A Scalable Server Architecture for Mobile Presence Services in Social Network Applications

Chi-Jen Wu, Jan-Ming Ho, *Member, IEEE,* Ming-Syan Chen, *Fellow, IEEE*

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.

Index Terms—Social networks, mobile presence services, distributed presence servers, cloud computing.

1 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 [1], Twitter [2], Foursquare [3], Google Latitude [4], buddycloud [5] and Mobile Instant Messaging (MIM) [6], 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 [7]. Currently, more than 500 million people use social network services on the Internet [1]. Given the growth of social network applications and mobile network capacity, it is expected that the number of mobile

- *Chi-Jen Wu is with the Department of Electrical Engineering, National Taiwan University, Taiwan and also with the Institute of Information Science, Academia Sinica, Taiwan.*
  E-mail: cjwu@iis.sinica.edu.tw
- *Dr. Jan-Ming Ho is with the Institute of Information Science, Academia Sinica, Taiwan.*
  E-mail: hoho@iis.sinica.edu.tw
- *Dr. Ming-Syan Chen is with the Research Center of Information Technology Innovation, Academia Sinica, Taiwan and also with the Department of Electrical Engineering, National Taiwan University, Taiwan.*
  E-mail: mschen@citi.sinica.edu.tw

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 PresenceCloud and different designs of distributed architectures, and evaluate them empirically to demonstrate the advantages of PresenceCloud.

The remainder of this paper is organized as follows. The next section contains a review of related works. In Section 3, we consider the buddy-list search problem in a distributed presence server architecture; and in Section 4, we describe the design of PresenceCloud in detail. The complexity analyses of PresenceCloud, the mesh-based scheme and the DHT-based scheme are presented in Section 5; and the performance results of the three approaches are detailed in Section 6. In Sections 7, we discuss performance issues related to PresenceCloud. Section 8 contains some concluding remarks.

2 Related Work

In this section, we describe previous researches on presence services, and survey the presence service of existing systems. Well known commercial IM systems leverage some form of centralized clusters to provide presence services [7]. Jennings III *et al.* [7] presented a taxonomy of different features and functions supported by the three most popular IM systems, AIM, Microsoft MSN and Yahoo! Messenger. The authors also provided an overview of the system architectures and observed that the systems use client-server-based architectures. Skype, a popular voice over IP application, utilizes the Global Index (GI) technology [8] to provide a presence service for users. GI is a multi-tiered network architecture where each node maintains full knowledge of all available users. Since Skype is not an open protocol, it is difficult to determine how GI technology is used exactly. Moreover, Xiao *et al.* [9] analyzed the traffic of MSN and AIM system. They found that the presence information is one of most messaging traffic in instant messaging systems. In [10], authors shown that the largest message traffic in existing presence services is buddy NOTIFY messages.

Several IETF charters [11]–[13] have addressed closely related topics and many RFC documents on instant messaging and presence services have been published, e.g.,XMPP [14], SIMPLE [15]. Jabber [16] is a well-known deployment of instant messaging technologies based on distributed architectures. It captures the distributed architecture of SMTP protocols. Since Jabber's architecture is distributed, the result is a flexible network of servers that can be scaled much higher than the monolithic, centralized presence services. Recently, there is an increase amount of interest in how to design a peer-to-peer SIP [17]. P2P-SIP [18] has been proposed to remove the centralized server, reduce maintenance costs, and prevent failures in server-based SIP deployment. To maintain presence information, P2PSIP clients are organized in a DHT system, rather than in a centralized server. However, the presence

service architectures of Jabber and P2PSIP are distributed, the *buddy-list search problem* we defined later also could affect such distributed systems.

It is noted that few articles in [19]–[21] discuss the scalability issues of the distributed presence server architecture. Saint Andre [19] analyzes the traffic generated as a result of presence information between users of inter-domains that support the XMPP. Houri *et al.* [20] show that the amount of presence traffic in SIMPLE [13] can be extremely heavy, and they analyze the effect of a large presence system on the memory and CPU loading. Those works in [21], [22] study related problems and developing an initial set of guidelines for optimizing inter-domain presence traffic and present a DHT-based presence server architecture.

Recently, presence services are also integrated into mobile services. For example, 3GPP has defined the integration of presence service into its specification in UMTS. It is based on SIP [23] protocol, and uses SIMPLE [15] to manage presence information. Recently, some mobile devices also support mobile presence services. For example, the Instant Messaging and Presence Services (IMPS) was developed by the Wireless Village consortium and was united into Open Mobile Alliance (OMA) IMPS [24] in 2005. In [25], Chen *et al.* proposed a weakly consistent scheme to reduce the number of updating messages in mobile presence services of IP Multimedia Subsystem (IMS). However, it also suffers scalability problem since it uses a central SIP server to perform presence update of mobile users [26]. In [27], authors presented the server scalability and distributed management issues in IMS-based presence service.

3 THE PROBLEM STATEMENT

In this section, we describe the system model, and the *buddy-list search problem*. Formally, we assume the geographically distributed presence servers to form a server-to-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 $n_i \in V$ represents a Presence Server and an element of E is a pair $(n_i, n_j) \in E$ with $n_i, n_j \in V$. Because the pair is ordered, $(n_j, n_i) \in E$ is not equivalent to $(n_i, n_j) \in E$. So, the edge (n_i, n_j) is called an outgoing edge of n_i, and an incoming edge of n_j. 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 $U = \{u_1, \ldots, u_i, \ldots, u_m\}$, where $1 \leq i \leq m$ and m is the number of mobile users. A mobile user u_i 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, $B_i = \{b_1, b_2, \ldots, b_k\}$ of user $u_i \in U$, is defined as a subset of U, where $0 < k \leq |U|$. Furthermore, B is a symmetric relation, *i.e*, $u_i \in B_j$ implies $u_j \in B_i$.

For example, given a mobile user u_p is in the buddy list of a mobile user u_q, the mobile user u_q also appear in the buddy list of the mobile user u_p. 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 u_i changes his/her presence status, the mobile presence service searches presence information of mobile users in buddy list B_i of u_i and notifies each of them of the presence of u_i and also notifies u_i 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.

3.1 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 λ 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, $\mu = \frac{\lambda}{n}$ 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 u_i to be attaching to the same PS node as u_i. It is the probability of having no need to send search messages when u_i attaches to a PS node. Thus,

$$h = \prod_{|B_i|} \frac{1}{n} = n^{-|B_i|}.$$

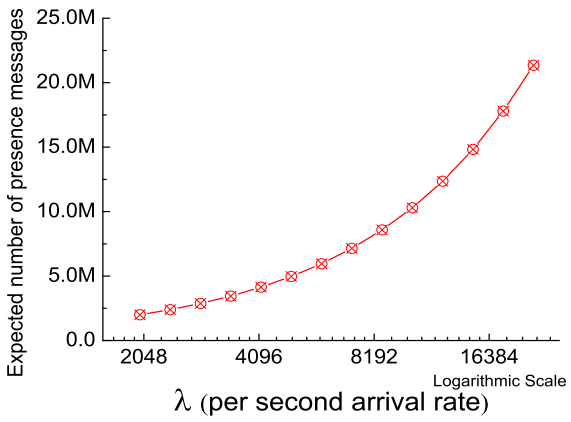
The expected number of search messages generated by this PS node per unit time is then

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

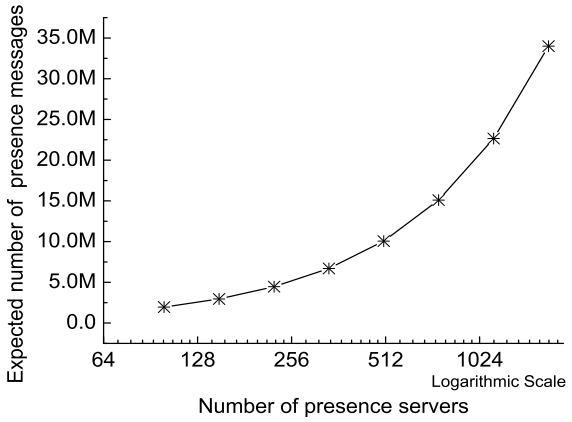
For a reasonable size of set B_i (e.g., $|B_i| \geq 3$) and $n \geq 100$, we consider the expected number Q of messages generated by the n PS nodes per unit time, then we have

$$\begin{aligned} Q &= n \times (n-1) \times (1-h) \times \mu \\ &= n \times (n-1) \times (1-h) \times \frac{\lambda}{n} \\ &\simeq (n-1) \times \lambda. \end{aligned}$$

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 λ is another important parameter having impact on the system



(a) The average arrival rate of mobile users increases from 2,000 to 21,000 per second in a 1,000 PS nodes mobile presence system



(b) The number of PS nodes increases from 100 to 1,700 in a mobile presence service with 20,000 per second arrival rate of mobile users

Fig. 1. The analysis of expected total transmissions when searching for buddy lists in a distributed mobile presence service (The x axis of both sub figures is in logarithmic scale)

overhead. When λ increases substantially, it has a major impact on the system overhead. However, a scalable presence system should be able to support more than 20,000 mobile user logins per second in burst cases as reported by [28].

In Fig. 1, we plot statistics for all expected queries transmitted in a mobile presence service to show the increase in number of buddy search messages as lambda increases. Fig. 1(a) shows that in a 1,000 PS nodes system, and the average arrival rate of mobile users increases from 2,000 to 21,000. From the results of analysis, the number of buddy searching messages increases with increasing average arrival rate of mobile users. Fig. 1(b) plots another scenario, the average arrival rate of user joining is 20,000 per second, and the number of PS nodes in a system increases from 100 to 1,700. It shows that the number of total buddy searching messages increases significantly with the number of PS nodes.

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.

4 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 buddy-list 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* [29]. 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.

4.1 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 Fig. 2. 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 [30]. The mobile user opens a TCP connection to the Presence Server (PS node) for 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. Our

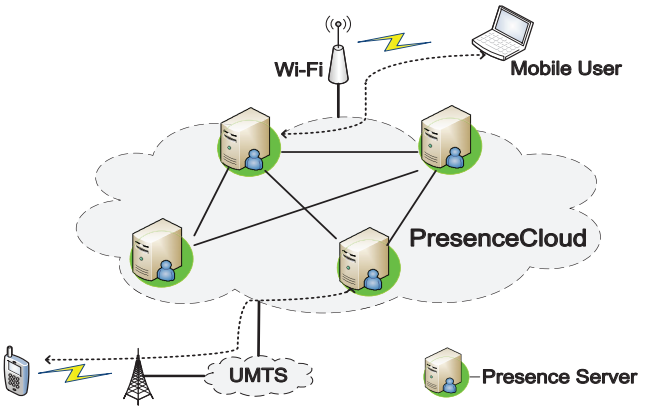


Fig. 2. An overview of PresenceCloud

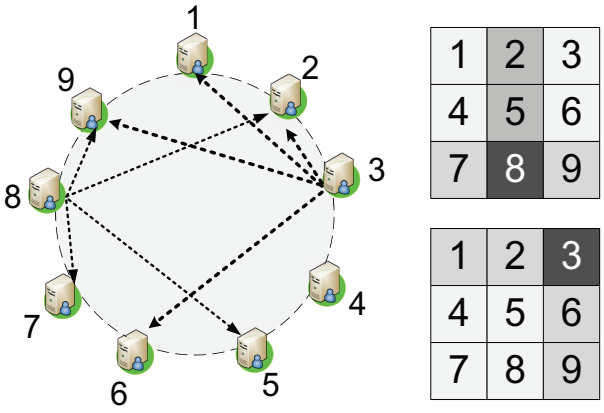


Fig. 3. A perspective of PresenceCloud Server Overlay

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.

4.2 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* [29], 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 Fig. 3 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 server[1]. 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. Fig. 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 Fig. 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, S_i 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 $|S_i|$ 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 $O(\sqrt{n})$. □

***Lemma** 2: Each presence server in a PresenceCloud server 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, S_i that contains a full column plus a full row of elements in the array. It is clear that $S_i \cap S_j \neq null$ for any grid point i and j, $1 \leq i, j \leq n$. Let K be the some grid points in $S_i \cap S_j$. Thus, for any grid point, $j \notin S_i$, the grid point i can reach j by passing one of grid points, $l \in K$ 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 Fig. 3, 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.e.*, a route $8 \rightarrow 2 \rightarrow 1$ or $8 \rightarrow 7 \rightarrow 1$. 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.

4.2.1 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.

1. The root server may become a scalability problem in a very large scale PresenceCloud, ex, a million PS nodes. However, it also can be extended to work with distributed root servers, such as the KAD system in BitTorrent [31].

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() returns 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 ith 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 ith 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 Fig. 3, 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, PresenceCloud piggybacks the heart beat message in buddy search messages for saving transmission costs.

The following result 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 $\frac{1}{2}$, 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 $(\frac{1}{2})^r$, 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) [30], 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 [32] 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 Fig. 3, 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.

4.3 One-hop Caching

To improve the efficiency of the search operation, PresenceCloud requires a caching strategy to replicate presence information of users. In order to adapt to changes in the presence of users, the caching strategy should be asynchronous and not require expensive mechanisms 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. Therefor, 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.

4.4 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

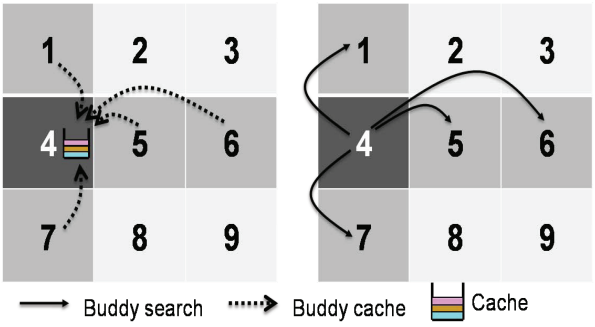


Fig. 4. An example of buddy list searching operations in PresenceCloud

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.

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 Φ of the queried buddy list at most one-hop.*

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.

$B = \{b_1, b_2, \ldots, b_k\}$: 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
S_j: set of $pslist[].id$ of PS node j

Directed Buddy Search Algorithm:

1) A mobile user logins PresenceCloud and decides the associated PS node, q.
2) The user sends a Buddy List Search Message, B to the PS node q.
3) When the PS node q receives a B, then retrieves each b_i from B and searches its user list and one-hop cache to respond to the coming query. And removes the responded buddies from B.
4) If $B = nil$, the buddy list search operation is done.
5) Otherwise, if $B \neq nil$, the PS node q should hash each remaining identifier in B to obtain a grid ID, respectively.
6) Then the PS node q aggregates these $b^{(g)}$ to become a new $B^{(j)}$, for each $g \in S_j$. Here PS node j is the intersection node of $S_q \cap S_g$. 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 Fig. 4, 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.

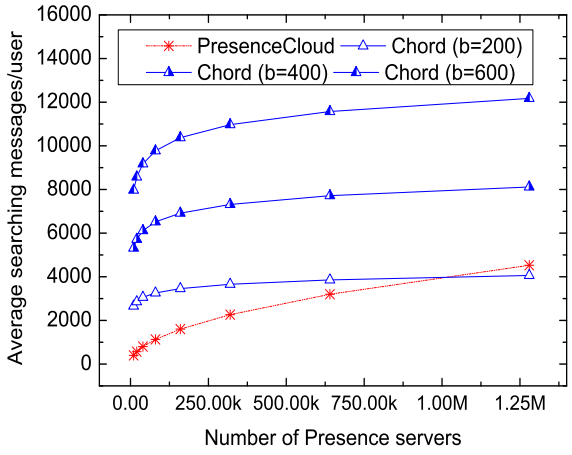


Fig. 5. Average searching messages vs. very large number of PS nodes

5 Cost Analysis

In this section, we provide a cost analysis of the communication cost of PresenceCloud in terms of the number of messages required to search the buddy information of a mobile user. Note that how to reduce the number of inter-server communication messages is the most important metric in mobile presence service issues. The *buddy-list search problem* can be solved by a brute-force search algorithm, which simply searches all the PS nodes in the mobile presence service. In a simple mesh-based design, the algorithm replicates all the presence information at each PS node; hence its search cost, denote by Q_{Mesh}, is only one message. On the other hand, the system needs $n-1$ messages to replicate a user's presence information to all PS nodes, where n is the number of PS nodes. The communication cost of searching buddies and replicating presence information can be formulated as $M_{cost} = Q_{Mesh} + R_{Mesh}$, where R_{Mesh} is the communication cost of replicating presence information to all PS nodes. Accordingly, we have M_{cost} =O(n).

In the analysis of PresenceCloud, we assume that the mobile users are distributed equally among all the PS nodes, which is the worst case of the performance of PresenceCloud. Here, the search cost of PresenceCloud is denoted as Q_p, which is $2 \times (\lceil\sqrt{n}\rceil - 1)$ messages for both searching buddy lists and replicating presence information. Because search message and replica message can be combined into one single message, the communication cost of replicating, $R_p = 0$. It is straightforward to know that the communication cost of searching buddies and replicating presence information in PresenceCloud is $P_{cost} = Q_p = 2 \times (\lceil\sqrt{n}\rceil - 1)$. However, in PresenceCloud, a PS node not only searches a buddy list and replicates presence information, but also notifies users in the buddy list about the new presence event. Let b be the maximum number of buddies of a mobile user. Thus, the worst case is when none of the buddies are registered with the PS nodes reached by the search messages and each user on the buddy list is located on different PS nodes. Since PresenceCloud must reply every online user on the buddy list individually, it is clear that extra b messages must be transmitted. In the worst case, it needs other $2 \times (\lceil\sqrt{n}\rceil - 1)$ messages (when $b >> 2(\lceil\sqrt{n}\rceil - 1)$). When all mobile users are distributed equally among the PS nodes, which is considered to be the worst case, the $P_{cost} = 4 \times (\lceil\sqrt{n}\rceil - 1)$. Consequently, we have the following lemma.

***Lemma** 6: For each buddy searching operation in PresenceCloud, the maximum communication cost of the buddy list search problem is $O(\sqrt{n})$, where n is the number of presence servers.*

Next, we discuss the search cost of the DHT-based presence architecture. We make the following assumptions to simplify the analysis: 1) the presence information of a mobile user is only stored in one PS node (i.e. no replication). Note that in some DHT systems, replicating data increases both the node workload and the maintenance complexity. 2) all mobile users are uniformly distributed in all PS nodes. Although our analysis is based on the Chord [33], it can be extended to other DHTs. Let n be the total number of PS nodes in a Chord. Chord nodes maintains $logn$ neighbors, *i.e.*, finger table, to provide a O($logn$) lookup operations. However, the lookup operation in DHT systems is based on exact-matching, each buddy must be searched one by one, the total search complexity of DHT is equal to D_{cost} = O($b \times \log n$). Note that some replica algorithms [34] are proposed for DHT systems, but these algorithms also increase the complexity of DHT.

Example. We Consider the following simple example to illustrate the efficiency of PresenceCloud. Assume that there are 1024 (n=1024) PS nodes in PresenceCloud and the maximum number of buddies is 100 (b=100). When a mobile user queries, the expected value of the number of messages which a PS node involves is less than $(4 \times (\lceil\sqrt{1024}\rceil - 1)) = 124$. It means that our PresenceCloud saves (124/1023) = 88% communication cost over the mesh-based approach. Then, we also have an example of DHT-based presence system, we assume that there are 1024 PS nodes in the DHT-based presence system and the maximum number of buddies is 100. The maximum number of messages which a PS node involves is $(100 \times \log_2 1024)$

TABLE 1
Presence Architecture Comparison

	Mesh	PresenceCloud	DHT-based
Search	O(n)	O($\sqrt{n}$)	O($b \times \log n$)
Replicas	O($\|U\|$)	O($\sqrt{n} \times u$)	O(u)
Latency	one hop	two hops	$\log n$ hops
Message Size	b	b/n	b/n
Message Reply Hops	O(1)	O(1)	O($\log n$)
Maintenance Overhead	O(n)	O($\sqrt{n}$)	O($\log n$)

= 1000 per user querying operation in the worst case (each buddy of the user is attached to different PS nodes).

Fig. 5 plots the average number of searching messages per searching operation in various very larger number of PS nodes, where b is the number of buddy sizes. As expected, the average message transmissions of Chord-based design increases with the buddy size. The reason is that based on the traditional DHT protocol, each buddy should be treated as a key by hash functions for routing operation in most of DHTs. And it also needs at most b messages for reporting the searching results to the source PS node. We can see that in the case (b=200), when the number of PS nodes grows, PresenceCloud performs better performance under one million PS nodes than Chord-based. However, PersenceCloud outperforms than Chord-based approach when the buddy size grows larger. Note that mesh architecture generates much more searching messages per user than the other architectures. It is thus difficult to put the comparisons in a single figure.

We summarize the comparison of different schemes in Table 1. The columns show the different schemes. The label "Search" means the maximum number of messages sent by a PS node when a mobile user joins; label "Replica" means the maximum number of buddy replicas in a PS node. label "Latency" means the buddy search latency, we quantify this metric by the diameter of the server overlay. This is reasonable because, in general, the search latency is dominated by the diameter of the overlay. label "Message Size" means the average number of required buddy in a message. label "Message Reply Hops" means that the maximum hop counts of a message relayed by PS nodes. label "Maintenance Overhead" means the number of PS nodes maintained by a PS node. In Table 1, none of the schemes is a clear winner. The mesh-based approach achieves good search latency at the expense of the other metrics. Our PresenceCloud yields a low communication cost in large-scale server architecture and small search latency. Note that u is denoted the average number of mobile users attached to a PS node and b is the number of buddy sizes. Meanwhile, the DHT-based method provides good features for low replica load, however it comes at a prices of increased searching latency.

6 Performance Evaluation

In this section, we present the details of the framework used for the experiments. Our implementation of the network simulator and the related architectures, including a Mesh-based, PresenceCloud and a Chord-based presence server architecture, was written in Java. The packet-level simulator allows us to perform tests up to 20,000 users and 2,048 PS nodes, after which simulation data no longer fit in RAM and makes our experiments difficult. In our experiments, the simulator first goes through a warming-up phase to reach the network size (both PS nodes and users), and the simulator starts of the 1,800 seconds test after the measurement approach has stabilized (the stabilized time is based on the system size).

We apply two physical topologies to simulate Internet networks. 1) King-topology: This is a real Internet topology from the King data set. The King data set delay matrix is derived from Internet measurements using techniques that described by Gummadi *et al* [35]. It consists of 2,048 DNS servers. The latencies are measured as RTTs between the DNS servers. 2) Brite-topology: This is an AS topology generated by the BRITE topology generator [36] using the Waxman model where alpha and beta are set to 0.15 and 0.2, respectively. In addition, HS (size of one side of the plane) is set to 1,000 and LS (size of one side of a high-level square) is set to 100. Totally, the Brite-topology consists of 1,000 nodes. In Fig. 6, we show the CDF of the King-topology and the CDF of the Brite-topology.

The simulated topology places every PS node at a position on the King-topology or the Brite-topology, chosen uniformly at random. Note that our simulations involve networks of less than 2,048 PS nodes, we use a pairwise latency matrix derived from measuring the inter-PS node latencies. And each mobile user also uniformly is attached to a random PS node, the propagation delay between mobile user and PS node is randomly assigned in the range [1,20] (ms). The King-topology is assumed as the default IP network topology. Every simulation result is the average 20 runs. The average delay of King-topology is 77.4 milliseconds and 96.2 milliseconds in the Brite-topology. Therefore, the number of users is set to be 20,000, unless otherwise specified.

Experiments were preformed on a Intel 2.8GHz Pentium machine with 4G RAM. The rest of this section is organized as follows. In Section 6.1, we discuss the three important criteria using in the evaluation. Finally, we report the performance results of the three server architectures.

6.1 Performance Metrics

Within the context of the model, we measure the performance of server architectures using the following three metrics: *1) Total Searching Messages:* This represents the total number of messages transferred between the query initiator and the other PS nodes during the simulation time. This is meat and potatoes metric in our experiments, since it is widely regarded to be critical in a mobile presence service that we discussed both in the Section 3 and the Section 5. *2) Average Searching Messages per-arrived user:* The number of searching messages used per arrived user. Moreover, this metric is independent of user arrival pattern.

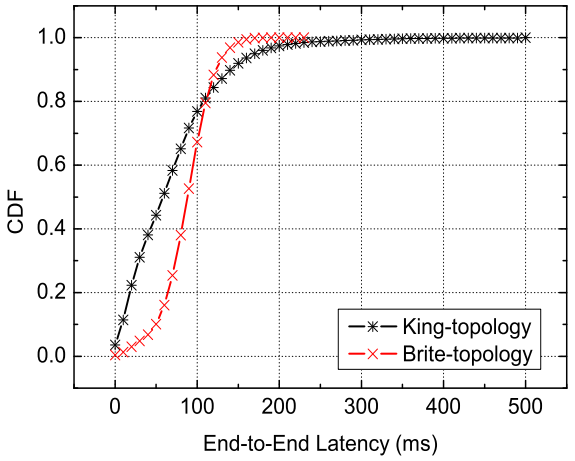


Fig. 6. End-to-end latency distribution over all pairs of King topology and Brite-topology

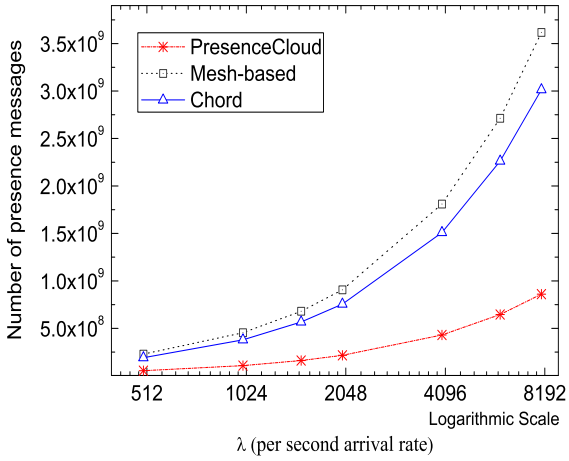


Fig. 7. The total message transmissions during simulation time (1,800s). (The x axis of this figure is in logarithmic scale)

3) Average Searching latency: This represents that average buddy searching time for a joining mobile user. This metric is a critical metric for measuring the search satisfaction of mobile presence services.

6.2 Simulation Results

We first evaluate and compare the three server architectures by considering the total buddy searching messages metric. We instantiated a server network of 256 PS nodes in our simulator, and ran a number of experiments to investigate the effect of scalability of PS nodes on involved searching messages. More precisely, we varied the user arrival rate from 100 per second to 8,000 per second to explore the relation between user arrival rate and the total searching messages. In this test, the number of buddies is set to 100.

Fig. 7 depicts the total number of searching message transmissions during simulation time (1,800 seconds) under various rates of user arrival patterns (100 to 8,000 per second). As the analytical results we discussed in Section 6, we shows that for a given number of PS nodes, the total number of searching messages is dominated by the user arrival rate (λ) significantly. In Fig. 7, the total number of searching messages significantly increased as the user arrival rate increased. We could see that PresenceCloud outperforms all other designs. Mesh-base and Chord both require an enormous number of messages for searching buddy lists in higher user arrival rates. However, vast message transmissions may limit the scalability of the server architecture in mobile presence services.

Fig. 8 shows the average number of searching message transmissions during simulation time (1,800 seconds) under various rates of user arrival patterns (100 to 8,000 per second). As shown in Fig. 8, the average number of searching message transmissions is independent of user arrival pattern. Increasing the rate of user arrival pattern does not increase the average searching message transmissions. For each design, the number of message transmissions is bounded as shown as Section 5. Our PersenceCloud

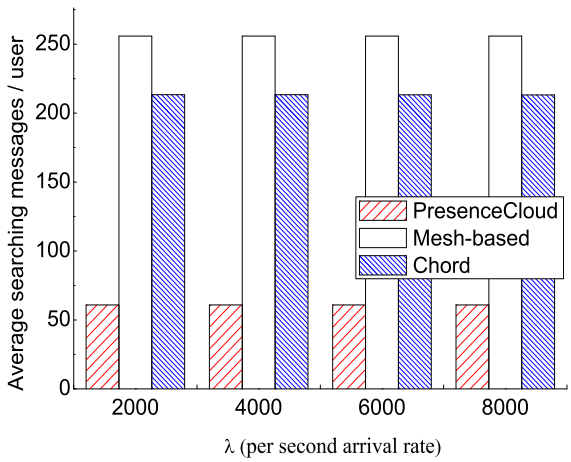


Fig. 8. The average message transmissions per searching operation

requires the least message transmissions. But mesh-based requires O(n) searching complexity (note that the number of PS node is set to 256), the experimental results fit our analysis in the Section 5. Chord-based design performs second highest message transmissions per searching operation. However, if the server architecture is not designed well, the scalability problem of servers may limit itself to scale more than thousands size, hence a poor server architecture may not support a very large number of servers.

In order to study the scalability of server architecture designs to the number of severs, we ran experiments in which the user arrival rate is fixed to 2,000 per second and the number of buddies is set to 120. In these experiments, we increase the number of presence sever nodes from 32 to 2,048. Fig. 9 plots the average number of searching messages per searching operation in various number of PS nodes. As expected, the average message transmissions of PresenceCloud increases gradually with the number of servers. However, the average message transmissions of

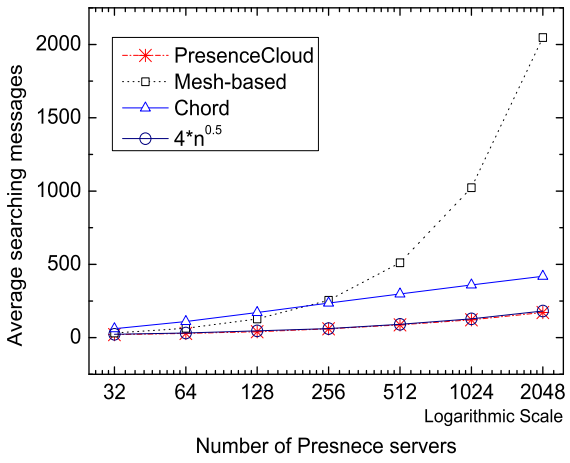


Fig. 9. Average searching messages vs. number of PS nodes (The x axis of this figure is in logarithmic scale)

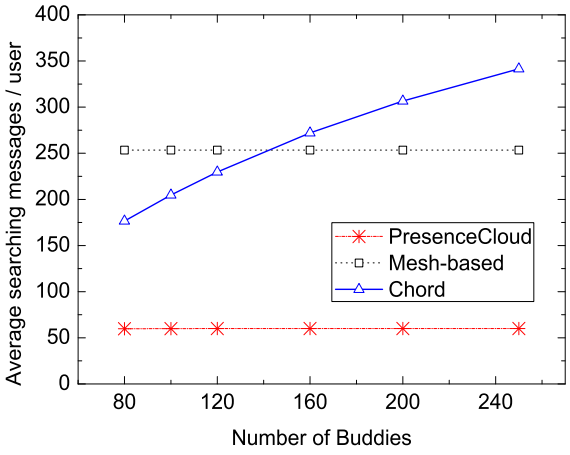


Fig. 10. Average searching messages vs. number of buddy in a 256 PS nodes system

PresenceCloud is bounded by $4 \times \sqrt{n}$. Recall that we had shown the searching complexity of our PresenceCloud in the Section 5. This results suggest good scalability with the number of servers for server architecture design. The simulation results also verify our analysis. Moreover, the mesh-based performs the poorest performance than other sever architecture designs. It requires O(n) searching complexity. Chord-based performs logarithmical performance in this metrics. Generally, the number of average message transmissions grows slowly with the network size in Chord-based and PresenceCloud designs.

In the following we studied the scalability of server architecture designs while varying the number of buddies per mobile user. We ran experiments in which the number of PS nodes is set to 256 and the user arrival rate is fixed to 2,000 per second. In these experiments, we increase the number of buddies per user from 80 to 250. Fig. 10 plots the average number of searching messages per searching operation in various number of buddy per mobile user. As expected, the average message transmissions of PresenceCloud and mesh-based are not impacted by the buddy size. However, the average message transmissions of Chord-based design increases with the buddy size. The reason is that based on the traditional DHT protocol, each buddy should be treated as a key by hash functions for routing operation in most of DHTs. And it also needs at most b messages for reporting the searching results to the source PS node.

Next, we investigate the search satisfaction of server architecture designs. We use our simulator to study the buddy searching latency while varying the number of PS nodes. The simulation environment is set as Fig. 9. As shown as Fig. 11, for PresenceCloud, the buddy searching latency grows gently with the number of PS nodes. However, the buddy searching latency of mesh-based design is significantly better than PresenceCloud. The reason is that, by using the mesh-based design, every PS node can retrieve all desired buddy information in its current replica and return the presence information of buddy to user in

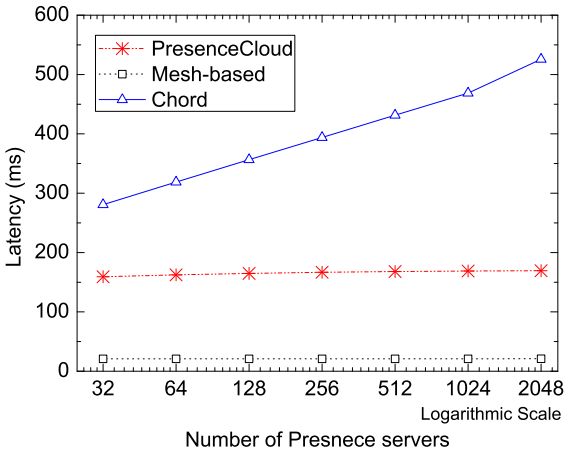


Fig. 11. Average buddy searching latency vs. number of PS nodes (The x axis of this figure is in logarithmic scale)

one hop RTT, and the one hop RTT is quite small in our assumption. PresenceCloud, on the other hand, needs to retrieve all available replicas from its neighbors, which affects the buddy search time. Although the mesh-based design achieves a faster buddy search time and a higher replica hit ratio than PresenceCloud, it sacrifices the scalability of the server architecture in mobile presence services. Under the Chord-based design, a search operation may need to visit a logarithmic number of PS nodes to find the buddies of users. Thus, for latency-sensitive applications, DHT-based designs may be unsuitable for mobile presence services due to their high lookup costs [37].

We also studied the buddy searching latency of server architecture designs while varying the number of buddies. The simulation environment is set as Fig. 10. In these experiments, we increase the number of buddies per user from 80 to 240. Fig. 12 depicts the buddy searching latency with the addition of buddy until 240. As shown in Fig. 12, in all designs, the buddy searching latency is not impacted

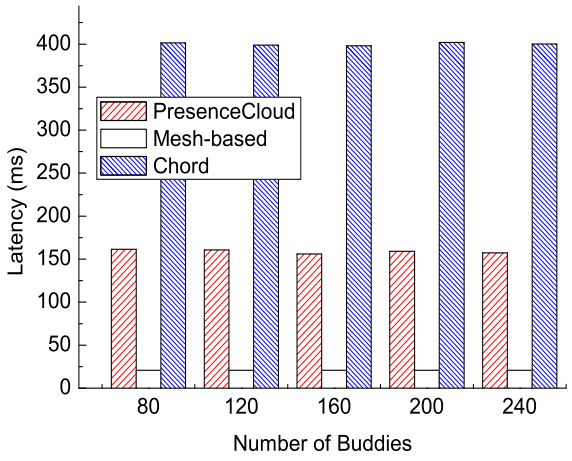


Fig. 12. Average buddy searching latency vs. number of PS nodes

by the number of buddies. The search latency is dominated by the diameter of the overlay, thus the buddy searching latency does not grows with the number of buddies. Clearly, it is a tradeoff, the experiment results show that mesh-based design performs best search satisfaction, but suffers heavily communication cost. However, our PresenceCloud reduces the significantly communication cost without sacrificing search satisfaction extremely.

Note that the buddy searching latency is a critical metric for measuring the search satisfaction of a mobile presence service. To the best of our knowledge, we are not aware any study about the buddy searching latency in mobile presence services. However, in our survey, we noticed that the average DNS lookup latency was 255.9 ms that reported by Ramasubramanian *et al.* [38]. The results is estimated in a large scale DNS in Planet Lab. This report could become a solid reference material for user satisfaction. Compared to the DNS lookup results in the article, the buddy searching latency of PresenceCloud is tolerable. With high probability, we could expect that our PresenceCloud appeases the user satisfaction basically.

Fig. 13 plots that the relative performance of various server architecture designs in the buddy searching latency for two different network topologies. We ran experiments in which the number of PS nodes is set to 512, the user arrival rate is fixed to 2,000 per second, and the number of buddies is set to 200. Fig. 13(a) shows the latency distribution results for the king-topology, while Fig. 13(b) shows the results for the Brite-topology specified earlier. As shown in the two Figures 13(a) and 13(b), the mesh-based design takes much less time to search buddy list than all two other designs under both King and Brite topologies. With the King-topology, the curve for Chord-based design has a flatter tail. This is due to the fact that the distribution of physically link latency in king-topology is more skewed. Note that the mean delay in King-topology is 77.4 milliseconds and is 96.2 milliseconds in the Brite-topology. Thus the expected search latency for PresenceCloud is about 150 milliseconds in King-topology and is approximately 200 milliseconds in Brite-topology. However, the high delay results of Chord-based design are caused by searches whose hops follow high delay paths.

7 Discussions

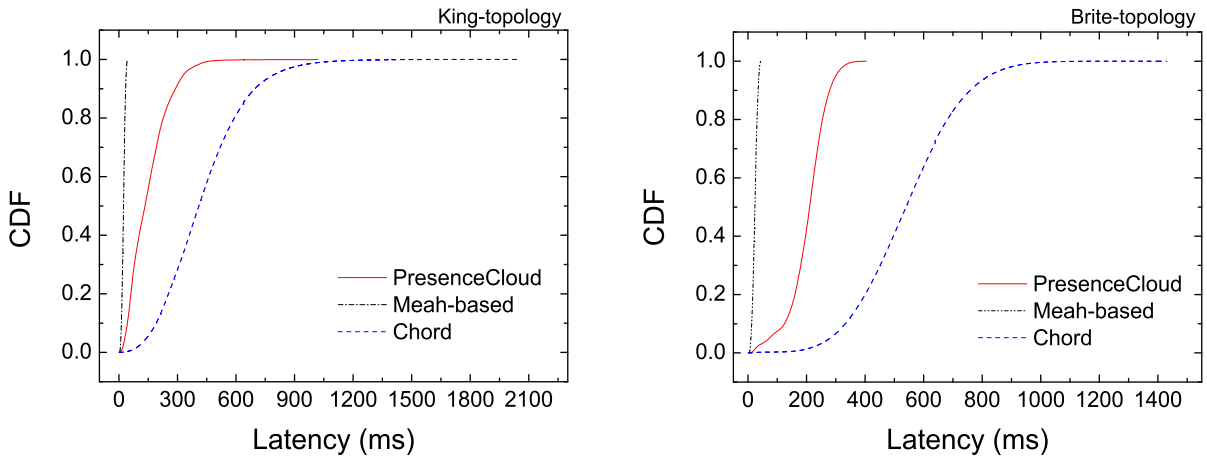
A number of issues require further consideration. Our current PresenceCloud does not address the communication security problem, and the presence server authentication problem, we discuss the possible solutions as follows. The distributed presence service may make the mobile presence service more prone to communication security problems, such as malicious user attacks and the user privacy. Several approaches are possible for addressing the communication security issues. For example, the Skype protocol offers private key mechanisms for end-to-end encryption. In PresenceCloud, the TCP connection between a presence server and users, or a presence server could be established over SSL to prohibit user impersonation and man-in-the-middle attacks. This end-to-end encryption approach is also used in XMPP/SIMPLE protocol.

The presence server authentication problem is another security problem in distributed presence services. In centralized presence architectures, it is no presence server authentication problem, since users only connect to an authenticated presence server. In PresenceCloud, however, requires a system that assumes no trust between presence servers, it means that a malicious presence server is possible in PresenceCloud. To address this authentication problem, a simple approach is to apply a centralized authentication server. Every presence server needs to register an authentication server; PresenceCloud could certificate the presence server every time when the presence server joins to PresenceCloud. An alternative solution is PGP web of trust model [39], which is a decentralized approach. In this model, a presence server wishing to join the system would create a certifying authority and ask any existing presence server to validate the new presence server's certificate. However, such a certificate is only valid to another presence server if the relying party recognizes the verifier as a trusted introducer in the system. These two mechanisms both can address the directory authentication problem principally.

In additional, the user satisfaction of mobile presence service is another search issue. Several studies have investigated the issues of user satisfaction in several domains, including VOIP [40], WWW search engine [41]. To the best of our knowledge, there is no study of exploring the user satisfaction issues, such as search response time, search precise, etc, about mobile presence services. Given the growth of social network applications and mobile device computing capacity, it is an interesting research direction to explore the user satisfaction both on mobile presence services or mobile devices.

8 Conclusion

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



(a) King-topology

(b) Brite-topology

Fig. 13. The CDF of buddy searching latency in King and Brite topology

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.

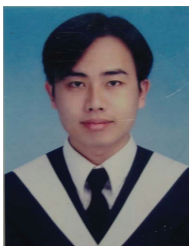
ACKNOWLEDGMENT

The work was supported in part by the National Science Council of Taiwan, R.O.C., under Contracts NSC99-2221-E-001-013-MY3.

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/simple-charter.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] A. 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.
[25] W.-E. Chen, Y.-B. Lin, and R.-H. Liou, "A weakly consistent scheme for ims presence service," *IEEE Transactions on Wireless Communications*, 2009.
[26] N. Banerjee, A. Acharya, and S. K. Das, "Seamless sip-based mobility for multimedia applications." *IEEE Network*, vol. 20, no. 2, pp. 6–13, 2006.
[27] P. Bellavista, A. Corradi, and L. Foschini, "Ims-based presence service with enhanced scalability and guaranteed qos for interdomain enterprise mobility," *IEEE Wireless Communications*, 2009.
[28] A. Houri, E. Aoki, S. Parameswar, T. Rang, , V. Singh, and H. Schulzrinne, "Presence interdomain scaling analysis for sip/simple," *RFC Internet-Draft*, 2009.
[29] M. Maekawa, "A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems," *ACM Transactions on Computer Systems*, 1985.
[30] D. Eastlake and P. Jones, "Us secure hash algorithm 1 (SHA1)," *RFC 3174*, 2001.
[31] M. Steiner, T. En-Najjary, and E. W. Biersack, "Long term study of peer behavior in the kad DHT," *IEEE/ACM Trans. Netw.*, 2009.
[32] K. Singh and H. Schulzrinne, "Failover and load sharing in sip telephony," *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, July 2005.

[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet," *IEEE/ACM Tran. on Networking*, 2003.
[34] X. Chen, S. Ren, H. Wang, and X. Zhang, "Scope: scalable consistency maintenance in structured p2p systems," *Proc. of IEEE INFOCOM*, 2005.
[35] K. P. Gummadi, S. Saroiu, and S. D. Gribble., "King: Estimating latency between arbitrary internet end hosts," *Proc. of ACM IMW*, 2002.
[36] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: An approach to universal topology generation," *Proc of ACM MASCOTS*, 2001.
[37] R. Cox, A. Muthitacharoen, and R. T. Morris, "Serving DNS using a peer-to-peer lookup service," *Proc. of IPTPS*, 2002.
[38] V. Ramasubramanian and E. G. Sirer, "Beehive: 0(1) lookup performance for power-law query distributions in peer-to-peer overlays," *Proc. of USENIX NSDI*, 2004.
[39] A. Abdul-Rahman and S. Hailes., "A distributed trust model," *Proc. of the workshop on New security paradigms*, 1997.
[40] K.-T. Chen, C.-Y. Huang, P. Huang, and C.-L. Lei, "Quantifying skype user satisfaction," *Proceedings of ACM SIGCOMM*, 2006.
[41] P. Anick, "Using terminological feedback for web search refinement: a log-based study," *Proceedings of ACM SIGIR conference on Research and development in informaion retrieval*, pp. 88–95, 2003.



Chi-Jen Wu is a Ph.D. student in the EECS department of National Taiwan University since September 2007. Chi-Jen also is a research assistant at the Institute of Information Science of Academia Sinica since October 2004. Chi-Jen received his M.S. degree in Communication Engineering from National Chung Cheng University, Taiwan in 2004. His research interests include Anycasting, Peer-to-Peer systems and Mobile networks. He is a student member of the ACM.

Jan-Ming Ho received his Ph.D. degree in electrical engineering and computer science from Northwestern University in 1989. He received his B.S. in electrical engineering from National Cheng Kung University in 1978 and his M.S. at Institute of electronics of National Chiao Tung University in 1980. Dr. Ho joined the Institute of Information Science, Academia Sinica as associate research fellow in 1989, and was promoted to research fellow in 1994. He visited IBM's T. J. Watson Research Center in summer 1987 and summer 1988, and the Leonardo Fibonacci Institute for the Foundations of Computer Science, Italy in summer 1992. In 2004-2006, he was jointly appointed by National Science Council, Taiwan, where he served as Director General of Division of Planning and Evaluation. He was Associate Editor of IEEE Transaction on Multimedia. His research interests cover the integration of theory and applications, including information retrieval and extraction, knowledge management, combinatorial optimization, multimedia network protocols and their applications, web services, bioinformatics, and digital library and archive technologies. Dr. Ho also published results in VLSI/CAD physical design.

Ming-Syan Chen received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, and the M.S. and Ph.D. degrees in Computer, Information and Control Engineering from The University of Michigan, Ann Arbor, MI, USA, in 1985 and 1988, respectively. He is now a Distinguished Research Fellow and the Director of Research Center of Information Technology Innovation (CITI) in the Academia Sinica, Taiwan, and is also a Distinguished Professor jointly appointed by EE Department, CSIE Department, and Graduate Institute of Communication Eng. (GICE) at National Taiwan University. He was a research staff member at IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA from 1988 to 1996, the Director of GICE from 2003 to 2006, and also the President/CEO of Institute for Information Industry (III), which is one of the largest organizations for information technology in Taiwan, from 2007 to 2008. His research interests include databases, data mining, mobile computing systems, and multimedia networking, and he has published more than 270 papers in his research areas. In addition to serving as program chairs/vice-chairs and keynote/tutorial speakers in many international conferences, Dr. Chen was an associate editor of IEEE TKDE and also JISE, is currently on the editorial board of Very Large Data Base (VLDB) Journal, Knowledge and Information Systems (KAIS) Journal, and International Journal of Electrical Engineering (IJEE), and is a Distinguished Visitor of IEEE Computer Society for Asia-Pacific from 1998 to 2000, and also from 2005 to 2007. He holds, or has applied for, eighteen U.S. patents and seven ROC patents in his research areas. He is a recipient of the NSC (National Science Council) Distinguished Research Award, Pan Wen Yuan Distinguished Research Award, Teco Award, Honorary Medal of Information, and K.-T. Li Research Breakthrough Award for his research work, and also the Outstanding Innovation Award from IBM Corporate for his contribution to a major database product. He also received numerous awards for his research, teaching,inventions and patent applications. Dr. Chen is a Fellow of ACM and a Fellow of IEEE.