ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/301790209

LARAS: Locality Aware Replication Algorithm for the Skip Graph

Conference Paper · April 2016

DOI: 10.1109/NOMS.2016.7502828

CITATIONS
11

READS
200

3 authors, including:

Yahya Hassanzadeh Nazarabadi
Dapper Labs
24 PUBLICATIONS **103** CITATIONS
SEE PROFILE

Oznur Ozkasap
Koc University
171 PUBLICATIONS **2,156** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Dual Sensor Platform Robot View project

Project Fair and Secure Computation View project

All content following this page was uploaded by Yahya Hassanzadeh Nazarabadi on 19 April 2018.

The user has requested enhancement of the downloaded file.

LARAS: Locality Aware Replication Algorithm for the Skip Graph

Yahya Hassanzadeh-Nazarabadi, Alptekin Küpçü and Öznur Özkasap
Department of Computer Engineering, Koç University, İstanbul, Turkey
{yhassanzadeh13, akupcu, oozkasap}@ku.edu.tr

***Abstract*—Skip Graph, a member of the distributed hash table (DHT) family, has several benefits as an underlying structure in peer-to-peer (P2P) storage systems. In such systems, replication plays a key role on the system's performance. The traditional decentralized replication algorithms do not consider the locations of Skip Graph nodes in the network. Negligence of node locations in the placement of the replicas results in high access delays between the nodes and their closest replicas. This negatively affects the performance of the whole storage system. In this paper, with the aim of making Skip Graph's replication locality aware, we propose dynamic fully decentralized LARAS approach, where the data owner can replicate itself based on the system size, possible data requester nodes' set and using local information of the storage system. Our extensive performance results show that LARAS improves replication access delay of the Skip Graph based storage system about 20% and 38% in comparison to the best known decentralized counterpart in the public and private replication scenarios, respectively.**

I. Introduction

Skip Graph is a member of the DHT-based distributed data structures [1]. The ability of performing concurrent search operations, scalability and fast searching, make the Skip Graph a suitable underlying infrastructure for the P2P storage applications [2–4]. The Skip Graph, can also be considered as an alternative for the other members of the DHT-based distributed data structures family in their storage applications [5], [6].

A P2P storage system consists of a group of nodes. In such systems, each node can be both a data owner and a data requester. The data owner, owns some data items and wants to share them with a specific sub-group of the data requester nodes. Each member of this sub-group may query any of those data items at any time. To reduce the query load of the data owner, provide data availability in the event of the data owner failure and data recovery, the data owner can replicate its data items on some nodes named replicas. The data requester nodes would then be divided into smaller groups. Each group is assigned to a certain replica. All data requesters in a certain group would then query their corresponding replica instead of querying the data owner.

Traditional decentralized replication algorithms aim at improving performance of P2P storage systems by reducing the response time of the data requester nodes' queries. Randomized replication [7], replication on the neighbors [8–10] and replication on the paths between the data owner and some data requester nodes [8], [11], [12] are the most common decentralized replication strategies for P2P storage systems. These algorithms employ randomness in their decisions, which prevents them to purely consider the locations of data requester nodes in the replica placement procedure. Hence, this would increase the access delay between data requester nodes and their replicas and leads P2P storage system to be inefficient in terms of query processing and response time.

978-1-5090-0223-8/16/$31.00 © 2016 IEEE

We propose a locality aware replication strategy (LARAS) that addresses the data requester nodes' access delay problems for the Skip Graph based storage systems using a landmark-based identifier assignment method like DPAD [13]. The locality aware replication is defined as clustering the data requester nodes based on their location into subgroups and assigning a replica to each subgroup such that the sum of latencies between each data requester node and its replica will be minimum. By employing LARAS algorithm, a data owner can replicate some parts of its resources considering a certain set of data requester nodes while the locality awareness of the replication is guaranteed. Since Skip Graph can be used as DHT alternative in the DHT based storage applications, by employing a locality aware replication strategy on the Skip Graph nodes, any storage application using DHT as its underlying infrastructure would also benefit from this approach.

Contributions of this study are as follows:

- We propose LARAS: **the first dynamic fully decentralized locality aware** replication algorithm for the Skip Graph nodes that use landmark based name id assignment strategy.
- As part of LARAS, we propose the **search by name id** algorithm for the Skip Graph nodes which can also be used to support different P2P storage mechanisms in addition to replication.
- We extended the Skip Graph simulator, SkipSim [14], for simulating the replication strategies and evaluating their performances.
- We simulated the best known decentralized replication algorithms on SkipSim and compared them with LARAS.
- The simulation results showed that LARAS improves the access delay of public and private replication scenarios with the gains of about **20% and 38%**, respectively.

In the rest of this paper, we describe the structure of the Skip Graph, our proposed *search by name id* algorithm and the landmark based name id assignment of the Skip Graph's nodes in Section II. In Section III, we present our LARAS approach. The related works and simulation setup are described

in Sections IV and V, respectively. The simulations' results are presented in Section VI, followed by conclusions in Section VII.

II. Skip Graph

A. Structure

Skip Graph [1] is the decentralized version of Skip List data structure [15]. Unlike Skip List, Skip Graph is not vulnerable to the single point of failure and advent of the hot spots [16] due to the heavy traffic load on certain nodes. Having N nodes in the system, Skip Graph can store data as a node and retrieve the address of the node that holds a certain data item in $O(\log N)$ time.

Our view for a Skip Graph-based storage system in this study is to map each peer in the real world to a unique node in Skip Graph. Like DHTs, each node has a key that is named numerical id. Numerical id of a node is the hash value of its corresponding peer's IP address. Prior works like [4] presume a similar view of the Skip Graph. Each node also has a name id that is a binary string which defines the connectivity (neighborhood) pattern.

Figure 1 shows an example of Skip Graph with 7 nodes and 3 levels. In general, a Skip Graph with maximum N number of nodes has exactly $\lceil \log N \rceil$ levels. Each node has exactly one element in each level. Furthermore, for a Skip Graph with maximum N nodes, name ids are binary strings of length at least $\lceil \log N \rceil$.

In the level $zero$, all elements are sorted based on their numerical id in non-decreasing order in a double linked list [17]. In the i^{th} level, there exist exactly 2^i double linked lists. Each of the nodes has exactly one element in exactly one of the lists in each level.

The nodes located in the same double linked list in the i^{th} level have at least i bits common prefix in their name ids. For instance, in Figure 1, nodes with numerical ids 12, 39 and 55 are located in the same list in the level 1. Since their name ids (000, 001, 011 respectively) have one bit prefix in common. The same situation happens for the case of the nodes with the numerical ids 28 and 93. Since they have two bits common prefix in their name ids (100, 101 respectively), they are located in the same lists in both levels 1 and 2.

Using Skip Graph as a DHT is feasible by mapping resources to an identifier space using a hash function. In this way, resource identifiers can be hashed and mapped to the Skip Graph nodes based on numerical id values. For instance, in order to perform a lookup for a file, the file name is hashed and the responsible node is found via a search by numerical id [1].

B. Search By Name ID

As part of LARAS, we propose *search by name id* algorithm for the Skip Graph nodes. Thus far, Skip Graph nodes were only able to search other nodes by their *numerical ids* [1]. In our approach, the data owner announces its replicas with their *name ids*. In order to find the address of the replicas, each data requester node needs to *search by name id* with the name id of its replica. Also, LARAS uses this search operation to map the replicas from its smaller size workspace to the real world system.

*1) **Algorithm Overview**:* The *search by name id* algorithm receives a target name id as a binary string, searches for it through the Skip Graph and returns the address of the node holding that name id. If the node who holds the target name id does not exist in the Skip Graph, the search algorithm returns the address of one of the nodes holding the most similar name id to the search target. The name ids similarity is defined by the *common prefix length* of them. The longer common prefix two nodes have in their name ids, their name ids are more similar to each other.

The *search by name id* is a distributed recursive algorithm where each recursion continues on a separate node. The basic idea of the algorithm is to find the node with the name id that has the longer common prefix with the search target than the current level number of the search. The algorithm then jumps to the corresponding level and continues the search in that level in the same manner recursively. The search is started by the node who initiates the search from a certain level. That certain level number corresponds to the common prefix length in the name ids of the search initiator and the search target. The search is terminated when either the search target has been found or when in a certain level no node is found with common prefix with the target name id greater than that level number.

As an example, in Figure 1, there is no node with the name id of 010. The most similar name id to 010 in the Skip Graph is 011. This is the name id of the node holding the numerical id 55. The 011 and 010 name ids have two bits prefix in common. Both of their name ids start with 01 prefix. If the node with the numerical id 55 and name id 011 does not exist in the Skip Graph, the most similar name ids to the target name id 010 are 000 and 001. These name ids both have only one bit prefix in common with the target name id.

Similar to the search by numerical id [1], the *search by name id* can be initiated and performed by any node of the Skip Graph. Furthermore, an external node that does not belong to the Skip Graph can initiate this search via an internal node of the Skip Graph.

*2) **Algorithm Description**:*

a) Inputs and Output: Algorithm II.1 shows the *search by name id* procedure that receives two pointers $Left$ and $Right$, the search target name id as a binary string ($searchTarget$) and the current level number ($Level$). It returns the address of the node who holds the target name id as the $Result$ pointer. We assume that each Skip Graph node has a lookup table [18] in the form of a two dimensional array defined as $lookup[levels][2]$, where $levels$ is the number of Skip Graph's levels that is equal to $\lceil \log N \rceil$ in a Skip Graph with N nodes. For a certain node, $lookup[i][R]$ and $lookup[i][L]$ return the right and left neighbors of the node in the i^{th} level of the Skip Graph, respectively.

Assume that the node α wants to perform a search for the $target$ name id. Then, it initiates the search by calling the $SearchByNameID(\alpha.lookup[cpl][L], \alpha.lookup[cpl][R], target, levels)$, where cpl is equal to the common prefix

Algorithm II.1: Search By Name ID

Input: pointer Left, pointer Right, String searchTarget, int Level
Output: pointer Result

```
1  pointer Buffer = Null;
2  while commonBits(searchTarget, Right) <= Level AND
     commonBits(searchTarget, Left) <= Level do
3      if Left.nameID == searchTarget then
4          return Left;
5      if Right.nameID == searchTarget then
6          return Right;
7      if Left ≠ NULL then
8          Buffer = Left;
9          Left = Left.lookup[Level][L];
10     if Right ≠ NULL then
11         Buffer = Right;
12         Right = Right.lookup[Level][R];
13     if commonBits(searchTarget, Right) > Level then
14         Level = commonBits(Right, searchTarget);
15         Left = Right.lookup[Level][L];
16         Right = Right.lookup[Level][R];
17         SearchByNameID(Left, Right, searchTarget,
             Level);
18     else if commonBits(searchTarget, Left) > Level then
19         Level = commonBits(Left, searchTarget);
20         Left = Left.lookup[Level][L];
21         Right = Left.lookup[Level][R];
22         SearchByNameID(Left, Right, searchTarget,
             Level);
23     if Left == NULL AND Right == NULL then
24         return Buffer;
```

length in the name ids of the search initiator and the search target.

b) Searching in a list (Lines 2-12): In the Algorithm II.1, in a certain level, while neither the search target has been found nor a node who holds common prefix length with the search target greater than the number of that level, the search will be continued by the $Left$ and $Right$ pointers in the left and right directions in that level concurrently. When each of the $Left$ or $Right$ pointers reach the left or right end of the list in a certain level, respectively, their values become $NULL$ and they can not go any further.

c) Jumping to the upper level (Lines 13-22): While the $Right$ and $Left$ pointers traverse a list in a certain level, if they find a node that has greater common prefix in its name id with the target name id than the current level number, the search in that level is terminated. The algorithm then jumps to the level that corresponds to the length of the common prefix in the name ids of that node and the search target, sets the pointers to their new values and continues the search recursively.

d) Returning one of the most similar name ids (Lines 23-24): There may be a case when both $Right$ and $Left$ pointers reach to the right and left end of a list in a certain level. In such case, there is not a more similar name id to the search target in the Skip Graph. After a jump to the upper level, all the nodes in the new level have the most similar name ids to the search target up to that point of the algorithm execution.

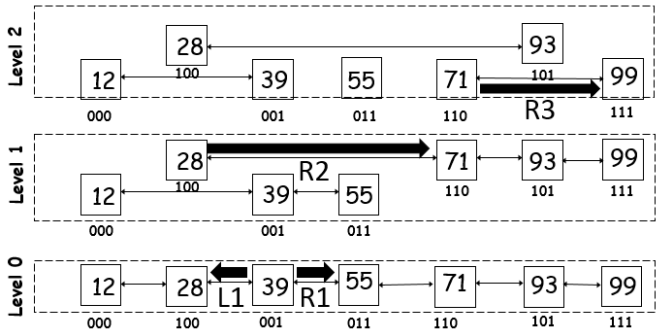

Fig. 1: An example of the *search by name id* algorithm. The search initiator is the node with name id 001 and search target is the node with name id 111

Reaching both ends of the list in a certain level means that there is no node with a better similarity to the search target. In this situation, the most similar existing name ids to the search target are the ones in the current level. All of the nodes in the current level have the common prefix in their name ids with the search target equal to the number of the current level. To return the most similar name id as the result of the search, it is enough to select one of the nodes in the current level. The algorithm performs this task by keeping the latest value of the $Right$ and $Left$ pointers in the $Buffer$ pointer at the time their value is going to be changed (Algorithm II.1, Lines 8 and 11). When both $Left$ and $Right$ pointers reach the end of a list, the algorithm returns the value of the $Buffer$ pointer as one of the most similar name ids to the search target.

*3) **Example**:* Figure 1 shows an example of the *search by name id* algorithm, where the node with name id 001 and numerical id 39 initiates a search for the target name id 111. The Rx and Lx notation corresponds to the values of the $Right$ and $Left$ pointers during the execution of the algorithm. Since the common prefix length of 001 and 111 is $zero$, the search is started at the level $zero$. The initiator sets the $Left$ and $Right$ pointers to its left and right neighbors in the level $zero$, respectively. The name id of the node who L1 holds its address has one bit common prefix with the search target which is greater than the current level number ($zero$). Therefore, the initiator passes the search to the node with numerical id 28 and name id 100, and the algorithm jumps to the level 1. In the level 1, the common prefix length between the name id of the node that R2 holds its address and the target name id is 2 bits which is more than the current level number. Therefore, the search is passed to the node with numerical id 71 and name id 110, and the algorithm jumps to the level 2. In the level 2, R3 holds the address of the node that has the search target name id (111). The search is therefore finished and the value of R3 is returned.

*4) **Time Complexity**:* Having N nodes in the Skip Graph, the *search by name id* algorithm traverses $O(\log N)$ nodes on average to search for a specific target name id. A full proof has been presented in [19].

C. DPAD name id assignment algorithm

DPAD [13] is the best known dynamic fully decentralized locality aware name id assignment algorithm for the nodes of Skip Graph. The name ids of Skip Graph nodes assigned by DPAD algorithm reflect their location information. The more

two nodes are closer to each other in the overlay network, the longer common prefix they have in their name ids.

DPAD assigns the name ids with the help of some nodes named landmarks. A landmark is *not* a node of the Skip Graph. Rather, landmarks are only used to measure the RTT delay of each node of Skip Graph to them. In DPAD, a system with N maximum possible number of nodes needs $\lceil \log N \rceil$ landmarks. The landmarks are assumed to be located in the regions with the highest probability of nodes manifestation. After the landmark placement, Skip Graph is divided into regions. A region is defined by a certain landmark and nodes whose closest landmark is that one. Each landmark receives a Huffman prefix based on the network distance between itself and the most dense landmark. The most dense landmark is defined as the landmark with the minimum sum of RTT to the rest of landmarks. The name ids of all the nodes in the same region start with their landmark's prefix.

III. LOCALITY AWARE REPLICATION ALGORITHM FOR THE SKIP GRAPH (LARAS)

A. *Algorithm Overview*

In our dynamic fully decentralized *Locality Aware Replication Algorithm for the Skip Graph (LARAS)*, a data owner can determine its replicas without the need of communicating with any special node as the coordinator. By mapping each peer to a Skip Graph node, LARAS considers replicating the set of resources that data owner wants to share, on another peer. LARAS enables backups or easier access to the resources. However, this does not necessarily convey that all the data is public as in a DHT. Access control is an orthogonal issue, which can be handled, for example, by encrypting files and distributing the keys as desired. The desired set of authorized parties is named as data requester set. This set is given as an input to LARAS which then optimizes the access delay of data requester nodes and replicates accordingly.

The aim of LARAS is to place the replicas so that the average access delay of a data requester node to its closest replica would be *close to minimum*. The exact minimum access delay can only be obtained by collecting the pairwise latencies of all nodes in the system. Obtaining such information from all nodes requires heavy communication load which would degrade performance of the storage system, and it would be nearly impossible to achieve in large scale systems.

Using LARAS, a data owner can replicate itself in two ways: public replication and private replication. In **public replication** it is assumed that all nodes of the system are potential requesters of the data owner's resources, like the P2P media sharing systems. On the other hand, in **private replication** a certain group of nodes are data requesters, for example the P2P file storage system of a company. In both cases, it is assumed that the only information the data owner knows about the system is the name id size of the system and the prefix of the landmarks.

In order to have a grasp about the maximum capacity of the system, LARAS receives the name id size of the system as one of its inputs. The other inputs are landmarks' prefixes and the number of replicas. For the case of private replication, LARAS also receives the set of data requester nodes (name

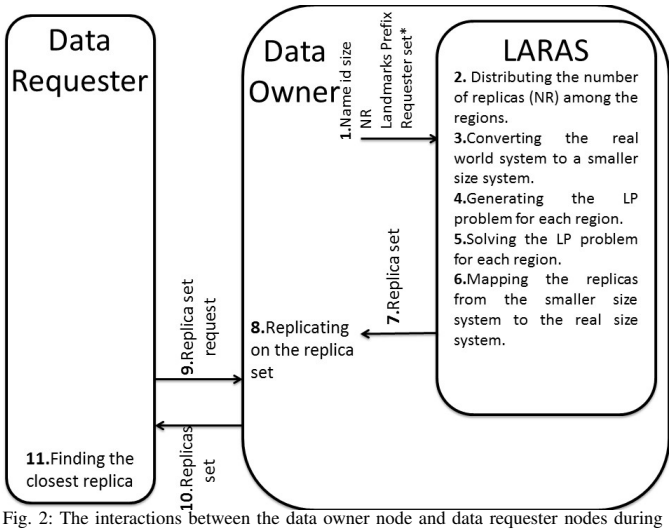

Fig. 2: The interactions between the data owner node and data requester nodes during and after the execution of LARAS. The *Requester Set* is only needed for the case of private replication*.

ids and addresses) that the data owner wants to share its data items with. To place the replicas faster, LARAS shrinks the whole real world system size to a system with smaller name id size, and then distributes the replicas among the regions based on their possible number of data requester nodes. After that, LARAS models the access delay based replication in each region with an integer linear programming (ILP) [20] problem, solves the LP relaxation version of it in each region and maps the replicas from the smaller size system to the real system. LARAS outputs a list of replicas. The data owner shares this list with its requesters. For a certain data requester node, the closest replica is the one that has the longest common prefix in its name id with the data requester node.

A single run of LARAS determines the set of replicas for a certain data owner who wants to share a set of resources with a set of data requester nodes. There may be the case where a data owner wants to share different sets of resources with different sets of data requester nodes. There can be overlaps between sets of data requester nodes as well as sets of shared resources. In this situation, LARAS should be executed once for each set of data requester nodes. Another example is where there exist different data owners who want to share their set of resources with a single set of data requester nodes. In this case, LARAS should be executed once for each data owner.

B. *Algorithm Description*

1) ***Inputs****:* Figure 2 shows the interactions between the data owner node and the data requester nodes during and after the execution of LARAS. As shown in the figure, the data owner runs the LARAS algorithm by giving the name id size of the real world system, the number of replicas (NR), the set of possible requester nodes (for the case of private replication) and the set of landmarks' prefixes (Figure 2, step 1).

2) ***Distribution of the replicas****:* After receiving the input arguments, LARAS distributes the number of replicas among the regions based on their possible number of nodes (Figure 2, step 2). Employing the locality aware name id assignment algorithm DPAD [13], the whole system is divided into regions based on the landmark locations. The higher a region has

the chance of emerging nodes, the longer prefix its landmark would have. For the case of public replication, the unit share per region (USPR) is defined by Equation 1. In this equation, l is the number of landmarks in the system and P_i is the prefix length of the i^{th} landmark. After the USPR value is determined, the number of replicas for each region is defined in Equation 2. In this equation, L is the set of all the landmarks and NR_i is the number of replicas for the i^{th} region.

$$USPR = \frac{NR}{\sum_{i=1}^{l} P_i} \tag{1}$$

$$\forall i \in L \quad NR_i = P_i \times USPR \tag{2}$$

For the case of private replication, distributing the number of replicas among the regions is similar to that was described for the public replication. However, instead of the length of landmarks' prefixes, number of data requester nodes in each region is considered.

3) ***Shrinking the problem size****:* After distributing the replicas among the regions, LARAS converts the real world system to a smaller size system (Figure 2, step 3). The number of regions of the smaller size system and the real world one are identical. The only difference between them is the maximum number of the nodes in each region. In the public replication, the name id size of the smaller size system is obtained by Equation 3. In this equation, S_i is the name id size of the i^{th} region of the smaller size system, P_i is the prefix length of the i^{th} landmark (that corresponds to the i^{th} region of the system), $maxP$ is the longest landmark prefix in the system and N is the system size. For a system with maximum N nodes and $\log N$ regions, the expected number of nodes in each region is $\frac{N}{\log N}$. As Equation 3 shows, for the case of public replication, the name id size of a region in the smaller size system is proportional to its landmark prefix length and expected number of the nodes in a region. Also, to place the higher number of replicas with better accuracy, we took the logarithm of the replicas number into the account.

$$S_i = \lceil \log(\frac{P_i}{maxP} \times \frac{N}{\log N} \times \log NR) \rceil \tag{3}$$

For the case of the private replication however, instead of the landmark prefix length of a region and the maximum landmark prefix length of the system, the number of data requester nodes of that region and the maximum number of data requester nodes of the system are considered.

4) ***Generating the LP model****:* For each region, LARAS generates the latency table L based on the name id size of the smaller size system. The $L_{i,j}$ is the number of the common prefix bits between the name ids of the i^{th} and j^{th} nodes in the smaller size system. By assigning locality aware name ids to the Skip Graph nodes, $L_{i,j}$ reflects the latency between the i^{th} and j^{th} nodes. LARAS then models the access delay based replication for each region based on its number of replicas by a LP model. The following shows the access delay based replication for a region of the system (Figure 2, step 4):

$$\min \forall i \in M, \forall j \in K \quad \sum_{i=1}^{m} \sum_{j=1}^{k} L_{i,j} X_{i,j} \tag{4}$$

$$\text{s.t.} \tag{5}$$

$$\forall i, j \in M \quad Y_i \geq X_{i,j} \tag{6}$$

$$\forall j \in M \quad \sum_{i=1}^{m} X_{i,j} = 1 \tag{7}$$

$$\forall i \in M \quad \sum_{j=1}^{m} X_{i,j} \geq Y_i \tag{8}$$

$$\sum_{i=1}^{m} Y_i = nr \tag{9}$$

$$\forall i, j \in M \quad Y_i \in \{0,1\}, \quad X_{i,j} \in \{0,1\} \tag{10}$$

a) Equation 4 (The Objective function): Equation 4 presents the objective function of the access delay based replication for a region of the system, where M is the set of all possible name ids in that region of the smaller size system. The size of M is represented by $|M| = m$, for the q^{th} region of the system $m = 2^{S_q}$. K is the set of the data requester name ids in that region of the smaller size system. LARAS converts the set of data requester nodes in a region of the real world system to the K in the same region of the smaller size system by performing a *search by name id* for each data requester node of that region of the real world system in the smaller size system. For the case of public replication, however, since each node in the system is a possible data requester, $K = M$. The output of this LP is the X matrix of size $m \times m$. $X_{i,j} = 1$ if and only if the i^{th} node is assigned as the corresponding replica for the j^{th} node, otherwise $X_{i,j} = 0$. The objective of this LP model is to minimize the sum of latencies between each data requester node and its replica. The other output of this LP is the Y vector of size m. $Y_i = 1$ if and only if the data owner replicates its data contents on the i^{th} node in the system, otherwise $Y_i = 0$.

b) Equation 6 (The replicas' constraint): The constraint presented by Equation 6 means that if the j^{th} node is assigned to use the i^{th} node as its corresponding replica, then the i^{th} node should be a replica itself. In other words, one data requester node can not be assigned to another node, unless the assigned node is a replica itself.

c) Equation 7 (The data requester nodes' constraint): The second constraint of this LP model is presented by Equation 7. Each node should be assigned to only one replica.

d) Equation 8 (The assignment constraint): If a node is a replica, then at least one node should be assigned to it. This constraint is defined by Equation 8.

e) Equation 9 (The number of replicas' constraint): The next constraint related to the number of replicas is defined by Equation 9. The nr variable is defined as the number of replicas of the region of the system modeled by LP. The number of 1s in the Y vector should be equal to the nr value.

f) Equation 10 (The allowed values for the variables constraints): Finally, the last constraint is shown by Equation

10. Based on this constraint, for all i, j values, Y_i and $X_{i,j}$ values can be either $zero$ or one.

In LARAS, modeling the access delay based replication for each region by a linear programming problem takes m^2 variables in the objective function and $2m^2 + 3m + 1$ constraints. We proposed another replication algorithm named LP which instead of modeling each region separately, models the whole system by a linear programming problem. This extra algorithm was only used for the purpose of performance comparison with LARAS. With $\log m$ bits name id size, a region of the system has $2^{\log m} = m$ number of all the possible name ids. Considering $\log m$ landmarks in the system, the whole system will have $m \log m$ possible number of name ids. Therefore, modeling this access delay based LP algorithm for the whole system instead of each region takes $m^2(\log m)^2$ variables in the objective function and $2m^2(\log m)^2 + 3m \log m + 1$ constraints. This larger number of constraints makes the linear programming slower to solve. This discussion justifies the reason of modeling the linear programming problem for each region instead of the whole system in LARAS.

*5) **Mapping the replicas**:* After LARAS models the access delay based replication for each region with an LP problem (Figure 2, step 4) and solves the LP model (Figure 2, step 5), it maps the replicas from the smaller size system to the real world system (Figure 2, step 6). LARAS does this by searching for the name id of each node that became a replica in the smaller size system in the real world system. The result of this search would be one of the most similar real world's nodes to that replica in the case of name id. LARAS would output the corresponding node of each smaller size system's replica using this strategy.

*6) **Finding the closest replica**:* After LARAS outputs the real world replicas (Figure 2, step 7), the data owner replicates its content on the replicas (Figure 2, step 8). After this phase, any data requester node can query the data owner to receive the list of replicas. When a data requester node contacts the data owner (Figure 2, step 9), the data owner sends back its replica list (Figure 2, step 10). The data requester then compares its name id with the name ids of the replicas and select the replica who holds the longest common prefix in its name id in comparison to the name id of the data requester (Figure 2, step 11). Since the name ids are assumed to be locality aware, the most similar replica in the case of the name ids would be the one with minimum latency to the data requester node.

IV. Related works

A. Decentralized algorithms

In this class of replication algorithms, the main goal is to reduce the access delay of the data requester nodes. By means of the decentralized algorithms, any node can replicate its content in the P2P system. When a node wants to replicate its content, it can do this in a distributed manner by using local information. There are several decentralized replication algorithms that consider certain aspects of the system:

1) Randomized replication: A randomized replication algorithm for client/server scenarios has been proposed with the goal of keeping the number of replicas as minimum as possible [7].

Strategy	Decentralization	Locality Awareness	Behavior
Randomized [7]	Full	No	Dynamic
On Path [8], [11], [12]	Full	No	Dynamic
On Neighbors [8–10]	Full	Hybrid	Dynamic
Consistent Hashing [8], [27–31]	Full	Hybrid	Dynamic
Objective-based [32], [33]	No	Full	Static
Genetic Algorithms [26], [34]	No	Full	Static
LARAS	**Full**	**Full**	**Dynamic**

TABLE I: Comparison of various methods of replication strategies

2) Replicating on path: The data owner replicates its content on the paths from some data requester nodes to itself considering the traffic load of the demands [8], [11], [12], [21]. Freenet [22], OceanStore [23] and Mojo Nation [24] use this replication strategy.

3) Replicating on neighbors: The data owner is assumed to only know its neighbors in the underlying system [8], [9], [25], [26]. In some P2P systems, the identifiers are chosen uniformly at random [10] and the nodes are connected to each other based on these random identifiers. Thus, each node is assumed to have neighbors that are almost uniformly distributed across the network. In such systems, replicating on the neighbors approach distributes a certain object across the network regardless of the amount of demands for that object. On the other hand, if the identifier assignment strategy is locality aware, as in DPAD [13], the data owner would have more nearby neighbors than the faraway ones. In this situation, replicating on the neighbors would mostly replicate the data items on the nearby neighbors than the faraway ones.

4) Consistent hashing replication: The content of a certain node with a certain id is replicated at the nodes with the ids of $h(i + id)$ for $1 \leq i \leq r$, where r is the replication degree and h is the hash function [8], [27–31]. In some cases, multiple hash functions are employed on a certain id resulting in several distinct replicas for a single node [8], [21].

B. Centralized algorithms

Centralized algorithms consider several metrics such as QoS, bandwidth, delay, and replication cost. Their aim is to replicate the data objects in the network such that one or some of these metrics achieve their minimum or maximum values or at least satisfy some constraints. One very common strategy for the centralized replication algorithms is to employ the genetic algorithms on the replica sets [26], [34].

As their main drawback, the centralized algorithms need global information of the network such as the capacity of the nodes, availability [35], [36], bandwidth and pairwise access delay of the nodes [33]. In this class of replication algorithms, some special nodes are coordinators [32] that collect all the required data from the rest of the nodes in the system, process these and decide where a data object should be replicated. Therefore, centralized algorithms need a huge flow of message exchanges while suffering from the single point of failure. Considering these drawbacks, the centralized replication algorithms are not suitable for P2P systems.

Table I shows a comparison of various methods of replication in the P2P systems. In the case of *decentralization*, a method is "Full" if any node can replicate its content in the

network by itself. A method is "Hybrid", if a node can replicate its content only with the help of some special nodes in the system. In the case of *locality awareness* a method is "Full" if it considers the location of the nodes purely without injecting any randomness in the replication. A replication method is "Hybrid" in this aspect, if it considers some randomness accompanied with the nodes location. Finally, in the case of *behavior*, a method is "Static" if it needs information of all the nodes to function, otherwise, it is "Dynamic".

C. Algorithms used for comparison

In addition to the best known decentralized replication algorithms described in Section III, we simulated another algorithm LP that is similar to LARAS. Implementation details of the algorithms used for comparison are as follows:

1) Randomized Replication: The data owner is selected from the set of nodes at random. The replication is done on the randomly chosen nodes as replicas, including the data owner, until all the replicas are determined.

2) Replication on path: A requester is selected from the set of data requester nodes at random, and replication is done on the path from the requester to the data owner, excluding the requester, until all the replicas are determined. Then, for each data requester node, the closest replica is assigned.

3) Replication on neighbors: The data owner replicates its content on its neighbors in the Skip Graph. It is assumed that the maximum number of replicas for a certain data owner can not go beyond the number of its neighbors.

On the other hand, the consistent hashing replication methods are mainly designed for other improvement purposes like availability [31], accessibility [27] and indexing instead of replicating [8], [21]. Since these goals are different than the access delay improvement, we did not use consistent hashing algorithm for comparison.

V. Simulation Setup

We extended the Skip Graph's simulator environment, SkipSim [14], to implement and evaluate the replication algorithms. In SkipSim, each system topology is generated in a 3000×3000 pixels environment. The RTT between two nodes was modeled as their Euclidean distance in SkipSim environment. Each pixel represents 1 millisecond delay in the real world. For each simulation setup, we generated 100 random topologies and simulated each replication algorithm for those topologies. We examined each algorithm in five system configurations with 64, 128, 256, 512 and 1024 nodes. The running times of the algorithms have been compared on a DELL Latitude E6330 laptop with Intel i5 2.60 GHz CPU and 8 GB of RAM. For solving the linear programming models, we used the lpsolve5.5 [37] on Windows 8.1.

VI. Performance Results

A. Access Delay

Access delay metric of a replication algorithm refers to the average latency between a data requester node and its closest replica. The behavior of each algorithm was similar in different simulations setup. Our results are presented for 1024-node network size and 10-bit name id size system configuration, and we observed similar behavior in different network size scenarios. Figures 3 and 4 show a comparison between the access delay performance of the replication algorithms versus the percentage of the replication in the public and private replication scenarios, respectively. In private replication evaluation, all replication algorithms were examined with an identical set of data requester nodes chosen uniformly at random. As shown in these figures, in comparison to the traditional decentralized replication algorithms, LARAS has the minimum access delay in both public and private replication scenarios. Also, in comparison to the replication on path (as the best known decentralized replication algorithm), LARAS improves the access delay about **20%** and **38%** on average in the public and private replication scenarios, respectively. In comparison to the optimal LP solution of the system, based on the physical

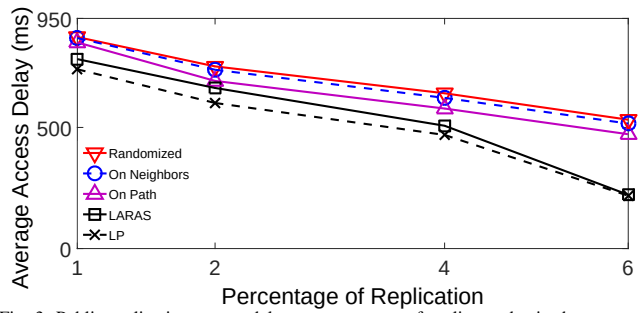

Fig. 3: Public replication access delay vs percentage of replica nodes in the system. System size: 1024 nodes, name id size: 10 bits. Y-axis shows average latency of a node to its closest replica and x-axis shows percentage of replica nodes in the system.

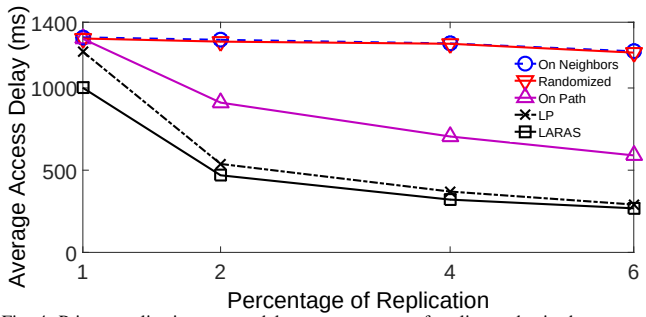

Fig. 4: Private replication access delay vs percentage of replica nodes in the system. System size: 1024 nodes, name id size: 10 bits, size of data requester nodes' set: 100. Y-axis shows average latency of a node to its closest replica and x-axis shows percentage of replica nodes in the system.

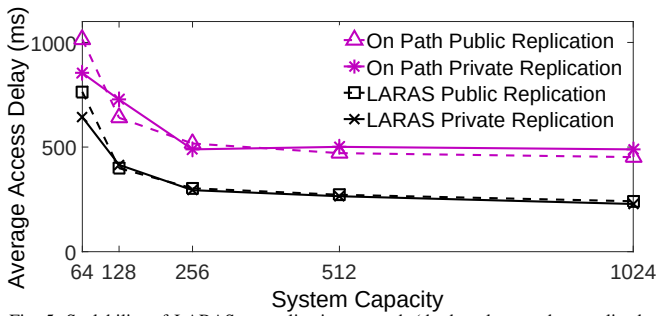

Fig. 5: Scalability of LARAS vs replication on path (the best known decentralized counterpart). Y-axis shows average latency between a node and its closest replica. X-axis presents system size. For each system setup, the number of replicas is about 5% of the system size. For private replication, size of data requester nodes' set is about 30% of the system size.

pairwise distances between the nodes in the SkipSim, the access delay obtained by the LARAS is about 2.68 times of the optimal solution on average.

Although the LP algorithm performs slightly better than LARAS in the case of public replication, LARAS works significantly faster. From the running time point of view, LARAS and LP are asymptotically the same. But since LARAS shrinks the problem size, it is about **7.5 times faster** than LP, on the average, considering all the simulation setups (6, 7, 8, 9 and 10-bit name id sizes). For instance, LP algorithm takes about **1 hour and 28 minutes** to place the replicas in a 10-bit name id size system (maximum 1024 nodes). However, in the same case, LARAS takes only about **12 minutes**. This speed up is expected to become larger as the system size increases.

B. Scalability

We compared LARAS with the best decentralized counterpart, replication on path algorithm, from the scalability point of view. In each system setup, the number of replicas was about 5% of the system size. Also, for the case of private replication, the size of data requester nodes' set was about 30% of the system size. Figure 5 shows the scalability comparison of LARAS and replication on the path algorithms. Considering the access delay similarity of an algorithm in the public and private replication scenarios as the scalability metric, as the size of the system scales up, in comparison to the replication on the path, LARAS has better performance of about **48%** on average.

VII. Conclusions

To reduce the access delay between the nodes and their closest replicas in the Skip Graph based storage systems, we propose a novel locality aware replication algorithm. Our dynamic fully decentralized approach, LARAS, determines the placement of replicas for both public and private replication scenarios. LARAS uses the name id size of the system, set of possible data requester nodes, number of replicas and the system landmarks' prefixes. To the best of our knowledge, LARAS is the only locality aware replication algorithm proposed for Skip Graph nodes that use DPAD [13] as their identifier assignment strategy. As part of LARAS, we also proposed an $O(\log N)$ time *search by name id* algorithm for Skip Graph nodes, which may be of independent interest.

To evaluate the performance of LARAS, we extended the Skip Graph simulator, SkipSim [13], [14] to be able to simulate and evaluate the replication algorithms. We simulated and compared LARAS with the state-of-the-art decentralized replication algorithms in terms of access delay improvement and scalability aspects. The simulation results showed that LARAS improves the access delay of public and private replication scenarios with the gain of about **20%** and **38%**, respectively in comparison to the best known decentralized counterpart. Also, from the scalability point of view, in comparison to the best known decentralized replication algorithm, LARAS has about **48%** improvement on average. Finally, LARAS models each region of the system with a smaller size LP problem. This helps LARAS to perform about **7.5 times** faster than modeling the whole system via LP problem with a very slight difference in performance.

References

[1] J. Aspnes and G. Shah, "Skip graphs," *ACM TALG*, 2007.

[2] E. Udoh, *Cloud, grid and high performance computing: emerging applications*. Information Science Reference, 2011.

[3] S. Batra and A. Singh, "A short survey of advantages and applications of skip graphs," *IJSCE*, 2013.

[4] T. Shabeera, P. Chandran, and S. Kumar, "Authenticated and persistent skip graph: a data structure for cloud based data-centric applications," in *ACM CSS 2012*.

[5] W. Galuba and S. Girdzijauskas, "Distributed hash table," in *Encyclopedia of Database Systems*. Springer, 2009.

[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM*, 2001.

[7] J. Su and D. Reeves, "Replica placement algorithms with latency constraints in content distribution networks," in *ACM, May*, 2004.

[8] S. Ktari, M. Zoubert, A. Hecker, and H. Labiod, "Performance evaluation of replication strategies in dhts under churn," in *6th international conference on Mobile and ubiquitous multimedia*. ACM, 2007.

[9] J. Paiva and L. Rodrigues, "Policies for efficient data replication in p2p systems," in *IEEE ICPADS, 2013*.

[10] I. Abraham, D. Malkhi, and O. Dobzinski, "Land: Locality aware networks for distributed hash tables," The Hebrew University, 2003, Tech. Rep.

[11] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher, "Adaptive replication in peer-to-peer systems," in *Distributed Computing Systems 2004*. IEEE.

[12] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM CSUR*, 2004.

[13] Y. Hassanzadeh-Nazarabadi, A. Kupcu, and O. Ozkasap, "Locality aware skip graph," in *ICDCSW*. IEEE, 2015.

[14] "Skipsim: https://github.com/yaahaanaa/skipsim."

[15] M. T. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an authenticated dictionary with skip lists and commutative hashing," in *DISCEX 2001*. IEEE.

[16] T. Crain, V. Gramoli, and M. Raynal, "No hot spot non-blocking skip list," in *33rd ICDCS 2013*. IEEE.

[17] M. T. Goodrich and R. Tamassia, *Data structures and algorithms in Java*. John Wiley & Sons, 2008.

[18] A. S. Tanenbaum and M. Van Steen, *Distributed systems*. Prentice-Hall, 2007.

[19] "Search by name id: https://crypto.ku.edu.tr/downloads."

[20] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, "Network flows," DTIC Document, Tech. Rep., 1988.

[21] V. Martins, E. Pacitti, and P. Valduriez, "Survey of data replication in p2p systems," HAL, 2006.

[22] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Designing Privacy Enhancing Technologies*. Springer, 2001.

[23] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer *et al.*, "Oceanstore: An architecture for global-scale persistent storage," *ACM Sigplan Notices*, 2000.

[24] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE JSAC*, 2004.

[25] T. Chang and M. Ahamad, "Improving service performance through object replication in middleware: a peer-to-peer approach," in *IEEE P2P 2005*.

[26] O. A.-H. Hassan, L. Ramaswamy, J. Miller, K. Rasheed, and E. R. Canfield, "Replication in overlay networks: A multi-objective optimization approach," in *Collaborative Computing: Networking, Applications and Worksharing*. Springer, 2009.

[27] A. Harwood and D. E. Tanin, "Hashing spatial content over peer-to-peer networks," in *University of Melbourne*, 2003.

[28] J. Paiva and L. Rodrigues, "On data placement in distributed systems," *ACM SIGOPS Operating Systems Review*, 2015.

[29] Z. Xiaosu, W. Xiaolin, and H. Hao, "Caching, replication strategy and implementions of directories in dht-based filesystem," in *ICPCA 2011*. IEEE.

[30] T. Pitoura, N. Ntarmos, and P. Triantafillou, "Replication, load balancing and efficient range query processing in dhts," in *Advances in Database Technology-EDBT*. Springer, 2006.

[31] P. Knežević, A. Wombacher, and T. Risse, "Dht-based self-adapting

replication protocol for achieving high data availability," in *Advanced Internet Based Systems and Applications*. Springer, 2009.
[32] Y. Chen, R. H. Katz, and J. D. Kubiatowicz, "Dynamic replica placement for scalable content delivery," in *Peer-to-peer systems*. Springer, 2002.
[33] A. S. Vijendran and S. Thavamani, "An efficient algorithm for clustering nodes, classifying and replication of content on demand basis for content distribution in p2p overlay networks," *International Journal of Computer & Communication Technology*, 2013.
[34] S.-Q. Long, Y.-L. Zhao, and W. Chen, "Morm: A multi-objective optimized replication management strategy for cloud storage cluster," *Journal of Systems Architecture*, 2014.
[35] A. Pace, V. Quéma, and V. Schiavoni, "Exploiting node connection regularity for dht replication," in *IEEE SRDS, 2011*.
[36] R. Rodrigues and B. Liskov, "High availability in dhts: Erasure coding vs. replication," in *Peer-to-Peer Systems IV*. Springer, 2005.
[37] "lpsolve5.5: http://lpsolve.sourceforge.net/5.5/."

View publication stats