This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the ICC 2007 proceedings.

Skip Tree Graph: a Distributed and Balanced Search Tree for Peer-to-Peer Networks

Alejandra González Beltrán, Paul Sage and Peter Milligan
School of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast, BT7 1NN Belfast, UK
Email: {a.gonzalez-beltran, p.sage, p.milligan}@qub.ac.uk

***Abstract*—Skip Tree Graph is a novel, distributed, data structure for peer-to-peer systems that supports exact-match and order-based queries such as range queries efficiently. It is based on skip trees, which are randomised balanced search trees equivalent to skip lists and designed to provide improved concurrency. Skip tree graphs constitute an extension of skip graphs enhancing their performance in both, exact-match and range queries. Moreover, skip tree graph maintains the underlying balanced tree structures using randomisation and local operations, which provides a greater degree of concurrency and scalability.**

I. Introduction

Recent research in structured Peer-to-Peer (P2P) systems has provided distributed data structures that satisfy several desirable properties for large-scale environments, such as decentralisation, scalability, load balancing, self-stabilisation, fault-tolerance, efficient and guaranteed searches, support for dynamic addition and deletion of nodes. These P2P systems implement a dictionary abstract data type, allowing for the insertion, deletion and searches of keys stored in the nodes of the P2P network. Naturally, these solutions have followed the steps of dictionary implementations in the sequential domain, which include hash tables, tries, binary search trees, 2-3 trees, AVL trees, skip lists and B-trees. P2P systems based on hashing provided a Distributed Hash Table (DHT) [11], [10] functionality in which the order of the data stored is not preserved. Consequently, the implementation of order-based queries such as range queries over these DHT structures is not directly supported. Support for range queries is important for, for instance, data management and grid information services, applications that can use P2P systems as a routing substrate. In this paper, the focus is on improving the efficiency of the searches considering P2P applied to grid environments. Some systems have layered range query schemes over DHT systems [2], [5]. An alternative approach has involved building P2P systems based on tries or different types of search trees. They provide a Distributed Search Tree (DST) functionality, allowing for order-based searches. Examples of these systems are: P-Grid [1], skip graph [3], SkipNet [6], Hyperring [4], BATON [7].

Skip graph (SG) and SkipNet [3], [6] were presented independently and are basically the same distributed data structure based on skip lists [9]. Nodes are connected as a collection of circular lists, constituting a series of overlapping skip lists. Each node requires only logarithmic state to store information about its neighbours. Key order is preserved and thus, it is possible to perform queries on ordered data, such as range queries or nearest neighbour searches.

By using a skip tree [8] as a starting point, which is a structure equivalent to a skip list created for improving concurrency over skip lists, this paper presents a new structured P2P network with DST functionality called *skip tree graph* (STG). STGs extend SGs by increasing the nodes' state with what is defined as a conjugate node. These extra links arise from considering the path traversed by SG's searches and adding that information to the nodes' state during the insertion procedure. Even with the addition of conjugate information, the nodes' state remains logarithmic in the number of nodes. This paper shows that although there is a small increase in the cost of the insertion procedure, two different schemes for exact-match search allow a significant performance improvement with respect to SG search. This increment is not significant for relatively stable grid environments, where the benefit is gained on very fast search operations. Moreover, STG is based on balanced trees using local operations, which implies it allows a greater degree of concurrency [8] in the insertion and deletion of nodes with respect to other balanced-tree structures that require expensive restructuring procedures to maintain the balance [7]. The range queries are also known to outperform the correspondent SG operation.

II. Related Work

Structured P2P networks based on (consistent) hashing, such as Chord [11] and CAN [10], provide a DHT functionality. They support efficient key lookups but lack locality, implying that order-based queries are not directly supported. Some systems have layered an order-preserving scheme over DHTs [2], [5]. Andrzejak and Xu [2] used a space-filling curve on top of CAN and P-tree [5] used a B^+-tree on top of Chord. These systems architectures use the DHT as a routing substrate and delegate responsibility about order-based queries to the upper layer scheme.

On the other hand, DSTs systems are based on order-preserving structures and, therefore, directly support queries where key order is relevant. P-Grid [1] is based on a trie structure, skip graph [3] on a skip list, Hyperring [4] on a deterministic 2-3 tree and BATON [7] on an AVL tree.

Most of these DST systems are based on balanced search trees, which can be classified into two groups depending on the

1-4244-0353-7/07/$25.00 ©2007 IEEE

set of local rules they apply [8]: 1) B-trees and its derivatives which use split and join operations, and 2) AVL-trees and red-black trees which use rotations.

As BATON [7] organises the nodes as an AVL-tree, it is a member of the second group and there is a unique path between every two nodes. To provide fault-tolerance, the structure adds alternative paths between every pair of nodes by maintaining sideways routing tables for each peer. These tables include links to selected nodes on its left and right hand sides at the same level, making the BATON structure very rigid. Each peer joining or leaving the system causes a restructuring process (a rotation in the AVL-tree) that affects several nodes in the structure (it is not a local operation). This restructuring process affects the possibility of several concurrent insertions or deletions, and to avoid inconsistencies a mutual exclusion mechanism must be implemented. This mechanism would inhibit the insertion of nodes, making the network not easily scalable.

Skip lists belong to the first group of balanced trees because they are equivalent to skip trees [8], which use split and join operations. Moreover, in the skip tree these operations are local, meaning that a small set of tree-nodes are accessed and the balancing is achieved through randomisation. Therefore, SG and STG also belong to this first group and their operations are also local. It will be shown that STG outperforms SG in the exact-match search operation, while providing the advantages of the underlying tree structure.

III. Skip Tree Graph Structure

This section introduces *skip tree graphs*, a distributed extension of skip trees [8] that provide an efficient model for P2P networks, preserving the order of the stored keys. This extension is performed in a similar way to which SGs [3] extend skip lists [9], i.e. incorporating redundancy to build a robust decentralised system. As skip trees are isomorphic to skip lists [8], it can be shown that STGs are isomorphic to SGs. However, STGs allow for important performance improvements on the operations with respect to SGs.

In order to introduce the new overlay structure, an overview of the previous related randomised data structures (skip lists, skip graphs and skip trees) is presented. All these structures consider the elements from a totally ordered domain $(K, <_K)$, where K is the set of keys and $<_K$ a total order over K. Skip lists maintain a list with all the keys at the bottom level, and build increasingly sparse lists on upper levels by choosing a key on the immediate lower level with probability p (see Fig. 1(a)). Then, a series of random choices is made for each key to determine to which levels it belongs. By using the higher levels, it is possible to traverse the keys quickly. Searching for a node with a particular key takes an average search time of $O(\log_{1/p} n)$ [9]. Skip graphs extend skip lists by maintaining multiple doubly-linked lists at each level, resulting in a set of overlapping skip lists (see Fig. 2(a)). This redundancy eliminates single points of failure and hot spots, and allows the system to work efficiently in dynamic and distributed P2P systems [3]. The keys identify the peers in the network. The random choices made for each key k are denoted by a random word $m(k)$ from an alphabet Σ whose cardinality is the reciprocal of the probability, i.e. $1/p$. This word is referred to as *membership vector* [3] and is potentially infinite but practically its length is $O(\log_{1/p} n)$. The key k belongs to particular list at level ℓ depending on the ℓ-length prefix of $m(k)$, denoted $m(k)|_\ell$.

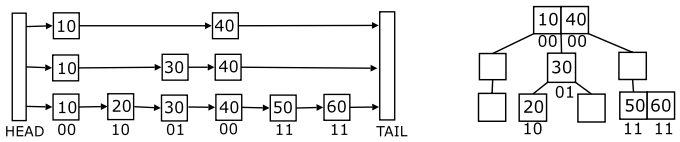
Messeguer presented a skip tree as a search tree structurally equivalent to a skip list [8], in a concurrent approach. The idea behind its construction is to incorporate into the structure information about the path followed by the sequential search algorithm on skip lists. Skip trees group together consecutive keys at the same level of the skip list while satisfying the search tree property. When no keys satisfy the search tree property, the tree-node contains no keys and it is called a *white node*. All the leaves have the same depth as in a B-tree but the number of keys in a node is unbounded and follows a random distribution. Consequently, a skip tree is an *unbounded random B-tree* [8]. Fig. 1(b) shows the skip tree isomorphic to the skip list in Fig. 1(a). Searches take logarithmic time in the number of keys, as in the equivalent skip list.

A set of transformations is applied to a skip tree to use it as the underlying structure in a STG. These transformations are:

1) each key is repeated into the left child from its maximum level until reaching the leaves of the skip tree are reached; this first step ensures that a level in the skip tree can be built from the information in the immediate lower level (Fig. 3(a))
2) white nodes are eliminated while preserving the levels of the nodes (Fig. 3(b))
3) due to the first transformation step, left children point to nodes with the same key and these edges can be eliminated (tree-nodes with the same key represent the same node in the P2P network) and edges to sibling nodes are added (Fig. 3(c))
4) a root node is determined by, if necessary, extending the membership vector of the upper-level keys so that they can be uniquely identified and conjugate keys, as defined below are identified; the membership vector of the root identifies the whole skip tree (Fig. 3(d))

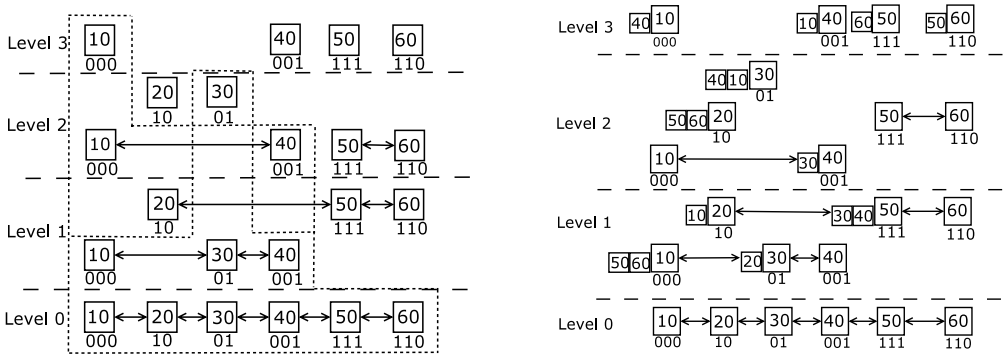
Two keys are conjugate if their membership vectors share the same prefix of length ℓ but differ in the symbol at position $\ell + 1$. If k_1 and k_2 are two keys that share the same prefix $m(k_1)|_\ell = m(k_2)|_\ell$ in a skip tree, they are at the same level ℓ. If ℓ is the maximum level of a key k, k will appear in level $\ell + 1$ as a conjugate key of the first key on its right, by following the order $<_K$ and considering the list circular at level ℓ. In the figures, keys are identified by larger squares and their conjugate keys are identified by smaller squares. Fig. 3(d) shows the transformed skip tree identified by the word 000, the word of its root with key 10, being 40 a conjugate key at that maximum level.

It was mentioned that SGs are equivalent to a collection of skip lists. Similarly, STGs result from overlapping a collection of (transformed) skip trees. As skip trees are isomorphic to



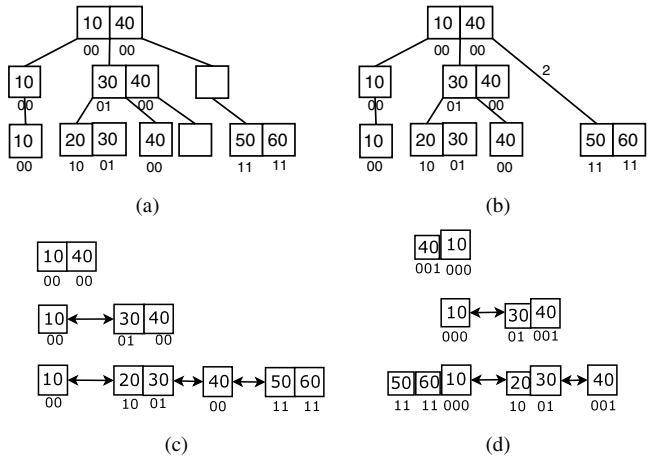
(a) Skip list (b) Skip tree

Fig. 1. Two isomorphic structures: (a) a skip list, and (b) its equivalent skip tree



(a) Skip graph (b) Skip tree graph

Fig. 2. A skip graph and its equivalent skip tree graph



(a) (b) (c) (d)

Fig. 3. Skip tree transformations

skip lists and the transformation steps introduced before are all reversible, transformed skip trees can be obtained by starting with a skip list. Given a SG, its corresponding STG is obtained by superimposing the transformed skip trees obtained from the set of skip lists. Consequently, STGs are isomorphic or structurally equivalent to SGs. Fig. 2(b) displays the STG equivalent to the SG in Fig. 2(a).

Given that each key represents a peer in the P2P network, it is possible to define the *conjugate at level* ℓ of a peer u, which is denoted ℓ*-conjugate(u)*. Thus, the ℓ*-conjugate(u)* is a peer v that satisfies that $m(v)|_{\ell-1} = m(u)|_{\ell-1}$ and $m(v)|_{\ell} \neq m(u)|_{\ell}$, for all $\ell \geq 1$.

By definition, the bottom level contains all the nodes in both structures and level ℓ of the STG contains the same information as levels ℓ and $\ell - 1$ in the SG. In particular, each list at level ℓ of a STG contains all the information from the list at level $\ell - 1$, differing from the corresponding list in SG because it is augmented with conjugate nodes.

A. Skip Tree Graphs Properties

Given the isomorphism between SGs and STGs, demonstrated with the set of transformations, it is clear that the height of the STG follows the same probability distribution as the height of a SG. Thus, on average the height will be $O(\log_{1/p} n)$ [3].

The number of neighbours also coincides with its SG counterpart, being on average $O(\log_{1/p} n)$.

The number of conjugate nodes of a given node u at level ℓ is the number of subsequent nodes v until the neighbour of u at level $\ell + 1$ is found (considering the list circular). Then, if $m(v)_{\ell+1}$ denotes the symbol at position $\ell + 1$ of $m(v)$, $m(u)_{\ell+1} \neq m(v)_{\ell+1}$. The probability that the symbols at position $\ell + 1$ differ is $q = 1 - p$. If G is the random variable representing the number of conjugate nodes, then G follows a geometric distribution, i.e. $Prob[G = k] = q^{k-1}p$. Consequently, the number of conjugate nodes on average is $\frac{1}{p}$.

Each node stores information about the neighbours and the set of conjugates at each level. It results that the state of a node is on average: $O(\log_{1/p} n) + O(1/p \log_{1/p} n) = O((1 + 1/p) \log_{1/p} n)$, i.e. $O(\log n)$ for a fixed p.

IV. Skip Tree Graph Algorithms

The search algorithms for STG achieve better performance than SG algorithms by exploiting the information about conjugate nodes. This section presents two schemes for exact-match search in skip tree graphs: $searchWCOp$ and $searchTreeOp$, which are compared to $searchOp$, the SG search scheme [3]. The insertion algorithm is also described

and compared with the respective one for SG. Each peer maintains at each level one neighbour in each direction, denoted $neighbour[dir][\ell]$ in the pseudocode, and a set of conjugate nodes, which in the algorithms are accessed through different functions explained below.

A. Exact-match Search Operation with Conjugates

The search with conjugates operation $searchWCOp$ (see Algorithm 1) is similar to $searchOp$ as it retrieves the node containing the searched key if it exists in the structure, or the largest key that does not overshoot it otherwise. The search can start at any node and it will remain to one side of the result (either to the left or to the right). It differs from $searchOp$ because conjugate nodes are selected whenever they can provide a shorter route to the destination node.

At level ℓ, if the neighbour in the search direction exists and does not overshoot the *searchKey* (verified with the method $checkOrder(dir, a, b, c)$ that returns true if the order of its parameters is a, b, c going from dir to the opposite direction) the search operation continues through this neighbour in the same level. Otherwise, the best conjugate node at level ℓ that does not overshoot the searched key k is selected, by using the method *getBestConjWOO* $(direction, \ell, k)$. If such a conjugate exists, the search operation is forwarded to it, continuing at a lower level. If there is neither an appropriate neighbour nor conjugate node, the procedure remains in the current node at a lower level. When continuing at a lower level, if possible, two levels down are skipped. The reason is that level ℓ contains the same information as levels ℓ and $\ell - 1$ in the corresponding SG, and it is not necessary to traverse all the levels.

As $searchWCOp$ only uses information about conjugate nodes, when these provide a better skip to the destination, the upper bound for the cost of $searchOp$ is also valid. Then, the search operation $searchWCOp$ on skip tree graph takes on average $O(\log n)$ messages and hops. It is assumed that each message takes at most unit time (one hop) to be delivered, while internal processing at a peer takes no time (same model as in [3]).

An example in which $searchWCOp$ reduces the cost with respect to $searchOp$ can be seen in Fig. 2(b), when starting the search from node 20 (whose maximum level is 2) and searching key 60. As 20 has no neighbours at level 2, $searchOp$ goes down a level and then follows the right neighbour link to 50, which then moves to the right arriving to 60 in 2 steps. On the other hand, $searchWCOp$ follows the conjugate link 60 while going down 2 levels, reaching 60 at the bottom level in 1 step.

Algorithm 1 $searchWCOp$ in node v

1: **upon receiving** $\langle\ searchWCOp, startNode, searchKey, level\ \rangle$:
2: **if** $(searchKey = v.key)$ **then**
3: **send**$\langle\ foundOp, v\ \rangle$ **to** startNode ; **return**
4: **end if**
5: **if** $(v.key < searchKey)$ **then**
6: $dir \leftarrow R$
7: **else**
8: $dir \leftarrow L$
9: **end if**
10: **while** $(level \geq 0)$ **do**
11: $nextNode \leftarrow neighbour[dir][level]$
12: **if** $((nextNode \neq \perp) \wedge checkOrder(dir, v.key, searchKey, nextNode.key))$ **then**
13: **send**$\langle\ searchWCOp$, startNode, searchKey, level $\rangle$ **to** nextNode ; **return**
14: **end if**
15: $nextNode \leftarrow$ ***getBestConjWOO*** $(dir,$
16: $level, searchKey)$
17: **if** $(level >= 2 \wedge (nextNode \neq \perp \vee dir = L))$ **then**
18: $nextLevel \leftarrow level - 2$
19: **else**
20: $nextLevel \leftarrow level - 1$
21: **end if**
22: **if** $(nextNode \neq \perp)$ **then**
23: **send**$\langle\ searchWCOp$, startNode, searchKey, nextLevel $\rangle$ **to** nextNode
24: **return**
25: **else**
26: $level \leftarrow nextLevel$
27: **end if**
28: **end while**
29: **if** $(level < 0)$ **then**
30: **send**$\langle\ notFoundOp, v\ \rangle$ **to** startNode
31: **end if**

B. Tree-Based Exact-match Search Operation

In a STG, it is possible to search for a particular key by following the underlying tree structure. While $searchWCOp$ does not overshoot the searched key (as in $searchOp$), $searchTreeOp$ operation (see Algorithm 2) may jump from one side to the other of the searched key. The search can start at any node at it will start at its maximum level. Each node v at each level ℓ is responsible for an interval of the key space of the form $(u.key, v.key]$, where u is: 1) the left-conjugate node with largest key at level ℓ, if it exists; or 2) the right-conjugate node with smallest key at level ℓ, if it exists; or 3) the right neighbour at level ℓ, if it exists. In this way, the whole key space is divided up at each level between the node and its connections (conjugates and neighbours). When a node receives a $searchTreeOp$ message, it checks if the key belongs to its interval. In that case, it continues the search at the immediate lower level. Otherwise, it finds which is the conjugate responsible for storing the searched key, using the method $getConjWKey$.

The $searchTreeOp$ cost is related to the average height of the skip tree, given that at each step it goes down a level. Consequently, the expected cost for $searchTreeOp$ is $O(\log n)$ in the number of messages and cost. The experimental evaluation section will show that, although the three search schemes are

bounded by logarithms, $searchTreeOp$ outperforms the rest of the schemes, having the smallest constant of proportionality.

To show an example of the improvement of the $searchTreeOp$ with respect to $searchOp$, searching the key 60 from the node 10 is considered in the skip tree graph of Fig. 2(b). As 10 has no neighbours at level 3 and $searchOp$ goes down a level, and after that follows the right neighbour edge to 40, which then goes down two levels, moves to the right arriving at 50 and again to the right to reach 60 at level 0 sending a total of 3 messages. On the other hand, $searchTreeOp$ finds out that node 10 is responsible for the (circular) interval $(40, 10]$ which contains 60 and goes down up to level 1 in which 60 is a conjugate. Consequently, it reaches the destination forwarding only one message.

C. Insert Operation

A new node that wants to join the network needs to know a node already in the network, i.e. an *introducer*. Similarly to a join operation in a SG, the new node u will insert itself in one linked list at each level until it finds itself in a singleton list at the topmost level [3]. However, the insertion procedure in STGs needs also to consider conjugate nodes. For the alphabet $\{0, 1\}$, the new node has to be not only added as a neighbour in one list at each level but also as a conjugate node in one list at each level. In general, conjugate nodes will be determined by storing the path traversed at each level to find the appropriate neighbour at the next level. When searching for the right neighbour, the conjugates are the nodes traversed until the right neighbour is found. The new node u will be added as a conjugate to the first of these nodes traversed. When searching for the left neighbour, the nodes traversed will be conjugate nodes of the new node u.

It might happen that the right neighbour at level $\ell + 1$ coincides with the right neighbour at level ℓ. This means that no conjugate nodes were found in the traversal and consequently the new node is not added as conjugate node. If that is the case, a special message to insert u as a conjugate at the next level is required. This message will be forwarded in the right direction from $u.neighbour[R][\ell]$ onwards to search for a ℓ-conjugate(u). The list at level ℓ is again considered circular. Let c be such a conjugate node, if it exists. Then, u can be added as a conjugate of c. Only if there is no such c after traversing the whole list, u is not added as conjugate node in any list.

When setting a new neighbour node, the existing conjugate nodes may need to be split and transferred to the new neighbour. This split operation is as in a skip tree [8].

D. Repair Mechanism

In the presence of node or link failures, a repair mechanism is required to reorganise the structure, healing the disruptions. This mechanism is similar to the one proposed for SGs [3], with the addition of consideration of conjugate nodes. SG implements two types of operations: *checkNeighOp* to verify neighbours at each level and *zipperOps* that merge two lists at a level, if the constraints with the immediate lower level list are violated. For STG, these operations are extended to record the conjugate nodes and split them when setting new neighbours, if necessary. The split operation is as in the insertion algorithm.

Algorithm 2 $searchTreeOp$ in node v

1: **upon receiving** $\langle$ $searchTreeOp$, $startNode$, $searchKey$, $level$ $\rangle$:
2: **if** $(searchKey = v.key)$ **then**
3: **send**$\langle$ $foundOp$, v $\rangle$ **to** startNode ; **return**
4: **end if**
5: **while** $level \geq 0$ **do**
6: $nextLevel \leftarrow level - 1$
7: $leftConj \leftarrow getLargestConj(L, level)$
8: $rightConj \leftarrow getSmallestConj(R, level)$
9: $leftNeigh \leftarrow neighbours[L][level]$
10: **if** $leftConj \neq \perp$ **then**
11: $nodeRange \leftarrow (leftConj.key, v.key]$
12: **if** $searchKey \in nodeRange$ **then**
13: $goDown \leftarrow TRUE$
14: **end if**
15: **else if** $rightConj \neq \perp$ **then**
16: $nodeRange \leftarrow (rightConj.key, v.key]$
17: **if** $searchKey \in nodeRange$ **then**
18: $goDown \leftarrow TRUE$
19: **end if**
20: **else if** $leftNeigh \neq \perp$ **then**
21: $nodeRange \leftarrow (leftNeigh.key, v.key]$
22: **if** $searchKey \in nodeRange$ **then**
23: $goDown \leftarrow TRUE$
24: **end if**
25: **end if**
26: **if** goDown **then**
27: $nextNode \leftarrow v$
28: $level \leftarrow nextLevel$
29: **else**
30: $nextNode \leftarrow getConjWKey(searchKey, level)$
31: **if** $nextNode \neq \perp$ **then**
32: **send**$\langle$ $searchTreeOp$, startNode, searchKey, nextLevel $\rangle$ **to** nextNode
33: **else**
34: $level \leftarrow nextLevel$
35: **end if**
36: **end if**
37: **end while**
38: **send**$\langle$ $notFoundOp$ $\rangle$ **to** startNode

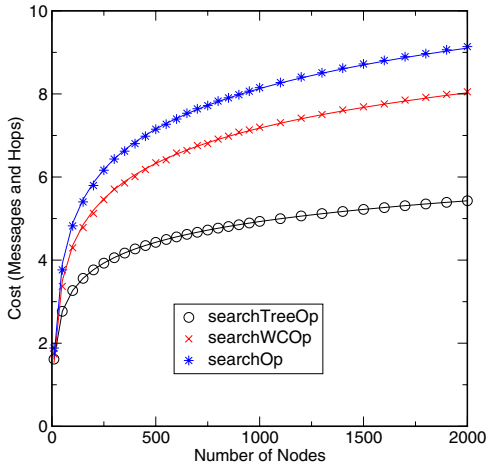
V. Experimental Evaluation

STG was implemented and evaluated under event-driven simulations. The experiments show the performance improvement of the two novel exact-match search operations for STG compared to SG's exact-match search operation. This improvement is achieved with only a slight increment in the insertion cost.

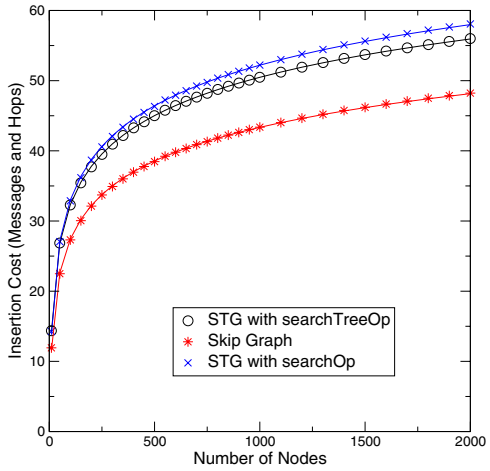
The number of peers n was varied from 10 to 2000 (more points were generated close to the origin to appreciate the

logarithmic behaviour). The keys identifying each peer where generated uniformly at random from the key space $[0, 10^5]$. As STG is a randomised data structure, its cost and performance results have to be considered on average on a number of structures generated for each fixed number of peers. The results presented next are based on 1000 STGs structures generated for each value of n. For each STG, 1000 exact-match search operations of each scheme were performed. Each exact-match search operation started at a random peer in the STG network and searched for a random key in the key space.

Non-linear regression was used to fit the data. Fig. 4(a) shows the cost of the three schemes for exact-match search operation: $searchOp$ (for SG), $searchWCOp$ and $searchTreeOp$ (for STG). All the schemes have the same logarithmic asymptotic performance, supporting the analytical results. However, the constants accompanying the logarithms differ. Costs are compared by percentage differences, valid for a large enough value of n. In the case of $searchWCOp$, the number of messages and hops is reduced by approximately 12 % with respect to $searchOp$. Moreover, $searchTreeOp$ outperforms the rest of the schemes, reducing by half the constant accompanying the logarithm of $searchOp$, i.e. with an improvement of approximately 50 %. These improvements are achieved at the expense of increasing the insertion operation cost, shown in Fig. 4(b). The increment corresponding exclusively to finding the conjugates is around 20 % and in total (when using the cost effective $searchTreeOp$) is around 15 %.



(a) Search Cost



(b) Insert Cost

Fig. 4. Trade-off between improvement in search cost and slight increase in insert cost

VI. Conclusion

This paper contributes to the ongoing research on providing structured P2P systems with the functionality for performing complex queries, such as range queries, in a efficient way. A novel distributed and balanced tree data structure called *skip tree graph* was introduced. This new P2P system is based on skip trees, which is a concurrent approach for skip lists using local operations. This avoids the expensive restructuring required for other balanced-trees based P2P networks. Skip tree graphs constitute an extension of skip graphs by storing in the nodes information about the path traversed by the searches in the form of conjugate nodes. It was also shown that there exists an isomorphism between the two structures. The state of the nodes continues to be logarithmic in the number of nodes. The expected costs for insertion operation and exact-match searches have also logarithmic asymptotic performance. However, exact-match search schemes were presented reducing the skip graph search cost up to its half.

Acknowledgment

The authors are grateful to APG QUB for funding the work of A. González Beltrán, Prof. Gabarró for bringing Messeguer's skip trees to our attention and Dr García for useful comments on this work.

References

[1] K. Aberer, "P-Grid: A Self-Organizing Access Structure for P2P Information Systems," in *Proceedings of CoopIS*, Trento, Italy, 2001.

[2] A. Andrzejak and Z. Xu, "Scalable, Efficient Range Queries for Grid Information Services," in *Proceedings of IEEE P2P*, 2002, pp. 33–40.

[3] J. Aspnes and G. Shah, "Skip Graphs," in *Proc. 14th ACM-SIAM SODA*, Baltimore, MD, USA, 2003, pp. 384–393.

[4] B. Awerbuch and C. Scheideler, "The Hyperring: A low-congestion deterministic data structure for distributed environments," in *Proc. 15th ACM/SIAM SODA*, 2004.

[5] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram, "Querying Peer-to-Peer Networks Using P-Trees," in *Proc. WebDB*, Paris, France, June 2004.

[6] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "SkipNet: A Scalable Overlay Network with Practical Locality Properties," in *Proc. of USITS*, Seattle, WA, USA, 2003.

[7] H. V. Jagadish, B. C. Ooi, and Q. H. Vu, "BATON: A Balanced Tree Structure for Peer-to-Peer Networks," in *Proceedings of the 31st VLDB Conference*, Trondheim, Norway, 2005.

[8] X. Messeguer, "Skip Trees: an alternative data structure to Skip Lists in a concurrent approach," *Informatique Théorique et Applications*, vol. 31, no. 3, pp. 251–269, 1997.

[9] W. Pugh, "Skip Lists: A probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, June 1990.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A Scalable Content-Addressable Network," in *Proceedings of ACM SIGCOMM*, USA, 2001, pp. 161–172.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proceedings of the ACM SIGCOMM*, 2001, pp. 149–160.