IEICE TRANS. INF. & SYST., VOL.E89–D, NO.2 FEBRUARY 2006

PAPER Special Section on Parallel/Distributed Computing and Networking

PChord: Improvement on Chord to Achieve Better Routing Efficiency by Exploiting Proximity

Feng HONG[†a)], **Minglu LI**[†], **Minyou WU**[†], ***and*** **Jiadi YU**[†], ***Nonmembers***

SUMMARY Routing efficiency is the critical issue when constructing peer-to-peer overlay. However, Chord has often been criticized on its careless of routing locality. A routing efficiency enhancement protocol on top of Chord is illustrated in this paper, which is called PChord. PChord aims to achieve better routing efficiency than Chord by exploiting proximity of the underlying network topology. The simulation shows that PChord has achieved lower RDP per message routing.

key words: *peer-to-peer, routing, Chord, proximity routing, RDP*

1. Introduction

Several overlays (Tapestry [1], Pastry [2], CAN [3] and Chord [4]) provide this kind of location-independent routing substrate for large scale peer-to-peer applications. These overlays can be viewed as providing a scalable, fault-tolerant distributed hash table, in which any item can be located within a bounded number of routing hops, using a small per-node routing table.

Routing efficiency is a key performance metric for these overlay infrastructures as well as other distributed object location and routing systems. All overlays mentioned above have achieved a number of polylogarithmic overlay hops per message routing. However, even if the object is near the source of a query, it is often the case that one or more hops through the overlay will be needed for the object to be found. Since a node with complete routing knowledge could reach this data with a simple direct hop through IP, extra overlay hops may cause a severe relative blowup in the location time of the query, compared to the minimum possible. Therefore, routing efficiency should be measured quantitatively by Relative Delay Penalty (RDP) of the query, also known as *stretch*, when the latency of message delivering in the underlying network is concerned. In this paper, we define RDP as the ratio of the distance a query travels through the overlay network to an object and the minimal distance to that object (i.e. through IP). Chord shows poor performance under RDP currently, for it does not consider network proximity at all [5]. As a result, its protocol for maintaining the overlay network is very light-weight, but messages may travel arbitrarily long distances on the Internet for each routing hop.

Meanwhile data and services are mobile and replicated widely for availability, durability, and locality in today's chaotic network. Therefore, object locating problem should be considered as finding the nearest replica for the object query. We introduce the object pointer indirection layer of Tapestry into Chord overlay (i.e. the objects stored in the overlay are pointers to the location of the actual data) and look for ways to make object location in such an overlay efficient. To address the routing problem under such conditions, this paper presents a routing scheme combining proximity routing with basic routing algorithm of Chord, which aims to achieve low RDP on Chord overlay and keeps the light-weight merit of Chord in its maintenance cost. We name this routing efficiency enhancement overlay as PChord.

The rest of this paper is organized as follows. Related work is discussed in Sect. 2. The core design of PChord is illustrated in Sect. 3. The simulation is given out in Sect. 4, which shows the result of routing efficiency of PChord. Conclusion is discussed in Sect. 5.

2. Related Work

Many efforts have been made to improve routing efficiency in decentralized structured peer-to-peer overlays. In general, the most widely used algorithms in structured peer-to-peer overlays for routing locality is topology-based node identifier assignment and proximity neighbor selection.

Topology-based node identifier assignment attempts to map the overlay's logical key space onto the physical network such that neighboring nodes in the key space are close in the physical network. The technique has been successfully used in a version of CAN, and has achieved RDP of $O(1)$ [6], [7]. However, this approach has several drawbacks. First, it destroys the uniform population of the key space, causing load balancing problems in the overlay. Second, the approach does not work well in overlays that use the one-dimensional key space, because the mapping is overlay constrained.

Proximity neighbor selection uses proximity criteria in building node's routing table, but not in choosing the next hop of routing. The entries of routing table are chosen as the topologically nearest among all nodes with node's identifier in the desired portion of the key space [5]. The success of this technique depends on the degree of freedom an overlay protocol has in choosing routing table entries without affecting the expected number of routing hops. In prefix-based protocols like Tapestry [1] and Pastry [2], the upper levels of the routing table allow great freedom in this choice, with

Manuscript received March 31, 2005.
Manuscript revised August 17, 2005.

[†]The authors are with the Department of Computer Science & Engineering, Shanghai Jiao Tong University, 200240 China.
a) E-mail: hongfeng@cs.sjtu.edu.cn

DOI: 10.1093/ietisy/e89–d.2.546

lower levels leaving exponentially less choice. As a result, the expected delay of the first hop is very low, it increases exponentially with each hop, and the delay of the final hop dominates. This technique can lead to RDP of *O(1)* in theoretical analysis too.

The main challenge to this approach is to construct a selection list of all nodes with their identifiers in the desired portion of key space only using locally available information about the whole overlay. After building the selection list, it still needs election operations to choose the nearest neighbor from this list to create one entry of the routing table for the new Tapestry node. So the procedure of building the routing table of new joining node will cause severe communication cost. Another limitation of this technique is that it does not work for overlay protocols like CAN and Chord, which require that routing table entries refer to specific points in the key space.

When a node joins Tapestry overlay, it needs to get the global information to create its own routing table [8]. A new Tapestry node will communicate all Tapestry nodes in the desired portion of the key space to create one entry of the routing table, which sums up as $O(\log^2 N)$ communications of the whole overlay. Therefore, it needs $O(\log^3 N)$ communications to build up the whole routing table for this new Tapestry node. Here N is the number of nodes in the overlay network. On such basis, Tapestry can achieve $O(1)$ RDP in the theoretic analysis. Three optimizations on Tapestry are provided to improve routing efficiency in [9]: publishing to backups, publishing to nearest neighbors, and publishing to the local surrogate. But these methods do not decrease the communication cost of building the routing table for the new Tapestry node.

Pastry uses runtime heuristics in neighbor selection algorithm which needs $O(\log N)$ communications when new node joining, but only ensures that routing table entries are close but not necessarily the closest comparing to Tapestry [5]. Pastry uses heuristics approach to achieve optimal routing table states gradually. i.e. During its lifetime, a Pastry node periodically performs routing table maintenance and improvement by asking other nodes for "better" routing table entries. To solve the problem of the final hop is the longest distance hop of Pastry, closest neighbor prefix assignment is introduced into Pastry in [10]. Closest neighbor prefix assignment is a kind of topology-based node identifier assignment method. However, this approach increases the communication cost of building the routing table for new Pastry node and still does not guarantee optimal routing table states.

Such two methods mentioned above can not be used in Chord to improve routing efficiency. Because Chord adopts a one-dimensional key space and each entry in its routing table is the node whose identifier most closely follows a certain *key* value (numerically) in key space. Meanwhile Chord is light-wight in its maintenance protocol, which is its most important feature. Therefore, the routing efficiency enhancement protocol on top of Chord should add as little cost as possible to Chord's original maintenance cost.

3. Core Design of PChord

This section shows how to achieve routing locality on Chord. First we give a short illustration on Chord protocol. Then we concern on the description of the main modification of PChord to Chord, which includes PChord's extra routing information *proximity list*, special routing algorithm and object pointer indirection layer.

3.1 Preliminary

Chord is one of the typical DHT peer-to-peer overlay [4]. Chord uses a one-dimensional circular key space. The node responsible for the *key* is the node whose identifier most closely follows the *key* (numerically); that node is called the *key*'s *successor*. Routing correctness is achieved with the pointer *successor*. Routing efficiency is achieved with the *finger list* of $O(\log N)$ nodes spaced exponentially around the key space. Routing procedure consists of message forwarding to the node which is closest preceding to the *key* and not past the *key*; routing path length is of $O(\log N)$ hops.

3.2 Proximity Routing on Chord

PChord's main modification to Chord is to include a new *proximity list* into Chord's routing table, i.e. PChord inherits Chord's *successor* and *finger list* to use in PChord's routing algorithm and maintenance algorithm. Therefore, the differences between PChord and Chord are the procedures which *proximity list* takes part in. We will focus our illustration on what is *proximity list*, how to create and maintain *proximity list*, as well as how *proximity list* functions in routing procedure.

3.2.1 Proximity List

Proximity routing is that the routing choice of next hop is based not only on which neighboring node makes the "most" progress towards the *key*, but is also on which neighboring node is "closest" in the sense of latency [11]. *Proximity list* is included in PChord to evaluate the topology of the underlying network. Proximity is weighed by RTT which can be easily got when communications happen between two nodes. RTT is an acronym for Round Trip Time, which is measured as the time it takes for a simple specified message to travel from one PChord node, across the network to another PChord node, and back. An entry in the *proximity list* contains the IP and identifier of the proximate node. The number of entries in the *proximity list* can grow dynamically as long as the latency to the new node is lower than some specified value, which shows the diameter of network partition, e.g. 30 ms for LAN environment. Here network partition is defined as the nodes in the same stub domain, as we use transit-stub model [12] to represent the Internet in this paper. This topology mimics the structure of large networks observed in nature by dividing the graph into two

classes of nodes, called transit nodes and stub nodes. So the upper limit of the length of *proximity list* is the number of the nodes in the same partition of the underlying network which is decided by the network topology. Here the number of the nodes in the same network partition is the number of the nodes in the same stub domain.

The *proximity list* is created and modified under runtime heuristics mechanism. When a new PChord node joins the overlay, it holds an empty *proximity list*. In such condition, the routing procedure of this new PChord node is the same as a common Chord node. This PChord node will find some other PChord nodes near to it with RTT lower than certain predefined value through routing or maintenance message communication. Then it will add such kind of nodes to its *proximity list*. Meanwhile, these two PChord nodes will copy all different entries of the *proximity list* from each other. The length of *proximity list* will increase until the PChord node finds all PChord nodes in the same network partition which it belongs to.

The heartbeat operation for *proximity list* is the same as the heartbeat of *finger list* of Chord. Every entry in the *proximity list* will be probed in a certain interval to make sure that this proximate node is still active. If the node is inactive, the entry will be deleted from the *proximity list*. As the *proximity list* holds the nodes with lowest communication latency, the heartbeat cost for *proximity list* is lower than the cost for the *finger list* in Chord. And it's lower than the cost of probing proximity neighbors in Tapestry or Pastry too, because nodes in *proximity list* are closer than proximity neighbor which must have certain prefix on communication latency. In general, nodes joining and leaving, and heartbeat probing comprises all the cost of overlay maintenance. It can be concluded that PChord keeps the merit of lightweight maintenance cost as Chord does.

```
//ask node n to find the node which holds object with identifier key
n.find_object(key){
    //check whether n holds the object with identifier key
    for i=1 to objects_list.length{
      if(objects_list[i].identifer=key)
        return this;
    }
    //check whether n holds the object pointer
    //which points to the object with identifer key
    for i=1 to object_pointer_list.length{
      if(object_pointer_list[i].identifer=key)
        return object_pointer_list[i].node;
    }
    //if current node's successor is also key's successor,
    //pass the query to current node's successor
    if(key∈(n,successor))
      return successor.find_object(key);
    //forward query for key to next hop n'
    n'=closest_preceding_node(key);
    return n'.find_object(key);
}

//ask node n to find the closest preceding node to key
n.cloest_preceding_node(key){
    //find the closest finger to key
    for i=finger_list.length downto 1{
      if(finger_list[i].identifer∈(n.identifier,key)){
        closest_finger=finger_list[i];
        break;
      }
    }
    //put all proximate nodes preceding to key into temp_list
    for i=1 to proximate_list.length{
      if(proximate_list[i].identifer∈(n.identifier,key))
        temp_list.add(proximate_list[i]);
    }
    //find the closest preceding node to key in temp_list
    closest_proximate_node=closest_preceding(temp_list,key);
    //find the closest preceding node to key
    //between closest_finger and closest_proximate_node
    n'=closest_preceding(closest_finger,closest_proximate_node,key);
    return n';
}
```

Fig. 1 Pseudocode of routing algorithm in PChord.

3.2.2 Routing Algorithms

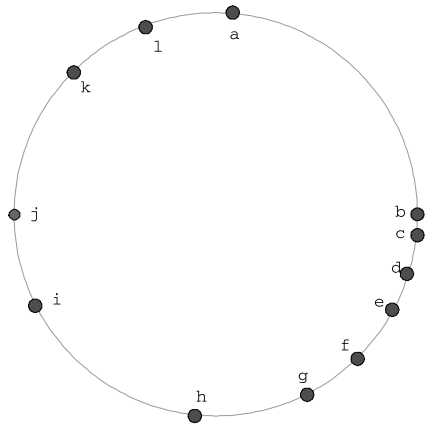
The key modification of routing algorithm in PChord is the choosing of next hop. Next hop is not only decided by the entries in the *finger list*, but also decided by the entries in the *proximity list*. The most closest preceding node to the target *key* in key space will be found out as the next hop from local entries of the union of both *finger list* and *proximity list* of the current node. Then the entry with that identifer will be chosen as the next hop. The routing algorithm is described in Fig. 1.

If the next hop is decided as the entry from the *finger list*, this hop of routing process is the same as Chord's, which comes along to the node with the identifier closer preceding to the target *key* than current node. Otherwise, the next hop is decided as the entry from the *proximity list*. In such condition, the next hop not only follows the proximity link which is of low network latency, but also adopts a hop with identifier closer to target *key*'s *successor* than Chord's hop in key space. i.e. the hop following proximity link plays two roles in routing process, as described in following:

- making the most progress to the target *key* from local routing information
- following the proximity link with low latency

As the routing process is composed of the hops getting closer to target *key*'s *successor* one by one, the routing procedure of PChord will results in less hop number than Chord's routing procedure, for every hop is larger or at least equal in key space in PChord than in Chord. As peer-to-peer nodes are nodes at the edges of the Internet [13], it's of great importance that less hop number can help in getting lower RDP. Because every hop of routing will transit from LAN to Internet backbone and back to LAN again, which is of great communication latency, when the two nodes are in the different network partitions.

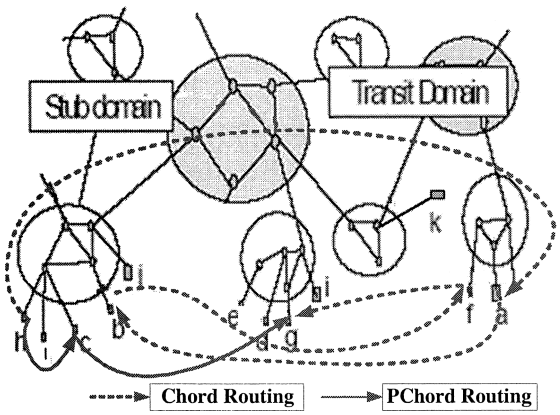
Routing examples are given out to describe how PChord decreases the hop number and RDP in Fig. 2. Figure 2 (a) shows the nodes' identifier distribution on the Chord ring. The full Chord ring is organized with full key space as 64. The exact identifers of these 12 nodes is listed in Fig. 2 (b) with their *finger lists*. Figure 2 (c) shows the network topology of these 12 nodes. The topology is generated



(a)

node	ID	f(5)	f(4)	f(3)	f(2)	f(1)	f(0)	neighbor list
a	1	h	b	b	b	b	b	f
b	17	k	h	f	e	d	c	c,h,j,l
c	18	k	i	g	f	e	d	b,h,j,l
d	19	k	i	g	f	e	e	e,g,i
e	21	k	i	h	f	f	f	d,g,i
f	25	l	i	h	h	g	g	a
g	27	l	j	i	h	h	h	d,e,i
h	33	a	k	i	i	i	i	b,c,j,l
i	42	b	l	k	j	j	j	d,e,g
j	48	b	a	k	k	k	k	b,c,h,l
k	56	f	b	a	l	l	l	null
l	60	h	b	b	a	a	a	b,c,h,j

(b)



(c)

Fig. 2 Routing example of PChord comparing to Chord. (a) The identifiers of nodes on the Chord ring, (b) *finger list* of every node, (c) network topology of the nodes and routing examples of Chord and PChord.

from Transit-Stub Internet Model with 2 Transit domains and 6 Stub domains.

Routing examples of Chord and PChord are illustrated in Fig. 2 (c) too, which is node *h* trying to locate node *g*. For standard Chord overlay, the hops of routing process is the dotted arrowhead line in Fig. 2 (c), which are *h,a,b,f,g* of 4 hops. For PChord, as node *h* holds a *proximity list* of *l,c,b,j* and *h*'s *finger list* is shown in Fig. 2 (b), *h* will choose *c* as its next hop for this routing process, for *c* is closest preceding hop to the target *g* in key space from all *h*'s local routing information. This hop of *h* is chosen from the entry of the *proximity list*, which is a larger hop in key space than *h*'s Chord hop *a*, and follows the proximity link. Then *c* will choose the next hop from its *proximity list* and *finger list* the same as *h* does. Node *g* is in the *c*'s *finger list*, which will be chosen as next hop of *c*. This routing process on PChord is composed of only 2 hops, *h,c,g*, shown with the common arrowhead line in Fig. 2 (c), which is 2 hops less than the routing process on Chord. If the latency of network link is defined as default of GT-ITM simulator [14], which is 1 for the link in the same stub domain or same transit domain, 2 for the link between transit domains and 6 for the link between stub domain and transit domain, the delay of these two routing path can be calculated. The delay of Chord's routing path is 84 and the delay of PChord's routing path is 22. The delay of direct hop by IP from *h* to *g* is 20, so the RDP of Chord's routing process above is 4.1 and the RDP of PChord's routing process is only 1.1. This is because the Chord routing path includes 8 transit-stub links and 4 transit-transit links, while the PChord routing path only contains 2 transit-stub links.

Meanwhile, it shows that the routing path of PChord only passes the same stub domain only once in Fig. 2 (c). When choosing the next hop, the current PChord node can check all its neighbors' identifiers if the length of *proximity list* is the same as the number of the nodes in this network partition. Therefore there will be no possibility of routing path coming back to the same network partition, for the closest preceding node to the target *key* in the same network partition has already been evaluated when choosing the next hop of current PChord node. This feature can be summarized as PChord's routing process prevents the routing hops from jumping back to the same network partition which the current node belongs to, when the length of *proximity list* is the same as the node number of the network partition, i.e. the routing path includes at most one hop in the same network partition.

In general, PChord's routing scheme can result in three optimizations as following:

- Routing path is shorter than Chord's of overlay hop number.
- Routing path passes through proximity link of the underlying network.
- Routing path passes the same partition of the underlying network only once.

This three optimizations will definitely reduce RDP of message routing.

3.3 Object Pointer Indirection Layer

Because there is an additional object pointer indirection layer added on common Chord overlay, the routing problem of PChord is changed to find the nearest replica for the target object. The object pointer indirection layer of PChord consists two main functions: object publishing and object querying.

3.3.1 Object Publishing

PChord nodes (storage servers) publish the fact that they are storing a replica of the object with identifier *key* by routing a publishing message toward *key*'s *successor*. Publishing messages are routed along to the *key*'s *successor* with the routing algorithm as shown in Fig. 1. At each hop, publishing messages deposit object pointers to the PChord node. An object pointer holds two items of object's identifier *key* and the IP of the PChord node holding that object. Each PChord node holds an object pointer list for all object pointers storing on it. This object pointer list is organized by ascending the object pointers' *key* field. And those object pointers with the same *key* will be stored in this object pointer list by ascending the latency between current node and the node published this object pointer. This kind of sort method of the object pointer list ensures the proximity criteria of routing, which is that the first pointer with the target *key* founded in the list will direct to the node most proximate to the current PChord node which holds the target object. The routing algorithm in Fig. 1 has used this technique to deal with object pointer list. Regarding churn of nodes, PChord assumes that object pointers are soft-state as Tapestry does. i.e. object pointers expire and objects must be republished at regular intervals.

3.3.2 Object Querying

Queries for object with identifier *key* route toward *key*'s *successor* until they encounter an object pointer of *key*, then route to the located replica by the pointer. If multiple pointers are encountered, the query proceeds to the closest replica to current node (i.e. the node where the object pointer is found), which is ensured by the organization of object pointer list of PChord node. In the worst case a location operation involves routing all the way to *key*'s *successor* and follows the object pointer to the node which publishes that object. If the desired object is close to the client, the query path will be very likely to intersect the publishing path before reaching *key*'s *successor*.

Especially the routing process of object query doesn't end at the *successor* which holds the object pointer, but ends at the node which holds the object, which results in that this process of routing will be one more hop than the common Chord overlay for the reason that there is an object pointer indirection layer above common Chord. In the following simulation, this one more hop is always calculated when we count hop number and RDP of routing on Chord or PChord.

4. Simulation

In order to perform large-scale and repeatable experiments, we constructed our simulator on GT-ITM transit stub model to provide an event driven network layer that simulates network delays. The transit-stub graph we used for this experiment consists of 1024 stub nodes, with approximately 16 nodes per stub domain. The network latency between nodes are chosen as default as GT-ITM model which has been illustrated in Sect. 3.2.2. And the specified latency for PChord's proximity node is chosen as 6, which causes *proximity list*'s length to increase to the number of nodes in the same network partition during running.

1024 objects has been published twice by the stub nodes randomly on the PChord overlay when every PChord's *proximity list* has achieved its full length. Each PChord node publishes 2 random objects from the set of all published objects. After that, each node locates 2 objects, chosen randomly from the set of all published objects. The experiment is repeated five times using five different transit-stub graphs with the same number of nodes to show that the result is not concerned with the topology of the underly network. The average routing hop number and RDP in the following figures are calculated from all the results of these five different experiments.

Figure 3 shows the comparing of routing efficiency between Chord and PChord overlay when every PChord node achieves its full length of *proximity list*. Figure 3 (a) shows the probability density function (PDF) of the number of overlay hops per object publishing. It shows that PChord decreases the number of overlay hops during object publishing. As the object pointers will be deposited to overlay nodes on each publishing hop, the number of all object pointers in PChord is lower than Chord. There are 12038 pointers in Chord and 9750 pointers in PChord, i.e. the overhead of PChord is 18.52% lower than Chord. Figure 3 (b) shows PDF of the number of overlay hops per query message routing. Figure 3 (c) shows PDF of RDP per query message routing. It shows that there is clear improvement of routing efficiency in PChord. Table 1 shows the quantitative comparison of routing efficiency per message routing between PChord and Chord. It can be concluded that PChord decreases the number of overlay hops and RDP greatly in query message routing comparing to standard Chord overlay, which reflects that *proximity list* has played its role in PChord's routing scheme as expected. Meanwhile, it should be emphasized that this improvement of routing is achieved with less object pointers on PChord, for more object pointers will help to shorten the routing path of object querying.

In the routing process described above, we classify all the query requests according to the distance between query source and target document. The distance is defined as the delay of direct hop by IP from the query source node to the node which holds the queried object. After all these routing hop numbers and RDP have been logged, the average hop number and RDP can be calculated and shown in Fig. 4. It shows that there is great decrease on routing hop number and RDP when the target document is near the query source. This comes from that the PChord node holding the target document will deposit a pointer to its neighbor node when the publishing path passes that neighbor as first hop through proximity link, which will help other node in the same network partition to find that document. When the distance is far from query source to target document, Fig. 4 (a)

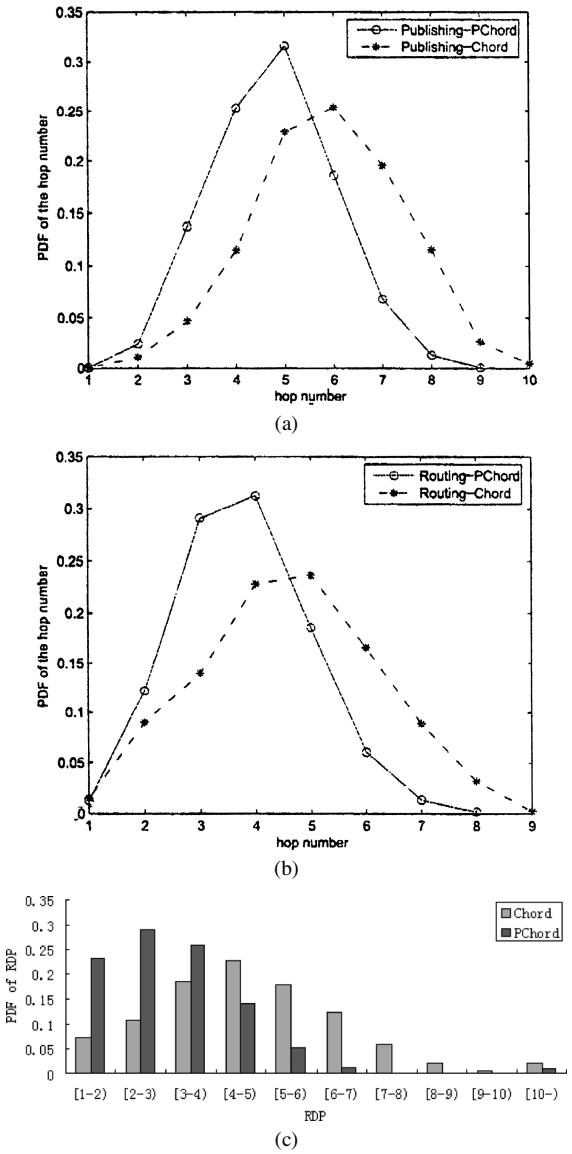


Fig. 3 Comparing routing efficiency between Chord and PChord when full length of *proximity list* on each PChord node. (a) PDF of routing hop number per object publishing, (b) PDF of routing hop number per message routing, (c) PDF of routing RDP per message routing.

Table 1 Quantitative comparison of routing efficiency per message routing between PChord and Chord.

overlay	pointers	average hop number	average RDP
Chord	12038	4.61	4.79
PChord	9750	3.78	2.67

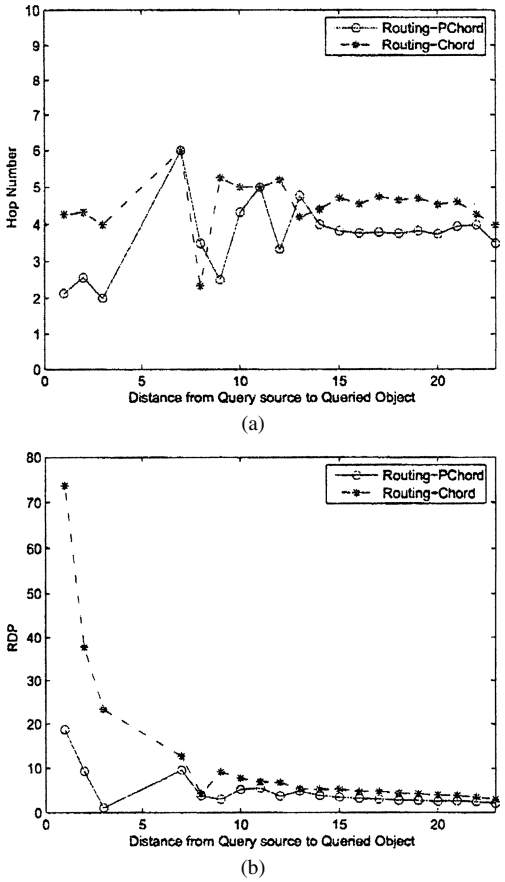


Fig. 4 Routing efficiency as a function of the query source's distance from the queried document. (a) Routing hop number according to the distance from querying node to target document, (b) routing RDP according to the distance from querying node to target document.

shows that both Chord and PChord achieve the feature of scalability i.e. the average hop number doesn't increase. It can also be got that the hop number of PChord is lower than Chord's as expected. Figure 4 (b) illustrates the average RDP of PChord is lower than Chord when the distance between query source and target document is long, too. Combining the result from Fig. 4 (a) and Fig. 4 (b), it proves that PChord increase the routing efficiency by decreasing the hop numbers and limiting the routing path crossing the same network partition only once.

In order to show the help of *proximity list* to routing efficiency clearly, we repeated the above experiments independently on 10 snapshots of PChord overlay in one transit-stub graph, which reflects the evolution of the whole overlay during the process of every node's *proximity list*'s length increasing. These 10 snapshots are distinguished from the sum length of all PChord nodes' *proximity list* in the overlay. We define the snapshot that all PChord nodes' *proximity list* achieving their full length as the reference point. The first snapshot of the overlay is defined as the sum length of all PChord nodes' *proximity list* being about 10% of the reference point. Therefore the other 9 snapshots are the ones when their sum length of all PChord nodes' *proximity list* achieving about 20%, 30% . . . . . 90% and 100% of the reference point. The simulation process is as same as the simulation process above on each snapshot. After the hop number and RDP of each query has been logged, the average hop number and RDP on each snapshot can be calculated.

Figure 5 shows the routing result on these ten snapshots. Point "0" on X axis is the point of snapshot when there is no entry in all PChord nodes' *proximity list*. This overlay's routing process is the same as common Chord's in

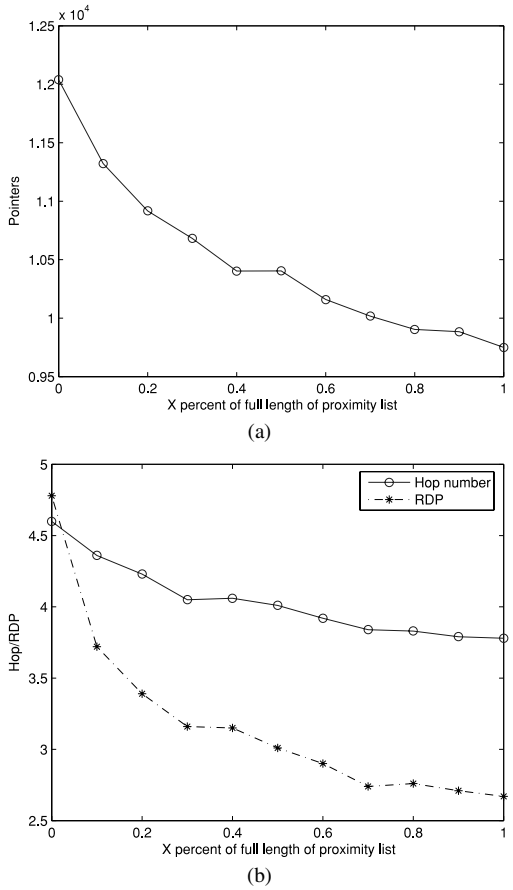


Fig. 5 Overhead and routing efficiency as a function of the sum length of all PChord nodes' *proximity list*. (a) Number of pointers of the whole overlay as a function of the sum length of all PChord nodes' *proximity list*, (b) routing hop number and RDP as a function of the sum length of all PChord nodes' *proximity list*.

the snapshot. Figure 5 (a) shows the sum pointers after object publishing processes of all objects on the overlay. Because PChord decreases the hop number in object publishing process with the help of PChord nodes' *proximity list*, the pointers of the whole overlay will keep decreasing when the sum length of all PChord nodes' *proximity list* increases. The pointers all over the overlay reflects the overhead of the overlay. So PChord will decrease its overhead when *proximity list* getting longer.

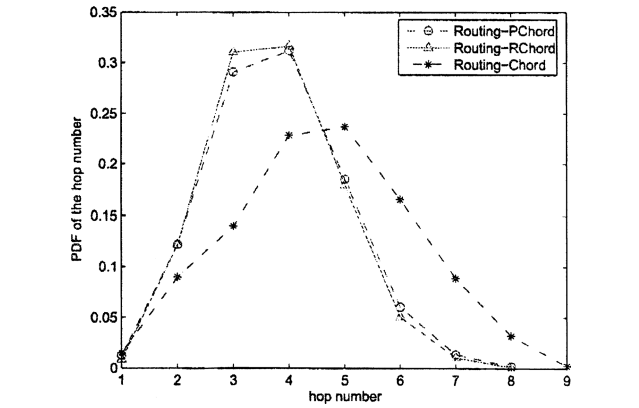
Figure 5 (b) shows the average routing hop number and RDP on these snapshots. It can be concluded that the routing hop number and RDP decrease quickly along with the increasing of the sum length of all PChord nodes' *proximity list*. The hop number curve decreases in about liner fashion and the RDP curve decreases in about an negative exponent curve type. It should be emphasized that these routing efficiency achievement have been got at the same time with low overhead on the whole overlay according to Fig. 5 (a), which definitely reflects the power of *proximity list* of PChord.

However, question which may be brought up here is whether the improvement of routing efficiency only results from the adding of more routing information on Chord. i.e. can the Chord overlay also achieve the routing efficiency like PChord does by adding a list with the same number of entries as PChord's *proximity list*, but these entries are chosen randomly? Another simulation is performed to answer this question clearly. In this simulation, every Chord node is added with a list of randomly chosen entries of nodes from the overlay, and the length of this list is the same as the full length of that node's *proximity list* on PChord overlay. We named this kind of overlay as RChord to distinguish from Chord and PChord in the following descriptions. The experiment process on RChord is the same as the process on PChord overlay. Figure 6 shows the result of RChord comparing to that of PChord's routing efficiency with full *proximity list* on each node above.

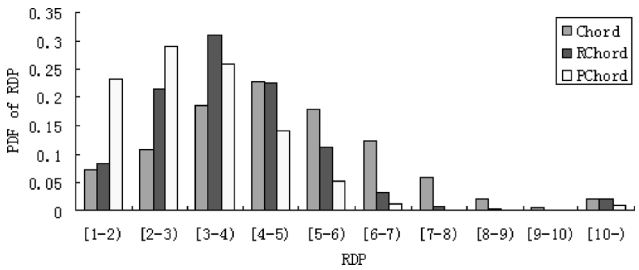
Figure 6 (a) shows that PDF of RChord's hop number per message routing equals to PChord's, which is 3.74 as the average hop number of RChord and the pointers on RChord is 9749. Comparing the avarage hop number of Chord and PChord in Table 1, it illustrates that the extra list on Chord can help decreasing the hop number per message routing, no matter whether the entries in this list is chosen randomly as RChord or chosen specifically as PChord. However, the routing efficiency of RDP is quite different between RChord and PChord. Figure 6 (b) shows that RChord is better than Chord on RDP per message routing, but is worse than PChord. The average RDP per message routing is 3.54 in RChord, while it's 2.67 in PChord and 4.79 in Chord, which proves that: a) less hop number will help decreasing the RDP per message routing which is illustrated by comparing between RChord and Chord, b) *proximity list* helps a lot in decreasing RDP when comparing PChord to RChord, for the full length *proximity list* will guarantee that routing path passing through proximity link of the underly network and passing the same partition of the underly network only once, which cannot be achieved by the randomly chosen list of RChord.

Figure 6 (c) and Fig. 6 (d) shows the average routing hop number and RDP as a function of the query source's distance from target document. RChord's average routing hop number and RDP are both lower than Chord's, which shows extra routing information plays its role in routing process. Though RChord's average routing hop number curve is like PChord's when the distance between query source and target document is long, RChord's average routing hop and RDP is much higher than PChord's when the distance between query source and target document is close. This is because PChord node holding the target document will deposit a pointer to its neighbor node, when the publishing path passes that neighbor as first hop through proximity link. This will help other node in the same network partition to find that document, which is the feature of *proximity list* that RChord does not hold.

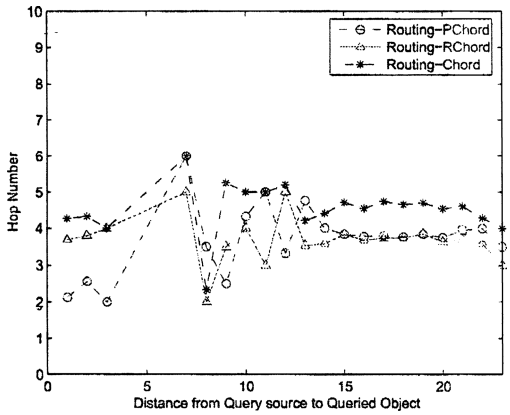
Summarizing up all these simulation results, we can further classify the three features of PChord's improving routing efficiency of RDP into two categories, according to the roles that *proximity list* plays in routing procedure:



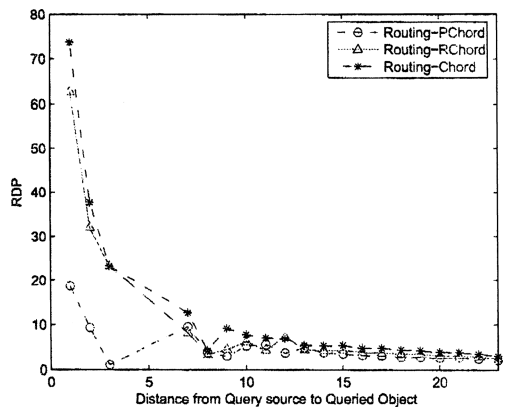
(a)



(b)



(c)



(d)

Fig. 6 Comparing routing efficiency among PChord, RChord and Chord. (a) PDF of routing hop number per message routing, (b) PDF of RDP per message routing, (c) average routing hop number as a function of the query source's distance from the queried document, (d) average RDP as a function of the query source's distance from the queried document.

- *proximity list* adds extra routing information on Chord node, which results in that the hop number of message routing is less than common Chord protocol.
- *proximity list* reflects the proximity relationship of the topology of the underly network, which causes that 1) routing path passes through the proximity link of the underly network and 2) routing path passes the same partition of the underly network only once.

5. Conclusion

PChord's main routing optimizations are of less overlay hops, passing proximity links of the underly network and passing the same network partition only once in the routing path, which directly results in the lower RDP of routing efficiency. Meanwhile, PChord keeps light-weight maintenance cost as Chord does.

Acknowledgement

This work is supported by National Natural Science Foundation of China (grant No.60433040, No.60442004).

References

[1] B.Y. Zhao, J. Kubiatowicz, and A.D. Joseph, "Tapestry: A resilient global-scale overlay for service deployment," IEEE J. Sel. Areas Commun., vol.22, no.1, pp.41–53, Jan. 2004.
[2] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," Proc. 18th IFIP/ACM International Conference on Distributed Systems Platforms, pp.329–350, 2001.
[3] S. Ratnasamy, P. Francis, and M. Handley, "A scalable content-addressable network," Proc. SIGCOMM, pp.161–172, 2001.
[4] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet," IEEE/ACM Trans. Netw., vol.11, no.1, pp.17–32, 2003.
[5] M. Castro, P. Druschel, Y.C. Hu, and A. Rowstron, "Topology-aware routing in structured peertopeer overlay networks," MSR-TR-2002-82, 2002.
[6] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Topologically-aware overlay construction and server selection," Proc. 21st IEEE INFOCOM, pp.1190–1199, 2002.
[7] S. Ren, L. Guo, J. Song, and X. Zhang, "SAT-Match: A self-adaptive topology matching method to achieve low lookup latency in structured P2P overlay networks," IPDPS, pp.83–92, 2004
[8] K. Hildrum, J. Kubiatowicz, S. Rao, and B.Y. Zhao, "Distributed object location in a dynamic network," Theor. Comput. Syst., no.37, pp.405–440, March 2004.
[9] J. Stribling, K. Hildrum, and J.D. Kubiatowicz, "Optimizations for locality-aware structured peer-to-peer overlays," Report no.UCB/CSD-03-1266, 2003.
[10] R. Winter, T. Zahn, and J. Schiller, "Topology-aware overlay construction in dynamic networks," Proc. 3rd International Conference on Networking (ICN), 2004.
[11] S. Ratnasamy, S. Shenker, and I. Stoica, "Routing algorithms for dhts: Some open questions," Proc. First International Workshop on Peer-to-Peer Systems (IPTPS'02), pp.42–52, 2002.
[12] E.W. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork," Proc. IEEE INFOCOM, pp.594–602, 1996.
[13] http://www.openp2p.com/lpt/a/p2p/2000/11/24/shirky1-whatisp2p.html, 2000.

[14] GT-ITM, http://www.cc.gatech.edu/projects/gtitm/

Feng Hong Ph.D. candidate in the Department of Computer Science & Engineering, Shanghai Jiao Tong University, China. His major research interest is of routing algorithm of peer-to-peer overlay.

Minglu Li professor in the Department of Computer Science & Engineering, Shanghai Jiao Tong University, China. Prof. Li is subeditor of International Journal of Grid and Utility Computing and on edit board of International Journal of Web Services Research. He is in executive committee of Technical Community for Services Computing of IEEE. His major research interests includes grid computing and peer-to-peer computing.

Minyou Wu professor in the Department of Computer Science & Engineering, Shanghai Jiao Tong University, China. He is a member of the ACM and of ISCA, and is a senior member of the IEEE. His research interests include multimedia networking, multimedia systems, and parallel and distributed systems.

Jiadi Yu Ph.D. candidate of Department of Computer Science & Engineering, Shanghai Jiao Tong University, China. His major research interests include peer-to-peer computing and grid computing.