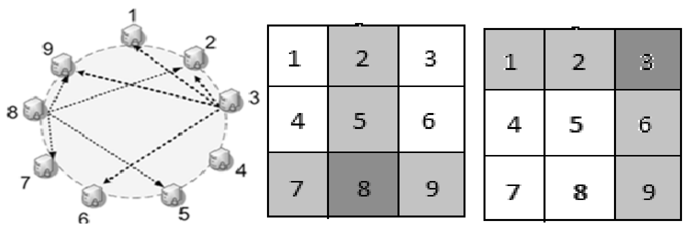

Figure 2: A Perspective of PresenceCloud Server Overlay

Lemma 2: *Each presence server in a PresenceCloudserver overlay can reach any other presence servers in a two hops route.*

Proof: In a grid quorum system of $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$, for each grid point *i* has a quorum set, *Si* that contains a full column plus a full row of elements in the array. It is clear that $|Si \cap S| \neq$ *null* for any grid point *i* and *j*, $1 \leq i; j \leq n$. Let *K* be the some grid points in $Si \cap Sj$. Thus, for any grid point, *j not ε Si*, the grid point *i* can reach *j* by passing one of grid points, *l ε K* set. Then it is clear that for any PS node *i* in a PresenceCloud, can reach any other PS node in a two hops route.

Here, we give an example for *Lemma* 2. In Figure.1 the PS list of PS node 1 is the set *{*2,3,4,7*}* and one of PS node 8 is the set *{*2,5,7,9*}*. PS node 8 can reach PS node 1 by *K* set *{*2, 7*}*, *i.g*., a route 8 → 2 → 1 or 8 → 7 → 1. Note that *K* set is the intersection set of two PS lists Consequently, PresenceCloud can hold the two-hop diameter property based on a grid quorum system.

Stabilization of Incomplete Quorum Systems

Our algorithm is fault tolerance design. At each PS node, a simple *Stabilization*() process periodically contacts existing PS nodes to maintain the *PS list*. The *Stabilization*() process is elaborately presented in the Algorithm 1. When a PS node joins, it obtains its PS list by contacting a root. However, if a PS node *n* detects failed PS nodes in its PS list, it needs to establish new connections with existing PS nodes. In our algorithm, *n* should pick a random PS node that is in the same column or row as the failed PS node.

Algorithm 1: PresenceCloud Stabilization algorithm

```
1. /* periodically verify PS node n's pslist */
2. Definition:
3. pslist: set of the current PS list of this PS node, n
4. pslist[].connection: the current PS node in pslist
5. pslist[].id: identifier of the correct connection in pslist
6. node.id: identifier of PS node node
7. Algorithm:
8. r ← Sizeof(pslist)
9. for i = 1 to r do
10. node ← pslist[i].connection
11. if node:id ≠ pslist[i].id then
12. /* ask node to refresh n's PS list entries */
13. findnode ← Find_ CorrectPSNode(node)
14. if findnode = nil then
15. pslist[i].connection ← RandomNode(node)
16. else
17. pslist[i].connection ← findnode
18. end if
19. else
20. /* send a heartbeat message */
21. bfailed ← SendHeartbeatmsg(node)
22. if bfailed = true then
23. pslist[i].connection ← RandomNode(node)
24. end if
25. end if
26. end for
```

In Algorithm 1, the function Find CorrectPSNode() re-turns the correct PS list entry, *i.e.*, *findnode:id* is equal to *pslist*[*i*]*:id*. Moreover, it is an easy implemented function based on *Lemma* 2**.** The function RandomNode(*node*) is designed to pick a random PS node that is in the same column or row as the failed PS node, *node*. This function can retrieve substituted nodes by asking the existing PS node in PS list.

In our design, we define *pslist*[*i*]*:id* as the id of the *i*th logical neighbor of a PS node *p*, while *pslist*[*i*]*:connection:id* as the id of the physical PS node that handles presence information for the *i*th logical neighbor of *p*. In other words, if the PS node with *ID* = *pslist*[*i*]*:id* does not exist on the overlay, then the Stabilization algorithm will assign the PS node with ID=*pslist*[*i*]*:connection:id* to take its place.

For example, in Figure 2, we let PS node 8 be the node running the Stabilization algorithm. In general, in PS node 8, the values of *pslist*[]*:id* should be *{*2, 5, 7, 9*}* and the values of *pslist*[]*:connection* should be *{*node 2, node 5, node 7, node 9*}*. If the PS node 2 is failed then PS node 8 can choose node 1 or node 3 to take over PS node 2 after running the Stabilization algorithm. Clearly, several attractive features of our algorithm are that it is simple to implement, is naturally robust to failures, and is with a two-hop diameter property.

The heart beat messages overhead is another performance factor in distributed server overlays. Anyone distributed server overlay architecture requires heart beat message to maintain the connectivity of each server for recovering system from server failure. However, in order to reduce the maintenance overhead,the presence cloud piggybacks the heart beat messages in buddy serach messages for saving transmission costs.

The following results quantifies the robustness of the PresenceCloud protocol.

Lemma 3*: if a presence server uses a PS list of size r in a PresenceCloud that is initially stable, and if every presence server fails with probability* 1/2*, then the Stabilization algorithm sustains the PS list with high probability.*

Proof: Before any nodes fail, a node was aware of its *r* immediate presence servers. The probability that all of these presence servers fail is (1/2)^*r*, thus every presence server is aware of these presence servers in its PS list with high probability. As was implied by ***Lemma*** **2**, if the current node *n* starts stabilization procedure and if the correct PS node *p* exists, then node *n* retrieves *p* in one hop query. Therefore, the Stabilization algorithm can sustain the PS list by asking these immediate presence servers.

Then, we describe a mobile user join procedure. When a mobile user joins a PresenceCloud, he/she uses a hash function, such as Secure Hash Algorithm (SHA-1),to hash his/her identifier in the mobile presence service to get a grid ID in the grid quorum. PresenceCloud assigns mobile users to PS nodes with a hash function, e.g., SHA-1 algorithm such that, with a high probability, the hash function balances loads uniformly and thus all PS nodes receive roughly the same number of mobile users.

So that the mobile user can decide that he/she has to associate with the specific PS node, *q* with the hashed grid ID. Note that PresenceCloud uses overlay network architecture. In other words, although the mobile user knows ID of the PS node *q*, it does not have IP address of *q*. In order to obtain IP address of *q*, the mobile user first hashes his/her identifier into the grid ID of the specific PS node *q*.

Then the mobile user contacts randomly with a of PS node, *p* redirect he/she to the specific PS node, *q*. Note that PresenceCloud does not require a centralized server to response the IP address lookup of PS nodes for every user joining. However, if the specific PS node *q* does not exist, the contacted PS node *p* can redirect the mobile user to those PS nodes that are in the same row as the PS node q. For example, in Figure 2, a mobile user obtains a grid ID 6 by hashing its identifier in the mobile presence service, then he/she randomly contacts with PS node 2. And PS node 2 should redirect the mobile user to PS node 6. If PS node 6 is not existing, then PS node 2 should redirect the mobile user to PS node 4 or 5.

One-Hop Caching

To improve the efficiency of the search operation, PresenceCloud requires a caching strategy to replicate presence information of users. Inorder to adapt to changes in the presence of users, the caching strategy should be asynchronous and not require expensive mechanism for distributed agreement. In PresenceCloud, each PS node maintains a *user list* of presence information of the attached users, and it is responsible for caching the *user list* of each node in its PS list, in other words, PS nodes only replicate the *user list* at most one hop away from itself. The cache is updated when neighbors establish connections to it, and periodically updated with its neighbors. Therefore, when a PS node receives a query, it can respond not only with matches from its own *user list*, but also provide matches from its caches that are the user lists offered by all of its neighbors.

Our caching strategy does not require expensive overhead for presence consistency among PS nodes. When a mobile user changes its presence information, either because it leaves PresenceCloud, or due to failure, the responded PS node can disseminate its new presence to other neighboring PS nodes for getting updated quickly. Consequently, this one-hop caching strategy ensures that the user's presence information could remain mostly up-to-date and consistent throughout the session time of the user.

More specifically, it should be easy to see that, each PS node maintains roughly $2(\lceil\sqrt{n}\rceil-1)\times u$ replicas of presence information, due to each PS node replicates its *user list* at most one hop away from itself. Here, u is denoted the average number of mobile users in a PS node. Therefore, we have the following lemma.

Lemma 4: *Each presence server in PresenceCloud only maintains the one-hop replicas of presence information of size* $O(u\times\sqrt{n})$*, where u is denoted the average number of mobile users in a presence server and n is the number of presence servers.*

By maintaining $O(u\times\sqrt{n})$ replicas of presence information at each PS node and the simple two-hop overlay design, PresenceCloud has sufficient redundancy to provide a good level of buddy searching service. Furthermore, this caching mechanism can significantly reduce the communication cost during searching operation. In the next section, an analysis studies will reveal that the one-hop caching mechanism brings PresenceCloud great improvement in buddy searching cost.

Directed Buddy Search

We contend that minimizing searching response time is important to mobile presence services. Thus, the buddy list searching algorithm of PresenceCloud coupled with the two-hop overlay and one-hop caching strategy ensures that PresenceCloud can typically provide swift responses for a large number of mobile users. First, by organizing PS nodes in a server-to-server overlay network, we can therefore use one-hop search exactly for queries and thus reduce the network traffic without significant impact on the search results.

Second, by capitalizing the one-hop caching that maintains the user lists of its neighbors, we improve response time by increasing the chances of finding buddies. Clearly, this mechanism both reduces the network traffic and response time. Based on the mechanism, the population of mobile users can be retrieved by a broadcasting operation in any PS node in the mobile presence service. Moreover, the broadcasting message can be piggybacked in a buddy search message for saving the cost.

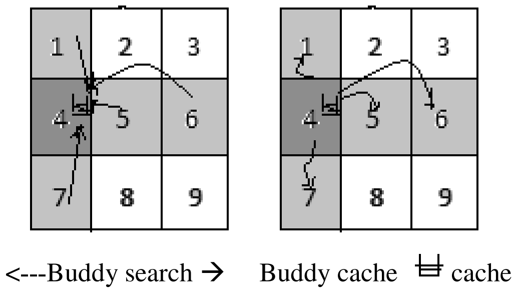

Figure 3: An Example of Buddy List Searching Operations in PresenceCloud

As previously mentioned, PresenceCloud does not require a complex searching algorithm, the directed searching technique can improve search efficiency. Thus, we have the following lemma.

Lemma 5: *For each buddy list searching operation, the directed buddy search of PresenceCloud retrieves the presence information _ of the queried buddy list at most onehop.*

Proof: This is a direct consequence of *Lemma* 2 and *Lemma* 3.

Before presenting the directed buddy search algorithm, lets revisit some terminologies which will be used in the algorithm.

B = *{b*1*, b*2,…,*bk}*: set of identifiers of user's buddies

B^(*i*): Buddy List Search Message be sent to PS node *i*

B^(*i*): set of buddies that shared the same grid ID *i*

Sj: set of *pslist*[]*:id* of PS node *j*

Directed Buddy Search Algorithm:

- A mobile user logins PresenceCloud and decides the associated PS node, *q*.
- The user sends a Buddy List Search Message, *B* to the PS node *q*.
- When the PS node *q* receives a *B*, then retrieves each *bi* from *B* and searches its user list and one-hop cache to respond to the coming query. And removes the responded buddies from *B*.
- If *B* = *nil*, the buddy list search operation is done.
- Otherwise, if *B≠nil*, the PS node *q* should hash each remaining identifier in *B* to obtain a grid ID, respectively.
- Then the PS node *q* aggregates these *b*^(*g*) to become a new *B*^(*j*), for each *g ε Sj*. Here PS node *j* is the intersection node of *Sq*∩*Sg*. And sends the new *B*^(*j*) to PS node *j*.

Following, we describe an example of directed buddy search in PresenceCloud. When a PS node 4 receives a Buddy List Search Message, *B* = *{*1,2,3,4,5,6,7,8,9*}*, from a mobile user, PS node 4 first searches its local *user list* and the buddy cache, and then it responds these searched buddies to the mobile user and removes these searched buddies from *B*.

In Figure 3, these removed buddies include the user lists of PS node *{*1,4,5,6,7*}*. Then PS node 4 can aggregates *b*^(3) and *b*^(9) to become a new *B*^(6) and sends the new *B*^(6) to PS node 6. Note that the *pslist*[]*:id* of PS node 6 is *{*3,4,5,9*}*. Here, PS node 4 also aggregates *b*^(2) and *b*^(8) to become a new *B*^(5) and sends the new *B*^(5) to PS node 5. However, due to the one-hop caching strategy, PS node 6 has a buddy cache that contains these user lists of PS node *{*3,9*}*, PS node 6 can expeditiously respond the buddy search message *B*^(6). Consequently, the directed searching combined with both previous two mechanisms, including PresenceCloud server overlay and one-hop caching strategy, can reduce the number of searching messages sent.

CONCLUSIONS

In this paper, we have presented PresenceCloud, a scalable server architecture that supports mobile presence services in large-scale social network services. We have shown that PresenceCloud achieves low search latency and enhances the performance of mobile presence services. In addition, we discussed the scalability problem in server architecture designs, and introduced the buddy-list search problem, which is a scalability problem in the distributed server architecture of mobile presence services. Through a simple mathematical model, we show that the total number of buddy search messages increases substantially with the user arrival rate and the number of presence servers. The results of simulations demonstrate that PresenceCloud achieves major performance gains in terms of the search cost and search satisfaction. Overall, PresenceCloud is shown to be a scalable mobile presence service in large-scale social network services.

REFERENCES

1. Facebook, http://www.facebook.com.
2. Twitter, http://twitter.com.
3. Foursquare http://www.foursquare.com.
4. Google latitude, http://www.google.com/intl/enus/latitude/intro.html.
5. Buddycloud, http://buddycloud.com.
6. Mobile instant messaging, http://en.wikipedia.org/wiki/Mobile instant messaging.
7. R. B. Jennings, E. M. Nahum, D. P. Olshefski, D. Saha, Z.-Y. Shae, and C. Waters, "A study of internet instant messaging and chat protocols," *IEEE Network*, 2006.
8. Gobalindex, http://www.skype.com/intl/en-us/support/user-guides/p2pexplained/.
9. Z. Xiao, L. Guo, and J. Tracey, "Understanding instant messaging traffic characteristics," *Proc. of IEEE ICDCS*, 2007.
10. C. Chi, R. Hao, D. Wang, and Z.-Z. Cao, "Ims presence server: Traffic analysis and performance modelling," *Proc. of IEEE ICNP*, 2008.
11. Instant messaging and presence protocol ietf working group http://www.ietf.org/html.charters/impp-charter.html.

12. Extensible messaging and presence protocol ietf working group http://www.ietf.org/html.charters/xmpp-charter.html.
13. Sip for instant messaging and presence leveraging extensions ietf working group. http://www.ietf.org/html.charters/simplecharter.html.
14. P. Saint-Andre., "Extensible messaging and presence protocol (xmpp): Instant messaging and presence describes instant messaging (im), the most common application of xmpp," *RFC 3921*, 2004.
15. B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, and D. Gurle, "Session initiation protocol (sip) extension for instant messaging," *RFC 3428*, 2002.
16. Jabber, http://www.jabber.org/.
17. Peer-to-peer session initiation protocol ietf working group, http://www.ietf.org/html.charters/p2psip-charter.html.
18. K. Singh and H. Schulzrinne, "Peer-to-peer internet telephony using sip," *Proc. of ACM NOSSDVA*, 2005.
19. P. Saint-Andre, "Interdomain presence scaling analysis for the extensible messaging and presence protocol (xmpp)," *RFC Internet Draft*, 2008.
20. A Houri, T. Rang, and E. Aoki, "Problem statement for sip/simple," *RFC Internet-Draft*, 2009.
21. Houri, S. Parameswar, E. Aoki, V. Singh, and H. Schulzrinne, "Scaling requirements for presence in sip/simple," *RFC Internet- Draft*, 2009.
22. S. A. Baset, G. Gupta, and H. Schulzrinne, "Openvoip: An open peer-to-peer voip and im system," *Proc. of ACM SIGCOMM*, 2008.
23. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "Sip: Session initiation protocol," *RFC 3261*, 2002.
24. Open Mobile Alliance, OMA instant messaging and presence service, 2005.