LinkNet: A New Approach for Searching in a Large Peer-to-peer System

Kunlong Zhang, Shan Wang

School of Information, Renmin University of China, Beijing, China, 100872
zhangkl@ruc.edu.cn, suang@public.bta.net.cn

Abstract. Searching a file by its name is an essential problem of a large peer-to-peer file-sharing system. Napster and Gnutella have poor scalability. DHT-based systems do not keep the order of the keys. Skip graphs and SkipNet have too many links. In this paper, we present a new scalable distributed data structure LinkNet for searching in a large peer-to-peer system. In LinkNet, all elements are stored in a sorted doubly linked list, and one node stores many elements. LinkNet uses virtual link to speed search and enhance fault tolerance. Because LinkNet is based on a sorted list, it benefits operations such as range query, bulk loading of data, and merging of two LinkNets.

1 Introduction

For a large peer-to-peer file-sharing system, searching a file by its name is an essential problem. One initial approach is to setup up a server which maps a file name to its location. Napster ([6]) uses this approach. The problem of this approach is that it uses an unscalable central database to index all files. Another initial approach is that a node broadcasts the search request to all its neighbors when it does not find the file in its local database. Gnutella ([7]) adopts this approach. However this approach doesn't scale well because of its bandwidth consumption and unrelated search in many nodes.

To overcome the scalability problem, several algorithms based on a distributed hash table (DHT) approach are presented ([1], [8], [9], [10], [11]). In these algorithms, each node in the system maintains a small routing table to form an overlay network and each data item is associated with a unique key which is hashed to determine which node it will be stored at. When a search request is received by a node that does not store the search key, the node will use its routing table to routing the request to a neighbor which is closer to the key. Because hashing does not keep the order of the keys, DHT systems do not support range queries efficiently.

Two recent papers [2] [3] try to build a peer-to-peer system on the skip list data structure. The paper [2] describes a distributed data structure called skip graphs. In a skip graph, search, insert and delete operations are done in logarithmic time. Because of no hashing, skip graphs support range queries more efficiently than DHT. Skip graphs also are highly resilient to node failures because they have many redundant links among nodes. The paper [3] describes a distributed data structure called SkipNet which is very similar to skip graphs. SkipNet supports controlled data placement and guaranteed routing locality by organizing data primarily by string names. One problem of these two data structure is there are too many links. With *N* resources in the network, there is a total of O(*N*log*N*) links.

From a list to a skip list, and from a skip list to a skip graph, it is true that more links leads to better performance. In a skip graph, a node stores only one data item. But in a file-sharing peer-to-peer system, each node stores many files. Based on these observations, this paper introduces a new scalable distributed data structure LinkNet which is expected to be a powerful competitor to the above methods.

The rest paper is organized as follows: section 2 describes the design of LinkNet; section 3 describes the basic algorithms for LinkNet; section 4 gives the search performance evaluation of LinkNet; and section 5 concludes the paper.

2 LinkNet

To help our discussion, we briefly define some terms first. A peer-to-peer system consists of many **nodes**. Each node has a unique **location**. Typically a node's location is its IP address or domain name. A node stores many data items or **elements**. An element is a file, an object, or one row of a database table. Each element has a **key**. An element is mapped to a **pointer**. A pointer is a pair <*location*, *key*>. Two elements form a **link** if their order is known. If a link is physically stored in main memory or second memory, it is a **physical link**; otherwise it is a **virtual link**.

Our purpose is to find the location of a given key based on list data structure. Because the search can start from any node, a sorted doubly linked list is a nice choice. Figure 1 shows a network-based sorted doubly linked list.

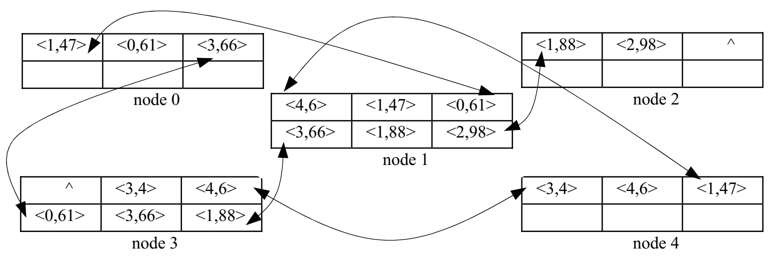

Figure 1. A network-based sorted doubly linked list

Search in a network-based sorted doubly linked list is simple. For example, in figure 1, to find key 98 starting from key 4 goes through key 4, 6, 47, 61, 66, 88, 98 and node 3, 4, 1, 0, 3, 1, 2. This search method is slow. When the search starts from key 4, it is a better choice to use key 66's forward pointer <1, 88> rather than using key 4's forward pointer <4, 6>. In this way the search only goes though key 4, 88, 98 and node 3, 1, 2 as if there is a virtual link <4, 88>. Because following the virtual link skips some keys, the search is faster. This idea can be generalized. For all keys in the same node, one key can share another key's pointers to form virtual links. Then a new data structure is built, it is named as LinkNet. Figure 2 shows a list-based LinkNet corresponding to figure 1. In figure 2, virtual links are marked by dot lines. The virtual links not only speed search, but also enhance fault tolerance. For example, if node 4 fails, it is still possible to find key 98 starting from key 4.

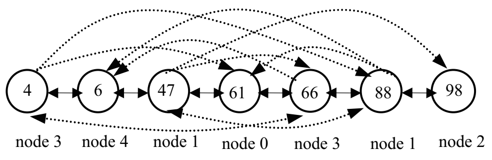

Figure 2. A list-based LinkNet with 7 keys on 5 nodes

A LinkNet is consisted of many nodes. Each node has a local manager to manage its elements. A local manager knows at least one other local manager. Two local managers communicate through message passing. There are no global coordinators, No local manager knows about global information. All global operations are finished by the cooperation of these local managers. In an implementation of list-based LinkNet, a local manager can store three pointers for an element: a pointer that points to the element, a forward pointer, and a backward pointer. There is a link between the elements of two pointers. If this link is not in the sorted doubly linked list, it is a virtual link; otherwise it is a physical link. When a node receives a search request, the local manager does a local search first. If the search key is not found, the local manager forwards the request to next closer node following a selected link (virtual or physical).

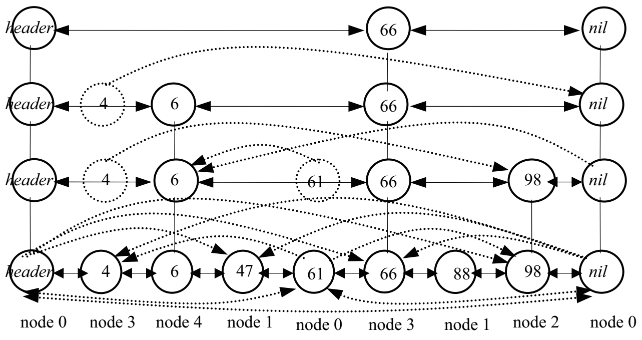

Figure 3. A skip-list-based LinkNet with 7 keys on 5 nodes

The disadvantage of list-based LinkNet is that its performance depends on the distribution of elements heavily. If each node stores only one element, the list-based LinkNet is degenerated into a network-based sorted doubly linked list. Skip-list-based LinkNet is built to avoid this problem. Figure 3 shows a skip-list-based LinkNet corresponding to figure 2 (not all virtual links are showed). If there are M elements in the network, a list-based LinkNet needs a total of O(M) space, and a skip-list-based LinkNet needs an expected total of O(M) space.

Because LinkNet keeps the order of the keys, it provides better support for range query than DHT. LinkNet also benefits many other operations. For example, it is more efficient than DHT to merge two LinkNets, split a LinkNet, and bulk load data into a LinkNet. The penalty is that LinkNet needs more space, and must do concurrent operations carefully.

3 Algorithms for LinkNet

In this section, we focus on the search, insert and delete operations of skip-list-based LinkNet. The corresponding operations of list-based LinkNet are similar to the description of this section.

In LinkNet, a node can store many elements. These elements are stored in a sorted doubly linked list. A new LinkNet has only two special elements: *header* which is given a key less than any legal key, and *nil* which is given a key greater than any legal key. Each element has a level which is a random number capped at *MaxLevel*. It is not necessary that all elements are capped at the same value of *MaxLevel*. Our approach for concurrent operations is similar to the approach in paper [5], thus we omit the proofs of correctness in this paper.

To describe the operations for LinkNet, we denote node as $\boldsymbol{u}$, $\boldsymbol{v}$ and element as $\boldsymbol{x}$, $\boldsymbol{y}$. A node has a location ($\boldsymbol{u}$.location). When there is no confusion, the location of a node refers to that node. An element has a key, a location and a level ($\boldsymbol{x}$.key, $\boldsymbol{x}$.location, $\boldsymbol{x}$.level). The successor and the predecessor of element $\boldsymbol{x}$ at level l is denoted as $\boldsymbol{x}$.neighbor[R][l] and $\boldsymbol{x}$.neighbor[L][l]. If $\boldsymbol{x}$.location is equal to $\boldsymbol{u}$.location, the element $\boldsymbol{x}$ is stored on the node $\boldsymbol{u}$; otherwise $\boldsymbol{x}$ is a pointer that points to the element $\boldsymbol{x}$.

3.1 Key search

The search operation (algorithm 1) can be started from any node. In algorithm 1, if no elements are stored on the node, function isEmpty() creates a new LinkNet on this node and returns true. Function chooseSide() is used to decide the search direction. If there is an element whose key equals to the search key, function localSearch() returns that element, otherwise it returns an element which is the nearest element to the search key following the search direction.

There are expected $2N$ pointers in an N elements skip list. Because LinkNet uses doubly linked list, an M elements skip-list-based LinkNet stores expected $4M$ pointers and an N elements node in it stores expected $4N$ pointers. Therefore an M elements skip-list-based LinkNet needs expect O(M) space overall and expected O(N) space for an N elements node in it.

The number of messages exchanged among nodes is used to evaluate the performance of the search algorithm. When each node stores only one element, the skip-list-based LinkNet is degenerated into a skip-list-like structure. So the worst performance of skip-list-based LinkNet is similar to skip list. To search an N elements skip list, the expected number of comparisons is O(log$N$). Therefore a search for an M elements skip-list-based LinkNet takes expected O(log$M$) messages.

Algorithm 1: search for node ***u***

```
upon receiving <search, key> from v
    if (u.isEmpty() = true)
        send <retNotFound, header> to v
    else
        side ← u.chooseSide(key)
        send <searchOp, v , key, side> to u

upon receiving <searchOp, startNode, searchKey, side>
    x ← u.localSearch(searchKey, side)
    if (x.location ≠ u.location)
        send <searchOp, startNode, searchKey, side> to x.location;
    else if (x.key = searchKey)
        send <retFound, x> to startNode
    else if (side = L)
        send <retNotFound, x.neighbor[L][0]> to startNode;
    else
        send <retNotFound, x > to startNode
```

Algorithm 2: insert for node ***u***

```
upon receiving <insert, key, value> from v
    send <search, key> to w
    wait until receipt of <retSearchResult, y>
    maxElementLevel ← u.randomLevel()
    x ← u.makeElement(u.location, key, value)
    u.lock(x.level)
    u.insertOp(x, y, 1)
    for level ← 2 to maxElementLevel do
        y ← x.neighbor[L][level-1]
        send <searchNeighbor, u,  y, level> to y.location
        wait until receipt of <retSearchResult, y, z>
        u.insertOp(x, y, level)
    u.unlock(x.level)
    send <retInsertSucess> to v

u.insertOp(x, y, level)
    send <getLock, u, y, x, level> to y.location
    wait until receipt of <retLockResult, y, z>
    u.lock(x.neighbor[R][level])
    x.neighbor[L][level] ← y
    x.neighbor[R][level] ← z
    x.level ← level
    send <setPointer, y, R, level, x> to y.location
    wait until receipt of <retSetPointerResult>
    send <setPointer, z, L, level, x> to z.location
    wait until receipt of <retSetPointerResult>
    send <unlock, y, R, level> to y.location
    wait until receipt of <retUnlockResult>
    u.unlock(x.neighbor[R][level])

upon receiving <searchNeighbor, startNode, x, level>
    if (x.level ≥ level)
        send <retFound, x, x.neighbor[R][level]> to startNode
    else
        y ← x.neighbor[L][level-1]
        send <searchNeighbor, startNode, y,  level) to y.location
```

3.2 Key insert

The first step of the insert operation (algorithm 2) is to find the place of new element in the level 1 of LinkNet. The search returns an element ***y*** which is the predecessor of the new element ***x*** and ***y***'s forward pointer point to ***x***'s successor.

The next step is to insert the new element into a node. A random level generated by function randomLevel() is assigned to the new element. To keep the level of the new element ***x*** from being changed by other concurrent operations (for example, try to delete ***x***) before the end of insert, it is necessary to put a lock on ***x***.level.

Then the new element is inserted into level 1. Because the search does not lock the forward pointer of element ***y***, ***y*** may be not the ***x***'s predecessor now. The getLock operation in algorithm 3 is used to lock the forward pointer of ***x***'s current predecessor. Because the backward pointer of ***x***'s successor is to be changed, the forward pointer of ***x*** is also locked after the forward pointer of ***x***'s predecessor is locked. To lock in this way for insert and delete operations avoids deadlock.

```
Algorithm 3:  Additional operations for node u

upon receiving <setPointer, x, side, level, y> from v
  x.neighbor[side][level] ← y
  send <retSetPointerSuccess> to v

upon receiving <unlock, x, side, level> from v
  u.unlock(x.neighbor[side][level])
  send <retUnlockSuccess> to v

upon receiving <getLock, startNode, y, x, level>
  z ← y.neighbor[R][level];
  if (z.key < x.key)
     send <getLock, startNode, z, x, level> to z.location
  else
     send <getLockOp, startNode, z, x, level> to z.location

upon receiving <getLockOp, startNode, y, x, level>
  u.lock(y.neighbor[R][level])
  z ← y.neighbor[R][level];
  if (z.key < x.key)
     u.unlock(y.neighbor[R][level])
     send <getLockOp, startNode, z, x, level> to z.location
  else
     send <retLockSuccess, y, z> to startNode
```

If the random level given to the new element is greater than 1, the new element will be inserted into the LinkNet level by level. The level of the element *header* and *nil* is not less than the level of any element in the LinkNet. The new element is inserted into level k (k>1) by the same way used to insert it into level 1. This level by level insertion brings an advantage that an element's level can be increased at any time.

There are two types of search for insert operation. One is searching for the predecessor at level 1 of the *M* elements skip-list-based LinkNet. It takes expected O(log*M*) messages. Another is searching for the predecessor at level k (k>1). It takes average 2 messages because 1/2 level k-1 elements appear in level k (assume *p*=1/2 in the skip list, see paper [4] for more details). Because an *M* elements skip-list-based LinkNet will have an average of O(log*M*) levels, to insert a new element into all upper levels also takes expected O(log*M*) messages. Therefore an insert operation takes expected O(log*M*) messages in an *M* elements skip-list-based LinkNet.

3.3 Key delete

Delete operation (algorithm 4) for LinkNet is simple. A node can only delete the element stored on it. To delete an element ***x***, it is not correct to immediately garbage collect ***x*** because other operations may have a pointer to ***x***. Instead, function putOn-GarbageQueue() is used to put ***x*** onto a garbage queue. An element can be taken off the garbage queue any time after the completion of all searches/insertions/deletions that were in progress. To prevent x.level from being changed by other operations before the end of the current delete operation, it is necessary to put a lock on x.level.

Algorithm 4: delete for node ***u***

```
1   upon receiving <delete, key> from v
2     x ← u.localSearch(key, u.chooseSide(key))
3     if (x.location = u.location and x.key = key)
4       maxElementLevel ← x.level
5       u.lock(x.level)
6       for level ← maxElementLevel down to 1 do
7         y ← x.neighbor[L][level]
8         send <getLock, u, y, x, level> to y.location
9         wait until receipt of <retLockResult, y, z>
10        u.lock(x.neighbor[R][level])
11        z ← x.neighbor[R][level]
12        send <setPointer, y, R, level, z> to y.location
13        wait until receipt of <retSetPointerResult>
14        send <setPointer, z, L, level, y> to z.location
15        wait until receipt of <retSetPointerResult>
16        x.level ← level-1
17        x.neighbor[L][level] ← z
18        x.neighbor[R][level] ← y
19        send <unlock, y, R, level> to y.location
20        wait until receipt of <retUnlockResult>
21        u.unlock(x.neighbor[R][level])
22      u.putOnGarbageQue(x)
23      u.unlock(x.level)
24    send <retDeleteSucess> to v
```

The deletion is done level by level. The element is deleted with a top-down style (i.e. from the topmost level to level 1). This level by level deletion brings an advantage that an element's level can be decreased at any time.

The algorithm 4 assumes there are no duplicate keys stored in the LinkNet. To release this limitation, The getLock operation in algorithm 3 must be modified to lock the correct element. The way is to replace the condition "z.key < x.key" with "z $\neq$ x" in line 9 and line 16 of algorithm 3.

The delete operation for an *M* elements skip-list-based LinkNet costs expected O(logM) messages if the getLock operation can be finished with O(1) messages.

4 Performance evaluation

The search algorithm of LinkNet is simulated on a PC. The LinkNet is built with two parameters: one is *N*, the number of nodes; another is *M*, the number of keys. The keys are generated by a uniform random number generator. Each node has a random number of keys. The simulation search random keys starting from random selected nodes for 10,000 times to evaluate the search algorithm by the average number of hops. A hop is a message passing from one node to another node.

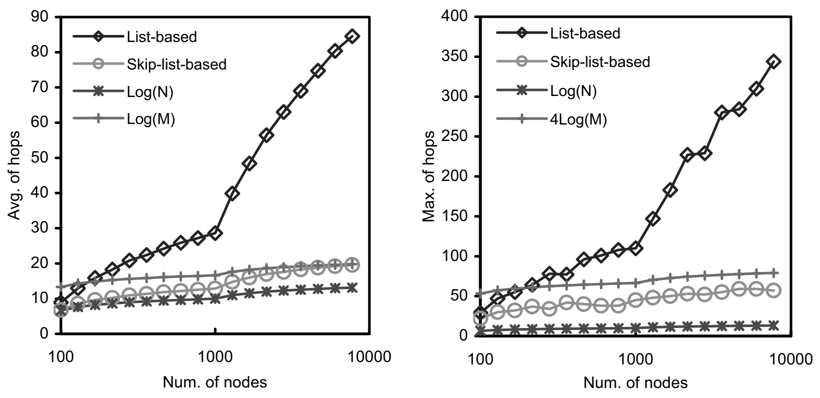

Figure 4. The number of hops vs. the number of nodes

The first experiment reveals the influence of different number of nodes. Figure 4 shows the relation between the number of hops and the number of nodes. Each node has average 100 keys. In figure 4, when the number of nodes increases, the average (maximum) number of hops on the list-based LinkNet also dramatically increases, this means its search performance is worse than skip-list-based LinkNet whose average (maximum) number of hops increases slower.

Figure 4 also shows that when the number of nodes increases from 100 to 10000, the average number of hops on skip-list-based LinkNet increases from log(*N*) to log(*M*), and the maximum number of hops on skip-list-based LinkNet approaches 4log(*M*). It is expected that the average number of hops on skip-list-based LinkNet approaches 2log(*M*) with the increase of nodes number. Therefore when the number of nodes increases and the average number of keys of each node is a constant, the average (maximum) number of search hops on skip-list-based LinkNet is O(log(*M*)).

The second experiment reveals the relation between the number of hops and the number of keys. As mentioned in section 2, to insert a new key into the LinkNet may lead to building new link between two nodes. Therefore more keys leads to less hops. Figure 5 shows that this is correct when the average number of keys on each node increases from 1 to 10, but when the average number of keys on each node increases from 10 to 10,000, the average number of hops approaches a constant. There are 100 nodes in the second experiment, and the partial graph from 100 keys to 10,000 keys is omitted.

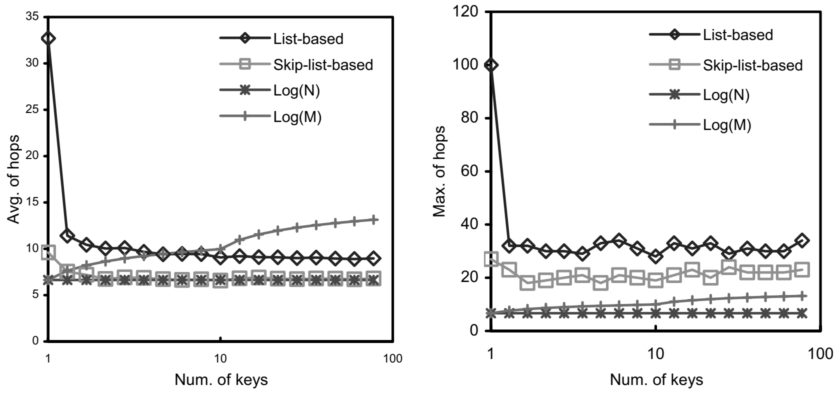

Figure 5. The number of hops vs. the number of keys

Figure 5 also shows that when the average number of keys on each node increases from 1 to 100, the average number of hops on skip-list-based LinkNet approaches log(*N*). It is expected that with the increase of the average number of keys on each node, the average number of hops on skip-list-based LinkNet approaches log(N). Therefore when the average number of keys on each node increases and the number of nodes is a constant, the average number of search hops on skip-list-based LinkNet approaches log(*N*).

5 Conclusions

We have defined a new scalable distributed data structure LinkNet. By adding virtual links to a skip list, we build a skip-list-based LinkNet. In an *N* nodes *M* elements network, the expected total space this data structure takes is O(*M*), and when *M* is big enough, the search operation takes expected O(log*N*) messages among nodes. Additionally, the virtual links enhance fault tolerance of LinkNet.

The scalability of skip-list-based LinkNet is worth an emphasis. Our design eliminates the global parameters of skip list. Every node in the LinkNet is full of autonomy. They don't need to know the global status. This benefits operations such as merging of two LinkNet, but it also let the self-organization LinkNet become a research focus. For example, if we have chosen an improper constant *MaxLevel* for the random level generator, how the system finds this automatically?

In this paper, we use lock to solve the problem caused by concurrent operations. This lowers the performance of the system. One future work is to find a more efficient way to do the concurrent operations correctly.

Acknowledgments

The work described in this paper was carried out during a visit to database group at school of computing, National University of Singapore. The design presented here is the result of much fruitful discussion with Beng Chin Ooi.

References

1. H. Balakrishnan, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. Communications of the ACM, 46(2), February 2003.
2. J. Aspnes and G. Shah. Skip Graphs. In Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, January 2003.
3. N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS), March 2003.
4. W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. Communications of the ACM, 33(6):668-676, June 1990.
5. W. Pugh. Concurrent Maintenance of Skip List. Technical Report CS-TR-2222, Department of Computer Science, University of Maryland, June 1990
6. Napster. http://www.napster.com/.
7. Gnutella. http://www.gnutelliums.com/.
8. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM), San Diego, CA, USA, August 2001.
9. I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Frans Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. Tech. Rep. TR-819, MIT LCS, 2001.
10. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms, November 2001.
11. B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, Apr.2001.