Research Journal of Recent Sciences _____ ISSN 2277-2502
Vol. **4(12),** 95-105, December **(2015)** *Res.J.Recent Sci.*

Dynamic Discovery of Resources in Structured P2P Systems

Saeed Arbabi[1,2] and Ali Akbar Keikha Javan[2]
[1]Computer Engineering Department, University of Zabol, Zabol, IRAN
[2]Computer Engineering Department, School of Eng., I.A.U. Zabol Branch, Zabol, IRAN

Available online at: www.isca.in, www.isca.me
Received 25th December 2013, revised 11th April 2014, accepted 22nd May 2015

Abstract

A distributed system is a collection of autonomous computers that appear to their user as one single coherent system. The main goal of any distributed system is sharing resources in a controlled and efficient way. But before any resources can be shared, they should be located. Structured peer-to-peer (P2P) systems have been recognized as an efficient approach to solve the resource locating and discovery problem in large-scale dynamic distributed systems. Efficiency of structured P2Presource discovery approaches attributed to their structured property. However, system dynamism (a.k.a. Churn) caused by changes in the system membership, i.e., nodes that join or leave the system or simply fail, perturbs the structure of the system and endangers the expected correctness and efficiency of resource discovery solutions. In this paper we propose an approach to dynamic searching and discovery of resources that adapts its operation dynamically with the dynamism in the system by using a structure maintenance technique that we have already presented in our recent paper. Although our approach is general enough to be applied to a lot of structured P2P systems, for the sake of brevity here we implemented this resource discovery approach for a well-known structured P2P system called Chord. We analyzed the efficiency of our presented resource discovery mechanism using master equation approach of physics and by experiments. We see how the simulation results and theoretical analyses both show the improved efficiency of our resource searching and discovery mechanisms.

Keywords: Distributed systems, Peer-to-Peer systems, Dynamic Resource Discovery, Churn.

Introduction

Distributed systems have emerged with the main goal of sharing resources in a controlled and efficient way. But before any resources can be shared, they should be located. The process of locating a resource in any distributed system is called resource searching and discovery[1]. Designing an efficient resource discovery mechanism becomes more challenging when the scale of the system gets larger and the dynamism becomes a part of the system behavior[2].

Peer-to-Peer (P2P) systems have emerged as a type of distributed system to partly resolve this issue. From the point of view of their proposed resource discovery mechanism, P2P systems evolved through three generations[3].

Resource discovery solutions proposed in the last generation of P2P systems also known as structured P2P systems have been praised because of their efficient behavior and guarantees for discovering queried resources[4]. These properties have been attributed to the structured property of these systems. A system is considered to be structured if a specific constant pattern of relation exists between system entities. Since nodes and resources are the two main entities in a P2P system, there should be a specific and durable pattern of relation between nodes and resources in a structured P2P system. This pattern should be maintained in the whole lifetime of the system so as to guarantee the accuracy and efficiency of any resource searching and discovery solution for such systems.

The structure of a structured P2P system is created by consistent hashing[5], i.e., the application of a hash function on a unique property of a node or resource, resulting in a unique identifier[6,7]. So a unique identifier from a common identifier space is assigned to each node and each resource in the system. These identifiers determine which resources are placed on which nodes, and this is the pattern of relations between nodes and resources. At the same time, each node in the system maintains some pointers to some other nodes in the system that is the pattern of relations between nodes in the system. As such, resource discovery in such systems becomes a routing problem wherein each node uses its routing pointers to choose the next node to forward a received query. Given a node identifier, a message can be delivered in few logical hops. Resources in structured P2P systems can be discovered correctly and efficiently only if the structure of the system is properly maintained.

Continuous and arbitrary arrivals and departures of nodes in structured P2P systems is the source of system dynamism (also known as Churn) that perturbs the system structure as well as the accuracy and performance of any resource discovery therein. The reason is that with arrival or departure of a node, some pointers in some other nodes may now point to wrong nodes. So

maintaining the structure of a structured P2P system with low overhead and high robustness in the presence of churn is a great challenge to be resolved[7].

Most of current structured P2P systems are based on periodic protocols for structure maintenance. Structure maintenance techniques based on periodic approach update the system structure at specific time intervals. The main challenge in techniques that are based on periodic approach is that a trade-off between robustness and bandwidth consumption has to be made on selecting the maintenance period durations. If the routing information is not maintained frequently enough, the system will not be robust as the routing information becomes outdated quickly. On the other hand, if the routing information is maintained too often, bandwidth consumption will be high[9].

Some other structured P2P systems use structure maintenance techniques that we categorize them in an approach that is based on only system traffic. In these type of structure maintenance techniques, there is no separate procedure for maintaining the routing pointers; instead, any out-of-date or erroneous routing entry is eventually corrected on-the-fly thereby, eliminating periodic bandwidth consumption. However, this approach assumes that the ratio of the number of routing messages to the dynamism in the system is high enough such that there are enough routing messages to correct the routing information. The routing information will become outdated if this ratio is low. Hence, the performance will be poor since a routing hop might lead to a failed node.

In this paper we propose a new structure maintenance approach that allows the system to automatically adapt to the dynamism, while avoiding unnecessary periodic bandwidth consumption. Then we present a family of efficient resource discovery mechanisms by applying a structure maintenance technique based on this new structure maintenance approach.

We analyze the efficiency of our presented resource discovery mechanisms using master equation approach of physics and experiments. We see how the simulation results confirm the theoretical analyses.

Related Works

As stated in Section 1, the arrivals and departures of nodes to/from the system disrupt the system structure and so jeopardize the desired properties of any good structured P2P system. To maintain the system structure under churn, different systems have adopted different techniques to return the system into its ideal structure. By far different structure maintenance techniques used in current structured P2P systems has been studied[8]. We categorize these techniques into two approaches (table-1).

As stated in table -1, most structured P2P systems such as Chord[10], Koorde[20], Viceroy[12], CAN[13], Ulysses[14], Pastry[15] and Tapestry[16] use structure maintenance techniques that fall in the periodic stabilization approach wherein all routing pointers are periodically looked up and updated. The main challenge of this approach is that a trade-off between robustness and bandwidth consumption has to be made. If the routing information is not maintained frequently enough, the system will not be robust as the routing information becomes outdated quickly. On the other hand, if the routing information is maintained too often, bandwidth consumption will be high.

Table-1
Structure Maintenance Approaches and Techniques

Maintenance Approach	Maintenance Technique
Periodic Stabilization	Chord[10], Koorde[11], Viceroy[12], CAN[13], Ulysses[14], Pastry[15], Tapestry[16]
On-Traffic Correction	DKS[17], Self-Contained Techniques[18,19], Kademlia[20]

In contrast, some structured P2P systems[17-20] maintain the system structure with techniques that are based on system traffic. These structure maintenance techniques fall in the category of on-traffic correction approach. These techniques maintain the system structure by piggy-backing technical information on common system messages instead of periodically maintaining the system structure. Although these techniques have lower maintenance traffic compared to techniques that use the periodic stabilization approach, the correction of each out-of-date routing entry depends highly on how frequently this routing entry is used. So techniques that use the on-traffic correction approach do not work efficiently in systems wherein the ratio of the dynamism of the system to the number of transferred messages is high.

In techniques that use the periodic stabilization approach, choosing an appropriate maintenance frequency entails a tradeoff between robustness and bandwidth consumption. A solution suggested by Mahajanet al.[7] is to self-tune the system by dynamically adapting to the operating conditions of the system instead of configuring the maintenance frequency statically and conservatively. Self-tuning requires knowledge about the global state of the system such as the number of nodes in the system and the rate of dynamicity in the system[6]. As there is no central authority in such systems, global system state is figured out by estimation. Additionally, self-tuning is done periodically and has very high communication overhead.

The structure maintenance technique proposed in Chord2[21] uses more stable and powerful nodes as superpeers to reduce the maintenance costs in a structured P2P system. The proposed technique relies on superpeers for correction of routing information, so the failures of any superpeer can jeoperdise the proper working of the maintenance technique.

In Section 4 we propose a new approach for structure maintenance and then present an efficient resource discovery

mechanism that maintains its structure based on this approach under churn. But before that, as we want a common infrastructure for our analysis, we introduce the Chord structured P2P system in the next section.

Chord Overview

In this section, we present an overview of the Chord[10] P2P system that we have used to investigate the feasibility of our proposed approach to be reported in Section 4.

Chord is a distributed lookup protocol and a well-known structured P2P system. It provides a primary operation: it maps a given key to the node that is responsible for that key. A hash function assigns each node and each data item to an identifier in a ring modulo 2^m, called identifier space.

Structure in Chord: Chord is a structured P2P system with a constant pattern of relations between its entities – that is data items and nodes. A data item with the identifier *k* is assigned to the first node whose identifier is equal to or follows *k* in the identifier space denoted by *successor (k)* (pattern of relations between nodes and data items). Each node needs only to maintain a link to its successor (pattern of internode relations). In order to lookup a desired data item with the identifier of *k*, a node can forward the query through the successor links until it reaches the *successor (k)*. It's just like a linear search for an identifier in a list of identifiers.

To make the lookup process scalable, each node also maintains links to *m*other nodes called fingers. The *i*'th finger of the node *n* points to the node *successor* $(n+2^i)$, $1 \le i < m$. By maintaining fingers, linear search turns into a binary search that can locate a node in an *N* node network size in at most *O(log N)* sent messages and by *O(log N)* hops of forwarding queries through fingers.

```
n.join (n′)
1. predecessor := nil;
2. s := n′.find_successor(n);
3. successor := s;
4. build_fingers(s);
n.find_successor(x)
1. if (x ε (n,n.successor])        returnn.successor;
2. else
3.n_nextHop := closest_preceding_node(x);
4.returnnnextHop.find_successor(x);
n.closest_preceding_node(x)
1. forі := m-1 downto 1
2.if (finger[i] ε (n,x))        return finger[i];
3.return n;
n.build_fingers(s)
1.i_0 := [log(successor-n)] + 1;
2.for i_0 ≤ i< m-1
3.finger[i] := s.find_successor(n + 2i);
```

Figure-1
Pseudo code for the Join operation[10]

Node Joins: When a node *n* wishes to join the system, it first contacts an existing node *n′* in the network and asks *n′* to find *n*'s immediate successor. Then *n* can build its finger table with the help of its successor. Figure-1 shows the pseudo-code for the join operation.

Structure Maintenance: To ensure that discovery process executes correctly in a dynamic system that nodes join and fail continuously, each node's successor pointer must always be up-to-date. This is assured by using a "stabilization" protocol that each node periodically executes and updates successor pointers (figure-2)[10].

```
n.stabilization()
1.check_predecessor();
2.   x := successor.predecessor;
3.   if (x ε (n, successor))
// successor changed due to new node
4.successor := x;
5.successor.notify(n);
s.notify(n)
1.if (predecessor = nil or n ε (predecessor, s))
2.   predecessor := n;
n.check_predecessor()
1.if (predecessor has failed)   predecessor := nil;
n.fix_successor_list()
1. <s1, . . . ,sr>:= successor.successor_list;
2.successor_list := <successor, s1, . . . , sr-1>;
n.fix_successor()
1.  if (successor has failed)
2.     successor := smallest alive node in successor_list;
```

Figure-2
Pseudo-code for the structure maintenance[10]

After node *n* joins the system, some nodes that are pointing to *n*'s successor in their finger table, should update their finger table. Because intrinsically, the join operation does not make the remainder of the network aware of *n*, nodes have no idea when fingers should be updated. To solve this problem, Chord allows each node to periodically execute *fix_fingers()* to keep fingers updated (figure-3).

```
n.fix_fingers()
1.    build_fingers(n);
```

Figure-3
Periodically refreshing the whole finger table[10]

Analysis: In the Chord maintenance algorithm, the execution of *stabilization()* costs four messages, while the execution of *fix_fingers()* costs $O(log2^N)$ messages (because *logN* executions of *find_successor()* are generated, each of which costs at most *logN* messages). So the maintenance of finger tables accounts for most of the overhead.

Proposed Approach and Technique

Working with structure maintenance techniques that use the periodic stabilization approach, all pointers in the system are updated periodically, creating a lot of overhead. They do not guarantee the robustness of the system either. In our proposed structure maintenance approach, each node needs not to periodically update all its pointers. Each node only checks a very small number of pointers periodically and by detection of a change in them, calls a mechanism that updates all the other pointers in the system that need to be updated.

Overview: As we want to show that a family of efficient resource discovery mechanisms can be constructed by applying our approach on existing structured P2P systems, we first need to show its applicability to a specific structured P2P system. We selected the Chord structured P2P system as it is very popular and commonplace. So here we present a technique derived from our approach for structure maintenance and apply it to the Chord system.

Application on Chord: The Chord maintenance technique presented in Section 3 uses a periodical scheme for routing table maintenance that frequently refreshes all the routing table entries of all nodes. To lessen the maintenance overhead and increase the robustness of Chord, we remove the costly periodic routing table maintenance in our technique for Chord system by using a lighter event-oriented version of the periodic successor maintenance.

To illustrate what exactly structure perturbation means, let's consider two cases in a ring (figure-4.). When a node *b* joins the system between the nodes *a* and *c*, the responsibility of the ring area that lies between *a* and *b* transfers from *c* to *b*, so some nodes that have a routing table entry targeting in this range of ring, should update some of their finger table entries from *c* to *b*. On the other hand, when a node *b* that lies between nodes *a* and *c* leaves the system or fails, node *c* becomes responsible for the ring area between *a* and *b*, so some nodes that had routing table entries pointing to *b*, should update some of their fingers to point to *c* instead of the failed *b*.

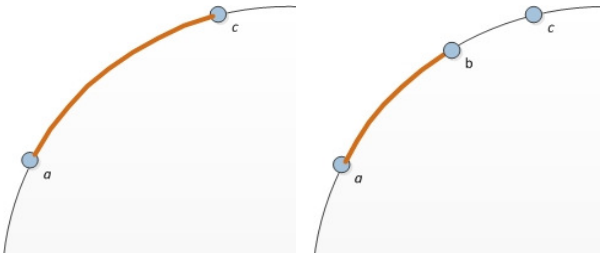


Figure-4
Changes in the responsibility of ring areas when node *b* joins or leaves the system

Upon detection of a structure perturbation, we return the system into its structured shape by activating a structure maintenance protocol to identify the affected nodes and notify them to update their affected fingers. To effectively identify the affected nodes, each node and its predecessor, store objects called pointer objects. Upon every structure perturbation and membership change, these pointer objects identify and notify the affected nodes in parallel.

In the next section we look closely at node join and failure operations in our technique and illustrate how the system structure is maintained in case of membership changes.

Structure Maintenance Technique: In order to reduce maintenance costs in our technique, we have removed executing *fix_fingers()* and made the predecessor of a finger's target responsible for maintenance of the finger. More precisely, each node and its predecessor store objects called pointer objects. An object has the following format:
pointer_object = <*source ,levels*>

where *source* is a node handle (a triple of <*IP address; UDP port; Node ID*>) to a node that has at least one routing table entry pointing to *n* and *levels* is a binary string of length *logN* and of the form:
levels[i]=1 if(source. finger[i]==n), i=0…logN

By means of these pointer objects, in case of joining, *n* can know which nodes in the system have fingers targeting to *n. successor* that now should update their fingers, and then can direct them to replace *n. successor* with *n* in their finger tables. In addition, when *n* departs the system or fails, its predecessor can know which nodes have fingers pointing to *n* with the help of its successor pointer objects, and then can direct them to replace *n* with *n. successor* in their finger tables.

The join operation in our system is similar to the one in Chord. In addition, we require *n* to create its pointer objects with the help of its *successor*. The following is the new join procedure.

```
n.join(n′)
1.   successor = n′.findSuccessor(n);
3.predecessor = successor.predecessor;
4.predecessor.ChangeSuccessor(n);
5.successor.notifyJoin(n);
6.buildFinger();
n.notifyJoin(x)
1.predecessor.successor = x;
2.predecessor = x;
3.for each pointer object obj
4.transfer a copy of obj to x;
5.if(obj is not pointing between x and n)
6.obj.UpdateFinger(x);
7.removeobj;
```

Figure-5
Node joining with agile structure maintenance in our technique

Note that when a node *n* joins the system, three operations should be done to bring the system states up-to-date:
-Update successor and predecessor pointers of its neighboring nodes
-Update the fingers in the system that were pointing to *n.successor* and now should point to *n* itself.
-Update and set pointer objects of *n* and its neighbors

In the above pseudo-code, the first three lines of *Join()* function and the first two lines of *notifyJoin()* update successor and predecessor pointers of nodes *n*, *n.successor* and *n.predecessor*. Also line 4 of the *join()* function process notifies node *n.successor*. Each node checks its pointer objects upon receiving the notification and updates its pointer objects, doing so creates and transfers the proper pointer objects to *n*. The pointer objects are examined as follows: For each object <*source, levels*> (meaning that some fingers of source point to *n*), if *levels[i]==1* and $source+2^i \varepsilon(predecessor,n]$, the object should be transferred to *n* and this object should be deleted from *n.successor*'s pointer objects.

As stated in Section 3, it takes some time for nodes *a* and *c* to detect the joining of node *b* in Chord, but this makes system less robust and makes resource discovery faulty in some occasions when some successor pointers become invalid between periods. In our proposed mechanism we modified the join operation to immediately make both *a* and *c* aware about the arrival of *b*. Doing so we ensure that all the successor and predecessor pointers are updated nearly immediately after a node joins the system[21,22].

In our technique, each node *n* still executes the stabilization procedure periodically to detect the failures of nodes. But in contrast to Chord, our stabilization is a very light process that for each node only checks the aliveness of its successor. Figure-6 shows the pseudo-code for stabilization process wherein each node starts informing affected nodes upon detection of its successor's failure by using its successor's pointer objects.

```
//node n periodically executes stabilization to detect failed successors
n.stabilize()
1. succold=successor;
2. n.fix_successor(); //if successor has failed fixes it
3. if succold ϵ (n, successor)  //node succold has left
4.successor.notifyLeave(n);
5.     for all successor pointer objects obj
6.         obj.updateFinger(successor);
7.         successor.addPointerObj(obj);
//restructuring after leave of failure detection
n.notifyLeave( x)
1.  predecessor = x;
2.for all pointer objects obj
3.transfer a copy to x;
```

Figure-6
Stabilization and failure detection

In our maintenance technique, in order to return the system to its structure upon detection of a node failure, three operations should be done:v i. Update successor and predecessor pointers of its neighboring nodes, ii. Update the fingers in the system that were pointing to *n* and now should point to *n.successor*, iii. Update pointer objects of *n.successor*and *n.predecessor.*

Again with respect to figure-4 when a node *b* that lies between nodes *a* and*c* leaves the system, node *a* detects *b*'s failure after *a*'s first stabilization process. Then as all nodes that were pointing to *b* should now point to *c*, *a* notifies all of its successor pointer objects to change their fingers to *c* instead of the failed *b*. At last *a* transfers a copy of its successor pointer objects to *c* and receives *c*'s pointer objects. After that, pointer objects of both *a* and*c* are eventually updated.

Efficiency Analysis

In order to being able to compare the efficiency of resource discovery mechanisms that makes use of each structure maintenance approach, we need to first apply each approach on the same specific structured P2P system and then compare their efficiency. Asa periodic approach we have the main implementation of Chord that is studied in section 3. As an example of using the on-traffic correction approach, we assume the technique used in DKS[17]system to be applied to Chord with minimal modification. At last, as an example of using our approach, we have the Chord system using the technique proposed completely in Section 4.

Now in this section we want to make a comparison and show that by applying our structure maintenance approach to any structured P2P system, we can make an efficient resource discovery family. Our analysis are based on constructing and working with master equations[23], a widely used tool wherever the mathematical theory of stochastic processes is applied to real-world phenomena.

Previously a master equation analysis of the main Chord structured P2P system with the periodic structure maintenance technique has been presented[10,21,22]. Here we analyze the Chord resource discovery mechanism that uses our structure maintenance technique and then the Chord resource discovery based on the on traffic correction approach and at last compare the efficiency of the resource discovery mechanism in the presence of churn.

As stated in section 2, the efficiency of each resource discovery mechanism is completely depended on the number of incorrect pointers in the system, so here we are going to compare these three resource discovery mechanisms by computing the average number of incorrect node pointers in the Chord system with a structure maintenance technique based on each approach.

Basic Assumptions: Here we introduce the notation used in our theoretical analysis. We use *K* to mean the size of the Chord key

space and N the number of nodes. Let $M = log_2 K$ be the number of fingers of a node and S the length of the immediate successor list, usually set to a value of $O\ (log(N))$. We refer to nodes by their keys, so a node n implies a node with key $n \in 0 \cdots K-1$. We use p to refer to the predecessor, s for referring to the successor list as a whole, and s_i for the ith successor. Data structures of different nodes are distinguished by prefixing them with a node key e.g. $n'.s_i$, etc. Let $fin_i.start$ denote the start of the ith finger (Where for a node n, $\forall i \in 1 \cdots M$, $n.fin_i.\ start = n + 2^{i-1}$ and $fin_i.node$ denote the actual node pointed to by that finger.

λ_j is the rate of joins per node, λ_f the rate of failures per node, λ_s the rate of stabilizations per node and λ_q is the rate of queries raised per node. We carry out our analysis for the general case when the rate of doing successor stabilizations is $\alpha\lambda_s$, is not necessarily the same as the rate at which finger stabilizations $(1-\alpha)\lambda_s$ are performed. In all that follows, we impose the steady state condition $\lambda_j = \lambda_f$. Further it is useful to define $r \equiv \frac{\lambda_s}{\lambda_f}$ and $r' \equiv \frac{\lambda_q}{\lambda_f}$ which are the relevant ratio on which all the quantities we are interested in will depend, e.g, $r = 50$ means that a join/fail event takes place every half an hour for a stabilization which takes place once every 36 seconds. The parameters of the problem are hence: K, N, α and r. All relevant measurable quantities should be entirely expressible in terms of these parameters.

Analysis of Efficiency Based on Our Technique: In this section we first want to compute the number of incorrect pointers in Chord system with the structure maintenance technique presented in section 4 based on our approach.

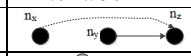
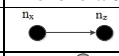
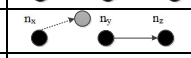
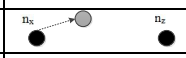
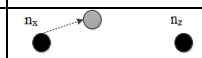
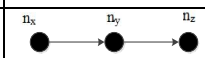
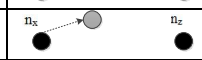
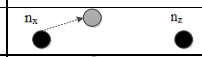
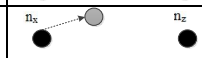
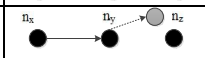
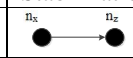
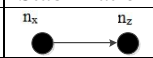
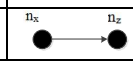
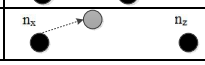
In order to get a master-equation description which keeps all the details of the system and is still tractable, we make the definition that the state of the system is the product of the states of its nodes, which in turn is the product of the states of all its pointers. Now we need only consider how many kinds of pointers there are in the system and the states these can be in. Consider first the successor pointers:

Let's assume $w(r, \alpha)$ and $d(r, \alpha)$ the number of nodes that their successor pointer is incorrect of failed and $W(r, \alpha)$ and $D(r, \alpha)$ the corresponding size of these sets.

In our structure maintenance technique, each node periodically contacts its first successor, possibly correcting it and reconciling with its successor list. Therefore, the numbers of wrong kth successor pointers are not independent quantities but depend on the number of wrong first successor pointers[24].

We consider only s_1 here. We write an equation for $W_1(r, \alpha)$ by accounting for all the events that can change it in a micro event of time Δt. An illustration of the different cases in which changes in W_1 take place due to joins, failures and stabilizations is provided in table-2.

Table-2
Changes in W_1, number of incorrect successors

$W_t(t+\Delta t)$	After a Join	Before a Join
+1		
0		
$W_t(t+\Delta t)$	After a Failure	Before a Failure
+1		
-1		
0		
+1-1=0		
$W_t(t+\Delta t)$	After Successor Stabilization	Before Successor Stabilization
0		
-1		

In some cases W_1 increases/decreases while in others it stays unchanged. For each increase/decrease, table -3 provides the corresponding probability.

By the implementation of the join protocol, a new node n_y, joining between two nodes n_x and n_z, has its s_1 pointer always correct after the join. However the state of $n_x.s_1$ before the join makes a difference. If $n_x.s_1$ was correct (pointing to n_z) before the join, then after the join it will be wrong and therefore W_1 increases by 1. If $n_x.s_1$ was wrong before the join, then it will remain wrong after the join and W_1 is unaffected. Thus, we need to account for the former case only. The probability that $n_x.s_1$ is correct is $1-w_1$ and from that follows the term c_1.

For failures, we have 4 cases. To illustrate them we use nodes n_x, n_y, n_z and assume that n_y is going to fail. First, if both $n_x.s_1$ and $n_y.s_1$ were correct, then the failure of n_y will make $n_x.s_1$ wrong and hence W_1 increases by 1. Second, if $n_x.s_1$ and $n_y.s_1$ were both wrong, then the failure of n_y will decrease W_1 by one, since one wrong pointer disappears.

Third, if $n_x.s_1$ was wrong and $n_y.s_1$ was correct, then W_1 is unaffected. Fourth, if $n_x.s_1$ was correct and $n_y.s_1$ was wrong, then the wrong pointer of n_y disappeared and $n_x.s_1$ became wrong, therefore W_1 is unaffected. For the first case to happen, we need to pick two nodes with correct pointers, the probability of this is $(1 - W_1)^2$. For the second case to happen, we need to pick two nodes with wrong pointers, the probability of this is W_2 From these probabilities follow the terms c_2 and c_3.

Table -3
Gain/Loss functions for W1

Rate of Changes	$F_k(t+\Delta t)$
$c_1 = (\lambda_j \Delta t) f_k P_{join}$	$F_k(t) + 1$
$c_2 = (\alpha \lambda_s \Delta t) D(r, \alpha)$	$F_k(t) - 1$
$c_3 = (\lambda_f \Delta t)(1 - f_k)^2 [1 - p_1(k)]$	$F_k(t) + 1$
$c_4 = (\lambda_f \Delta t)(1 - f_k)^2 [p_1(k) - p_2(k)]$	$F_k(t) + 2$
$c_5 = (\lambda_f \Delta t)(1 - f_k)^2 [p_2(k) - p_3(k)]$	$F_k(t) + 3$

Finally, successor stabilization does not affect W_1, unless the stabilizing node had a wrong pointer. The probability of picking such a node is w_1. From this follows the term c_4.

Hence the equation for $W_1(r, \alpha)$ is: $\frac{dW}{dt} = \lambda_f(1-W) + \lambda_f(1-W)^2 - \lambda_f W^2 - \alpha\lambda_s W$ That after solving this equation and letting $\lambda_f = \lambda_j$ we have: $W(r, \alpha) = \frac{2}{3+r\alpha} \approx \frac{2}{r\alpha}$

And as half of incorrect successor pointers are failed and half are live wrong ones, the number of failed successor pointers is: $D(r, \alpha) \approx \frac{1}{r\alpha}$.

We now turn to estimating the fraction of finger pointers which point to failed nodes. This is an important quantity for predicting lookups. Let $f_k(r, \alpha)$ denote the fraction of nodes having their k-th finger pointing to a failed node and $F_k(r, \alpha)$ denote the respective number. For notational simplicity, we write these as simply F_k and f_k. We can predict this function for any k by again estimating the gain and loss terms for this quantity, caused by a join, failure or stabilization event, and keeping only the most relevant terms. These are listed in table-4.

A join event can play a role here by increasing the number of F_k pointers if the successor of the joinee had a failed k-th pointer (occurs with probability f_k) and the joinee replicated this from the successor (assume that occurs with probability $p_{join}(k)$[10]).

Successor stabilization here detects a change in successor. On detection of a failed successor (with the probability of $D(r, \alpha)$) the node notifies all the nodes in the system that point to the failed node to update their affected finger.

Given a node n with an alive k-th finger (occurs with probability of $(1-f_k)$, when the node pointed to by that finger fails, the number of failed k-th fingers (F_k) increases. The amount of this increase depends on the number of immediate predecessors of n that were pointing to the failed node with their k-th finger. That number of predecessors could be *0, 1, 2,..*etc.

As shown in[21], the respective probabilities of those cases are: $1 - p_1(k)$, $p_1(k) - p_2(k)$, $p_2(k) - p_3(k)$,... etc.

Table -4
Changes in F_k, the number of incorrect k-th finger

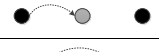
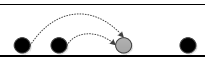
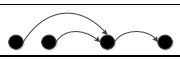
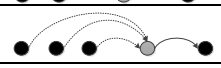
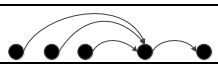
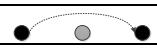
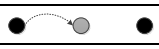
$F_k(t+\Delta t)$	After Join	Before Join
+1		
$F_k(t+\Delta t)$	After Failure	Before Failure
+1		
+2		
+3		
…	…	…
$F_k(t+\Delta t)$	After Successor Stabilization	Before Successor Stabilization
-1		

Table-5
Gain/Loss Terms for $F_k(r,\alpha)$

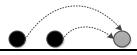
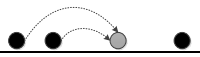
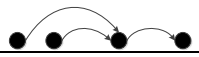
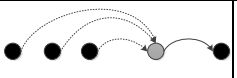
Rate of Change	$F_k(t+\Delta t)$
$c_1 = (\lambda_j \Delta t) f_k P_{join}$	$F_k(t) + 1$
$c_2 = (\alpha \lambda_s \Delta t) D(r, \alpha)$	$F_k(t) - 1$
$c_3 = (\lambda_f \Delta t)(1 - f_k)^2 [1 - p_1(k)]$	$F_k(t) + 1$
$c_4 = (\lambda_f \Delta t)(1 - f_k)^2 [p_1(k) - p_2(k)]$	$F_k(t) + 2$
$c_5 = (\lambda_f \Delta t)(1 - f_k)^2 [p_2(k) - p_3(k)]$	$F_k(t) + 3$

Solving in the steady state, we get an equation for F_k. here we don't care that equation but, we want only compare it with that in those other approaches, so we look at these with another sight: With a careful attention on the term c_2 we can see that the finger stabilization in our technique is like a complete periodic technique with the period of $\alpha\lambda_s D(r,\alpha)$ that is $\alpha.\lambda_s/r.\alpha$ that is: λ_f. This result completely confirms the simulation results presented in section 6 that shows the system pointers remaining updated all the times.

In next part of this section we overview the number of incorrect pointers in the Chord system that uses structure maintenance technique based on the on traffic correction approach.

Analysis of Efficiency based on On-Traffic Correction Structure Maintenance Technique: Here we want to show the results of our theoretical analyses of Chord system with a structure maintenance technique that is based on the traffic correction approach. A technique like this is presented for DKS system[17] that is a system based on Chord. As in this technique all the system pointers are considered the same, we don't talk about successor and finger maintenance separately. First let's look at changes in system pointers in the face of each process in the system life-time in table-6.

Table-6
Changes in F_k, the number of incorrect k-th finger

$F_k(t+\Delta t)$	After Join	Before Join
+1		
$F_k(t+\Delta t)$	Before Failure	After Failure
+1		
+2		
+3		
…	…	…
$F_k(t+\Delta t)$	After Receiving a Query	Before Receiving a Query
-1		

Here the difference with our analysis in first part is only in stabilization part. As there is no periodical stabilization in this approach, we eliminate the stabilization from those calculations but we should consider the fact that the cause of correction for pointers here is receiving a query in a node. As stated before in A, considering the rate of queries that each node raises in the system to be λ_q, with the knowledge that in average each query will be live in the system for $logK$ hops and so can correct $(logK-1)$ pointers in its way to destination. So first, the number of current queries in the system in each time will be $N.\lambda_q.\log K$ and so the rate of reaching a node and being able to correct a system pointer is $\lambda_q.(\log K-1)$, that this node in the case of using its kth level finger for forwarding the query (with the probability of K) and failure of this finger (with the probability of $f_k(r, \alpha)$), that finger will be corrected and so the number of incorrect fingers in kth level will become minus one.

Table -7
Gain/Loss Terms for $F_k(r,\alpha)$

Rate of Change	$F_k(t+\Delta t)$
$c_1 = (\lambda_j \Delta t) f_k P_{join}$	$F_k(t)+1$
$c_2 = \left(\lambda q(\mathrm{M}-1) \times \frac{1}{M}\Delta t\right) f_k$	$F_k(t)-1$
$c_3 = (\lambda_f \Delta t)(1-f_k)^2[1-p_1(k)]$	$F_k(t)+1$
$c_4 = (\lambda_f \Delta t)(1-f_k)^2[p_1(k)-p_2(k)]$	$F_k(t)+2$
$c_5 = (\lambda_f \Delta t)(1-f_k)^2[p_2(k)-p_3(k)]$	$F_k(t)+3$

Again here let's look at this approach at a point of view of comparison with other approaches. We can see that a system that uses a structure maintenance technique of this family will behave like a periodical technique with a stabilization rate of $\lambda q \times \frac{\mathrm{M}-1}{\mathrm{M}}$.

Comparison: In this part we want to compare these three techniques in the sense of the efficiency of the resource discovery mechanism based on each technique.

With respect to our analyses in previous parts we presented the theoretical analysis of the efficiency of resource discovery mechanism based on three techniques each from one approach to structure maintenance.

As explained in section 4, we can compare the efficiency of two resource discovery mechanisms by comparing the number of correct routing pointers in them in the steady state.

As studied in section 5, for the sake of comparison, we can consider all of three approaches like a periodic approach but with different stabilization rate. The rate of stabilization for routing table pointers in the periodic technique is $(1-\alpha)\lambda_s$, for the on-traffic correction is $\frac{M-1}{M}\lambda_q$, and for our technique is λ_f. With these values we can make good comparison on the efficiency of resource discovery mechanisms.

For example, for a comparison between the efficiency of the resource discovery in a system using on-traffic technique and our technique, we will have better performance in situations that: $\frac{M-1}{M}\lambda_q \geq \lambda_f$
After solving this equation we will have:
$r' \geq \frac{M}{M-1}$
And this means that for example in a peer-to-peer system with a 1024 number of nodes, if $r' = \frac{\lambda_q}{\lambda_f} \geq 1.1$, then the performance of a resource discovery using the on-traffic structure maintenance technique will be higher than the resource discovery mechanism that uses our approach.

Table -8
Theoretical comparison of Structure Maintenance Approaches

Lower Overhead	Higher efficiency				
	If $r' < \frac{M}{M-1}$		If $r' \geq \frac{M}{M-1}$		
	If $\frac{r'}{r} < \frac{M(1-\alpha)}{M-1}$	If $\frac{r'}{r} \geq \frac{M(1-\alpha)}{M-1}$	If $r < \frac{1}{1-\alpha}$	If $r \geq \frac{1}{1-\alpha}$	
3'rd	2'nd	3'rd	3'rd	2'nd	Periodic
1'st	3'rd	2	1'st	1'st	On-Traffic
2'nd	1'st	1'st	2'nd	3'rd	Our Approach

Above shown data states that our approach will be the best in the mean comparison.

Maintenance Overhead: Assume a Chord ring with maximum of *N* nodes. When a node joins the ring, we need to build its pointer objects by receiving pointer objects of its successor. The number of nodes pointing to a node is *O(log N)* with high probability. With this in mind, the number of messages will be *O(log N)*.[10] After that we should notify the affected nodes to update their affected fingers with the cost of *O(logN)* messages. After transferring these messages, all system states will become up-to-date. But the dominating task in the join operation is still construction of the finger table, that like Chord needs $O(\log^2 N)$ messages. So all together the cost of join is of the order of $O(\log^2 N)$.

When a node *n* leaves the system, its departure will be detected on the next stabilization of its predecessor. First *O(log N)* nodes should be notified to update their fingers to point to *n.successor* instead of *n*. It then takes *O(logN)* messages to transfer pointer objects from *n. predecessor* to *n.successor* and vice versa to maintain the pointer objects. When a node*n* leaves the system, its finger table entries should delete *n* from their pointer objects. As this will be done at working time of system, the leaving of a node costs *O(log N)* messages.

Each node runs a stabilize process periodically that costs only a message for each node to check the availability of its successor. So the stabilize process in our system is of the order *O(1)*.

Table -9 summarizes the comparison. Our system has reduced the maintenance cost by removing the costly periodic maintenance of fingers with the cost of $O(\log^2 N)$ to the only successor maintenance with the cost of *O(1)*. As shown in results section, in each unit of time the number of finger table entries in the system that are pointing to the correct node is far more than the number of correct fingers in Chord.

Table-9
Maintenance overhead of Chord using periodic structure maintenance and our proposed system

	Chord	**Our Proposed Mechanism**
Join	$O(\log^2 N)$	$O(\log^2 N)$
Leave	--	O(log N)
Periodical Maintenance	$O(\log^2 N)$	O(1)

Finally, we comment the load added to nodes. For storage, recall that in our system each node stores *O(log N)* fingers and *2*O(log N)* pointer objects. So again all together each node in our system still stores *O(log N)* states which is aligned with the structured nature of P2P systems. In addition we can make this real world assumption that storage is not a critical concern in nodes at all.

Simulation Results

We first show the difference between periodic stabilization in main Chord system and our approach to structure maintenance. Figure-7 shows a simulation of a system with *N=29* nodes. The nodes arrive and depart every *2* time units. The curves show the amount of maintenance bandwidth consumed by a Chord system running periodic stabilization and a system running our structure maintenance approach.

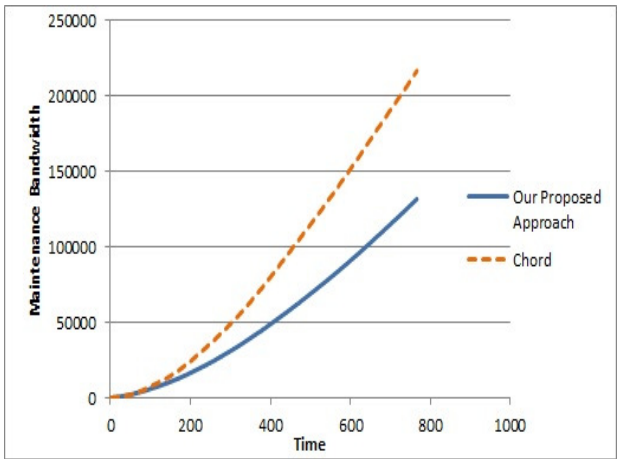


Figure-7
The maintenance traffic for the Chord system running periodic successor stabilization every *10* time units and *fix_finger* every *30* time units and our proposed system running light successor stabilization every *5* units. A leave and join event occured every *2* time units on average. The robustness of the system for these simulations is shown in Figure-8.

Stabilization rate in our proposed mechanism was set to *5*. In Chord, the stabilization rate was set to *10* and fix finger period was set to *30* such that the amount of maintenance bandwidth became equal in both systems. Figure-8 shows the robustness of these systems. Although both systems had the same maintenance cost, our system was maintained approximately in legitimate state as expected while approximately half of the routing pointers in Chord were incorrect.

We now show the reverse by fixing the robustness close to optimal for Chord and our proposed system, i.e. the routing state in both systems is set to a legitimate state, to investigate the amount of maintenance bandwidth consumed in respective systems. We experimented with many different stabilization and finger table maintenance rates to find one which matched the dynamism such that the system would be in approximately legitimate state at all times.

Figure-9 clearly shows that both systems are approximately maintained in a legitimate state. However, as Figure-10 shows, periodic stabilization consumes significantly more traffic than our proposed system.

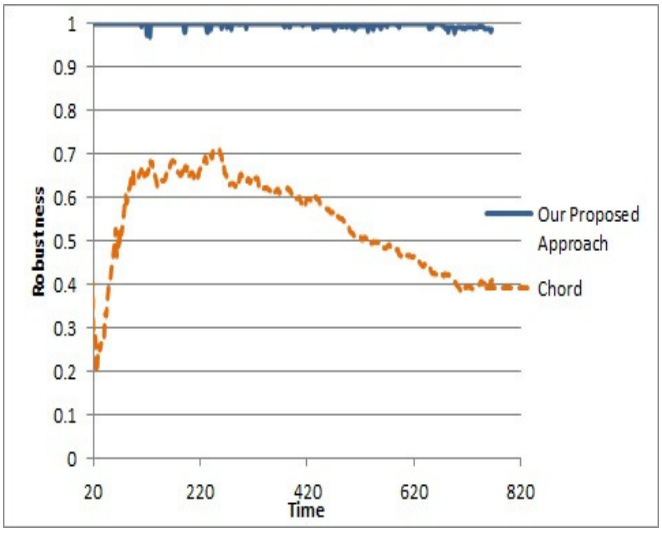


Figure-8

Robustness of the Chord system running periodic successor stabilization every *10* time units and periodic fixing of fingers every *30* time units, and the robustness of our proposed system running periodic stabilization every *5* time units. A leave and join event happened every *2* time units on average. Both systems consumed approximately the same amount of maintenance bandwidth as shown in figure-10

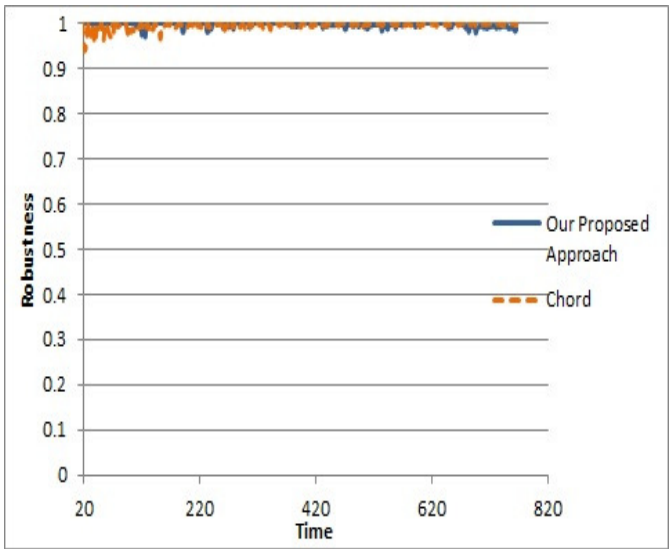


Figure-9

Robustness of the Chord system running periodic stabilization every 1 time unit and periodic fixing of fingers every 4 time units, and the robustness of our proposed system running stabilization every 5 time units. A leave and join event occurred every 2 time units on average. Both systems had a legitimate state deviation close to 0 indicating that the system is approximately in a legitimate state (robust)

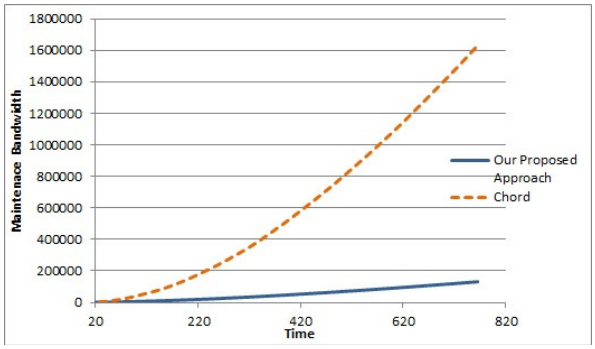


Figure-10

The maintenance traffic for the Chord system running periodic stabilization every *1* time unit and periodic fixing of fingers every *4* time units, and for our proposed system running stabilization every *5* time units. A leave and join event occurred every *2* time units on average. Figure-10 shows the robustness for these simulations

Conclusion

We presented a general approach to the maintenance of structured P2P overlay networks in the face of membership change (churn) and showed how the approach can be applied to the Chord structured Peer-to-Peer system. We showed that the proposed approach increases system robustness while keeping the maintenance cost low. In our system, bandwidth was consumed only when necessary.

Experimental results showed that for the same amount of maintenance bandwidth, our proposed approach made the system by far more robust when compared to periodic stabilization. Moreover, even when a periodic stabilization that adapts itself perfectly to the dynamism in the system was used, our system yielded the same performance but with a small fraction of the maintenance cost of periodic stabilization.

By applying our approach on every structured P2P system we will have an efficient resource discovery, like the one showed and experimented for Chord system.

References

1. Tanenbaum AS and Van Steen M., Distributed Systems: Principles and Paradigms, 2nd ed., New Jersey: Prentice Hall Press, **(2006)**
2. Mewada Shivlal and Singh Umesh Kumar, Performance Analysis of Secure Wireless Mesh Networks*, Res.J.Recent Sci.,* **1(3)**, 80-85 **(2012)**
3. Yao Z. and Loguinov D., Analysis of Link Lifetimes and Neighbor Selection in Switching DHTs, IEEETransactions on Parallel and Distributed Systems, **22(11)**, 1834-1841, **(2011)**
4. Rao W, Chen L, chee Fu AW and Wang G, Optimal Resource Placement in Structured Pee-to-Peer Networks,

IEEE Transactions on Parallel and Distributed Systems, **21(7)**, 1011-1026, **(2010)**

5. Karger E. Lehman, T. Leighton, R. Panigrahy and M. Levine, Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web, in In STOC '97: Proceedings of the 29th annual ACM symposium on theory of computing, New York, **(1997)**

6. Zhang Q., Miao Z., Zhang Y., Xu W. and Du Y., Multi-Attribute Resource Discovery in Structured P2P Networks, Proceedings of the 9th International Symposium on Linear Drives for Industry Applications, **2(1)**, Lecture Notes in Electrical Engineering, **271,** 501-508, (**2013)**

7. R. Mahajan, M. Castro and A. Rowstron., Controlling the Cost of Reliability in Peer-to-Peer Overlayss, in 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), Berkeley, CA, USA, (**2003)**

8. S. El-Ansary, Designs and Analyses in Structured P2P Systems, Ph.D. Thesis, Department of Microelectronics and Information Technology, The Royal Institute of Technology (KTH), Stockholm, Sweden, **(2005)**

9. Krishnamurthy S., El-Ansary S., Aurell E. and Haridi S., Comparing Maintenance Strategies for Overlays, in Parallel, Distributed and Network-Based Processing, Toulouse, France, (**2008)**

10. Stoica, Morris R., Liben-Nowell D., Karger D., Kaashoek M.F., Dabek F. and Balakrishnan H., Chord: A scalable Peer-to-Peer Lookup Service For Internet Applications," in Transactions on Networking, (**2003)**

11. Karger, F. Kaashoek and D.R, Koorde: A simple degree optimal distributed hash table, in 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), Berkeley, CA, USA, February, (**2003)**

12. Malkhi NR, Viceroy: A Scalable and dynamic Emulation of the Butterfly, in Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC '02), Monterey, California, August (**2002)**

13. Ratnasamy S, Francis P., Handley M, Karp R and Shenker S, A Scalable Content Addressable Network, in ACM SIGCOMM '01 Conference, Berkeley, CA, (**2001)**

14. Kumar, S. Merugu, J. Xu and E. W. Ze, "Ulysses: A Robust, Low-Diameter, Low-Latency Peer-to-Peer Network," in ICNP '03 11th IEEE International Conference on Network Protocols, Washington, DC, USA, (**2003)**

15. Druschel P and Rowstron A, Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-To-Peer Systems, in IFIP/ACM International Conference on Distributed Systems Platforms, (**2001)**

16. Zhao Y., Huang L., Stribling J. and Rhea S.C., Tapestry: A Resilient Global-Scale Overlay for Service Deployment, IEEE Journal on Selected Areas in Communications, **22(5)**, 41-53, (**2004)**

17. Alima L.O., El-Ansary S., Brand P. and Haridi S., DKS(N,k,f): A Family of Low Communicatio, Scalable and Fault-Tolerant Infrastructures for P2P Applications, in CCGRID2003- International Workshop on Global and Peer to Peer Computing on Large Scale Distributed Systems, Tokyo, Japan, (**2003)**

18. Aberer K., Datta A. and Hauswitrh M., "Route Maintenance Overheads in DHT Overlays," The 6th Workshop on Distributed Data and Structures, EPF Lausanne, Switzerland, July 8-9, (**2004)**

19. Aberer K, Datta A and Hauswirth M, Efficient, Self Contained Handling of Identity, IEEE Transactions on Knowledge and Data Engineering, **16(2)**, 36-54, (**2004)**

20. Maymounkov P. and Mazières David, Kademlia: A Peer-to-Peer Information System Based on the XOR Metric, in Peer-to-Peer Systems, 2429, Springer Berlin / Heidelberg, 53-65, (2002)

21. Arbabi M. Sharifi et.al., Mirtaheri SL and Mousavi Khaneghah SE, A Low Overhead Structure Maintenance Approach for Building Robust Structured P2P Systems, IST, Tehran, (**2012)**

22. Analysis of G-CSF Treatment of CN using Fast Fourier Transform, Balamuralitharan S. and Rajasekaran S., Res. J. Recent Sci., **1(4)**, 14-21**(2012)**

23. A branch-and-bound procedure for resource leveling in multi-mode resource constraint project scheduling problem, Afshar-NadjafiBehrouz, NajjarbashiHojjat and MehdizadehEsmaeil, *Res.J.Recent Sci.,***1(7),** 33-38 **(2012)**

24. Krishnamurthy S., El-Ansary S., Aurell E. and Haridi S., An Analytical Study of a Structured Overlay in the Presence of Dynamic Membership*, IEEE Transactions on Networking,* **16(4)**, 814-825, (**2008)**

25. Krishnamurthy S., El-Ansary S., Aurell E. and Haridi S., A Statistical Theory of Chord under Churn, in 4th Int. Workshop on Peer-to-Peer Systems (IPTPS'05), Ithaca, NY, (**2005)**

26. El-Ansary S. and Hardi S., An Overview of StructuredOverlay Networks, in Handbook of Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, Stockholm, Auerbach, 665-683, (**2006)**

27. Behmaneshfar Ali, Shahbazi S. and Vaezi S., Analysis of the Sampling in Quality Control Charts in non uniform Process by using a New Statistical Algorithm, *Res.J.Recent Sci.,***1(8),** 36-41 **(2012)**

28. Iyer K. and Khan Z.A., Depression-AReview, *Res.J.Recent Sci*., **1(4)**, 79-87 **(2012)**