Vol.2012-DPS-150 No.47
Vol.2012-CSEC-56 No.47
2012/3/1

A Proposal of Skip Graph extension for load balancing distributed interval matching

Tran Anh Phuong,[†1] Yuuichi Teranishi,[†2,†1]
Kaname Harumoto [†1] and Shojiro Nishio[†1]

In this research, we propose a new overlay network for processing distributed intervals matching, which is an important task in distributed Content Based Publish Subscribe system (CBPS) or sensor data sharing. Previous methods often required high cost in maintaining the balance of the delivery tree, especially in systems where the subscribing intervals are skewed. A few approaches based on Skip Graph addressed this problem, but they imposed highly unbalanced workload on nodes or introduce high latency during retrieving value. By extending the Skip Graph with additional states on each level, we enable it to respond to request faster with lower cost in a more balanced fashion. We evaluated the proposal method by simulation and confirmed the method's effectiveness compared to existing approaches.

1. Introduction

In the recent years, many new advances have been made in the field of wireless network infrastructure and device manufacture. Internet accessing devices such as PDAs, cell phones, sensors or surveillance cameras pervade widely in human life, increasing the amount of data over the Internet at an unprecedented rate. Also, the emergence of smart phones furthermore empowers users, enables them to take more advantages of the abundance of data. Convenient, timely and inexpensive way to access relevant information at a very large scale has become essential.

As the system continues to grow, the traditional client-server method proves to be infeasible because of the bottle neck problem, which hinders scalability. Therefore, much research has tried to achieve higher level of scalability through the use of distributed overlay network in which devices interconnect independently without any central server; hence, there's less worry about the central servers become overload. There have been much researches on distributed hash table (DHT)[6),7)], in which single key-based queries are forwarded to the matching node. Skip Graph[2)], a overlay network based on skip lists, is designed to perform queries based on key-ordering, which enables range matching queries. These overlay networks allow only single-key matching. That is, these overlay networks supports an event delivery for single-value subscription of each node.

However, there are many cases that single-value subscription is inadequate. For example:

- Handling complex constraints such as finding data which is larger (smaller) than a value, value between ranges in Content Based Publish Subscribe (CBPS) systems (ex: find all subscriptions interested in computer with OS=Windows; RAM $\leq$ 4GB, Price$\in$($500-$700)
- Handling value covering queries in a sensor network(ex: find nodes with sensor data covering a specific geographic area)

In such applications, a message delivering system which can deal with interval subscriptions instead of single-value subscriptions is required.

There have been some researches on handling interval subscriptions using familiar overlay networks such as DHT, tree structured overlay or Skip Graph. However, these overlay networks pose rather critical problems in real life situations. A DHT-based overlay network such as Mercury[3)] often becomes unbalanced among nodes when the subscriptions are skewed (ex. hot spot). Tree structured overlay networks such as Pyracanthus[1)] often have to be built upon other overlay network for message transporting purpose. Also, if the tree itself is unbalanced, number of hops will vary largely between routes. Therefore, the tree structured overlays require tree balancing mechanism. Skip Graph-based approaches overcome such unbalances but if nodes overlap, and some nodes have very long range of data, the latency during event delivery process can become very high.

In the present research, we address these problems and propose the *Interval Tree Skip Graph (ITSG)*, an overlay network based on Skip Graph that handles interval subscriptions. Our contribution is as below:

- Proposal of ITSG that handles interval subscriptions in a fast, balanced fash-

†1 Osaka University
†2 National Institute of Information and Communications Technology

ion.

- Proposal of fully distributed algorithms to construct the ITSG structure
- Simulation results and proofs of ITSG's effectiveness over previous methods.

2. Related Research

Even though the original Skip Graph can only handle single value subscriptions, some extensions such as Interval Skip Graph (ISG) and Range Key Skip Graph (RKSG) are able to handle interval subscriptions. Here, we first take a look at these researches.

(1) Skip Graph

Skip Graph is a Skip List adaptation in the distributed environment. Skip Graph is suitable to a volatile P2P environment, i.e. it maintains connectivity even when a large fraction of nodes fail. Figure 1 shows the structure of an overlay network based on Skip Graph. The numbers in the rounded rectangles represent the keys of the nodes, while each circle represents that node at a level. Black lines represent links between nodes. Nodes at the same level connect to each other using Membership vector generated randomly from an alphabet (in this case, size of alphabet is 2), so that any nodes have Membership vectors with the same first i-digit connect to one or 2 other nodes at level i. The Membership vector is represented by the small number below each circle in the figure.

The retrieval of an individual key is carried out from the top most level of the node seeking a key. At each node, the query is compared to the node's key to decide whether it should be forwarded left or right. It then continues at the same level without overshooting the key, continuing at lower levels if needed, until it reaches level 0 (black arrow in Figure 1). From there, either the address of the node storing the search key, if exists, or the address of the node storing the key closet to the search key, is returned.

Skip Graph also supports range query. Since keys are arranged in the total order relation, one simple method is to find the smallest (largest) key possible that belongs to the range of the query and then forward the query to neighbor nodes on the left (right) side until the node's key gets out of the range. By making use of the higher levels' links, this process can finish in averagely $O(\log n)$ hops, where n is the number of nodes in the system.

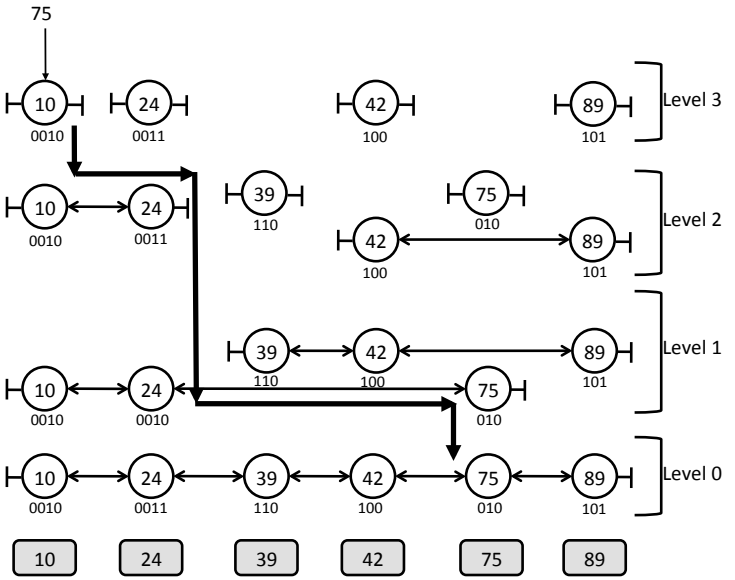


Fig. 1 Search example in Skip Graph

(2) Interval Skip Graph (ISG)

Interval Skip Graph(ISG)[4] combines Interval Tree, an interval-based search tree with Skip Graph to produce an order, distributed data structure aiming at finding all intervals that contain a particular point or ranges of values. In ISG, the Skip Graph is extended to store intervals of $[low_i, high_i]$ instead of a single-value key where i is the ID of node. Node indexing is based on the lower bound low_i, i.e $low_i \leq low_{i+1}$. Another extension in ISG is that, each node now maintains a secondary key max_i, which is the cumulative maximum of the upper bound $high_i$, i.e $max_i = max_{k=0...i}(high_i)$. Figure 2 shows an example of the ISG data structure.

The search algorithm for all intervals containing a value u consists of 2 phases. First, it searches for u on the secondary key, max_i, and locates the left-most node with $max_i \geq u$ (by definition, this data element will have $max_i = high_i$). If $low_i > u$, then this interval does not contain u, and no other intervals will, so

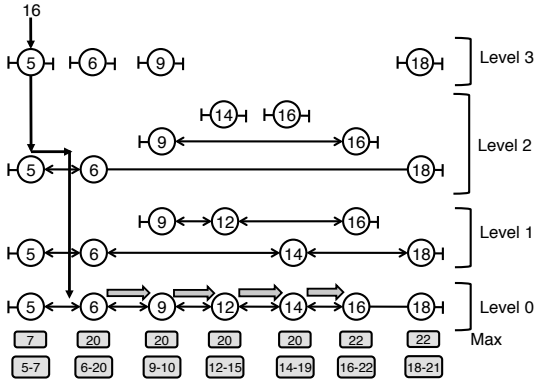


Fig. 2 Interval Skip Graph illustration

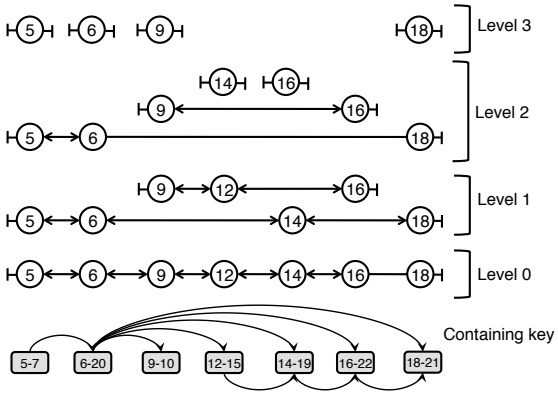


Fig. 3 Ranke Key Skip Graph illustration

the search ends here. Otherwise, the second phase starts with traversing every node in increasing order of low_i (moving to the right side), returning matching intervals on the way, until it reaches a node with $low_i > u$, then the search finishes.

Searching for the first matching interval of a given value is performed in a manner very similar to an interval tree. The complexity of search operation is $O(\log n)$, and the number of intervals that match a range query varies depending on the amount of overlaps. However, in cases that exists an especially long interval, the number of hops can become very high, like in Figure 2. In real life situation, this high number of hops is translated into high latency over network.

In order to reduce number of hops in case of long interval, we can apply the search algorithm with a range-forwarding mechanism, so that in the second phase, query can be delivered to all potential nodes after a small number of hops. We name this method Range-forwarding ISG, and denote it as RISG. However, this method still has lots of no-match nodes, since queries are sent to all the intermediate nodes.

(3) **Range Key Skip Graph (RKSG)**

Rather similar to ISG, RKSG[8] also uses a pair of (low,high) other than just a single-key value. The difference is that, RKSG maintain information about all the overlapping intervals (called the containing keys) instead of just the maximum value. In responding to a search query, RKSG will first find one interval that contains the queried value. It then looks up among the node's containing keys to see which interval contains the value, and forwards the query to those nodes (Figure 3).

RKSG promises very low latency in retrieving values, as it only takes $O(\log n)$ hops on average to reach the first matched node (if exists) and one another hop to all the other matched nodes. However, if there are many overlapping intervals, the first matched nodes will suffer a high workload in finding matched intervals. Also, insertion procedure requires every newcomer node to computeits own containing keys, resulting in high number of messages exchanged during insertion.

3. Interval Tree Skip Graph (ITSG)

We propose a new structure Interval Tree Skip Graph (ITSG), which can overcome problems in the existing Skip GRaph's extensions for interval matching. ITSG involves augmenting nodes with extra maximum values, making it possible to handle range retrieval with faster speed, lower cost and in a more balanced fashion. Here, we describe how it can be implemented.

(1) **The model**

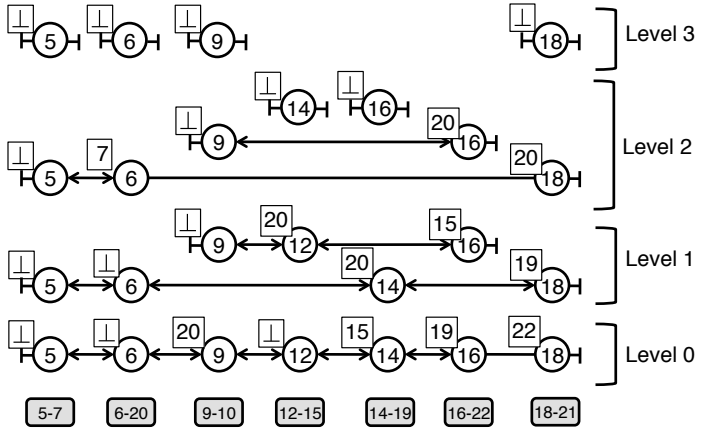


Fig. 4 Interval Tree Skip Graph illustration

Each node in ITSG keeps a pair of $[low_u, high_u]$ as its data element. Indexing order still bases on lower bound low_u, i.e. node u will be on the left side of node v if $low_u < low_v$. It also maintains some maximum values, each ties to a different region depends on the position of the nodes, the level at which the value accounts for, and the neighboring nodes. Figure 4 demonstrates the data structure of ITSG.

As shown in the figure, each node now maintains a maximum value called the *LeftNeighborMax (LNM)* value (represented by the number in the square at upper left corner of each circle). This value representsthe maximum value of nodes lying between current node's left neighbor at a level and the node's left neighbor at 1 level above. To reason about this structure formally, we need some denotations. Let u be the node, $key(u)$ be its key value (here, $key(u) = low_u$), let i ($i \leq 0$) be the level, and we denote the left(right) neighbor of node u at level i as $left_u(i)$ ($right_u(i)$). The node's denotation also refers to its key value in some cases, which is easily to tell from the context. Now we can define this *LNM* values mathematically as following:

$$LNM_u(i) = max(high_v | left_u(i+1) < key(v) | left_u(i) \text{at level } 0)$$

Note that these intermediate nodes are nodes that actually have their keys fall into this range, which means nodes at level 0, not just nodes at level i. If node has a left neighbor, this $LeftNeighborMax$ value will be defined. Else, it will take the value of $\perp$. Besides, if node connects to the same node at more than 1 level, this value also becomes $\perp$. (If $left_u(i+1) = left_u(i)$, exists no v such $left_u(i+1) < v \leq leftu(i)$).

(2) **Search algorithm**

Given the above-mention data structure, the search operation is rather simple. First, it looks for the right-most node (highest key) that doesn't exceed the query. This step is exactly the same with the search operation in Skip Graph. Then we broadcast the queries on all levels that have the *LNM* value higher than *searchKey* (Algorithm 1). Each query carries a boundary value, making sure that query doesn't overshoot the range that other query has already come to; hence reduces the number of wasted messages.

Algorithm 1 Search at node u

```
upon receiving <searchOp, startNode, searchKey, boundary>:
level ← level-1
if self.range contains searchKey then
  send <onRangeMatch, u> to startNode
end if
while level > 0 do
  if LNM_u(level) ≥ searchKey then
    if boundary = ⊥ then
      send <searchOp, startNode, searchKey, ⊥ > to left_u(level)
      level ← level-1
    else {boundary ≠ ⊥&&left_u(level) > boundary}
      send <searchOp, startNode, searchKey, ⊥ > to left_u(level)
    end if
    level ← level-1
  end if
end while
```

In this algorithm, we have:

Lemma 3.1. *The search operation takes expected O(log n) time in a system with n nodes*

Proof: The first step of the search operation in ITSG is exactly the same with that in the original Skip Graph, so it runs in $O(\log\ n)$ time. In the second step, a range-forwarding mechanism is employed to broadcast the messages to nodes. Queries with $\perp$ boundary always takes place at the highest level possible, therefore it takes about $O(\log\ n)$ time to reach the last possible node. In the mean time, queries with specific boundary are also forwarded to nodes. Since they are limited by the boundary, they are expected to run in $O(\log\ n)$ time (where m is the number of node limited by boundary values, and m<n). Therefore, the whole search operation takes expected $O(\log\ n)$*($O(\log\ n)$+$O(\log\ n)$) = $O(\log\ n)$ time to finish. □

Lemma 3.2. *In a system with n nodes, each node has to process at most* $O(\log\ n)$ *messages during search operation*

Proof: During the first step of search operation, intermediate nodes (nodes that the query passes through) have to process at max 1 messages, while latter during the second step; each node that receives the query needs to send out at most the same number of messages with its number of levels. Since ITSG employs the same structure with Skip Graph, which has $O(\log\ n)$ levels on average at each node, each node in ITSG has to send out at most $O(\log\ n)$ messages. Therefore, each node has to process at most $O(\log\ n)$ messages during search operation. □

(3) Insert Algorithm

Insert algorithm is also divided into 2 phases. In the first phase, newcomer node inserts itself into the graph, creating links with neighboring nodes. In the second phase, request update messages are sent to nodes of the right side of the newcomer, asking them to update their maximum values accordingly. First, we introduce another maximum value called the *LeftMax (LM)* value. This value represents the maximum of high value among nodes lying between the node itself and its left neighbor. We denote this value as $LM_u(i)$, with u being the node, and i being the level to which this value belongs. Mathematically:

$$LM_u(i) = \max\ (high_v \text{ — } left_u(i) < v \leq u \text{ at level 0})$$

$LM_u(i)$ can also be defined as the maximum of all LeftMax value of nodes lying between the 2 nodes (itself and its left neighbor) at level i-1:

$$LM_u(i+1) = \max\ (LM_v(i) \text{ — } leftu(i) < v \leq u \text{ at level i})$$

Lemma 3.3. *The LeftMax value at level i+1 equals the higher value among LeftMax value at level i and LeftNeighborMax value at level i.*

$$LM_u(i+1) = max\ (LM_u(i)\ , LNM_u(i))$$

Proof: By definition, we have:
$LM_u(i) = \max\ (high_v \mid left_u(i) < v \leq u)$
$LNM_u(i) = \max\ (\ high_v \mid left_u(i+1) < v \leq left_u(i)\ )$
$LM_u(i+1) = \max\ (high_v \mid left_u(i+1) < v \leq u)$
$= \max(\max(high_v | left_u(i+1) < v \leq left_u(i)), \max(high_v | left_u(i) < v \leq u\ ))$
$= \max\ (LNM_u(i), LM_u(i))$ □

With this LM value, we are able to insert newcomer node u into the Graph and update its maximum values at the same time. The messages that are used to find neighbors at each level now carry an extra piece of information, which is the maximum value of the intermediate nodes. For the left side, that value will become $LNM_u(i)$ at level i. Using Lemma 3, $LM_u(i+1)$ can be easily calculated. The process is continued, until newcomer node finds itself alone at a level. The newly calculated LM value becomes the value at $levelMax$, reflecting the maximum value of all nodes lying on u's left side, similar to Max value in ISG.

Update operation starts with the newcomer node u sending itself a request update message. On receiving this message, u asks all its right neighbors update their LNM values at corresponding levels. Then u forwards this request to neighboring nodes on its right side. Each node receives this request message will check its maximum values at every levels. At level i, if u lies between node v and its left neighbor, v will have to update its LM, and then ask its right neighbor to update its LNM. Else, v will update its LNM. This process continues until request message reaches a node v with $high_v \leq high_u$. There the process stops, and all nodes now have the correct, up-to-date information of their left side nodes.

(Algorithm 2)

Lemma 3.4. *The update phase of newcomer node u can safely end when update message reaches the first node v with $high_v \leq high_u$ i.e there's no node needed to update nodes lying beyond v.*

Proof: Let v be the left-most (smallest) node that satisfy $high_v \leq high_u$ ($low_v \leq low_u$ is obvious since update message is sent to the right)
Let w be any node that is one the right side of v (i.e $low_v < low_w$)
We now prove that, in any cases, w needs not to update its Max value.
If, at any level, the left neighbor of w lies on the left of v ($left_w(i) < v$), it means v lies between w and its left neighbor at level i. Therefore, by definition of $LM_w(i) : LM_w(i) \geq high_v \geq high_u$: u doesn't affect w.
If, at any level, the left neighbor of w lies on the right of (or equals to) v ($left_w(i) \leq v$). Since we have $high_v \geq high_u$ and $low_v \geq low_u$: u doesn't affect w. Therefore, in any cases, a newcomer node u doesn't require an update in w.

□

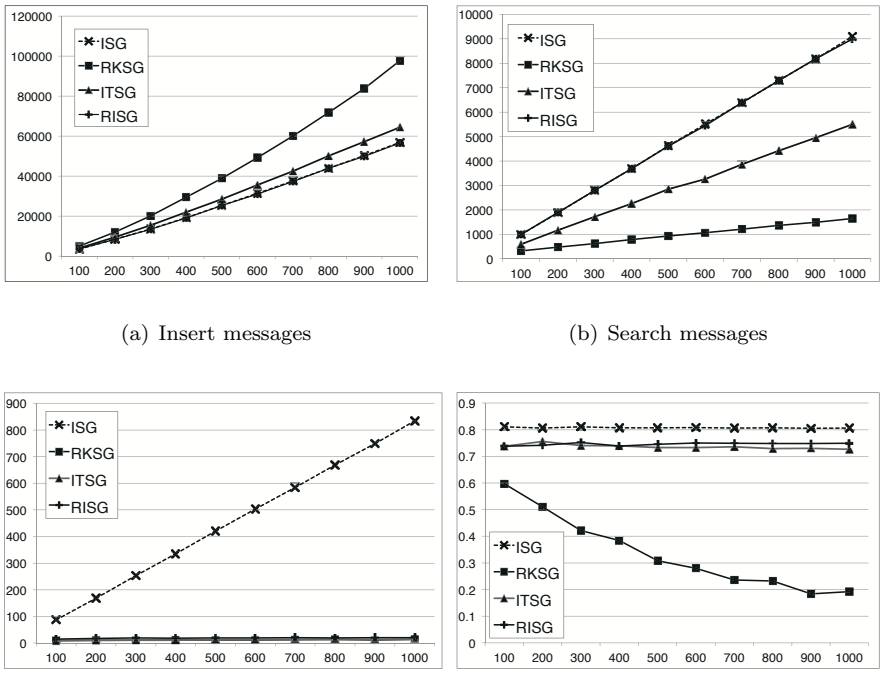
(4) Delete Algorithm

Delete operation is similar to that of insert operation. When node u wants to leave the network, it first links together its 2 neighbors (if available), and then starts sending (delete) update message toward the right side. The message carries u's high value $high_u$, and its left neighbor v's $LM_v(levelMax)$. Nodes that receive the message will remove those values equal to $high_u$, and change it with $LM_v(levelMax)$. The process also ends when update message reaches node v with $high_v \geq high_u$.

Algorithm 2 Update phase for newcomer node u at v

```
upon receiving <updateOp, newcomer, max>
level ← levelMax
if self equals u then
  while level > 0 do
    level ← level-1
    send <updateOp, newcomer, max> to right_v(level)
  end while
else {self not equal u}
  if LM_v(level) < max then
    update LM_v(levelMax)
    send <updateLevelMaxOp, LM_v(levelMax)> to nodes that connect to
    u at (levelMax-1)*1
  end if
  while level > 0 do
    level← level-1
    if left_v(level) < u then
      update LM_v(level)
      send <updateLeftNeighborMaxOp, LM_v(level) > to right_v(level)
    else {left_v(level) ≥ u}
      LNM_v(level)
    end if
  end while
end if
if high_newcomer < high_v then
  send <updateOp, newcomer, max> to right_v(0)
end if
```

⋆1 If node u doesn't have any left neighbor at level $i+1$, u's $LNM_u(i)$ needs to point to its left neighbor v's $LM_v(levelMax)$. Whenever $LM_v(levelMax)$ is updated, v needs to inform u about the change to maintain the correct data structure. However v doesn't know which node to send this kind of update message to. Therefore, we introduce the LevelMaxNode list, which holds information of nodes that connect to the current node at level i, i.e if a node v belongs to node u's LevelMaxNode list at level i, $v = right_u(i)$ and $left_v(i+1) = \perp$. When node v updates its $LM_v(levelMax)$ value, it send a `updateLevelMaxOp` message to every node in its LevelMaxNode list, making sure that the data structure is correct.



(a) Insert messages

(b) Search messages

(c) Number of hops

(d) Fairness Index

Fig. 5 Performance of ISG, RISG, ITSG and RKSG in equally distributed system: horizontal axis represents number of nodes in the system.

4. Evaluation

We performed a series of experiments to verify the operation and performance of Interval Tree Skip Graph (ITSG), in comparison with Interval Skip Graph (ISG), Range Key Skip Graph (RKSG) and Range-retrieval ISG (RISG, as mentioned in section 2(3)). Data related to traffic during insertion and search operation, latency over the network and load-balance were measured.

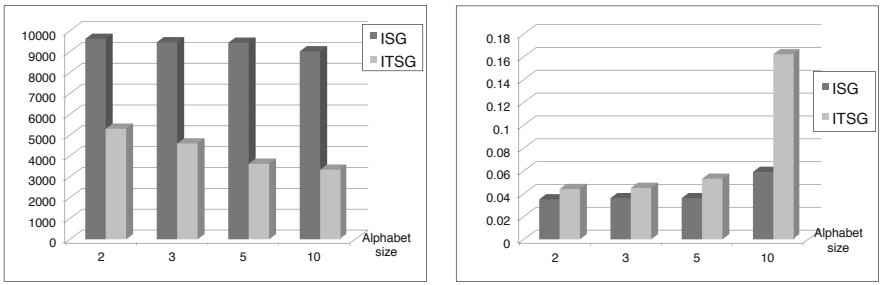
In the first experiment, nodes were equally generated over the data domain in terms of their postions on the network,i.e. the values of low_u acting as the key value. Most of the intervals are short (12%-15% of the domain range), only a few are long (100%). This is to reflect the situation when some "master" nodes store all the data for redundancy purpose, or in Content Based PubSub system, some subscribers don't specific their interest for some attributes. Network size varied from small (100 nodes) to large (1000 nodes). Figure 5(a) shows the relation between number of insert messages and number of nodes in the network, while Figure 5(b) shows the total number of search messages in case of 10 queries. Figure 5(c) exhibit the varying trends in number of hops until queries reached all matched nodes. Shown in Figure 5(d) is the Jain fairness index[5] of message processing loads of each methods. This metric is defined as

$$f(x) = \frac{(\sum x_i)^2}{N \sum x_i^2}$$

where x_i is the number of messages processed by each node in a system of N nodes. A fairness index of 1 indicates that the system is equally fair to all nodes; smaller values indicate less fairness. Here we used number of received and sent messages at each node as an indicator for workload, and only took into acount nodes that received (or sent) messages, not the total number of nodes in the system.

All the four methods exhibit the trade-off between insert and search operation. In case of 1000 nodes, RKSG required about 97k messages during insertion, but only 1650 messages during search operation. ISG (RISG) required about 57k/9000 messages, while ITSG needed about 64k/5500 messages. In term of load balancing, RKSG had the lowest value, since the first-found matched node forwarded queries to all the other matched nodes, suffering from high workload. ISG has the best performance in this aspect. ITSG and RISG employed rather the same mechanism, therefore they have roughly the same value. During search operation, ISG might need up to 900 hops to reach all matched nodes in system of 1000 nodes. In real-life situation, this translates into a very high latency. RKSG needed only around 13 hops, which is close to $O(\log\ n) + 1$ (in this case, n = 1000). ITSG and RISG followed in that order, both required around $O(\log\ n)$ number of hops.

One another factor that affects the performance of ITSG is the graph's struc-



(a) Search messages (b) Precision Rate

Fig. 6 Effect of different alphabets on the performance of ITSG and ISG

ture, in the sense that the further each node at high levels stays away from its neighbor, the better the search operation. To confirm this hypothesis, we tried to configure the Graph's structure by altering the size of alphabet used in Membership vector. Each Membership vector is a finite word consists of characters randomly chosen from the above-mentioned alphabet. The larger the size of alphabet is, the less chance that two randomly generated Membership vectors have the same first i-digit, and therefore, the higher chance for node to link with far away nodes. We conducted this experiment on a system with most of the nodes having very short intervals, and only a few nodes with long intervals.

Figure 6(a) shows that the larger the alphabet, the fewer messages ITSG required to perform search operation. This is because ITSG was able to forward the queries futher using high levels'links. This also resulted in fewer no-matched nodes (i.e intermediate nodes that don't match the query), as shown in Figure 6(b) under the term of Precision Rate. Precision Rate index was calculated by dividing the number of matched-nodes by the number of nodes that received any messages. The higher this value was, the fewer no-matched nodes were introduced during search operation.

5. Conclusion

In the present research we proposed the Interval Tree Skip Graph that extends the original Skip Graph for interval search purpose. An extra maximum value at each level of nodes acts as a message-forward indicator, enabling the graph to decide whether or not it is neccessary to forward the message to that link. A comparative evaluation with Interval Skip Graph and Range Key Skip Graph was carried out. Simulation results show ITSG's effectiveness comparing to previous methods in term of speed and load balancing. Future issue includes handling node failure mechanism to maintain the correct data structure of the graph.

References

1) Aekaterinidis, I. and Triantafillou, P.: Pyracanthus: A scalable solution for DHT-independent content-based publish/subscribe data networks *Information Systems*, Vol.36, Issue 3, pp. 655–674 (2011)
2) Aspnes, J. and Shah, G.:Skip Graphs, *ACM Transaction of Algorithms*, Vol. 3,No. 4, pp. 37, (2007)
3) Bharamble, A.,Agrawal, M. and Seshan, S.:Mercury: supporting scalable multi-attribute range queries i *ACM SIGCOMM Computer Communication Review*, Vol. 34, no. 4,pp. 353–366, (2004)
4) Desnoyers, P.,Ganesan, D. and Shenoy, P.:TSAR: a two tier sensor storage architecture using interval skip graphs, *Proc. of the 3rd international conference on Embedded networked sensor systems(Sensys)*,pp. 39–50 (2005)
5) Jain, R. and Chiu, D.M. and Hawe, W.:A quantitative measure of fairness and discrimination for resource allocation in shared computer systems, *DEC Research Report TR-301*, (1984)
6) Maymounkov, P. and Mazieres, D.:Kademlia: A peer-to-peer information system based on the xor metric. *Proc. of the 1st International Workshop on Peer-to-peer System (IPTPS'02)*, pp. 53–65, (2002)
7) Stoica, I.,Morris, R.,Karger, D.,Kaashoek, M.F.,Balakrishnan, H.:Chord: A Scalable Peer-to-peer Lookup Service for Internet Application,*Proc. of ACM SIGCOMM 2001*, (2001)
8) Ishi, Y.,Teranishi, Y.,Harumoto, K. and Nishio, S.:Range Key Extension of the Skip Graph, Globecom, (2010)