SEALA: Secure, Efficient, Availability and Locality Aware P2P Cloud Storage

A thesis proposal submitted in fulfilment of the requirements for the degree of PhD in Computer Science and Engineering
Koç University, 2017

Yahya Hassanzadeh-Nazarabadi

Graduate School of Science and Engineering
Koç University
Turkey

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Nowadays, the computational power of the end user's devices is developing fast toward a small server's power. An end-user device (e.g., smartphone, desktop computer, tablet, laptop, smart television, etc) is capable of maintaining communications to some other devices, and offer some specifically limited services for them. For example, an end-user device, which we call it a **peer**, can offer a chat service for the set of user's friend with no chat server in between. Tonic [18] and Squiggle [19] are examples of such chat application. Multiple peers can expand their limited offered service by coming together, forming a network of peers, and offering their specific service on a vast scale with no central server in the game. Such a system is called a peer-to-peer (P2P) system. The operations in a P2P system are accomplished in a decentralized manner by distributing the workload among a subset of peers. For example, in contrast to the centralized social networks like Facebook, where a centralized authority stores all the users' profiles, GNU Social [20] is a P2P social network where each peer is responsible for storing the profiles of some other users, and responding the queries of other peers in the system about those stored profiles.

Similarly, in contrast to the centralized cloud storage services like Dropbox [21], Amazon Cloud [22], and Mozy [23] where all the users' data objects

are stored in the company's data centers, in a P2P cloud storage, each peer is responsible for storing the data objects of a sub-set of users, and answering the queries of other peers about those stored data objects, accordingly. The P2P cloud storages follow the same goal of centralized ones in terms of data sharing with other users, synchronizing the data objects in a subset of peers, and providing backup. However, instead of trading space for a price, in P2P cloud storages, peers trade their own disk spaces to gain storage space on other peers. In addition to low-cost services, P2P cloud storages are more secure than the centralized ones. As there is no centralized authority, to damage the P2P cloud storage in an evident manner, an attacker needs to attack thousands of peers of the system. This slows down the attacker and makes the attack costly inefficient for the majority who are interested in attacking the centralized cloud storages. Likewise, most of the P2P cloud storages distribute the data objects geographically across the system, which provides more reliability especially in the event of natural disasters. Finally, in a P2P cloud storage, users' data objects are encrypted and stored in a few other peers of the system. This provides more privacy than the centralized cloud storage where a centralized authority is in charge of all the stored data of the users, which makes it able to extract sensitive information about the usage and sharing patterns of users, their offline/online time periods, geographical location, etc.

There exist several P2P cloud storages available online. Gnutella [24] and BitTorrent [25] are two P2P cloud storages where users benefit from each other publicly shared data objects. In Wuala [26] users can trade spaces from their own disks to store each others' files. CrashPlan [27] and CuckooDrive [28] are backup oriented P2P cloud storages, where friends are exchanging backup capacity from their disk spaces, and backup their own devices on the exchanged capacity. PowerFolder [29] and BitTorrent Sync [30] are sharing-based P2P cloud storages, which provide the synchronizing among a subset of selected peers by the user.

1.2 Original Contributions

This thesis proposal aims to present the contributions accomplished until the date of the proposal as well as the future works that are planned to be conducted. The original contribution of the final thesis is a **secure, efficient, availability and locality aware P2P cloud storage called SEALA**, which accommodates the following list of contributions. In the list, by a dynamic algorithm, we mean that the algorithm can generate the proper output with respect to the current state of the system, and does not need the existence of all the nodes in the system for execution. Likewise, by full decentralization, we mean that the algorithm can be executed by a single node which only has a local view of the system as well as some public information with small overhead.

- A structured P2P cloud storage is designed on the top of a Skip Graph routing infrastructure in which each peer is mapped to a Skip Graph node with two identifiers known as name ID and numerical ID. In our designed P2P cloud storage, nodes can store their data objects on other nodes of the Skip Graph, and share their stored data objects either with a subset of nodes of the system in a private manner or with all of the nodes of the system in a public manner.

- The first dynamic, fully decentralized and locality aware name ID assignment algorithm is proposed for the Skip Graph nodes. Using the proposed strategy, each node of the Skip Graph can assign a name ID for a new node in a fully decentralized manner. The assigned name ID is unique and reflects the locality information of the nodes in the underlying network i.e., the pairwise latency of nodes is efficiently comparable using their locality aware name IDs.

- An authentication method is proposed for the locality-aware name ID assignment strategy that was described in the previous item. Using the authentication scheme, each node can verify the validity of a name ID ownership claim by another node, as well as the validity of the

information that the name ID's owner has provided for the locality-aware name ID assignment algorithm. The verification is done in the presence of the colluding malicious adversarial model.

- A search for name ID algorithm is proposed for the Skip Graph, which enables each node to find the address of a name ID owner in a fully decentralized manner. The search for name ID is as efficient as the original search for numerical ID of Skip Graph in the term of communication complexity. However, benefiting from the locality-aware name ID assignment, a node can search for the existence of other nodes with some specific locality information in the underlying Skip Graph.

- The search for name ID and numerical ID algorithms are authenticated in a dynamic and fully decentralized manner. Each node of the Skip Graph can authenticate the validity of search results in the presence of colluding malicious adversaries who tray to perform a reply attack on the search result.

- The first dynamic, fully decentralized and locality aware replication strategy is proposed for the Skip Graph-based P2P cloud storage. Using the proposed replication strategy, each node of the Skip Graph can place its replicas in a fully decentralized manner such that the average access delay between each data requester and the closest replica to it becomes minimum. For a data owner node, a replica corresponds to another node of the Skip Graph which stores all its owned data objects. Similarly, for a data owner node, a data requester is a node of Skip Graph which the data owner aims to share its data objects with it. These terminologies are explained in more details in Chapter 5.

- The first dynamic, fully decentralized, energy efficient and locality aware aggregation tree is proposed for the Skip Graph-based P2P systems. In the resulted aggregation tree, the total latency on the path from the root and each of the leaves, as well as the average energy cost of non-leaf nodes of the aggregation tree are minimized.

- A dynamic, fully decentralized, and availability aware replication strategy is proposed for the structured based P2P cloud storage. By employing the proposed availability aware replication, each node of the P2P cloud storage can place its replicas such that the maximum availability of replication during a predefined time slot unit is achieved. The availability of replicas is affected by their dynamic arrivals and departures to and from the system, respectively, which is called churn. The proposed availability aware replication achieves the maximum availability of replication independent of the churn behavior of nodes.

- The availability information provided by nodes to the availability aware replication strategy that was described in the preceding item is authenticated in a fully decentralized and dynamic manner. Each node of the Skip Graph can authenticate the availability information obtained from other nodes in the presence of colluding malicious adversaries.

- A multi-objective dynamic, and fully decentralized replication algorithm is proposed, which accommodates the preceding locality and availability aware replication objectives. The proposed multi-objective replication algorithm also tries to minimize a load of each node in term of a number of replication duties that node holds.

- The first dynamic and fully decentralized fault-tolerant and energy-aware approach is proposed for the Skip Graph-based P2P systems to recover the paths failures between the nodes in the underlying routing infrastructure, which is caused by nodes' failure or churn. The resulted scheme provides a K-fault tolerant Skip Graph that is fully functional. By K-fault tolerant we mean that every pair of nodes can still reach each other from the underlying Skip Graph even after failure or departure of K randomly chosen nodes.

- An offline simulator of Skip Graph is developed to support simulations of Skip Graph based algorithms (e.g., name ID assignment, replication, aggregation tree, etc) in large scale.

Current contributions to *SEALA* cover the Skip Graph simulator, search for name ID, locality aware name ID assignment, locality aware replication, availability aware replication, and locality aware and energy efficient aggregation tree. As the ongoing research, the locality-aware name ID assignment and locality aware replication are extended to support the larger scale of nodes with better performance results.

In the rest of this thesis proposal, we describe the structure of the Skip Graph, its original search for numerical ID, and our proposed search for name ID in Chapter 2. We describe the *SkipSim*, our developed Skip Graph simulator in Chapter 3. In Chapter 4, we present our proposed locality aware name ID assignment. Our proposed locality and availability aware replications are presented in Chapters 5, 6, respectively. In Chapter 7, we present our proposed energy and locality aware aggregation tree. Finally, we present our future work and research timeline in Chapter 8.

Chapter 2

Skip Graph

2.1 Structure

Skip Graph [31] is the decentralized version of Skip List data structure [32]. Unlike Skip List, Skip Graph is not vulnerable to the single point of failure and advent of the hot spots [33] 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 by traversing at most $O(\log n)$ nodes.

Our view for a Skip Graph based storage system 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 hashed value of its corresponding peers IP address. Prior works like [34] presume a similar view of the Skip Graph. Each node also has a name ID that is a binary string which defines the connectivity (i.e., neighborhood) pattern. Considering the real world scenario, a peer can join the Skip Graph-based decentralized P2P system, search within that, or even search from the outside of the system -via a node of Skip Graph- in $O(\log n)$ time.

Figure 2.1 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, enumerated from 0 to $\lceil \log n \rceil - 1$. Each node has exactly one element in each level. Furthermore, for a Skip Graph with maximum n nodes, name

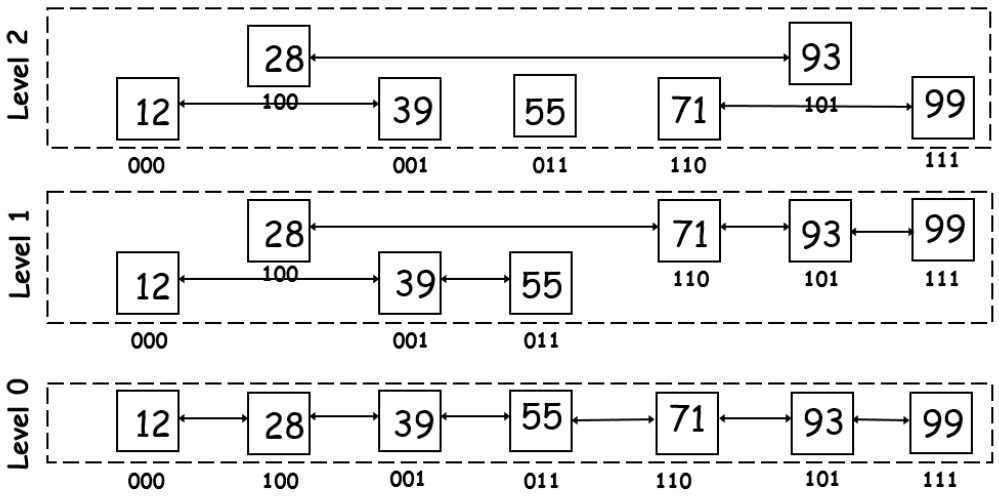

Figure 2.1: An example of a Skip Graph with 7 nodes, and 3 levels. Elements of a node in each level are presented by squares with the numerical IDs enclosed, and the name is located at the bottom.

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 [35]. 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 2.1, nodes with numerical IDs 12, 39 and 55 are located in the same list in the level 1. Since their name IDs (000, 001, and 011, respectively) have one-bit prefix in common. In the other word, their name IDs start with 0. 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, and respectively), they are located in the same lists in both levels 1 and 2.

2.2 Acting as a DHT

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 [31].

2.3 Search for numerical ID

As a member of DHT-based distributed data structure family, Skip Graph supports regular DHT operations such as *insertion* and *get*. In Skip Graph, *get(x)* operation is defined as a search for the address of the node that has x as its numerical ID [31]. This operation is defined as *search for numerical ID*, and is very similar to the searching in the Skip List, except that in the Skip Graph any node could initiate a search. On the contrary, in the Skip List, all the searches should be initiated by the left topmost node (start node). Skip Graph could perform this operation with logarithmic time order on the average with respect to the number of the nodes in the zero level.

Figure 2.2 shows an example of search for a numerical ID on the Skip Graph. The search is initiated by node-28 for the node with numerical ID 71. Since the search target 71 is greater than the numerical ID of the search initiator (28), the search in all the levels with continuing to the *right* side. In the figure, the arrows indicate the procedure of the search step by step. In the beginning of the search, at level 2, there are two consecutive nodes with numerical IDs 28 and 93 that are less than and greater than the search target (71), respectively. Therefore, the search will continue one level down (level 1) with node-28. In level 1, node-28 will find the search target just after checking its right neighbor in that level.

Another example of a search for a numerical ID is shown in Figure 2.3. A node with the numerical ID 55 initiates a search for the node with numerical ID 14. Since the target numerical ID *does not* exist in the Skip Graph, the algorithm returns the node with the numerical ID that is the greatest node in the system that is less than the search target (12 in this case). The search starts at the topmost level (2) by the initiator of the search (55 in this example). Since the search target (14) is less than the initiator, in all the lower levels the linear search will continue to the *left* direction. In level

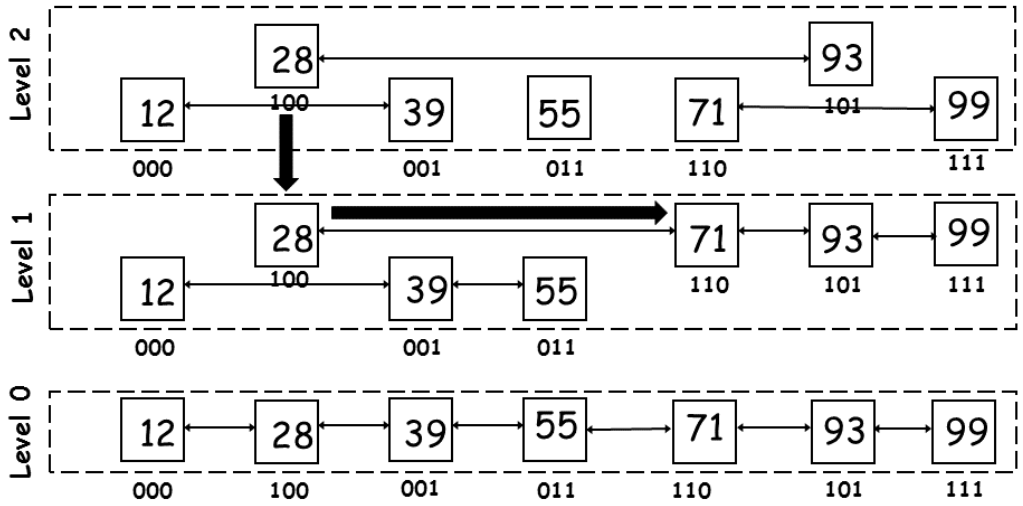

Figure 2.2: Search for numerical ID 71 starting at the node with numerical ID 28. numerical ID and name ID of a node are shown inside and below the node, respectively. The arrows show the exchanged messages during the execution of search algorithm.

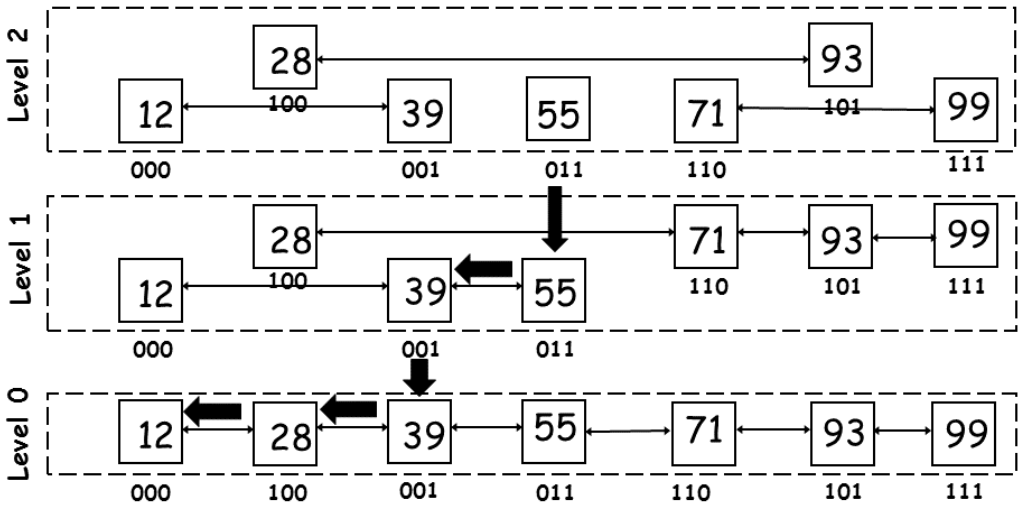

Figure 2.3: Search for numerical ID 14 starting at the node with numerical ID 55.

2, since 55 has no left neighbor, the search will switch to the level 1. In that level, the two consecutive elements (one greater than the search target and one less) will be found immediately by just checking the left neighbor of 55 (node with the numerical ID 39). Therefore, the search will continue at the lower level (0) with the node that has the smaller numerical ID (39). In level 0, node-39 checks its left neighbor and it will reach the node with numerical ID 12. At this point, the search will be terminated. Since we reach an element that is smaller than the search target, continuing the linear search will make no sense anymore and the node with the numerical ID of 12 will respond to the search initiator (55).

2.4 Search For Name ID

We propose *search for name ID* algorithm for the Skip Graph nodes [36]. Thus far, Skip Graph nodes were only able to search other nodes by their *numerical IDs* [31].

2.4.1 Algorithm Overview

The *search for 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 for 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 at 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 at 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 2.4, 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. In general, in a system with n bits name ID size, two nodes can have up to $n-1$ bits common prefixes in their name IDs. Here in this 3-bits name ID size example, there exists just one node in the Skip Graph that has the most similar name ID to 010 with 2 bits common 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. In the other word, since we assume that the only node has 2 common bits prefix with the search target (i.e., node with the name ID 011) does not exist in the Skip Graph, the most similar available name IDs to the target name ID will be the ones with *one* bit common prefix.

Similar to the search by numerical ID [31], the *search for 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.

2.4.2 Algorithm Description

Inputs and Output Algorithm 1 shows the *search for 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. If the node holds the target name ID does not exist in the Skip Graph, the algorithm will return the address of one of the nodes which has the

Algorithm 1: Search For 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;
```

most similar name ID to the search target in the case of common prefix (i.e., the name ID holder with the longest common prefix to the search target). We assume that each Skip Graph node has a lookup table [37] 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 length in the name IDs of the search initiator and the search target.

Searching in a list (Lines 2-12) In the Algorithm 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. During the time that $Left$ and $Right$ pointers traverse a list in a certain level if they found a node with the name ID equal to the target name ID, they will return the address of that node (Algorithm 1, Lines 3-6). Otherwise, the $Left$ and $Right$ pointers will continue traversing the list in their specific directions (Algorithm 1, Lines 7-12). When each of the $Left$ or $Right$ pointers to 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. At that time they will stop at the left or right end of the list in that certain level. We used another pointer named $Buffer$ to keep the previous value of the $Right$ and $Left$ pointers before they latest change.

Jumping to the upper level (Lines 13-22) While the $Right$ and $Left$ pointers to traverse a list in a certain level if they find a node that has a 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.

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 the end of a list at 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. Reaching both ends of the list at 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 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.

2.4.3 Example

Figure 2.4 shows an example of the *search for 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

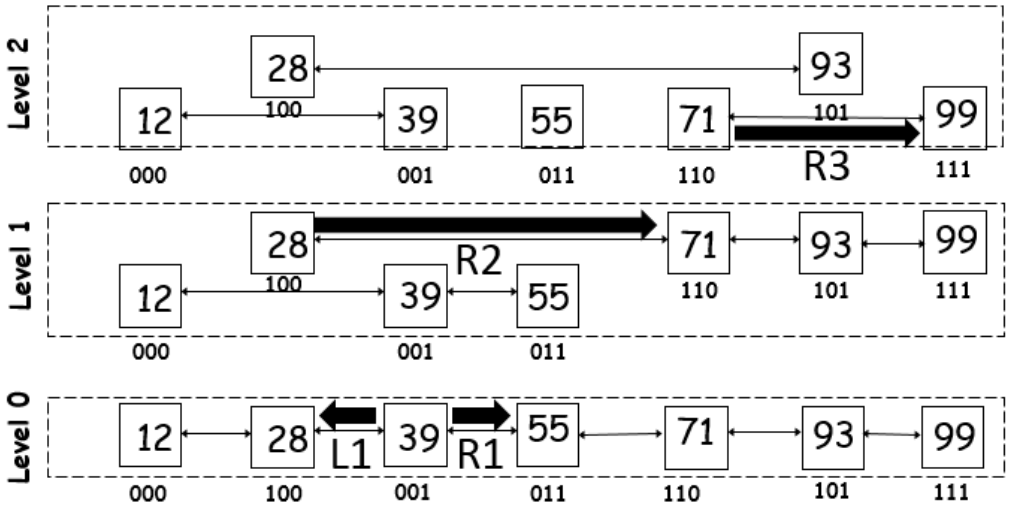

Figure 2.4: An example of the *search for name ID* algorithm. The search initiator is the node with name ID 001 and search target is the node with name ID 111

(*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 are 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. Since the node with name ID 100 has not any left neighbor in the level one, the $Left$ pointer will be set to $NULL$. The *Right* pointer in the level one (R2) holds the address of the node with the name ID 110 and numerical ID 71. The common prefix length between the name ID of the node that the *Right* pointer in the level one (R2) holds its address and the target name ID is 2 (Common prefix length of the 110 and 111 is 2 bits) which is more than the current level number. Therefore, the algorithm will jump to the node with name ID of 110 and numerical ID of 71 in the level 2.The node with name ID 110 does not have any left neighbor in the level 2. Therefore the $Left$ pointer will be again set to the $NULL$, while the *Right* pointer will be set to the right neighbor of the node with name ID 110 in the level 2 (R3). 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.

2.4.4 Time Complexity

Having n nodes in the Skip Graph, the *search for name ID* algorithm traverses $O(\log n)$ nodes on average to search for a specific target name ID. As was described, the basic idea behind the search for name ID algorithm is to find the node holds the most similar name ID to the search target in the current level, jump to the upper level and continue the search from that most similar node in the upper level. The main operations of the search for name ID are therefore jumping to the upper level and scanning there recursively.

Theorem 1: In the search for name ID algorithm, with high probability, the number of the scan operations in a certain level is $O(1)$.

Proof: In the certain k^{th} level, the search will be continued in the left and right directions until a node will be found such that its common prefix length with the search target is longer than k bits. Without loss of generality, assume then the $k+1^{th}$ bit of the search target name ID is *zero*. Based on the logic of the search for name ID algorithm all of the nodes placed in the same list in the k^{th} level have at least k bits common prefix with the search target. So the search in the k^{th} level continues until a node will be found such that its $k+1^{th}$ bit of name ID is equal to *zero*. Starting from a certain node in the k^{th} level, the probability that after scanning a single node in both directions, no node will be found with the $k+1^{th}$ bit equal to *zero* is $\frac{1}{4}$ (The probability is equal to $\frac{1}{2}$ for each direction). The probability that such a node **will not be found** after a **constant** number of scan operations in both directions is equal to the probability that such a node **will be found** after a **variable** number of scans say m scan operations. Therefore, the probability that such a node will be found after m scans in both directions is equal to $(\frac{1}{4})^m$. Based on the above discussion, the probability of having the constant number of scan operations in a certain level is equal to the complementary probability of terminating the scan operations after scanning m nodes in both directions which are equal to the $1-(\frac{1}{4})^m$.

Theorem 2: In the search for name ID algorithm, having n nodes in the

Skip Graph, the number of jumps to the upper level is $O(\log n)$.

Proof: The number of the jumps is bounded by the number of the levels in the Skip Graph which is $O(\log n)$.

Theorem 3: The asymptotic running time of the search for name ID in a Skip Graph with n nodes is $O(\log n)$.

Proof: Based on what was described in the Theorems 1 and 2, in the worst case, the search for name ID algorithm traverses all the levels one by one. In each level the number of the scan operations are $O(1)$. Therefore, the asymptotic running time of the search for name ID is bounded by the number of Skip Graph's level which is $O(\log n)$.

Chapter 3

SkipSim: A Skip Graph Simulator

3.1 Introduction

For the design and evaluation of Skip Graph-based algorithms, we developed *SkipSim*: a Skip Graph simulator [38]. ***SkipSim* is an offline, open source, and fully object-oriented platform, which supports large-scale simulations and evaluations of Skip Graph-based algorithms.** *SkipSim* is offline as it does not need to communicate with other machines in order to perform a simulation. *SkipSim* benefits from an embedded implementation of a Skip Graph data structure and its operations. *SkipSim* preserves distributed characteristics of Skip Graph similar to testbeds like PlanetLab, while improves the cost of simulations and provides various topologies. Likewise, *SkipSim* preserves the scalability of P2P simulations similar to offline simulators like PeerSim.

SkipSim contains name ID assignment algorithms such as DPAD, LMDS, randomized, and hierarchical assignments. Detailed descriptions of *SkipSim*'s name ID assignment algorithms is available in our prior work [39]. Furthermore, *SkipSim* places the set of well-known decentralized replication algorithms such as replication on the path, replication on neighbors, randomized replication and LARAS, with detailed description available in [36].

Complete list of available name ID assignment and replication algorithms as well as configuration guidelines are available in *config.txt* from *SkipSim* distribution bundle [38].

In the rest of this chapter, we first describe the *SkipSim* feature highlights in Section 3.2. Next, we present a sample demo scenario of *SkipSim*, provided with sample configuration and outputs in Section 3.3. Finally, the architecture of *SkipSim* is described in Section 3.4.

3.2 Feature Highlights

3.2.1 Scalability

The data structures and their interactions have been chosen and implemented with very low memory footprint. Empirically we have scaled *SkipSim* up to 100 topologies each consisting of 4096 nodes.

3.2.2 Modularity

All of the *SkipSim* classes can be configured independently. Any part of implementation can be customized and classes can be replaced with customized implementations.

3.3 Sample Demo Scenario

In *SkipSim*, each simulation is configured based on *config.txt* which contains characteristics of system accompanied with a list of algorithms to be simulated. A sample configuration with detailed descriptions is available in *config.txt* in *SkipSim* distribution bundle [38].

Figure 3.1 shows a sample *config.txt* file, which aims to measure the average latency of search for numerical IDs affected by selecting DPAD as the name ID assignment algorithm. Based on the value of the *iterations* field, this sample simulation is performed for 100 randomly generated topologies. Each topology consists of 512 nodes and 9 landmarks [39] determined by

```
//System Configuration
nodes.number = $512$;
nodes.nameIDSize = $9$;
nodes.generation = $landmark$;
landmarks.number  = $9$;
landmarks.prefix = $3$;
domain     = $3000$;
iterations = $100$;


//Simulation Configuration
nodes.numericalIDAssignment = $randomized$
nodes.nameIDAssignment = $DPAD$
search.numericalID = $100$;
```

Figure 3.1: A sample *SkipSim* configuration file (*config.txt*)

nodes.number and *landmarks.number* fields, respectively. Landmarks are not nodes of Skip Graph, they are super nodes placed randomly to make some locality features of nodes measurable with respect to them. For instance, DPAD [39] uses the latency between a node and each of the landmarks to assign a locality aware name ID to that node. In sample configuration of Figure 3.1, size of nodes' name ID and landmarks' prefix [39] binary strings are set to 9 and 3 bits, respectively. As determined by $nodes.numericalIDAssignment$ and $nodes.nameIDAssignment$ fields, numerical IDs and name IDs of nodes are assigned randomly and by invoking DPAD algorithm, respectively. Numerical IDs are non-negative integer values. Finally, 100 random searches by numerical ID [31] are initiated.

In each simulation's iteration, *SkipSim* distributes landmarks and nodes in a square plain of contiguous points with the size specified by the *domain* field of the configuration file. In sample configuration file in Figure 3.1, each side of the square plan consists of 3000 points. *SkipSim* distributes landmarks uniformly at random. Nodes can be either distributed uniformly at random or based on the landmarks' distribution [39]. In this sample configuration, *nodes.generation* field has the *landmark* value. This makes *SkipSim* to distribute nodes based on the node manifestation probability of each

Figure 3.2: An intermediate state of *SkipSim*'s graphical user interface for sample configuration of Figure 3.1

point, which is computed based on the locality state of each point regarding landmarks. The closer a point is to a landmark, with higher probability a node emerges at that point. The detailed descriptions of node manifestation probability are available in [39].

After the configuration is done and *config.txt* is saved, simulation can be started by executing *skipsim.jar* or compiling the source files together. Figure 3.2 illustrates an intermediate state of *SkipSim*'s graphical user interface for sample configuration of Figure 3.1. In this Figure, a randomly generated topology is depicted by solid red and pale blue hollow circles, which denote

```
num ID Search for 136 is 30 with name ID  136
The search started from 46 with numID 52
TotalTime Reset from 4711
num ID Search for 52 is 258 with name ID  52
The search started from 27 with numID 33
TotalTime Reset from 6453
num ID Search for 112 is 4 with name ID  112
The search started from 14 with numID 77
Average latency of 100 random searches for this topology: 4520.35
Total average latency: 5166.28 Standard Deviation: 715.21
```

Figure 3.3: Part of the generated results by *SkipSim* for sample configuration of Figure 3.1

the landmarks and nodes, respectively. Figure 3.3 shows part of results generated by *SkipSim* configured with the sample shown by Figure 3.1. Each line except the last shows the intermediate results of the simulation. For each randomly generated search for numerical ID, the search initiator, target, result, and latency are printed. As 100 searches for numerical IDs finish for each topology, the average latency for that topology is also reported. The last line shows the total average and standard deviation of the search for numerical IDs latency taken over all the topologies for total $100 \times 100 = 10000$ searches.

3.4 Architecture and Development

Main classes of *SkipSim* and their interactions are depicted in Figure 3.4. Each class is presented by a rectangle. The interactions between classes are represented by arrows where the callee and caller classes correspond to the head and tail of the arrows, respectively. The hierarchy among classes represented by enclosing the children classes into their parents where the parent classes employ their children as data objects. In *SkipSim*, the Skip Graph is modeled as a set of nodes described by the *Node* class. *Node* class contains all the characteristics of a single node such as numerical ID, name ID, and the neighborhood table. Latency between two nodes is modeled as the Euclidean distance between their coordination. For a certain topology, set of all Skip Graph nodes and their related functions are handled by the *Nodes* class.

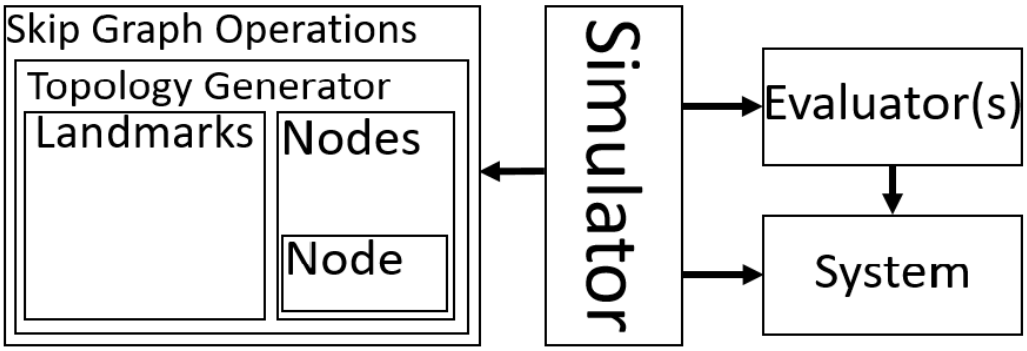

Figure 3.4: An overview on main classes of *SkipSim*

The *Landmarks* class keeps track of landmarks with their attributes like the location. *Nodes* and *Landmarks* classes are employed as data members of the *Topology Generator* class, which handles the creation of a topology by generating the coordination of nodes and landmarks.

The *Skip Graph Operations* has an embedded Skip Graph implementation defined by the *Topology Generator* class as well as the main operations of Skip Graph such as node insertion, node deletion, search for numerical ID and search for name ID. *Skip Graph Operations* class acts as a communication medium among the nodes by receiving the initiation of an operation from a node, routing the initiated operation's messages among the nodes, and returning the results back to the initiator. Novel Skip Graph operations can be implemented as a function inside this class. The main function of *SkipSim* is located in the *Simulator* class. Algorithms such as replication and name ID assignments are implemented as separate classes. System parameters such as number of nodes, landmarks, algorithms under simulation are loaded directly from *config.txt* into the *System* class. These parameters are also manually configurable from this class.

The *Evaluator(s)* classes are implemented separately and each enclose the evaluation routines of a certain set of algorithms. We have already developed two evaluator classes denoted by *repEvaluation* and *idEvaluation*, which contain the evaluation methods of replication and name ID assignment algorithms, respectively. Figure 3.5 visualizes the data points collected by the *idEvaluation* based on the sample configuration of Figure 3.1 for the

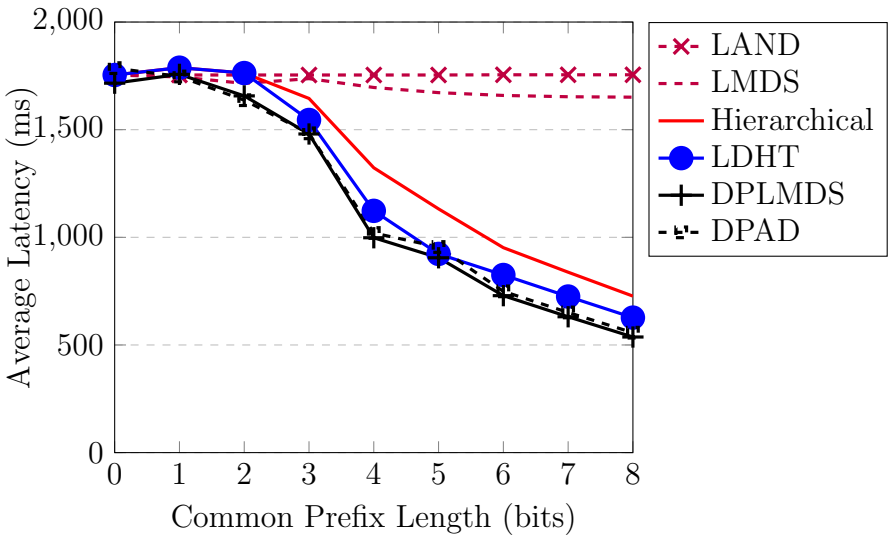

Figure 3.5: Average pairwise latency of nodes vs length of common prefix in name IDs for the simulation configuration of Figure 3.1. Y-axis shows the average pairwise latency of nodes in miliseconds. X-axis shows the common prefix length of nodes' name IDs.

available name ID assignment algorithms. Figure 3.5 represents the average pairwise latencies of Skip Graph nodes versus the common prefix length of their name IDs.

Chapter 4

Locality Aware Skip Graph

4.1 Introduction

Skip Graph is a distributed data structure that stores data using {key, value} pairs [31]. Data retrieval is performed by using the key associated with the specific data item. Scalability, fault tolerance, fast searching, correctness under concurrency, and load balancing [40] make Skip Graph a suitable underlying routing infrastructure for P2P cloud storage applications [34, 41]. Skip Graph can be considered as an alternative to DHT [42] in several DHT-based services such as P2P storage systems [43, 44], online social networks [45], and search engines [46].

In a Skip Graph, each node has a unique name ID and numerical ID. The nodes will be connected together based on their name IDs on multiple levels. The more two nodes have common prefix bits in their name IDs, they will be connected to each other in the more levels of Skip Graph. In a Skip Graph, the most common search query is to find the node with the specific numerical ID, that is called *search for numerical ID*. The Skip Graph with the total number of n nodes could perform a *search for numerical ID* query by traversing $O(\log n)$ nodes on the average.

The performance of a Skip Graph is measured by its query processing and response time. Since the most common query of a Skip Graph is the *search for numerical ID*, the average end-to-end latency (i.e., the latency

between the source and destination of a query) [47] of the search would be considered as the main performance metric. The Skip Graph topology is based on the name IDs of the nodes. Therefore, the network paths in the underlying Skip Graph infrastructure are heavily influenced by the name ID assignment strategy. In the existing solutions, nodes' name IDs in the Skip Graph are chosen uniformly at random from an identifier space. This makes the underlying network path between two nodes to be chosen uniformly at random. In such systems, there would be a huge difference between the underlying network paths in the Skip Graph infrastructure and the overlay network paths in the real system. The average hop-to-hop latency (i.e., the latency between two consecutive hops) could increase and hugely affect the average end-to-end latency per search query and therefore makes the whole system inefficient in the terms of query processing and response time.

One solution to improve the latency problem in the Skip Graph is to make it locality aware such that nodes' identifiers reflect their location in the overlay network. As much as two nodes are closer to each other in the overlay, their name IDs would be much more similar to each other in the underlying network. **We define the locality awareness of a Skip Graph as assigning name IDs to the nodes such that the more two nodes are close to each other, the longer common prefix they would have in their name IDs**.

Skip Graph has the potential to be used instead of DHT in various DHT-based applications. Therefore, making the Skip Graph locality aware and optimizing end-to-end latency in its search queries, would also help optimizing query processing and response time in such applications. Using a locality aware Skip Graph instead of a DHT will also enable DHT-based storage systems to perform data replications based on the location of the peers. Contrary to DHT, in the Skip Graph, each node has a name ID alongside with its traditional key (i.e., numerical ID). Therefore, when the name IDs reflect the locality, placement of the replicas could be handled based on their location information.

In addition to improving end-to-end latency, a locality aware Skip Graph could preserve security and privacy by handling the flow of data. Locality

awareness can be used to support the traffic flow between two nodes to be routed completely by local peers rather than far away ones. This helps keep the flow of data within a certain region. Besides, in a secure cloud storage system, using a locality aware Skip Graph would improve the challenge-response and update time [48–57].

Our contributions to the locality awareness problem of Skip Graph are as followings.

- We propose the first dynamic fully decentralized algorithm (DPAD) to assign locality aware name IDs to the nodes of a Skip Graph.
- We redesign the proposed identifier assignment algorithms for DHT nodes to be compatible with the Skip Graph features, simulate them in the SkipSim and compare their performance with our DPAD algorithm.
- Our simulation results show that our DPAD algorithm 40% improves the end-to-end latency of the search queries and performs about 82% better in preserving the locality awareness of the Skip Graph nodes than the previously best known dynamic, fully decentralized counterparts.

In the rest of the chapter, we present DPAD algorithm to provide locality aware identifiers for the nodes of Skip Graph in Section 4.2. In Section 4.3, we describe our simulations setup. We review the related works in Section 4.4. In Section 4.5, we evaluate DPAD algorithm performance in comparison to the best known static and dynamic algorithms, followed by conclusions in Section 4.6.

4.2 Dynamic Prefix Average Distance (DPAD) algorithm

In our proposed *Dynamic Prefix Average Distance* algorithm, we assume that there exist some supernodes in the system called landmarks. Landmarks are not members of the Skip Graph. Rather, each *new node* that wants to receive a name ID and join the Skip Graph uses the landmarks as reference points.

A *new node* contacts the landmarks to receive location information about itself with respect to the landmarks. The *new node* then, generates a name ID based on the location information it received. After generating its name ID, the *new node* will contact one of the existing nodes in the Skip Graph for checking the availability of its generated name ID. If the *new node*'s generated name ID was not available, it will receive the most similar available name ID to its generated name ID. Here by the most similar, we mean the one that has the longest common prefix with the *new node*'s generated name ID.

In DPAD algorithm, with n nodes in the system, we need to have at least $\log n$ landmarks. Landmarks are some supernodes in the system that are placed according to the expected density of the nodes in the system. To make dynamic hierarchical regions division among the landmarks considering their density, each landmark has a dynamic prefix. This prefix is determined before any node arrives at the system. To generate these dynamic prefixes, first, the landmark that has the minimum total distances to all other landmarks will be determined. This landmark considered as the densest landmark in the system. Then, the distance of each landmark to that densest landmark will be used as its weight for the Huffman [58] algorithm. By running the Huffman algorithm, the prefixes of landmarks will be determined.

Figure 4.1 summarizes DPAD algorithm step by step. When a *new node* wants to obtain its name ID, it first has to contact all the landmarks once (Figure 4.1, steps 1-2) to receive the *Round Trip Time (RTT)* distance between itself and each landmark (step 3). Alongside with the RTT distance, each landmark will also return the average RTT distance between itself and all the nodes that have already contacted that landmark (i.e., all the existing nodes in the system so far) to that *new node* (step 4). The landmark will then update the average RTT distance between itself and all existing nodes in the system considering the *new node* (step 5). The *new node* then finds the closest landmark to itself by a linear search in its RTT distances to the landmarks (step 6). The name ID of the *new node* will have the prefix of its closest landmark (steps 7-8). The *new node* then compares its RTT distance to each landmark with the average RTT distance of all the existing nodes in the system to that landmark and generates its name ID based on those

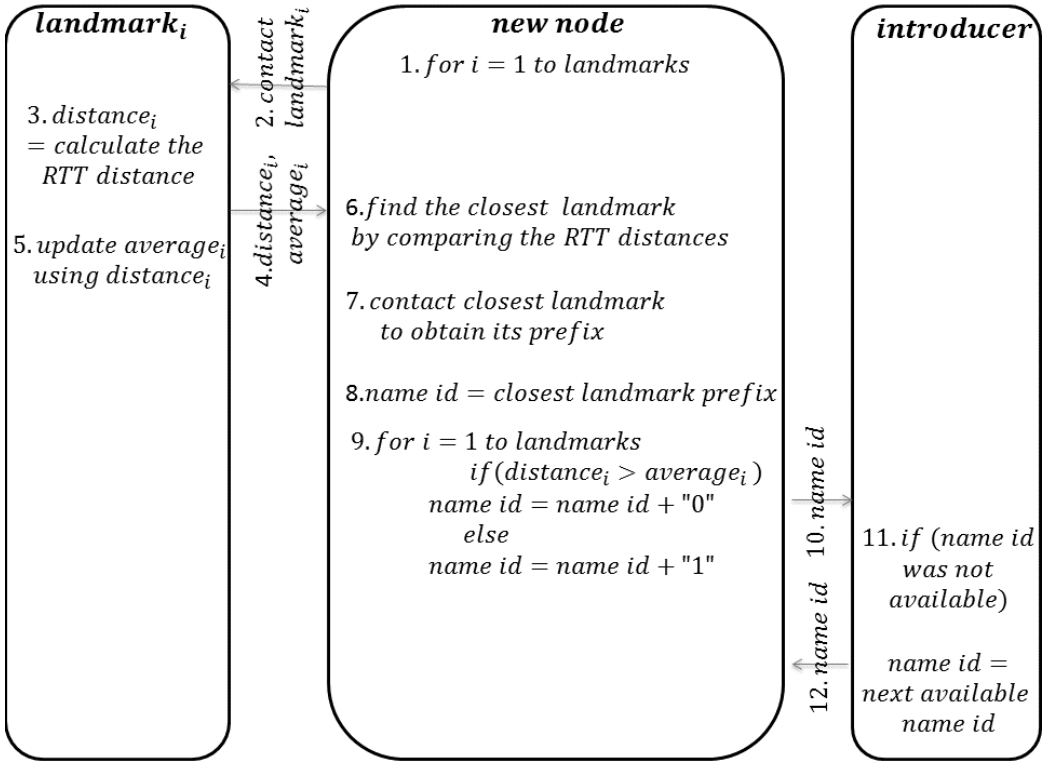

Figure 4.1: The interactions between a new node, its introducer and landmarks during the execution of DPAD algorithm.

elements (step 9).

There may already exist a node in the system with the same name ID that the *new node* generates for itself. This case occurs if the RTT distance of that node and *new node* to each landmark be in the same situation compare to the average RTT distance of the existing nodes to that landmark at the time that their name IDs was going to be assigned. In order to handle this situation, *new node* is assumed to know at least one node from the Skip Graph called its introducer. When the *new node* generates a name ID, it will send that to its introducer (step 10). The introducer checks whether the generated name ID by *new node* is already assigned to another node existing in the Skip Graph or not. If the generated name ID has been already assigned, the introducer tries to find the closest available name ID to the *new node*'s generated name ID. The introducer does this by finding the closest available leaf of the name ID tree to the generated name ID. The name ID tree is a binary tree with a null string as the root. The right child of a node has the name ID of its parent extended with a *one* bit and the left child of a node has the name ID of its parent extended with a *zero* bit.

DPAD is a dynamic algorithm that generates a name ID for each new node based on the current state of the system. Although by using the DPAD algorithm, the maximum size of name IDs in the system may increase, in order to preserve the logarithmic behavior of Skip Graph in responding the search queries, the number of levels in the skip graph will not be changed. Therefore, in a graph with 6 levels, two nodes may have more than 6-bits common prefixes but they will be in the same list up to the 6^{th} level.

4.3 Simulations Setup

In *SkipSim*, existence probability of the nodes in each point is proportional to their positioning with respect to the landmark points. Assuming nodes would appear near the landmarks with higher probability, Equation 4.1 shows the chance of each point regarding each landmark. In this equation, $chance_{ij}$ is the chance of i^{th} point in the simulation environment to be selected as a node location with respect to the j^{th} landmark and d_{ij} is the distance

between them. $maxDistance$ is the maximum possible distance between two nodes in the *SkipSim* (the diameter of simulation environment). In Equation 4.2 $chance_i$ is the sum of all the i^{th} point's chances with respect to all $|L|$ landmarks in the system where L denotes the set of all landmarks in the system with size $|L|$. The probability distribution of the nodes is obtained by Equation 4.3 where p_i is the probability that a node arrives at the point i in the screen and $\sum_{j=1}^{s} chance_j$ is the sum of all the points chance in the screen, where s corresponds to the number of points on the screen. We generated 100 random topologies, each with 64 nodes and 6 landmarks based on the described distribution, and evaluated each algorithm based on them. In our simulations, the physical distance between two nodes is assumed to reflect RTT between them.

$$chance_{ij} = 1 - \frac{d_{ij}}{maxDistance} \tag{4.1}$$

$$chance_i = \sum_{j=1}^{|L|} chance_{ij} \tag{4.2}$$

$$p_i = \frac{chance_i}{\sum_{j=1}^{s} chance_j} \tag{4.3}$$

4.4 Locality Aware Name ID Assignment: Related Work

Several approaches have been proposed to assign node identifiers in a distributed data structure. Most of these methods have been designed for DHTs, and there is no specific method for Skip Graph. We classify the existing identifier assignment approaches into two groups, namely dynamic and static algorithms.

4.4.1 Dynamic algorithms

In this class of algorithms, **the identifier of a new node could be assigned as the node arrives to the system**. *LAND* [59] chooses the identifiers of DHT nodes uniformly at *random*. In *LDHT* [60], DHT identifiers were considered as a conjunction of two independent binary strings where the first part is a prefix containing the *location information* of the node (Autonomous System Number (ASN) [61]), while the second part is chosen uniformly at *random*.

Employing a *hierarchical* addressing was proposed in [62], where the hierarchy consists of region, sub region, leaf region and a random part. In this method, name IDs have dynamic lengths. The more an area is crowded, the longer name IDs of the related nodes would have.

It was experimentally shown that *IP prefixes* would not reflect the relation between the locations of peers [63]. There are several examples of peers with at least 21 bits mask which are about 1000 miles away from each other, while according to their IP prefixes, they would assumed to be close neighbors.

4.4.2 Static algorithms

Static algorithms need locality information of all the nodes in the system before generating the name IDs. For each new node that joins the system, the static algorithm should be run and assign new name IDs to all the nodes by considering new relations between the location of new node and other existing nodes. Each time the static algorithm runs for a new arrival node, existing node identifiers may be changed.

LMDS [64] algorithm scales each node in 1-dimensional space using the distances between that node and some special nodes called landmarks. In general, LMDS does not provide the locality-based name IDs in the desired format of the skip graph. On the contrary to the skip graph where the name IDs are strings of binary digits containing locality information as their common prefix, the outcome of this algorithm is an integer which could not be considered as the name ID of the Skip Graph's nodes by itself and needs extra processing.

Several approaches were discussed based on *MDS* behavior to assign identifiers to some points in [65]. Instead of choosing landmarks and positioning all the nodes with respect to the landmarks, in *PMDS* some pivots will be selected from the ordinary nodes and the points will be positioned with respect to each other. The algorithm should be executed for each new pivot and also for each newly arriving node. *High dimensional scaling (HDS)* [65] selects a few nodes and measures distances of the other nodes to these selected nodes (e.g., K nodes). These distances would provide a K-dimensional coordination. Final coordination would be calculated using the adjacency matrix.

A method based on LMDS to convert the physical coordination of nodes to the network coordination was proposed in [66], such that the Euclidean distance between each two nodes in the network reflects the latency. The method also uses Hilbert Curve to improve the efficiency of LMDS in very dense networks. To reduce the effect of noise on LMDS in very large networks, another method [67] was suggested to divide the input into sub-parts, run LMDS on each part and combine the results, instead of running LMDS on the whole input.

Geo-Peer [68] method uses Delaunay triangulate [69] based on Voroni diagram [70] in order to generate a locality aware P2P system such that each peer is connected to its closest neighbor. In this method, several message exchange steps are used to provide a Delaunay triangle among each 3 neighbors. For each new node, several messages will be exchanged in the system for neighbor discovery, network maintenance and Delaunay triangulation. This method has some advantages such as a node will be connected to its closest neighbors, and the approach is completely P2P and routing would be efficient. However, it would not generate locality aware identifiers. Considering locality aware identifier assignment to each node, the maximum degree of each node (number of the links it has to its neighbors) is not static in this method and is related to the geographical location of the node. By employing this method in the Skip Graph, some nodes may have more than $O(\log n)$ links that is in contrast to the constraint of Skip Graph and will make the search query inefficient.

Method	Behavior	Decentralization	Locality Awareness
LAND [59]	Dynamic	Full	No
LDHT [60]	Dynamic	Hybrid	Hybrid
Hierarchical assignment [62]	Dynamic	Hybrid	Hybrid
LMDS family [64] [65] [66]	Static	Hybrid	Full
Dynamic Prefix Average Distance (DPAD)	**Dynamic**	**Full**	**Full**

Table 4.1: Comparison of various methods of identifier assignment

Table 4.1 shows our comparison of identifier assignment strategies in DHT-based data structures and proposed dynamic DPAD algorithm. One criteria to compare different methods is decentralization, which is Full if any peer could perform the identifier assignment, and Hybrid if only some super-nodes (for example landmarks) could assign node identifiers. For the locality awareness criteria, a method is Full if the node's identifier only contains its location information, and Hybrid if in addition to the location information, the identifier also contains a random part. As shown in Table 4.1, our DPAD algorithm is **the only dynamic, fully decentralized locality aware algorithm** among the others.

4.4.3 Algorithms used for comparison

We implemented related best known algorithms to compare their performance of locality awareness and query efficiency with our proposed DPAD algorithm. Implementation details of these algorithms are as follows.

4.4.3.1 LDHT

Name id of each node is its closest landmark prefix followed by a random sub-string. In LDHT, we assign a fix prefix to each landmark and assume each landmark defines a specific ASN by that prefix. All the nodes around a landmark will then belong to the same ASN group.

4.4.3.2 Hierarchical Assignment

In order to implement this algorithm, we employed the same strategy of generating Huffman prefixes for the landmarks as in DPAD algorithm. The second part of the name IDs would be chosen uniformly at random from the

available name IDs. In hierarchical assignment, density of the landmarks will determine the regions and sub-regions. The more an area is crowded by the landmarks, Huffman algorithm would generate more sub-regions by assigning similar prefix to those landmarks.

4.4.3.3 LMDS

With n nodes and $|L|$ landmarks in the system, a $n \times |L|$ *Distance* matrix will be generated such that $Distance[i][j]$ is the network distance of the i^{th} node to the j^{th} landmark. LMDS is run on the *Distance* matrix and will output a single value for each node. Nodes are ranked based on their LMDS values from zero to $n-1$ in the ascending or descending order. The name ID of each node is the conversion of its rank to the binary considering the system size of name IDs.

4.4.3.4 Dynamic Prefix LMDS (DPLMDS)

We designed and implemented this static algorithm by combining the Hierarchical assignment and LMDS algorithms together in order to obtain another static algorithm to compare with our proposed DPAD algorithm. In DPLMDS algorithm, name ID of each node consists of two parts: Huffman prefix of the closest landmark in the system to that node followed by the output of the LMDS algorithm as described above.

4.5 Performance Results

4.5.1 Locality awareness

As the locality awareness performance metric, we consider the average distance of each node to its neighbors in the overlay network. To evaluate each name ID assignment algorithm, we ran the algorithm on 100 random topologies. The average distance of each node to all its neighbors in the overlay network was computed for 100 random topologies, that is reported as the performance metric of locality awareness for each algorithm.

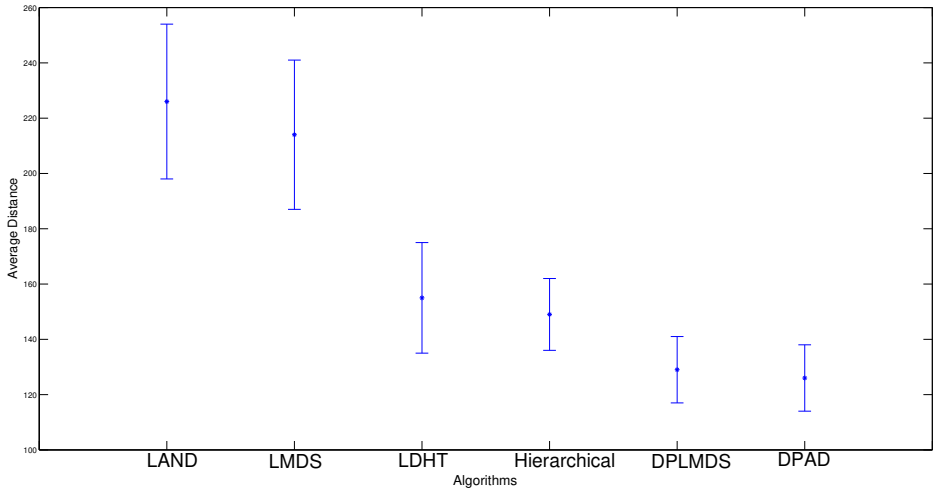

Figure 4.2: Average distance of the nodes to their look-up table's neighbors in the name ID assignment algorithms. The x-axis corresponds to the algorithms and the y-axis shows the average distance (measured in pixels).

Figure 4.2 shows the average distance of each node to its look-up table neighbors in each algorithm. As shown in the figure, our proposed **dynamic, fully decentralized** DPAD algorithm has the same performance as the **static** DPLMDS algorithm and improves the locality awareness with the gain of about 24% in comparison to the **hybrid decentralized** Hierarchical approach and with the gain of about 82% in comparison to the random assignment (LAND), which was the only dynamic, fully decentralized algorithm before DPAD. Figure 4.3 shows the average distances between all pairs of the nodes in the system based on their name IDs' common prefix produced by DPAD algorithm. As the figure indicates, **the more two nodes are closer to each other, the more common prefix they would have in their name IDs**.

4.5.2 End-to-end latency in search query

In order to evaluate the effect of name ID assignment algorithms on the end-to-end latency of search query, we ran 1000 random search for numerical ID

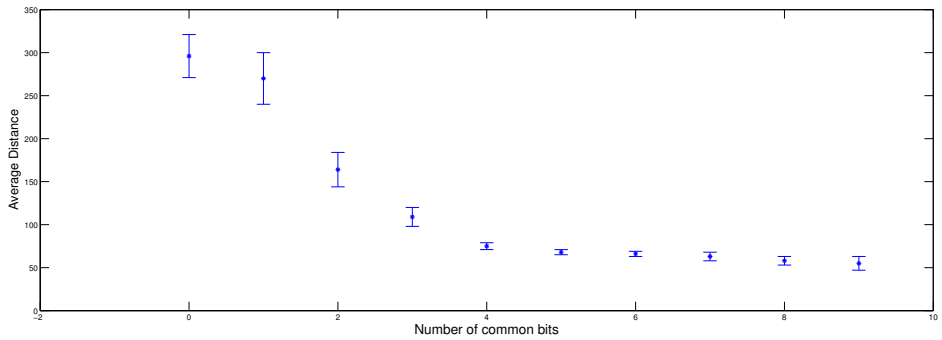

Figure 4.3: Average distance between the nodes for different number of common bit prefixes in DPAD algorithm.

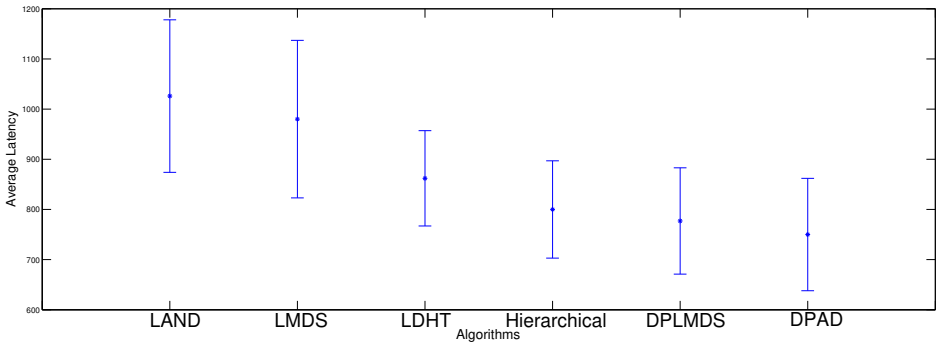

Figure 4.4: Effect of name ID assignment algorithms on the end-to-end latency of the search for numerical ID. The latency between two nodes was modeled as the physical distance between them in SkipSim simulation environment.

queries originated at random nodes per each topology for 100 random topologies (i.e., 100,000 total queries). By modeling the overlay network RTT with the physical distances between nodes, we measured the total distance that a packet travels on average to perform a search for numerical ID for 100,000 random queries as the performance metric of each algorithm. Figure 4.4 shows the performance of name ID assignment algorithms on the end-to-end latency of the search for numerical ID. The x-axis corresponds to the algorithms and the y-axis shows the average end-to-end latency.

As Figure 4.4 shows, our proposed **dynamic, fully decentralized** DPAD algorithm has similar performance to **static** DPLMDS in the case of improving search query end-to-end latency. In comparison to the randomized assignment algorithm (LAND) which was only dynamic and fully decentralized algorithm before ours, DPAD improves end-to-end latency with the gain of 40%. Although in comparison to the **hybrid decentralized** Hierarchical assignment of the identifiers, DPAD improves the end-to-end latency with the gain of 6%, the main difference between DPAD and Hierarchical assignment is in terms of their decentralize behavior. In contrast to the Hierarchical assignment, DPAD assigns node identifiers in a fully decentralized manner. In DPAD, all the existing peers in the system are able to assign a locality aware name ID for a new node, whereas in the Hierarchical assignment only some super-nodes that know the hierarchical division in the system can perform the assignment.

Before assigning the name IDs, LDHT, Hierarchical, DPLMDS and DPAD algorithms use a clustering approach and put the nodes into groups. On the contrary, Random and LMDS algorithms assign the name IDs without pre-clustering the nodes. This difference causes a gap between the cluster-based algorithms (LDHT, Hierarchical, DPLMDS and DPAD) and non-cluster based algorithms (Random and LMDS) as shown in Figures 4.2 and 4.4. Due to this reason, standard deviations of the measurements for the algorithms in each group resemble each other.

4.6 Conclusions

With the aim of reducing end-to-end latency in search queries, we proposed a novel algorithm to make Skip Graph nodes locality aware. We address the locality awareness problem caused by the name ID assignment method of the Skip Graph's nodes. The proposed approach DPAD is a dynamic and fully decentralized algorithm, which assigns the identifiers of the Skip Graph nodes based on their location information at their arrival time to the Skip Graph. DPAD, which is the only proposed locality aware identifier assignment algorithm for the Skip Graph, assigns the name IDs by analyzing the distance of the nodes with respect to some supernodes in the system called landmarks.

In order to analyze the performance of DPAD algorithm, we implemented a simulator called SkipSim. Through SkipSim, we analyzed the performance of DPAD algorithm and compared it with the state-of-the-art assignment algorithms in the terms of nodes locality awareness and end-to-end latency of search queries. DPAD algorithm improves locality awareness of Skip Graph nodes with the gain of about **82%**, and end-to-end latency of search queries with the gain of about **40%** in comparison to the best known static and dynamic algorithms. In the DPAD approach, distribution of the nodes is assumed as a function of the landmarks' distribution. The *optimal landmark placement* is left as an open problem.

Acknowledgement

The authors thank Türk Telekom for their support under grant number 11315-06.

Chapter 5

LARAS: Locality Aware Replication Algorithm for the Skip Graph

5.1 Introduction

Skip Graph is a member of the DHT-based distributed data structures [31]. Like the other members of the DHT-based distributed data structures, each node of a Skip Graph has a lookup table, which holds the $\{identifier, address\}$ pairs. The lookup operation is defined as searching for a specific $identifier$. The result of the lookup operation is the $address$ of the node holds that specific $identifier$ [31]. The ability to perform concurrent search operations, scalability and fast searching, make the Skip Graph a suitable underlying infrastructure for the P2P storage applications [34, 40, 41]. 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 [42, 43].

A peer-to-peer (P2P) storage system consists of a group of peers (i.e., nodes). In such systems, there is no specific client and server. Rather, a node can act as a server for a subgroup of nodes while it queries some other

nodes of the system as a client. In such systems, each node can be both a data owner and a data requester, simultaneously. 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. As this sub-group spreads out, the rate that the data owner is queried increases. The more the data owner is queried, its chance of failure will increase due to the heavy load of transactions. The moment a data owner is failed, its data items' become unavailable to its data requester nodes. In this situation, no data requester node can query any of its corresponding data items. Also, increasing the transaction load on the data owner will increase its query processing and response time. Increasing the query processing and response time will cause the performance of the system to decrease significantly.

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 directly. This will reduce the transaction load, query processing and response time of the data owner. Therefore, in a P2P storage system, replicating the data item of all the data owners will increase the performance of the storage system.

Traditional decentralized replication algorithms aim at improving the performance of P2P storage systems by reducing the response time of the data requester nodes' queries. Randomized replication [71], replication on the neighbors [2,59,72] and replication on the paths between the data owner and some data requester nodes [72–74] 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. In some cases, these randomnesses cause the assumptions about the system to be far from the reality. This increases 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.

Assuming the data requester nodes have been distributed uniformly at random, randomized replication helps the data requester nodes to be assigned to a closer replica. This will reduce the access delay of the data requester nodes to the replicas and hence improves the system performance. Other strategies employ this randomness accompanied with some assumptions about the data requester nodes, wishing that these assumptions may improve the latency by providing a better clustering of the data requester nodes. By assuming the high density of data requester nodes on the path between the data owner and some data requester nodes [72–74] replicate the data owner on the path between itself and some arbitrary data requester nodes. In the case the data owner has several underlay network neighbors in different network overlay locations, by assuming that most of the data requester nodes will be emerged around some notable neighbors of data requester, replicating on neighbors strategy [2, 59, 72] has been proposed. Employing some kind of randomness, however, prevents these traditional replication algorithms to purely consider the locations of data requester nodes in the replica placement procedure. In some cases, this randomness causes the assumptions about the system to be far from its reality. This will increase data requester nodes' access delay. Increasing the access delay will lead the P2P storage system to be inefficient in the terms of query processing and response time.

Our solution to improve the access delay of the data requester nodes in the Skip Graph based storage systems is to provide a locality aware replication strategy. We propose a locality aware replication algorithm for skip graph (LARAS). LARAS addresses the data requester nodes' access delay problems for the Skip Graph based storage systems using a landmark-based identifier assignment method like DPAD [75]. 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 is minimized. By employing LARAS, 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 achieved. Likewise, since Skip Graph can be used as a 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.
- We extended the Skip Graph simulator, SkipSim [38], 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 chapter, in Section 5.2, we present our LARAS approach. The related works and simulation setup are described in Sections 5.3 and 5.4, respectively. The simulations' results are presented in Section 5.5, followed by conclusions in Section 5.6.

5.2 Locality Aware Replication Algorithm for the Skip Graph (LARAS)

5.2.1 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 the 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 is 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 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) [76] 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.

5.2.2 Algorithm Description

5.2.2.1 Inputs

Figure 5.1 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 5.1, step 1).

5.2.2.2 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 5.1, step 2). Employing the locality-aware name ID assignment algorithm DPAD [75], the whole system is divided into regions based on the landmark locations. The higher chance of a region is in emerging nodes, the longer prefix its landmark owns. For the case of public replication, the unit share

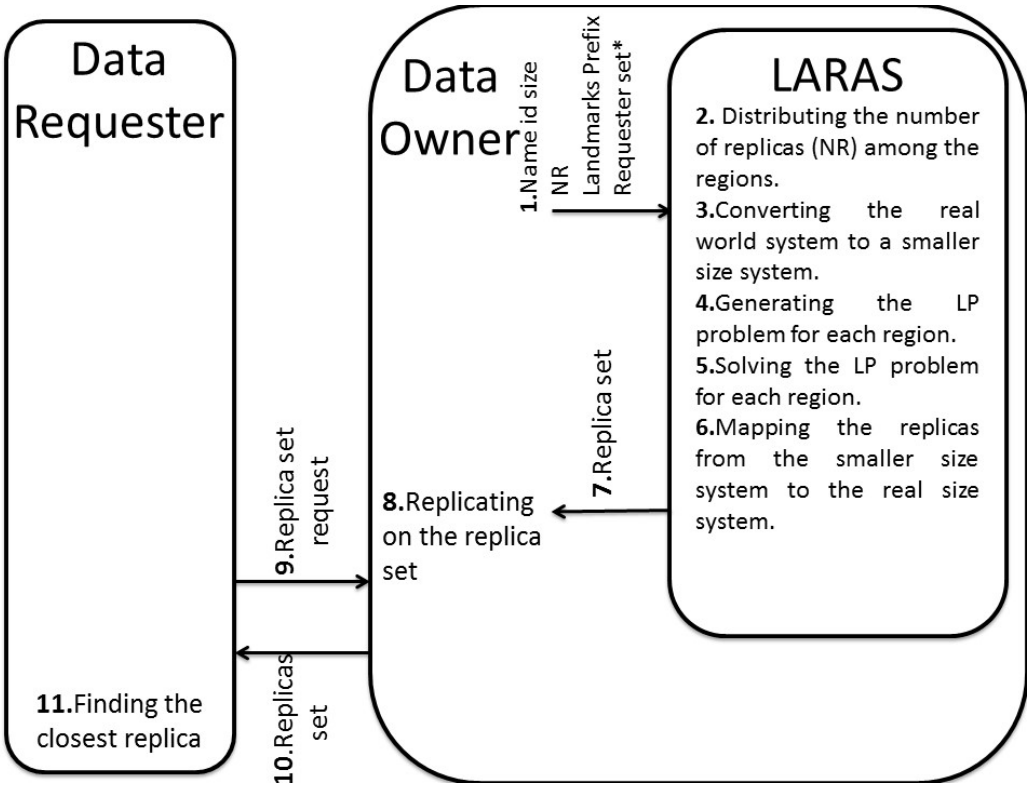

Figure 5.1: The interactions between the data owner node and data requester nodes during and after the execution of LARAS. The *Requester Set* is only needed for the case of private replication*.

per region (USPR) is defined by Equation 5.1. 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 5.2. In this equation, L is the set of all the landmarks and NR_i is the number of replicas for the i^{th} region.

$$USPR = \frac{NR}{\sum_{i=1}^{|L|} P_i} \tag{5.1}$$

$$\forall i \in L \quad NR_i = P_i \times USPR \tag{5.2}$$

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, a number of data requester nodes in each region is considered.

5.2.2.3 Shrinking the problem size

After distributing the replicas among the regions, LARAS converts the real world system to a smaller size system (Figure 5.1, 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 5.3. 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), max_P 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 5.3 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.

$$S_i = \lceil \log(\frac{P_i}{max_P} \times \frac{n}{\log n} \times \log NR) \rceil \tag{5.3}$$

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.

5.2.2.4 Generating the LP model

For each region, LARAS generates the latency table D based on the name ID size of the smaller size system. The $D_{i,j}$ 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, $D_{i,j}$ 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 5.1, step 4):

$$\min \forall i \in M, \forall j \in K \quad \sum_{i=1}^{m} \sum_{j=1}^{k} D_{i,j} X_{i,j} \quad \text{s.t.} \tag{5.4}$$

$$\forall i, j \in M \quad Y_i \geq X_{i,j} \tag{5.5}$$

$$\forall j \in M \quad \sum_{i=1}^{m} X_{i,j} = 1 \tag{5.6}$$

$$\forall i \in M \quad \sum_{j=1}^{m} X_{i,j} \geq Y_i \tag{5.7}$$

$$\sum_{i=1}^{m} Y_i = nr \tag{5.8}$$

$$\forall i, j \in M \quad Y_i \in \{0,1\}, \quad X_{i,j} \in \{0,1\} \tag{5.9}$$

Equation 5.4 (The Objective function) Equation 5.4 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 for name ID* for each data requester node of that region of the real world system in the smaller size system. As was discussed in Section 2.4, the search for name ID will return one of the most similar nodes in the case of name ID to each real world's data requester nodes 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 $Y_i = 0$.

Equation 5.5 (The replicas' constraint) The constraint presented by Equation 5.5 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.

Equation 5.6 (The data requester nodes' constraint) The second constraint of this LP model is presented by Equation 5.6. Each node should be assigned to only one replica.

Equation 5.7 (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 5.7.

Equation 5.8 (The number of replicas' constraint) The next constraint related to the number of replicas is defined by Equation 5.8. 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.

Equation 5.9 (The allowed values for the variables constraints) Finally, the last constraint is shown by Equation 5.9. 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.

5.2.2.5 Mapping the replicas

After LARAS models the access delay based replication for each region with an LP problem (Figure 5.1, step 4) and solves the LP model (Figure 5.1, step 5), it maps the replicas from the smaller size system to the real world

system (Figure 5.1, 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. Each Search for name ID in the real world system with n nodes will take $O(\log n)$ on average. Totally NR number of Search by Name ID needed to search for each replica in the real world. Therefore, the whole mapping phase will take $O(NR \times \log n)$. 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.

5.2.2.6 Finding the closest replica

After LARAS outputs the real world replicas (Figure 5.1, step 7), the data owner replicates its content on the replicas (Figure 5.1, 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 5.1, step 9), the data owner sends back its replica list (Figure 5.1, 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 5.1, 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.

5.3 Related works

There exist several locality-based replication strategies for the P2P systems trying to minimize the average access delay of replication. However, there is not any specific replication strategy for the Skip Graph nodes. In this section, we classify the existing replication strategies into the centralized and decentralized approaches.

5.3.1 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 each considers certain aspects of the system:

5.3.1.1 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 [71]. Initially, the set of replicas is empty. At each step, one server is chosen as a replica and all of the clients it covers will be removed from the client set. If a server would not cover any client, it will be ignored. This procedure continues until all the clients will be covered by the replicas.

5.3.1.2 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 [72–74,77]. Freenet [78], OceanStore [79] and Mojo Nation [80] use this replication strategy.

5.3.1.3 Replicating on neighbors

The data owner is assumed to only know its neighbors in the underlying system [2, 72, 81, 82]. In some P2P systems, the identifiers are chosen uniformly at random [59] 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 neighbor's approach distributes a certain object across the network regardless of a number of demands for that object. On the other hand, if the identifier assignment strategy is locality aware, as in DPAD [75], the data owner would have more nearby neighbors than the faraway ones. In this

Strategy	Decentralization	Locality Awareness	Behavior
Randomized [71]	Full	No	Dynamic
On Path [72–74]	Full	No	Dynamic
On Neighbors [2, 59, 72]	Full	Hybrid	Dynamic
Consistent Hashing [72, 83–87]	Full	Hybrid	Dynamic
Objective-based [88, 89]	No	Full	Static
Genetic Algorithms [82, 90]	No	Full	Static
LARAS	**Full**	**Full**	**Dynamic**

Table 5.1: Comparison of various methods of locality-based replication strategies

situation, replicating on the neighbors would mostly replicate the data items on the nearby neighbors than the faraway ones.

5.3.1.4 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 [72, 83–87]. In some cases, multiple hash functions are employed on a certain id resulting in several distinct replicas for a single node [72, 77].

5.3.2 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. As their main drawback, the centralized algorithms need global information of the network such as the capacity of the nodes, availability [91, 92], bandwidth and pairwise access delay of the nodes [89]. In this class of replication algorithms, some special nodes are coordinators [88] 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.

One very common strategy for the centralized replication algorithms is to employ the genetic algorithms on the replica sets [82, 90]. In a network with N nodes, the replica set of a certain data object can be modeled as a N-bits chromosome. If the i^{th} node of the system holds a replica of that data object, then the i^{th} bit of the chromosome becomes 1 otherwise *zero*. Modeling the replica set by the chromosomes, some random replica set will be generated as the input of the genetic algorithm. These initial solutions will be examined by the objective function and the suitable candidates will be determined. By employing a heuristic on the suitable candidates [93], new generations of the replica set with better objective function values will be generated. However, these centralized replication algorithms do not result in the optimal solution, it is claimed to be close to the optimal solution.

Table 5.1 shows a comparison of various methods of locality-based 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 location of the nodes. Finally, in the case of *behavior*, a method is "Static" if it needs information of all the nodes to function, otherwise, it is "Dynamic".

5.3.3 Algorithms used for comparison

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

5.3.3.1 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.

5.3.3.2 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.

5.3.3.3 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.

5.3.3.4 The LP algorithm

Instead of modeling and solving the LP problem for each region, the LP algorithm models and solves the whole access delay based replication for the whole system.

On the other hand, the consistent hashing replication methods are mainly designed for other improvement purposes like availability [87], accessibility [83] and indexing instead of replicating [72, 77]. Since these goals are different than the access delay improvement, we did not use consistent hashing algorithm for comparison.

5.4 Simulation Setup

We extended the Skip Graph's simulator environment, SkipSim [38], 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 a 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 [94] on Windows 8.1.

5.5 Performance Results

The performance results of the replication algorithms in the terms of access delay and scalability are presented in this section. As was described in the Section 5.3, the main algorithms to compare with our proposed LARAS algorithm are the randomized replication, replication on the neighbors and replication on the path. We also added another algorithm, knows as LP, which is similar to our proposed LARAS algorithm with some differences in its behaviors. The goal of considering this extra algorithm was to compare the effect of these differences on its performance.

5.5.1 Average 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-bits name ID size system configuration, and we observed similar behavior in different network size scenarios.

Figures 5.2 and 5.3 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 the path (as the best known decentralized replication algorithm), LARAS improves the access delay about **20%** and **38%** on average in the public and private replica-

tion scenarios, respectively. In comparison to the optimal LP solution of the system, based on the physical 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.

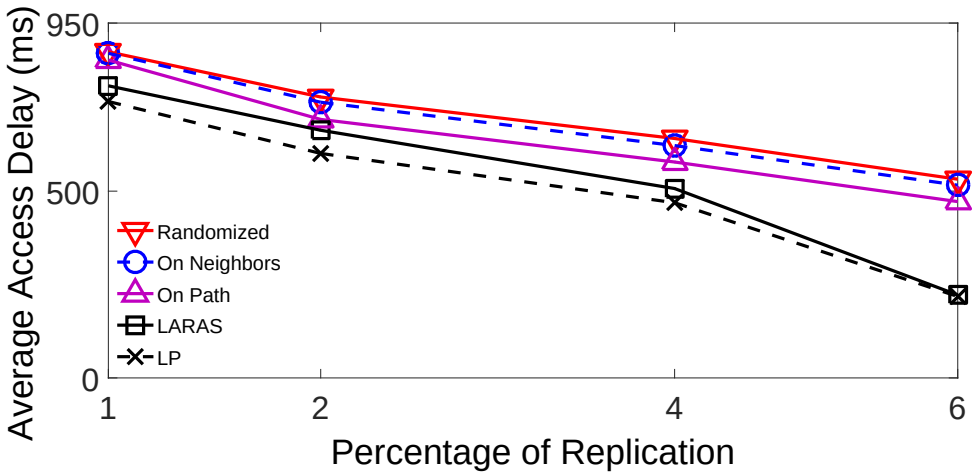

Figure 5.2: Public replication access delay vs percentage of replica nodes in the system. System size: 1024 nodes, name ID size: 10 bits. Y-axis shows average latency of a node to its closest replica and x-axis shows percentage of replica nodes in the system.

5.5.2 Scalability

We compared LARAS with the best-decentralized counterpart, replication on path algorithm, from the scalability point of view. In each system setup,

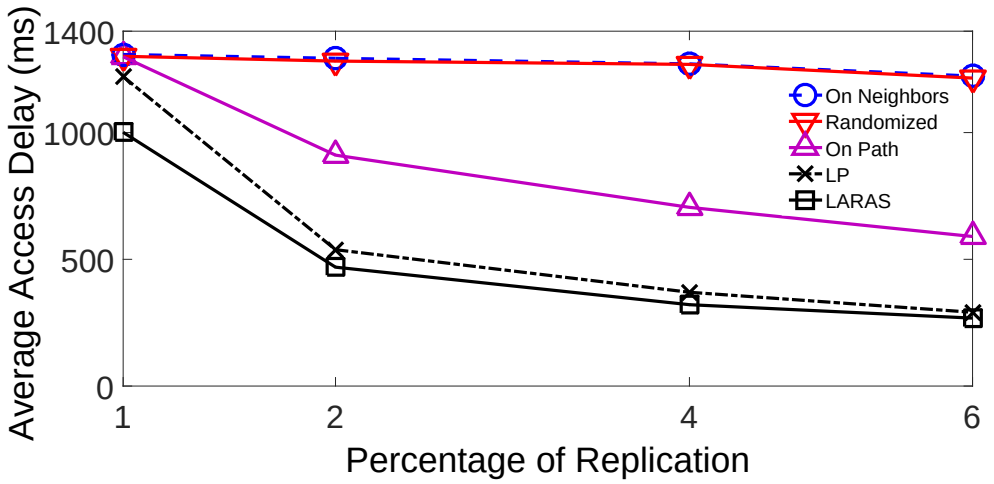

Figure 5.3: Private replication access delay vs percentage of replica nodes in the system. System size: 1024 nodes, name ID size: 10 bits, size of data requester nodes' set: 100. Y-axis shows average latency of a node to its closest replica and x-axis shows percentage of replica nodes in the system.

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. Each of the system setups was simulated for the 100 identical randomly generated topologies. Figure 5.4 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.

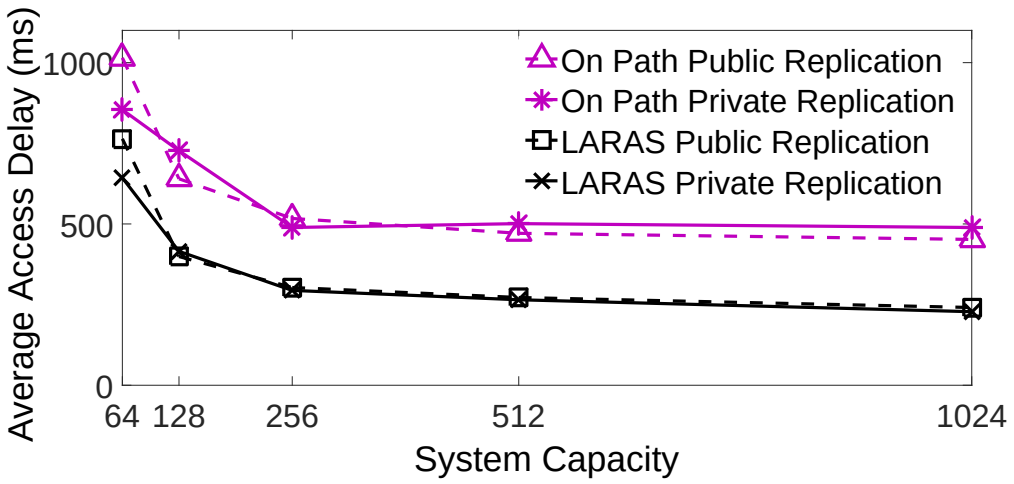

Figure 5.4: Scalability of LARAS vs replication on path (the best known decentralized counterpart). Y-axis shows average latency between a node and its closest replica. X-axis presents system size. For each system setup, the number of replicas is about 5% of the system size. For private replication, size of data requester nodes' set is about 30% of the system size.

5.6 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 [75] as their identifier assignment strategy.

To evaluate the performance of LARAS, we extended the Skip Graph simulator, SkipSim [38, 75] 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.

Chapter 6

Awake: decentralized and availability aware replication for P2P cloud storage

6.1 Introduction

Peer-to-peer (P2P) cloud storages consist of a set of nodes where a node can be both a data owner and a data requester simultaneously. Data owner possesses a set of data objects and aims to share them with a group of authorized nodes called data requester nodes. In P2P cloud storages, nodes oscillate between online and offline periods. A node arrives at the system and starts an online state. After a while, it terminates its session and departs from the system or fails. A departed or failed node may join the system at a later time or may leave the system forever. This dynamic behavior of the nodes is called churn [95]. Churn negatively affects the data availability in P2P cloud storage. When a data owner goes offline or fails, its data objects would no longer be available to its set of data requesters.

To remedy the data availability problem of P2P cloud storages under churn, the data owner makes copies of its data objects on the other nodes (i.e, replicas) and the operation of determining replicas is called replication [96, 97]. After replication is done, in case data owner becomes unavailable,

the data requesters can utilize the replicas.

The traditional decentralized availability-based replication algorithms aim at improving the performance of P2P cloud storage by either immediately resolving replica's failure or by providing an average number of available replicas for a long period of time e.g., providing three available replicas on average for one month. The average number of available replicas is defined as the average number of replicas that are available during a certain period of time. For example, considering the period as an hour, it is defined as the average number of replicas available at each hour. Probing the replicas and determining a new replica for each failure [72, 87, 98–100], randomized replication [1], cluster-based replication [4–7], and correlation-based replication [3] are the most common decentralized availability-based replication methods.

All of the decentralized availability-based replication algorithms employ some kind of randomness for replica selection. Employing randomness prevents the algorithms to purely consider the availability patterns of nodes in the replica selection procedure. This leads the system to cases where only a few number of replicas or even no replica is available. Poor availability of replicas results in performance degradation of the P2P cloud storage in the terms of data availability. Likewise, all of the decentralized availability-based existing solutions make explicit assumptions about the underlying churn behavior of the system for replica selection. This causes their performance to be extremely narrowed by the underlying churn behavior of the system. In the case where the churn behavior of the system is mispredicted or changes over the time, employing the existing solutions results in poor data availability.

To improve the data availability of P2P cloud storages under churn, **we propose a dynamic, fully decentralized, churn behavior independently, and availability aware replication algorithm named Awake**. We define availability aware replication as dividing a fixed size periodic time interval ($FPTI$) into a set of identical time slots (TS) and placing the replicas such that the maximum availability of replicas during each TS is achieved. By employing Awake, a data owner can replicate its data objects in a fully decentralized manner with the availability awareness of the replication is achieved. For instance, considering $FPTI$ as a day and TS as an hour,

compared to the best existing solutions our approach provides maximum availability of replicas during each hour of the day.

Awake can be employed in any structured P2P cloud storage given that the availability information of nodes is piggybacked on the messages that they route or initiate. The communication complexity of Awake is asymptotically identical to the communication complexity of the underlying structured P2P system. For instance, employing Awake in a Distributed Hash Table (DHT)-based [87] cloud storage with the capacity of n nodes that have the communication complexity of $O(\log n)$, costs the message overhead of size $O(\log n)$.

The contributions of this study are:

- We propose Awake: the first churn behavior independent, dynamic, fully decentralized availability aware replication algorithm for P2P cloud storages.
- Space complexity of Awake is linear in the number of registered users to the system.
- Communication overhead of Awake is the same as the communication complexity of the underlying P2P cloud storage that Awake is applied on.
- We extended the SkipSim [38, 75], for simulating and evaluating the availability-based replication algorithms.
- We modeled the best known decentralized availability-based replication algorithms on SkipSim and compared with Awake.
- The simulation results show that on average regardless of underlying churn behavior of the system, Awake improves the availability of replicas with a gain of about **21%** in comparison to the best known decentralized existing solutions.
- Awake is scalable in the sense that it achieves the same performance independent of the system size.

- Employing Awake in a system with 1 million nodes, charges a space consumption of **25 megabytes** on each node and a communication overhead of **480 kilobytes** on each message.

The rest of the chapter is organized as follows. Section 6.2 presents the system characteristics that Awake can be applied on. In Section 6.3, our approach Awake is described in detail. The related works and simulations setup are presented in Sections 6.4 and 6.5, respectively. Performance results are presented in Section 6.6, followed by conclusion in Section 6.7.

6.2 Preliminaries

6.2.1 System Architecture

Awake works on top of a structured P2P system such as a Distributed Hash Table (DHT). In a structured P2P system, the overlay network is based on a specific topology where the nodes can efficiently search for other nodes and data.

In our view of a structured P2P system, nodes just use the overlay network to efficiently search for other nodes and data. In a search operation, the node that initiates the operation is named the initiator. The node or data object that an initiator searches for is called search target. As the result of a search operation, the address of the target node or address of the node that owns the target data object is returned to the initiator. Once the search initiator receives the address of the search target, it communicates with the search target directly.

We consider a structured P2P cloud storage where each node corresponds to a unique peer in the real world. There exist two roles for each peer: data owner and data requester. Data owner owns a set of data objects and data requester is the authorized consumer of those objects. A peer can be both a data owner as well as a data requester for other data owners.

6.2.2 Availability Information

6.2.2.1 Availability Vector

Availability pattern of a node is represented by a vector of size s and is called the availability vector. s is defined as the number of time slots and is equal to $\frac{FPTI}{TS}$. Availability vector of a node i is denoted by $AV_i[]$. $AV_i[t]$ represents the availability probability of the node i in time slot t of the *FPTI*. Node i computes its $AV_i[t]$ by dividing the total duration of its availability during time slot t over the number of times that *FPTI* has repeated up to the time of computation. For example, consider a system with *FPTI* equal to a day and *TS* equal to an hour, which five days have elapsed since the system initialization. In such a system, s is equal to 24 and hence the availability vector of nodes is a vector of size 24. $AV_i[t]$ is computed as the total duration that node i was available at hour t, over five days. For example, if during the last five days the total availability duration of node i during hour t was 2.3 hours, $AV_i[t] = \frac{2.3}{5} = 0.46$.

6.2.2.2 Availability Table

The availability table of a node is the local availability view of the node for the system. Availability table of node i is represented as $AT_i[][]$, which is a two-dimensional array of size $n \times s$, where n denotes the maximum number of registered users to the system. $AT_i[j][t]$ is the knowledge of the node i about the availability probability of the node j in time slot t of *FPTI*.

A node piggybacks its *id* (i.e., name ID or numerical ID in Skip Graph case) and availability vector on the messages that it routes or initiates. On receiving a message to route or read, the receiver obtains the piggybacked availability vectors and updates its availability table accordingly. Nodes are assumed to be honest in computing and piggybacking their availability vectors as well as their *id*s.To emphasize the recent availability behavior of node j while considering its availability history, the node i updates $AT_i[j][t]$ using an exponential moving average mechanism [101].

$$\forall t \quad 0 \leq t \leq S \quad AT_i[j][t] = (1 - \beta) \times AT_i[j][t] + \beta \times AV_j[t] \qquad (6.1)$$

Equation 6.1 presents update procedure of node i's availability table on receiving a message that contains the availability vector of node j. In this equation, the right side and left side $AT_i[j][t]$ values are the (j,t) entry of node i's availability table before and after the update, respectively. $AV_j[]$ is the availability vector of node j and $0 \leq \beta \leq 1$ is the effect factor of amplifying the new availability probabilities over the old ones in learning the value of $AT_i[j][t]$. We call β as the **learning factor**.

6.2.3 Churn Model

In P2P cloud storage, a node oscillates between online and offline states. In each online state, the node is available for a certain while named the session length. Availability of a node corresponds to its session length. As the session length elapses, the node goes offline for a certain while that is called the downtime. After the downtime, the node rejoins the system and this cycle flows.

In this chapter, we assume that the connectivity of the structured P2P routing infrastructure is churn resilient. The departure or failure of a node results in connectivity disruption in the paths routed from that node. This negatively affects the connectivity of the whole system and yields failure in routing the request. To overcome this problem, it is assumed that a node maintains and frequently updates an alternative direct communication link between its consecutive predecessors and successors. When the node departs or fails, its consecutive predecessors and successors use the alternative direct communication to route a query. Hence, although the churn negatively affects the data objects availability of the system, it does not affect the system connectivity.

For a churn model, we define the **availability ratio** as the ratio of its session length over its downtime. Based on the availability ratio, a churn model is high available if its availability ratio is higher than one. A churn model is moderate available if its availability ratio is around one. And a churn model is low available if its availability ratio is less than one.

Session length and downtime distributions of the churn model define the

Churn Model	Distribution	SL λ	SL α	Average SL (Hour)	DT λ	DT α	Average DT(Hour)	Availability Ratio
High Available [102]	Weibull	0.35	0.34	1.96	0.179	0.34	0.99	1.97
Moderate Available [102]	Weibull	0.69	0.59	1.06	0.65	0.34	0.99	1.07
Low Available [105]	Exponential	0.118	—	8.42	0.017	—	58.32	0.14

Table 6.1: Characteristics of the BitTorrent-based churn models used. SL and DT correspond to the session length and downtime distributions, respectively.

availability pattern of nodes. There exist several studies on the churn models, each considering a certain system e.g. BitTorrent [102–105], Kad [102], Gnutella [102, 106], eDonkey [3], Kademlia [107], and PlanetLab [100, 108].

Among the churn models reviewed, the BitTorrent-based models are concrete, parametrically clear and consistent with each other, and hence were selected to model our system. Similar to [102–105] we considered the size of the system in the order of couple of thousands of nodes. Also, the size of data requester set is assumed to be in the order of a couple of thousand nodes. Table 6.1 summarizes the characteristics of these churn models where SL and DT correspond to the session length and downtime distributions, respectively. These models either follow a Weibull or an Exponential distribution. Followings provide an overview on the description of Weibull and Exponential distributions.

6.2.3.1 Weibull-based Churn Model

In a Weibull-based churn model [102], the session length and downtime of nodes are modeled with Weibull distributions. A Weibull distribution is defined with two parameters: shape and scale. The shape parameter affects the shape of the distribution and is denoted by α. The scale parameter is denoted by λ. A larger scale parameter results in the distribution to be less spread out and more concentrated around its expected value. Considering the random variable T as a representation of the session length (or downtime), Equation 6.2 represents the Cumulative Distribution Function (CDF) of the session length (or downtime) of a system that follows the Weibull distribution. Also, Equations 6.3 and 6.4 represent the mean and variance of the corresponding Weibull distribution, respectively. In those equations, Γ is the Gamma function. For a positive integer i greater than one, $\Gamma(i)$ is

defined as $(i-1)!$.

$$\Pr(T \leq t) = 1 - \exp(-(\frac{t}{\lambda})^{\alpha}) \tag{6.2}$$

$$E[T] = \lambda \times \Gamma(1 + \frac{1}{\alpha}) \tag{6.3}$$

$$var(T) = \lambda^2 \times \left[\Gamma(1 + \frac{2}{\alpha}) - \Gamma^2(1 + \frac{1}{\alpha})\right] \tag{6.4}$$

6.2.3.2 Exponential-based Churn Model

In an exponential-based churn model [105], the session length and downtime of nodes follow exponential distributions. An exponential distribution is specified with a rate parameter (λ). The rate parameter of the downtime and session length distributions shows how often a node arrives at the system or departs from the system, respectively. Considering random variable T as the session length (or downtime) of the nodes, Equation 6.5 represents the CDF of the session length (or downtime) of nodes in a system that follows the exponential distribution. Likewise, the mean and variance of the random variable T are shown in the Equations 6.6 and 6.7 respectively.

$$\Pr(T \leq t) = 1 - exp(-\lambda t) \tag{6.5}$$

$$E[T] = \frac{1}{\lambda} \tag{6.6}$$

$$var(T) = \frac{1}{\lambda^2} \tag{6.7}$$

6.3 Awake: Availability-Aware Replication

6.3.1 Scenario

As a data owner joins the system, it starts searching for other nodes as well as routing other nodes' search queries. The data owner sends its availability vector to other nodes by means of piggybacking on the query messages it initiates or routes. Likewise, the data owner obtains other nodes piggybacked

availability vectors and updates its availability table accordingly. This is called the **learning phase**. In the learning phase the data owner learns about the availability behavior of the system. The learning phase is common among all the existing availability-based solutions. A larger learning phase helps the data owner to obtain a more accurate availability view from the system. The data owner continues to this learning up to a point which is called the replication time. At the **replication time**, the data owner runs Awake using its availability table as well as the authorized number of replicas. As the Awake terminates, it provides the replica set to the data owner and data owner replicates on the replica set.

After the replication is done, the data owner cycles this procedure by collecting availability vectors of other nodes and piggybacking its availability vector up to the next replication time. By each execution of Awake, the replica set may be changed based on the availability table of the data owner. Upon obtaining a new replica set, the data owner removes the current replicas which are not listed in the new replica set as well as replicates on the new node in the replica set.

6.3.2 Motivations and Challenges

In Awake, the main challenge is to move toward maximizing the replica availability in a decentralized manner regardless of the underlying churn behavior of the system. The main disadvantages of the existing decentralized solutions are considering explicit assumptions about the underlying churn behavior of the system (and hence strong dependency on it) as well as employing randomness in replica selection. The hard parts in designing Awake were to achieve a formal formulation of availability aware replication that is refined from randomized decisions and any assumption about the underlying churn behavior of the system.

Awake is the first decentralized availability aware replication algorithm with mathematical objective, formulation, and constraints. This formulation is completely free of including any assumption about the underlying churn model of the system. Heart of Awake is its ILP model of availability aware

replication which its correctness is self-explanatory based on the nature of ILP models. Due to its ILP maximization objective that is clearly bounded by the constraints, being free of randomness, and being independent of underlying system's churn behavior, Awake is theoretically expected to outperform the existing solutions. This claim is supported by the empirical results.

6.3.3 Algorithm Overview

Awake is a dynamic fully decentralized availability aware replication algorithm for the P2P cloud storages. By employing Awake, a data owner can determine its replicas without the need for communicating with any special node as the coordinator. Since Awake is an availability aware algorithm, by employing it in a system, maximum replication availability during each time slot is acheived. Maximum replication availability corresponds to the maximum average number of replicas available during each time slot. Also, Awake is independent of the churn behavior of the system. That is, regardless of the churn behavior of the system, Awake always guarantees the availability awareness of replication. Awake considers the replication as copying the data objects that data owner wants to share on another node which is called a replica. Replicating data objects of a node in Awake does not necessarily mean that the data objects are publicly available. Access control is a separate problem that can be solved, for example, by means of encrypting the data objects and delivering the keys to the authorized parties. The authorized parties are called the data requester nodes. Figure 6.1 illustrates the interactions between the data owner, data owner's neighbor, and Awake. As the input, Awake receives availability table of the data owner as well as authorized number of replicas (Step 1). The authorized number of replicas is called the **replication degree** and is represented by R. Awake then models the availability aware replication as an integer linear programming (ILP) (Step 2), solves the LP relaxation of it (Step 3) and provides the replica set to the data owner (Step 4). Data owner replicates its data objects on the replica set provided by Awake (Step 5).

Data owner also sends backup of the replica set (i.e, address of replica

nodes) to its neighbors (i.e. nodes which are directly connected to the data owner in the structured P2P overlay) (Step 6). Thus, the replicas' addresses would be available even if the data owner becomes unavailable. Data owner runs Awake periodically with its updated availability table. Likewise, if the churn behavior of system changes and negatively affects the availability of replicas, better replicas in terms of availability are selected in the next run of Awake.

Data Replication vs. Replica Set Replication: The data owner does the data replication based on the replica set that is provided by Awake. However, in order to prevent the unavailability of the replica set by failure or departure of the data owner, similar to [109], the data owner makes a backup of its replica set accompanied by an expiration time on its neighbors in the underlying structured P2P (Step 6). This procedure is called replica set replication. The expiration time is defined as the time of next execution of Awake based on the updated availability information of nodes. Likewise, the data owner's neighbors are the nodes which are directly connected to the data owner in the structured P2P routing infrastructure. The data owner is assumed to check its neighbors list frequently and back up its replica set on the new neighbors. It worth to note that Awake only provides the replica set for data replication. While the replica set replication is done independently of Awake as a fault tolerance approach. When an intermediate node receives a search query requesting the replica set of a certain data owner, it checks whether it possesses a valid replica set backup of that data owner. A valid backup is the one that is not expired. If the intermediate node has a valid backup, it responds to the search query on behalf of the data owner by sending the replica set to the data requester node. Otherwise, the intermediate node routes the search query to the data owner via the next node.

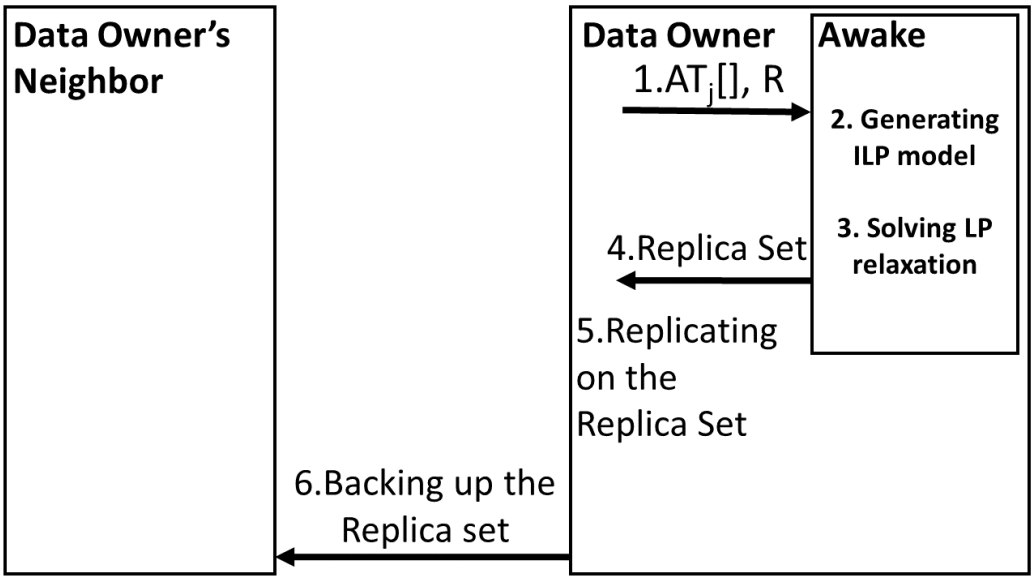

Figure 6.1: The interactions between the data owner, data owner's neighbor and Awake.

6.3.4 Algorithm Description

6.3.4.1 Input and Output

Considering data owner j running Awake, inputs of Awake are availability table of the data owner, $AT_j[][]$, as well as the replication degree of the data owner, R. The output of Awake is a binary vector Y with the size n where n is the capacity of the system in terms of nodes and is considered as a constant regardless of system arrivals and departures. The capacity of the system is defined as the maximum number of users that can register to that system. Although n is constant, new nodes can join the system, and old nodes may depart permanently. However, the total number of user registration to the system is assumed to be constant n. If node i is selected as a replica, $Y_i = 1$, otherwise $Y_i = 0$.

6.3.4.2 Generating and Solving the ILP model

Awake models the availability aware replication as an ILP represented by Equations 6.8-6.10. The only decision variable of this ILP is Y and constants

are n, R, and $AT[][]$.

$$\max \sum_{t=1}^{S} \sum_{i=1}^{n} Y_i \times AT_j[i][t] \quad \text{s.t.} \tag{6.8}$$

$$\sum_{i=1}^{n} Y_i = R \tag{6.9}$$

$$\forall i, 1 \leq i \leq n \quad Y_i \in \{0, 1\} \tag{6.10}$$

The objective function (Equation 6.8) The objective is to maximize the total availability of replicas over all the time slots. Availability of replicas in a time slot is defined as the summation of their availability probabilities from the availability table of the data owner. As represented by this equation, if node i is selected as a replica ($Y_i = 1$) it contributes to the availability of time slot t with its availability probability in that time slot ($AT_j[i][t]$). The summation over n represents the total availability of replicas during time slot t of *FPTI*. Subsequently, the summation over s represents the total availability of replicas over all the time slots. As the total availability of replicas during each time slot is non-negative, maximizing the total availability of replicas over all the time slots implicitly results in the maximum availability during each time slot as well.

The replication degree constraint (Equation 6.9) If node i is selected as a replica, Y_i is set to one, otherwise zero. Summing up the elements of vector Y hence results in the replication degree. Equation 6.9 shows the constraint on the replication degree.

The authorized values for output variable (Equation 6.10) This constraint represents that each element of vector Y should be either one or zero which shows whether the corresponding node is selected as a replica or not, respectively.

After Awake models the availability aware replication as an ILP, it solves the LP relaxation of ILP and outsources the vector Y to the data owner.

At each time, a data requester finds the closest available replica by pinging them and queries that replica instead of querying the data owner directly.

6.4 Related Works

Table 6.2 shows a comparison of state-of-the-art decentralized availability-based replication strategies. As shown in this table, among the availability-based replication algorithms, Awake is the only availability aware one. Also, while the main objective of all the proactive solutions is to provide an average availability of replicas, Awake provides the maximum replication availability. Compared to the existing solutions, Awake is the only algorithm that works independently of the underlying system's churn model. All of the replication algorithms listed in the table are decentralized. Considering the behavior feature, an algorithm is reactive, if it replicates regardless of availability behavior of nodes and reacts to a replica's failure by placing a new replica. Likewise, an algorithm is proactive, if it replicates based on the availability behavior of nodes. A replication algorithm is an availability aware if it places the replicas such that the maximum availability of replicas during each TS is achieved. All the algorithms are dynamic in the sense that a data owner can replicate at any time.

6.4.1 Reactive Replication

Reactive replication algorithms replicate regardless of availability patterns of the nodes. They react to a replica failure by placing a new replica. In this class of algorithms, after an initial replication is done, the replicas are periodically probed, for example, every 12 hours. If a replica does not answer the probe message in a certain while, it is presumed to be failed. Failed replicas are substituted by newly placed ones [72]. Probing of replicas is done by the data owner, data requesters or other replicas.

For a large number of replicas or small probing periods, probing all the replicas is inefficient. To resolve this problem while preserving the fairness, at each probing time a subset of replicas is chosen uniformly at random and

Strategy	Behavior	Availability Aware	Churn Model Independent	Objective
Probing [72, 87, 98–100]	Reactive	No	No	Resolving Failures
Randomized [1]	Proactive	No	No	Average Availability
Cluster-Based [4–7]	Proactive	No	No	Average Availability
Correlation-Based [8]	Proactive	No	No	Average Availability
Awake	**Proactive**	**Yes**	**Yes**	**Maximum Availability**

Table 6.2: Comparison of various availability-based replication strategies

probed [87, 98–100].

6.4.2 Proactive Replication

The goal of a proactive replication algorithm is to provide an average availability of replicas for a long period of time. For example, a proactive replication algorithm may provide an average availability of 2 replicas at each hour for 3 months. A proactive replication algorithm is supposed to be executed periodically to refresh the replica selection based on the updated availability pattern of nodes up to that period. In the previous example, the proactive replication algorithm is expected to be executed every 3 months. Likewise, when the availability behavior of replicas changes negatively and degrades the system performance, the proactive replication is supposed to be executed based on the new availability behavior of nodes. There are several proactive replication algorithms that depend on a specific churn behavior of the system:

6.4.2.1 Randomized Replication [1, 2]

In a randomized replication algorithm, a number of nodes denoted by the replication degree are selected as replicas uniformly at random. Randomized replications perform well in the highly available systems where the average availability of nodes is high. Choosing replicas uniformly at random hence results in an acceptable expected number of available replicas.

6.4.2.2 Cluster-Based Replication [3–7]

A cluster-based replication algorithm divides the nodes based on their common features such as time zone, load, and query rate into a set of cliques. The algorithm then distributes the replication degree among the cliques considering their availability pattern. Cluster-based replication algorithms perform well in high and moderate available systems.

6.4.2.3 Correlation-Based Replication [3, 8]

Correlation is defined as the similarity between the availability patterns of the nodes. The more two nodes are correlated with each other, the more similarity they have in their availability behavior. In other words, for a pair of highly correlated nodes, whenever one node is available, with a high probability the other node would be available as well. Similarly, two nodes that have reverse availability patterns with respect to each other, are called anti-correlated nodes. For a pair of anti-correlated nodes, when one node is unavailable, with a high probability, the other one is available. The correlation between two nodes is defined as the dot product of their availability vectors. The higher the dot product is, the higher two nodes are correlated with each other. A zero dot product shows a pair of anti-correlated nodes.

The goal of a correlation-based replication algorithm is to provide k-availability of replicas in the system. k-availability is defined as providing the availability of at least k replicas at any time in the system. To achieve this goal, a correlation-based replication algorithm replicates on k pairs of anti-correlated nodes. In this way, totally $2 \times k$ replicas are selected. Considering a pair of anti-correlated replicas, when one replica is offline, the other one is available with a very high probability. For each pair of anti-correlated replicas, hence, always one replica is available. Having k pairs of anti-correlated replicas results in the availability of at least k replicas at any time. However, finding k pairs of anti-correlated nodes may not be always feasible. Thus, empirically, a correlation-based replication algorithm selects k pairs of nodes with the minimum correlation as a pair of replicas. For this reason, the correlation-based replication algorithms are merely suitable for

the low available systems which experience a very low pairwise correlation of nodes.

6.4.3 Algorithms used for comparison

Followings are the implementation details of the algorithms that are selected for the sake of comparison with Awake.

6.4.3.1 Randomized Replication

The data owner pings nodes repeatedly at random until it finds as many as available nodes as the replication degree and replicates on them [1].

6.4.3.2 Cluster-Based Replication

The data owner performs a k-mean clustering [110] of the nodes based on the availability of the nodes, which is the second norm of their availability vector where k is equal to the replication degree (i.e., R). After the clustering is done, the replication degree is distributed among the clusters based on their availability. The availability of each cluster is defined as the second norm of the average availability vectors of nodes inside that cluster. Based on the assigned replication degree to each cluster, the replication is done on the most available nodes of that cluster [4–7].

6.4.3.3 Correlation-Based Replication

Considering the replication degree as R, the data owner aims to find $\frac{R}{2}$ pairs of nodes with the minimum correlation. Considering just the pairs with minimum correlation may fall the replication into a pitfall. Two nodes that are not available at all and hence have an availability vector of all zero have the minimum correlation respect to each other. However, they are the worst candidates for replication. To avoid falling into this pitfall, $\frac{R}{2}$ pairs of replicas are initialized with the $\frac{R}{2}$ most available nodes. After this initialization, each pair has exactly one node. Let's call that node the premier of the pair. Next,

to complete each pair, the most available node that has the least correlation with the premier of that pair is selected.

6.5 Simulation Setup

To simulate and evaluate the decentralized availability-based replication algorithms we extended SkipSim [38,75] by enabling it to simulate the availability-based replication algorithms. Table 6.3 represents a comparison between the versions of SkipSim before and after our extensions. The previous version of SkipSim was capable of generating static simulations i.e., the simulations where the topology is fixed and does not change. As the whole SkipSim architecture was designed in a static oriented manner, adding time feature to the simulations and making them dynamic was the most challenging part. We embedded the high, moderate and low available churn models into the SkipSim. In the extended version of SkipSim, nodes arrive and depart based on the selected churn model.

We implemented Awake as well as randomized replication, cluster-based replication, and correlation-based replication in the SkipSim. This package of the algorithms is called the dynamic replication package. By the extensions, SkipSim is able to evaluate the performance of availability-based replication algorithms from the scalability, average number of available replicas, replication time and learning factor points of view. We experimented with multiple learning factors and determined $\beta = 0.5$ as the best for all the algorithms.

For each simulation setup, we generated 100 random topologies. *FPTI* and *TS* were set to a day and an hour, respectively. Each topology was simulated for a lifetime of 3 months. A simulation step corresponds to an hour. At $t = 0$ all the nodes are available in the system. As the time proceeds, nodes depart from or come back to the system based on the session length and downtime distributions of the system's churn model. In each simulation setup, the replication algorithm is executed at the end of the second day.

For each topology, at each hour, a number of search transactions are initiated between the nodes. At hour t, the number of search transactions is determined as the binomial coefficient of the number of available nodes

Feature	Before extensions	After extensions
Static Simulations	✓	✓
Identifier Assignments	✓	✓
Identifier Assignment Evaluation	✓	✓
Static Replication	✓	✓
Static Replication Evaluation	✓	✓
Churn Models	✗	✓
Dynamic Simulations	✗	✓
Dynamic Replication	✗	✓
Dynamic Replication Evaluation	✗	✓

Table 6.3: A comparison of SkipSim features before and after our extensions.

at t, and 2. As shown in Equation 6.11, number of transactions at hour t represented by n_{trans_t} is computed as the binomial coefficient of number of available nodes at hour t (represented by $n_{available_t}$) and 2. For a search transaction done at hour t, the search initiator and search target are selected from the set of available nodes at hour t and set of all the registered nodes to the system, respectively. For each transaction, availability table of the nodes on the transaction path including the source and destination are updated by the aggregated piggybacked availability vectors. At the replication time, a data owner is selected uniformly at random that executes the replication algorithm.

$$n_{trans_t} = \binom{n_{available_t}}{2} \tag{6.11}$$

We simulated each algorithm with a system size of 128 nodes under the churn models presented in Table 6.1. Also, the scalability of each algorithm was evaluated under the moderate available churn model with system sizes of 128, 256, 512 and 1024 nodes. Additionally, following the moderate available churn model, running time, space consumption, and communication overhead of Awake were evaluated with the system sizes of 128, 256, 512, 1024, 2048, 4096, and 8192 nodes.

Correctness of Churn Modeling: To verify the correctness of our implementations Figures 6.2 and 6.3 show the extracted probability distri-

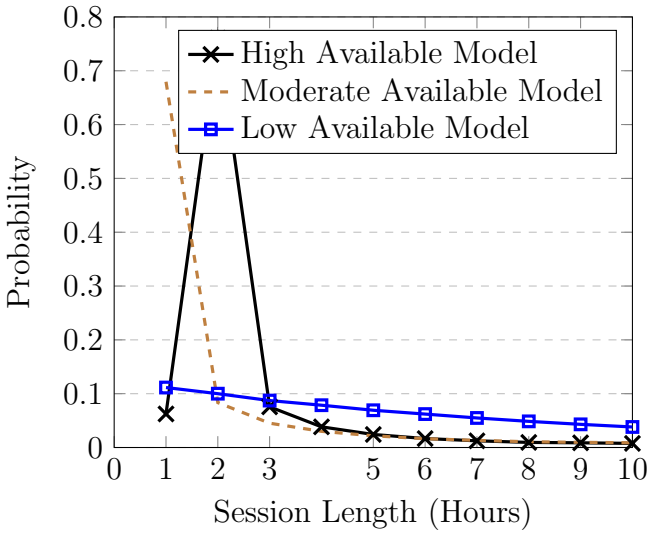

Figure 6.2: Extracted churn model session lengths probability distributions from SkipSim

butions of the session lengths and down times of churn models from SkipSim, respectively. Table 6.4 represents the corresponding average values of Figures 6.2 and 6.3. Comparing the empirical average values (Table 6.4) with the theoretical average values (Table 6.1) results in an average churn model implementation error of about 3%. For solving the linear programming models, we used the lpsolve 5.5 [94].

Churn Model	Average SL (Hour)	Average DT (Hour)
High Available	2.03	1.07
Moderate Available	1.11	1.01
Low Available	8.37	58.41

Table 6.4: Extracted average session length and downtime values from the SkipSim. SL and DT correspond to the session length and downtime distributions, respectively.

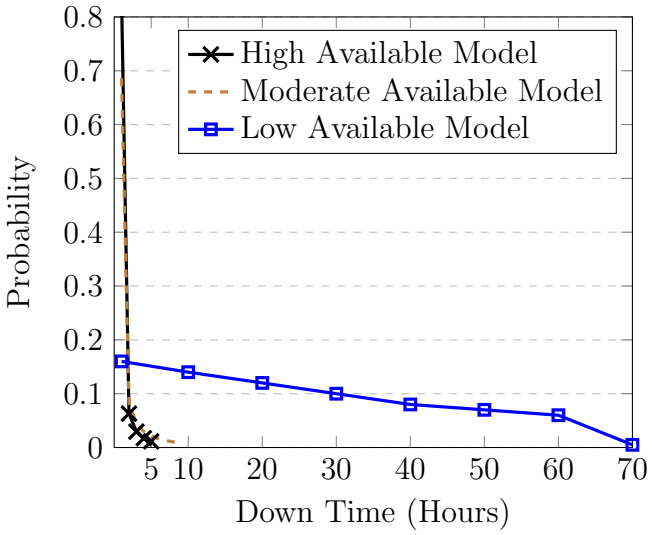

Figure 6.3: Extracted churn model down times probability istributions from SkipSim

6.6 Performance Results

6.6.1 Learning Factor Effect

Figure 6.4 shows the effect of the learning factor, β, that was presented in Equation 6.1, on the average available number of replicas. In this experiment, the replication degree was fixed to 4 and the algorithms were simulated under the moderate available churn model. As shown in this figure, as β varies from 0.1 to 0.9, Awake outperforms the cluster based replication as the best existing solution with the gain of about **20%**. Since all of the replication algorithms work at their best with $\beta = 0.5$, in all simulations the learning factor was set to 0.5.

6.6.2 Average Availability of Replicas

The average availability of replicas is defined as the average number of available replicas at each slot. The average is taken over the number of time slots from the replication time to the end of simulation time. Figures 6.5 and 6.6 represent the average availability of replication algorithms under the high and moderate available churn models. Compared to the cluster-based replic-

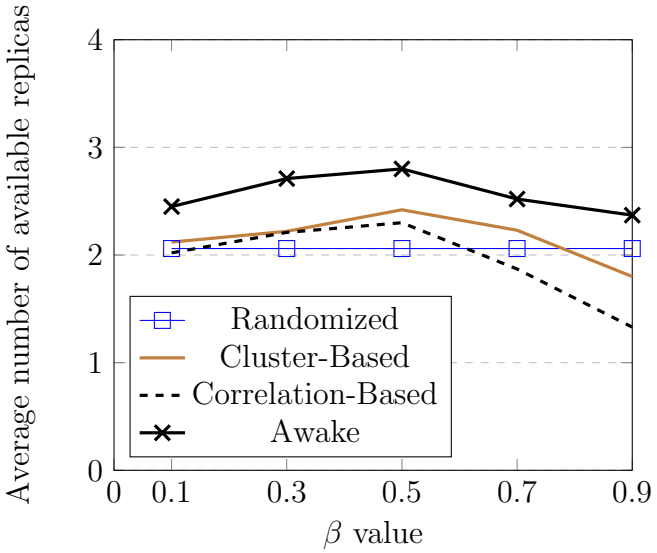

Figure 6.4: Effect of the learning factor, β, on the performance of replication algorithms. Awake performs about 20% better compared to the best existing solution.

ation algorithm as the best existing solution under these two models, Awake improves the average availability of replicas about **24%** and **14%** under the high and moderate available churn models, respectively. In the high available churn model, the correlation between nodes is high. Therefore, correlations in the pairs of nodes that are selected by the correlation-based replication algorithm are too far from zero. This causes the correlation-based replication to perform as the worst algorithm under the high available churn model. This problem is ameliorated in the moderate available churn model as the correlation between the nodes degrades, and the correlation-based replication beats the randomized replication.

Figures 6.7 shows the average availability of replication algorithms under the low available churn model. Under this model, the correlation between nodes reaches its minimum and many suitable replication candidate pairs with close to zero correlation emerge. In this case, the correlation-based replication algorithm conquers the cluster-based replication and becomes the best among the existing solutions. However, under the low available churn model, Awake outperforms the correlation-based replication with the gain of about **26%**.

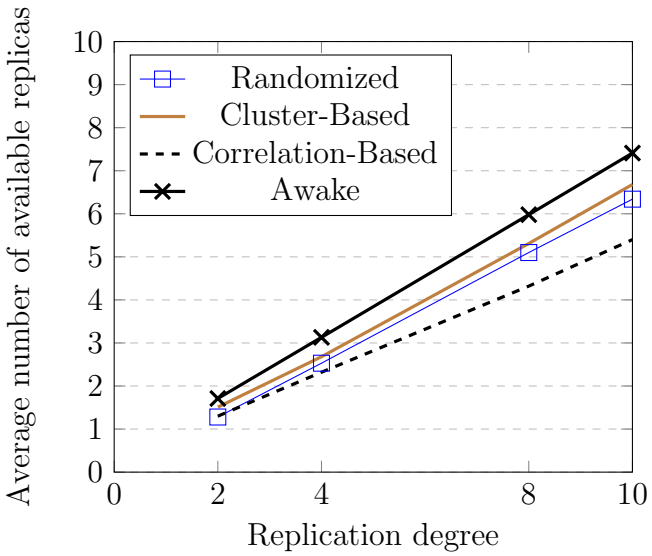

Figure 6.5: Average number of available replicas at each hour vs replication degree under the high available churn model. Compared to the best existing solutions Awake performs about 24% better under the high available churn model.

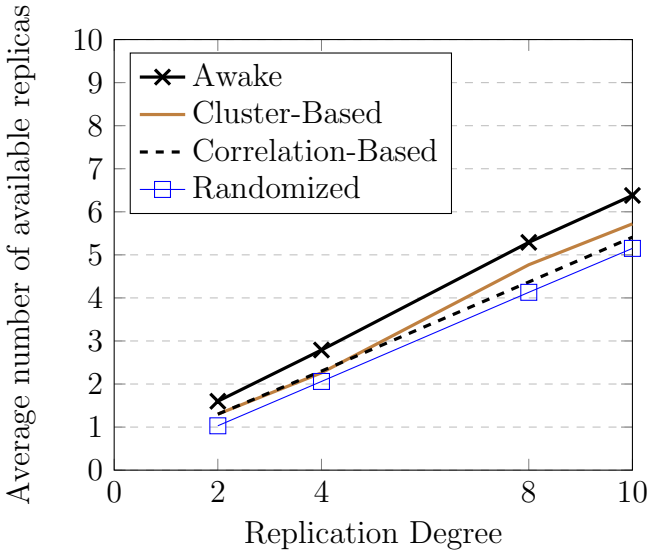

Figure 6.6: Average number of available replicas at each hour vs replication degree under the moderate available churn model. Compared to the best existing solutions Awake performs about 14% better under the moderate available churn model.

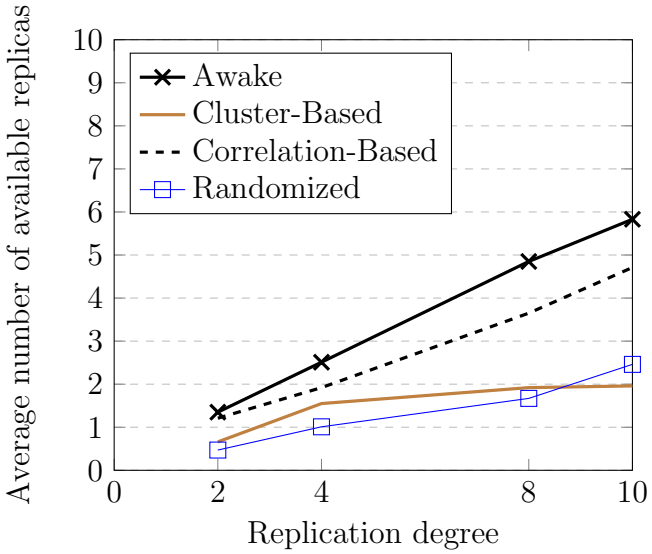

Figure 6.7: Average number of available replicas at each hour vs replication degree under the low available churn model. Compared to the best existing solutions Awake performs about 26% better under the low available churn model.

Considering the low available churn model, as the replication degree increases, cluster-based replication assigns replication quota to the low available cliques with higher probability. The replicas from low available cliques do not play a significant role in the performance of the cluster-based replication algorithm. This causes the performance of the cluster-based replication to reach a steady state and even is beaten by the randomized replication when the replication degree goes beyond 8 replicas as illustrated in Figure 6.7. This trend of randomized and cluster-based replication algorithms was verified with the higher replication degrees as well.

6.6.3 Scalability

We examined all the replication algorithms under the moderate available churn model with the system sizes of 128, 256, 512, and 1024 nodes. In all of the simulation setups, the replication degree was fixed to 4 replicas. Figure 6.8 shows the scalability behavior of the replication algorithms under the moderate available churn model when the replication degree is fixed and

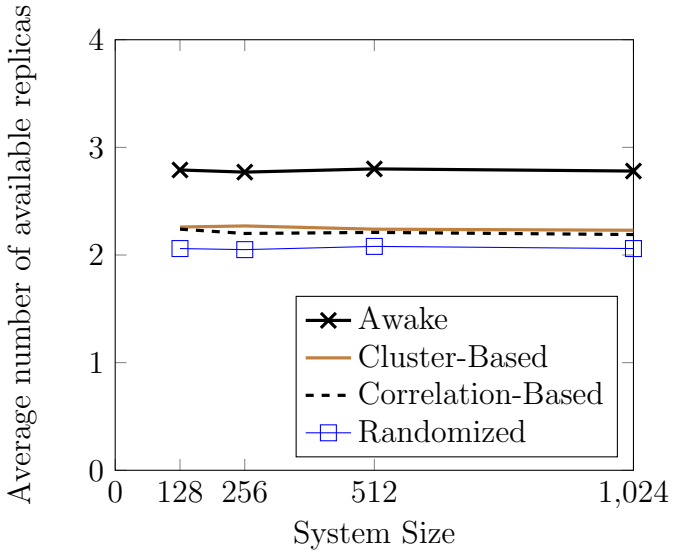

Figure 6.8: Scalability of replication algorithms over different system sizes. Replication degree is set to 4. Awake performs about 23% better compared to the best previous work.

system size is scaled up. As shown in this figure, for a fixed replication degree, the performance of all algorithms including Awake is independent of the system size. The same behavior is observed with the high and low available churn models. Besides, the scalability analysis confirms that the results obtained in the 128-node scenarios are consistent with larger system sizes. Under the moderate available churn model and fixed replication degree of 4 replicas, on the average Awake outperforms the cluster-based replication algorithm as the best-decentralized counterpart with the gain of about **23%**, where the average is taken over all the system sizes.

6.6.4 Replication Time

A later replication time gives more chance to the data owner to learn about the availability behavior of the system. For this sake, we examined all the algorithms with the replication times of a day, two days, a week, and a month under the moderate available churn model in a system with 128 nodes and replication degree of 4 replicas. On average, about 1702 random transactions were initiated per *TS*. As shown in Figure 6.9, a replication time around two

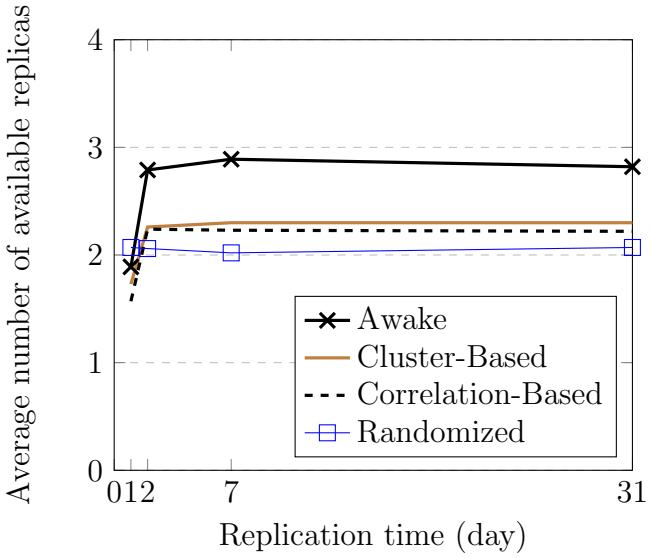

Figure 6.9: The effect of replication time on the performance of replication algorithms. Awake performs about 21% better compared to the best existing solution.

System Capacities (Nodes)	128	256	512	1024	2048	4096	8192
Running Time (Second)	0.12	0.26	0.49	1.08	2.13	4.61	9.01
Space Consumption (KB)	3	6	12	24	48	96	192
Communication Overhead (KB)	0.37	0.43	0.48	0.57	0.59	0.64	0.71

Table 6.5: Running time, space usage and communication overhead of Awake as the system size is scaled up. Replication degree is set to 4. Number of simulation runs = 100 times. Churn model = Moderate available model.

days -when each time slot has been updated at least once- is enough for all the replication algorithms to operate at their bests. A replication time longer than two days almost has the same effect on the performance of replication algorithms as a two days replication time does. The same behavior was observed with the high and low available churn models. Considering all the replication times, in comparison to the cluster-based replication as the best existing solution, on the average Awake performs around **21%** better.

6.6.5 Running Time

Table 6.5 shows the average running time of Awake in seconds under the moderate available churn model as the system size is scaled up. The average was taken over 100 simulation runs. The running times were measured with the system sizes of 128, 256, 512, 2048, 4096 and 8192 nodes. The replication degree in all of the simulations was fixed to 4 replicas. We also measured the running time of Awake under the high and low available churn models and similar results were obtained. As shown in Table 6.5, as the size of the system is scaled up, the running time of Awake increases linearly. At our simulation extreme with 8192 nodes system capacity, an execution of Awake takes around 9 seconds. Following the trend of the running time, in a system with 1 million nodes, Awake is supposed to select the replicas in about 17 minutes.

6.6.6 Space Consumption

In a system with n nodes and availability vector of size s, the availability table of a node contains $n \times s$ probability values. Since s is constant, the space complexity of Awake is $O(n)$ and hence is linearly dependent on the number of nodes registered to the system. For instance, by representing the probability values with integers between 0 to 100, and setting TS to 24, the value of s is equal to 24 bytes. Hence, in a system with 1 million nodes, each node needs about 24 megabytes to store its availability table which is a small memory overhead.

6.6.7 Communication Complexity

In order to employ Awake on a structured P2P cloud storage, nodes need to piggyback their availability vector on the messages they route or initiate. The communication complexity is, therefore, proportional to the number of the nodes that the message is traversed on average during the routing procedure. For instance, in a DHT-based [87] cloud storage with the capacity of n nodes, the number of nodes that a message is traversed during the routing procedure

is $O(\log n)$. By employing Awake on a DHT-based cloud storage, the number of piggybacked availability vectors on a message during its routing procedure (i.e, communication complexity) is $O(\log n)$.

We employed a Skip Graph [31] as the underlying routing infrastructure for our P2P cloud storage. In a Skip Graph with n nodes, a message traverses $O(\log n)$ nodes on average while it is routed from source to the destination. Hence, the communication overhead of employing Awake in such system is $O(s \times \log n)$ where s is the size of nodes' availability vector. Since s is constant, the communication overhead is simplified to $O(\log n)$ which is asymptotically the same as the number of nodes that a message is traversed on average during the routing procedure. Based on our simulation setup the communication overhead of Awake is **logarithmically** dependent on the system capacity. The average communication overheads of Awake per message over the system sizes are represented in Table 6.5. At our simulation extreme with 8192 nodes system capacity, an execution of Awake costs the communication overhead of about 0.71 kilobytes per message. Following the same trend, considering a system with 1 million nodes where the TS is 24, and by representing probability values with byte, Awake is presumed to charge the communication overhead of about 480 kilobytes per message on average.

6.7 Conclusions

To maximize the availability of replicas in the structured P2P cloud storages, we propose a novel availability aware replication algorithm. In order to select the replicas, our fully decentralized availability aware replication algorithm, Awake, employs the availability vectors of the nodes. The availability vectors are aggregated in a fully decentralized manner. The message overhead of this decentralized aggregation is asymptotically the same as the communication complexity of the structured P2P cloud storage that Awake is applied on. Awake has a linear space complexity in the number of registered nodes to the system. To the best of our knowledge, Awake is the only availability aware replication algorithm for the structured P2P cloud storages that maximizes

the availability of replicas regardless of the underlying churn model of the system.

We extended the SkipSim simulator and simulated the state-of-the-art decentralized availability-based replication algorithms. In comparison to the best known decentralized existing solutions, on the average, Awake improves the availability of replicas with the gain of around **21%**. From the scalability point of view, as the system size is scaled up, Awake shows the same performance regardless of scaling. Awake only needs two days as the learning phase to learn the availability behavior of the system. In a system with **1 million** nodes, Awake is expected to select the replicas in about **17 minutes**, charge the space consumption of about **25 megabytes** on each node, and charge the communication overhead of about **480 kilobytes** on each message.

Chapter 7

ELATS: Energy and Locality Aware Aggregation Tree for Skip Graph

7.1 Introduction

Skip Graph [31] is a prefix-based routing DHT in which nodes are connected to each other based on their identifiers' common prefixes. Each node in a Skip Graph knows a few other nodes of the system and keeps their information as $\{identifier, address\}$ pairs in a lookup table. Using the Skip Graph overlay, each node can efficiently find the address of any node by conducting a search for its identifier [36, 39].

Fast searching [31], low traffic [13], high guarantee [13], scalability, and load balancing features of Skip Graph make it a suitable routing approach for P2P cloud services [34, 36, 39–41, 111]. Likewise, Skip Graph can be used as an efficient underlying structure in applications ranging from distributed storage systems [43], to online social networks [112], and search engines [46].

Nodes in P2P systems employ data aggregation to obtain an identical view about a certain parameter of the system, which is referred to as the parameter of interest. Examples for parameter of interest in P2P systems are size of system [113], average available bandwidth, the reputation of a

data object [114], and a local clock [115]. Data aggregation is defined as repeatedly obtaining data about the parameter of interest from several peers, compressing or reducing the size of the obtained data to achieve a smaller sized aggregated data, and forwarding the aggregated data to other peers of the system, which causes all peers converge to a similar or identical view of the parameter of interest.

Skip Graph suffers from the lack of a data aggregation capability. The traditional distributed data aggregation solutions are based on forwarding aggregated data to a subset of nodes (i.e. gossiping [116]), passing the aggregated data one by one (i.e. random walk [117]), and forwarding the aggregated data to all nodes (i.e. flooding). The traditional solutions, with the drawback of high message overhead, result in high aggregation latency and response time. Likewise, discarding the constraint on the number of exchanged messages as well as the size of the messages, the traditional solutions cause an additional energy cost for the system.

The existing DHT-based solutions aim at constructing an aggregation tree on top of the underlying routing infrastructure in which the aggregation is done in two phases: request and reply. In the request phase, the root node initiates the aggregation request message and sends it to its children. The aggregation request message is forwarded in a top-down manner by each node to its corresponding children set until it reaches the leaves. Once a leaf node receives the aggregation request, it starts the reply phase by sending its state of the parameter of interest to its parent. Each node combines its parameter's state with all aggregated states obtained from its children and sends it to its parent. The reply phase continues in a bottom-up manner until it reaches the root node. The root node generates the final result of aggregation by combining the aggregated states obtained from its children. If all nodes need to be notified about the aggregation result, the root node sends the result in a top-down manner similar to the request phase.

The existing DHT-based aggregation trees are constructed based on the specific features of the system. So far, no aggregation tree is constructed specifically for the Skip Graph. Likewise, the existing aggregation tree schemes for other prefix-based DHTs have mainly focus on response time and do not

consider the energy cost enforced by aggregation tree to the peers.

In this study, we propose **ELATS which is the first energy and locality aware aggregation tree for Skip Graph**. We define the locality awareness of an aggregation tree as minimizing the latency on the path between the root and the leaves. This improves the performance of the aggregation tree in term of response time. We define the energy awareness as minimizing the average energy cost of the non-leaf nodes in the aggregation tree.

The original contributions of this chapter are as follows:

- We propose the first energy and locality aware aggregation tree algorithm, namely ELATS, for the Skip Graph. ELATS employs a fully decentralized and recursive approach while constructing the aggregation tree.
- We define the energy cost model of a generic aggregation tree, and extend the Skip Graph simulator SkipSim [38] to support the aggregation tree models.
- We redesign the proposed aggregation tree algorithms for other DHTs to be applicable on the Skip Graph, model them in the SkipSim and compare their performance with ELATS.
- Our simulation results show that compared to the best existing solutions, which are either locality aware or energy aware, *ELATS* provides both the energy and locality awareness, and improves the aggregation latency with gains of about **8%**.

The rest of this chapter is organized as follows. In Section 7.2, we present the energy cost model of a general aggregation tree. In Section 7.3, our solution ELATS is presented. We review the related works in Section 7.4. The simulation setup is described in Section 7.5. In Section 7.6, we evaluate ELATS's performance in comparison to the best known prefix-based existing solutions. We conclude in Section section:elats˙conclusion.

7.2 Aggregation Tree: Energy Cost Model

In our view of an aggregation tree, a node consumes energy while it exchanges request/reply messages, or aggregates the obtained data from its children [118]. All the aggregated data have the identical size, which implies identical size of all the exchanged messages. We assume that exchanging a single message between two nodes in the aggregation tree charges an energy cost of E units on each of them.

Energy Cost of Request Phase: The root node r initiates a request phase by sending an aggregation request to its children. Equation 7.1 indicates the energy cost of the root node in the request phase, where the children set of r is denoted by CS_r.

$$E_{request}(r) = |CS_r| \times E \tag{7.1}$$

During the request phase, an intermediate node i receives the request message from its parent, and forwards the request to its children. Equation 7.2 shows the energy cost of intermediate node i during the request phase.

$$E_{request}(i) = E + |CS_i| \times E \tag{7.2}$$

Equation 7.3 represents the average energy cost of non-leaf nodes in the aggregation tree T during the request phase, denoted by $E_{request}$. I_T corresponds to the set of all intermediate nodes of T.

$$E_{request} = \frac{E_{request}(r) + \sum_{i \in I_T} E_{request}(i)}{|I_T| + 1} \tag{7.3}$$

Energy Cost of Reply Phase: During the reply phase, an intermediate node i receives the aggregated states of nodes covered by sub-tree rooted at node i, adds its state of the parameter of interest to the set of obtained aggregated states, combines the states in the set, and sends the combined aggregated states to its parent. Equation 7.4 shows the energy cost of an intermediate node i during the reply phase. U denotes the energy cost of combining one pair of aggregated states. We assume that the combination is

done linearly. The first two aggregated states are combined, and the result is iteratively combined with the next aggregated states until no aggregated state remains. Doing so, combining $|CS_i|$ aggregated states as well as node's i state of parameter of interest is done in $|CS_i|$ steps, which costs the energy cost of $|CS_i| \times U$.

$$E_{reply}(i) = (|CS_i| \times E) + (|CS_i| \times U) + E \tag{7.4}$$

In the last step of the reply phase, the root node, r, receives the aggregated states of all nodes in the subtrees rooted at its children, adds its state of the parameter of interest to the set of obtained aggregated states, and combines the aggregated states. This results in the aggregated state of all nodes in the aggregation tree. Equation 7.5 shows the energy cost of the root node during the reply phase.

$$E_{reply}(r) = (|CS_r| \times E) + (|CS_r| \times U) \tag{7.5}$$

Equation 7.6 shows the average energy cost of non-leaf nodes in the aggregation tree T in the reply phase, denoted by E_{reply}.

$$E_{reply} = \frac{E_{reply}(r) + \sum_{i \in I_T} E_{reply}(i)}{|I_T| + 1} \tag{7.6}$$

7.3 Energy and Locality Aware Aggregation Tree for Skip Graph (ELATS)

In a Skip Graph, each node has two identifiers, a non-negative integer called numerical ID, and a binary string called name ID. In a Skip Graph with n nodes the search for name ID operation returns *address* the owns the most similar name ID to the search target by traversing $O(\log n)$. The similarity of a pair of name IDs is defined as the length of their common prefix. *ELATS* works on the top of a locality aware Skip Graph in which the pairwise laten-

cies of nodes correspond to the length of their name IDs' common prefix [39]. A longer prefix corresponds to the lower latency. In our view of an aggregation tree, all nodes in the Skip Graph are spanned by the aggregation tree. The node that starts the aggregation process by invoking *ELATS* is named the initiator, which becomes the root of the resulted aggregation tree. We define a sub-domain as the set of name IDs with a common prefix. The common prefix represents a sub-domain called its sub-problem. For example, in a Skip Graph with name ID size of 4 bits, corresponding sub-domain to the 01 sub-problem is $\{0100, 0101, 0110, 0111\}$.

ELATS is a recursive distributed algorithm where each recursion is executed on a single node. The node who executes a recursion of *ELATS* is called executor. *ELATS* is shown by Algorithm 2. As the inputs *ELATS* receives the address of the *executor* node as a pointer, *subProblem* that is a binary string defines the sub-problem, energy cost of current path denoted by E_c, latency of current path denoted by L_c, and the number of intermediate nodes on the current path denoted by *inernum*. The current path is defined as the path between the executor and root of aggregation tree. In *ELATS*, the latency of the current path is obtained by summing up the common prefix lengths of consecutive nodes on the path. Likewise, the energy cost current path corresponds to the average energy cost of non-leaf nodes on the path between the executor and root of aggregation tree.

ELATS first extracts the name ID of the executor denoted by $nameID$ (Algorithm 2, line 1). The corresponding sub-domain of the *subProblem* denoted by *subDomain* consists of the set of name IDs which start with the *subProblem* prefix. All name IDs in *subDomain* are as long as the executor's name ID denoted by $nameID$. Likewise, all name IDs in *subDomain* start with *subProblem* prefix of size $|subProblem|$ bits, and they differ only in their next $|nameID| - |subProblem|$ bits. Hence, size of *subDomain* that corresponds to its number of name IDs is obtained as $2^{|nameID|-|subProblem|}$ (line 2). A *subDomain* of size 1 represents the situation where $nameID$ is the only member of *subDomain*. In such a case, *ELATS* terminates since no parent-child relationship can be defined for one node (lines 3 and 4).

The core of *ELATS* is to decide on executor to be either the parent

of all the *subDomain* name IDs, or the parent at most half of them. We call the former as covering all case, and the latter as covering half case. In covering half case, only the name IDs have more than $|subProblem|$ bits common prefix with $nameID$ can become the children of executor. Moving from $|subProblem|$ to $|subProblem|+1$ ignores half of the *subDomain* name IDs i.e., the ones who have exactly $|subProblem|$ bits common prefix. Since Skip Graph is locality aware, the remaining name IDs -which have at least $|subProblem|+1$ common prefix length with the $nameID$- constitute the expected lower latency half portion of *subDomain*. To make this decision, *ELATS* computes the affected energy cost and latency of current path affected by each of cases (line 5). The energy cost of current branch of aggregation tree in the covering all and covering half cases are illustrated by E_a and E_h, respectively. Each of these energy costs are obtained as the sum of the corresponding $E_{request}$ and E_{reply} obtained from Equations 7.3 and 7.6, respectively. In both equations $|I_T| = inernum$.

Equation 7.7 represents the latency of the current branch in the covering all case denoted by L_a. The latency of current branch is defined as the minimum common prefix length of $nameID$ with the name IDs in *subDomain*, which is equal to $|subProblem|$. This minimum common prefix corresponds to the maximum latency between the executor and nodes of *subDomain*. If the *executor* becomes the parent of all *subDomain* name IDs, all of the name IDs in *subDomain* become leaves of the aggregation tree. The latency of the current branch is hence increased by the maximum latency between the executor and name IDs of *subDomain* as shown in Equation 7.7. In the covering half case, since the minimum common prefix length of $nameID$ and the new sub-domain is $|subProblem|+1$, the corresponding latency denoted by L_h is obtained in a similar way shown by Equation 7.8.

$$L_a = L_c + |nameID| - |subProblem| \tag{7.7}$$

$$L_h = L_c + |nameID| - (|subProblem| + 1) \tag{7.8}$$

After computing the energy cost and latency of current branch of aggreg-

Algorithm 2: ELATS

Input: pointer *executor*, String *subProblem*, double E_c, integer L_c, int *inernum*

```
1  nameID = executor.getNameID();
2  subDomainSize = 2^(|nameID|−|subProblem|);
3  if subDomainSize == 1 then
4  |  terminate;
5  computing L_a, L_h, E_a, E_h;
6  imp_L = improvement(L_a, L_h);
7  imp_E = improvement(E_a, E_h);
8  if imp_L < e or imp_E < e then
9  |  avnodes = extend(subDomain);
10 |  for i ∈ avnodes do
11 |  |  parent(i) = executor;
12 |  |  children(executor).add(i);
13 else
14 |  if commonBits(subProblem +'0', nameID) == |subProblem|
   |   then
15 |  |  α = searchByNameID(subProblem +'0');
16 |  |  if α ≠ NULL then
17 |  |  |  parent(α) = executor;
18 |  |  |  children(executor).add(α);
19 |  |  |  α.ELATS(α, subProblem +'0', E_h , L_h, inernum + 1);
20 |  |  ELATS(executor, subProblem +'1', E_h , L_h, inernum + 1);
21 |  else
22 |  |  α = searchByNameID(subProblem +'1');
23 |  |  if α ≠ NULL then
24 |  |  |  parent(α) = executor;
25 |  |  |  children(executor).add(α);
26 |  |  |  α.ELATS(α, subProblem +'1', E_h, L_h, inernum);
27 |  |  ELATS(executor, subProblem +'0', E_h , L_h, inernum);
```

ation tree in both covering all and covering half cases, *ELATS* computes the latency and energy cost improvements of covering all compared to covering half cases denoted by imp_L and imp_E, respectively (lines 6 and 7). imp_L and imp_E are floating point values between 0 and 1, and provided by the *improvement* function. On receiving two input arguments, the *improvement* function divides their absolute value of difference over the greatest input value, and outputs the result.

A smaller than the e value of imp_L or imp_E conveys that going deeper than the current depth does not provide lower latency or better energy cost. This makes *ELATS* to choose the covering all case. e is a close to zero local constant of executor determined based on its properties like bandwidth and computational power. To perform covering all case, *ELATS* first finds all available name IDs in *subDomain*. A name ID is available if it has been assigned to an existing node in the Skip Graph. To find the available name IDs, *ELATS* employs the *extend* function that applies a *searchForNameID* for each of the name IDs in *subDomain*, and checks their existence. The set of all available nodes of *subDomain* is denoted by *avnodes*. *ELATS* adds each node in *avnodes* to the children set of the executor, as well as makes the executor as the parent of all nodes in *avnodes* (lines 8-12).

Large improvements in both imp_L and imp_E promise to meet better energy cost and lower latency as going deeper. In this case, the executor node needs to divide the *subDomain* into two smaller sub-domains and choose covering half case (line 13). $nameID$ starts with the *subProblem* prefix of size $|subProblem|$. The $|subProblem| + 1^{th}$ bit of $nameID$ is either 0 or 1. If the $|subProblem| + 1^{th}$ bit of $nameID$ is equal to 1, the common prefix length of $subProblem +' 0'$ and $nameID$ computed by *commonBits* function is equal to $|subProblem|$ (line 14). For example, if $nameID = 01010011000$ and $subProblem = 0101001$, then $subProblem +' 0' = 01010010$, and the common prefix of $subProblem +' 0'$ and $nameID$ is 0101001 that is equal to the *subProblem* itself. In this case, the executor node should deliver the aggregation tree covering the sub-domain corresponds to $subProblem +' 0'$ to another node, which has the name ID prefix of $subProblem +' 0'$. The executor node performs a $searchForNameID$ to find address of node α has

the $subProblem +' 0'$ prefix in its name ID (line 15).

If node α exists, the executor node introduces itself as the parent of α and adds α to its children set (Algorithm 2, lines 16-18). Likewise, *ELATS* calls a fresh recursion of itself on the node α for the $subProblem +' 0'$. In the fresh recursion of *ELATS*, the current average energy cost of intermediate nodes on the path between the root of aggregation tree and node α is equal to the already computed E_h. Likewise, the latency of path from root node to α is equal to the already computed L_h. Since the executor is the parent of α in the aggregation tree, the number of intermediate nodes on the path between the root node and node α is one more than the number of intermediate nodes on the path between the root node and executor (line 19).

ELATS also calls a fresh recursion on the executor node for the $subProblem +' 1'$ and with the similar average energy cost and latency arguments. However, since executor node runs the fresh recursion, no intermediate node is added (line 20). Lines 21-27 show the similar procedure for the case where $|subProblem| + 1^{th}$ bit of the $nameID$ is equal to 0. Assuming that the initiator of aggregation tree is node β, the initial call to *ELATS* is $\beta.ELATS(\beta, NULL, 0, 0, 1)$.

Time Complexity: As shown by lines 2-4 of Algorithm 2, a sub-problem that is as long as the name ID size of Skip Graph terminates the recursion of *ELATS*. The total number *ELATS*'s recursions on a single node is therefore bounded by the name ID size of Skip Graph. In a Skip Graph with n nodes, the name ID size is $\lceil \log n \rceil$. Hence, *ELATS* enforces $O(\log n)$ recursions on each node of the Skip Graph. The recursions of *ELATS* are executed in parallel on different nodes.

7.4 Related Works

Distributed aggregation strategies are classified based on the underlying routing infrastructure into the unstructured and structured.

7.4.1 Unstructured Aggregation

Unstructured based aggregation algorithms are mainly applied to the unstructured P2P systems where there is no predefined topology. Each node knows only a finite subset of the network. The unstructured aggregation methods are flooding, random walk, gossip, and dominant set. In flooding, each node receives the aggregated result from other nodes, adds its state to the aggregated result, and broadcasts the new aggregated results to all nodes in the system. A single node starts a random walk [117, 119, 120] by sending the aggregated data to another node. Receiver adds its state to the aggregated data and forwards the result to the next node. This procedure continues sequentially in a one to one manner until it passes all nodes and reaches the initial node. A gossip aggregation [114, 121–123] is analogized to the spreading of an epidemic disease. It is similar to flooding, but instead of broadcasting to all known nodes, it forwards to a smaller subset. The Dominant Set [9, 118, 124] denoted by DS, is defined as a subgraph of the unstructured P2P graph such that each node is of the graph is either in the sub-graph or has a neighbor in the sub-graph. Neighbors of a node are the ones connected directly to that node in the underlying network. From the DS point of view, there are two types of nodes in the P2P system: DS-node and NDS-node. A node is called DS-node if it belongs to the DS, and NDS-nodes otherwise. DS nodes collect the data of NDS nodes they cover, exchange their aggregated data, compute the final aggregated information, and send the result back to the NDS nodes they cover. For example ProFID [121] provides a gossiping frequent item discovery in unstructured P2P systems. Each peer has a set of item types. Each node holds a certain frequency of each item type. The goal of the algorithm is to find the globally frequent items beyond a certain threshold. To do so, an aggregation takes place by means of gossiping. The basic idea for each node is to perform pairwise averaging with other nodes until all the nodes converge for all the item types.

7.4.2 Structured Aggregation

Structured-based aggregation algorithms are applied to the structured P2P routing infrastructures where there is a well-defined topology among the nodes, and nodes are able to search efficiently for each other by means of this underlying routing infrastructure.

7.4.2.1 Ring [9]

The very naive solution for structured aggregation is to use the underlying routing infrastructure and build a ring based on that. In ring-based aggregation, one node initiates an aggregation and routes to the other node. On receiving an aggregation, each intermediate node routes the aggregation in a way that it shapes a ring. The main difference between ring aggregation and the random walk is the underlying structure of the system which provides a deterministic flow in the ring. The disadvantages of applying a ring based aggregation are the single point of failure and time complexity. Failure or departure of a node while routes the aggregation message results in message loss as well as the failure of the whole aggregation query. With n nodes in the system, a ring aggregation is done in $O(n)$ cycles where each cycle is defined as the aggregation computation at a single node.

7.4.2.2 Hierarchical [9]

In hierarchical aggregation, a tree is defined on the top of the structured P2P system inspiring from the existing structure of the system. For instance, a parent function for the circular namespaces like Chord is proposed in [14] where each node covers all its predecessors. Given a node's identifier, the parent function returns back the parent's identifier for that node. However, the parent function is not applicable to the Skip Graph that does not follow a circular continuous namespace. The tree is constructed by defining a parent for each node. As soon as the tree is created and the root is determined, an aggregation request is made by the root in certain intervals. The aggregation request is delivered to children of root and they deliver to their children. This procedure continues until the aggregation request is received

by the leaves. Upon receiving an aggregation request, a leaf node responds back by a response message. The response messages are aggregated at each intermediate nodes from its children and the aggregated response message is delivered to the parent. This bottom-top approach continues until the root is reached. A parent function [14] for node a is the function P that satisfies the following properties:

$$P(r) = r \tag{7.9}$$

$$Distance(P^{i+1}(x), r) < Distance(P^{i}(x), r), \quad \forall i > 0 \tag{7.10}$$

$$P^{\infty}(x) = r, \quad \forall x \tag{7.11}$$

In Equation 7.9, r represents the root of tree. This equation states that the parent function evaluated at r results in r itself. Equation 7.10 denotes for a certain node a a higher power of its parent function results in a closer grandparent of r to root node. Finally, Equation 7.11 represents that infinite power of parent function for each node converges to the root node. [14] proves that if the function P exists for a P2P system, $P(x)$ represents the parent of node x. Likewise, there is a directed path from all nodes to the root node. The parent function was proposed for the circular continues namespaces like Chord [43], where each node covers all its predecessors, and it is not applicable to the Skip Graph which does not follow a circular namespace.

7.4.2.3 Breadth first search tree (BFS-tree) [10]

Applying a distributed BFS on the network's graph results in a spanning BFS-tree, which can be used as an aggregation tree. A BFS-tree guarantees the minimum number of intermediate nodes between each pair of nodes, which implicitly results in a shorter aggregation tree. A short aggregation tree helps saving energy by reducing the number of exchanged messages. However, the locality awareness of the resulted aggregation tree is completely ignored in the BFS-tree. For a Skip Graph with n nodes which each node has $O(\log n)$ neighbors, constructing a BFS-based tree causes the time complexity

and message complexity of $O(n)$ [125], which is inefficient to consider in large scale networks.

7.4.2.4 Shortest Path-based (SP-trees) [11]

An aggregation tree has two degrees of freedom: nodes' degree, and tree's height. It is claimed that the average nodes' degree in SP-based tree drops as the system size scales up, and converges to the degree of 3 [11]. Also, the degree of a node in this approach does not go beyond 8. An SP-tree provides locality awareness. However, limiting nodes' degree makes the aggregation tree to grow in height as the system size scales up. This results in increasing number of exchanged messages during the aggregation, which negatively affects the energy cost of the tree. The root node can be either selected randomly or among the nodes with the highest degree. A Higher degree of the root node increases the expected number of nodes added to the tree after each round of shortest path execution. Leaves also can be selected randomly, or among the nodes with the minimum degree. A lower degree of a node degrades its chance to be placed on a path and hence is better to be added manually by the algorithm. Although putting strict policies on the selection of root and leaves results in better performance of the tree construction, nevertheless, needs global state of the system and costs in high communication load to achieve in large scale systems.

7.4.2.5 Spanning broadcast tree [12–15]

In a spanning broadcast tree, each node broadcasts the received message to all its neighbors. The broadcasting procedure continues in a top-down manner until the message reaches the leaves. Leaf nodes are the ones without any neighbor have not received the broadcast message. A broadcast tree can be used as an aggregation tree by broadcasting the aggregation request from the root node. Each node broadcasts the request to its children, aggregates and replies their responses back to its parent.

7.4.2.6 Prefix-based tree [12, 16, 17]

Similar to Skip Graph, Kademlia is a prefix-based routing DHT. There are many prefix-based broadcast trees proposed for Kademlia. In a prefix-based broadcast tree, starting from level 0, upon receiving an aggregation message, each node at level i increases the level value by one, updates the aggregated data with its own state, and broadcasts the aggregated result as well as increased i to all neighbors with i bits common prefix length in their identifiers [16]. In another variation [17] similar to our landmark-based Skip Graph, In a prefix-based broadcast tree, the system is divided into disjoint regions, where each region covers a specific prefix. A broadcast initiator initiates the broadcast by contacting one node per region. Since in each region only one node is responsible for broadcasting and aggregating of information, the aggregation tree is unbalanced. Some regions may be very crowded and some may be very empty. Hence one node may have a degree of hundreds, while another node from another region has a degree in order of tens.

Table 7.1 provides a comparison between the decentralized aggregation tree schemes and our proposed *ELATS*.

7.4.3 Algorithm used for comparison

Followings are the implementation details of the algorithm used for comparison. They just differ in the in the way the tree is built.

7.4.3.1 Broadcast Tree

Consistent with [12, 13], the randomly chosen initiator initiates an aggregation request. It marks its neighbors as its children in the aggregation tree and broadcasts the aggregation request to all of them. Upon node i receives the first aggregation request from node j, node i marks node j as its parent, and notifies j. Node i then forwards the aggregation request to all neighbor nodes k which is not assigned a parent yet and adds k to its children list. This procedure continues until the aggregation request reaches leaves. A leaf node is the one that can not find any of its neighbors without a parent.

Strategy	Energy Aware	Locality Aware
Parent Function [14]	No	Yes
BFS-trees [10]	Yes	No
SP-trees [11]	No	Yes
Broadcast Tree [12–15]	Yes	No
Prefix-based Tree [12, 16, 17]	No	Yes
ELATS	**Yes**	**Yes**

Table 7.1: Comparison of decentralized aggregation trees

7.4.3.2 Prefix-based Broadcast Tree

Consistent with [16] the randomly chosen initiator initiates the aggregation request with $Level = 0$ and sends it to all its neighbors. Once each neighbor node i receives the aggregation request from node j, if it has been already assigned a parent, discards the aggregation request. Otherwise, it first marks node j as its parent and notifies j. Node i then increases the *Level*, and forwards the aggregation request to all its neighbors who have *Level* bits in common with node i in their name IDs. This procedure continues until the aggregation request reaches leaves. A leaf node is the one that can not find any of its neighbors with the *Level* bits common prefix. The forwarded nodes which reply by a parent notification message constitute the children list of node i.

7.5 Simulation Setup

We extended the Skip Graph simulator SkipSim [38] and developed models to simulate the aggregation tree algorithms. We implemented *ELATS*, broadcast tree, and prefix-based tree in SkipSim, and compared their performance in terms of energy and locality awareness. Each simulation consists of 100 random topologies. In our simulations, we consider $E = 8J$, $U = 1J$, and for *ELATS* $e = 0.01$.

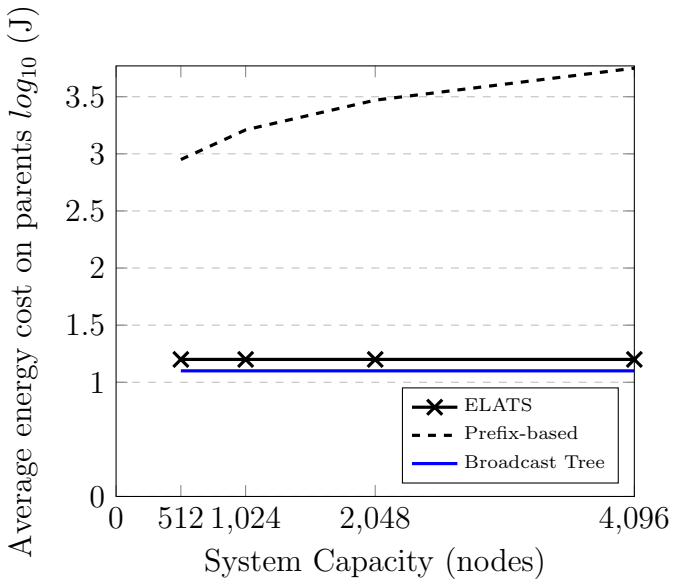

Figure 7.1: Average energy cost on parent nodes in the aggregation tree vs System capacity. Y-axis shows the base 10 logarithms of the average energy cost.

7.6 Performance Results

7.6.1 Energy Awareness

Figure 7.1 shows the average energy cost of non-leaf nodes of the aggregation tree as the system capacity scales up. The system capacity is defined as the maximum number of registered users to the system. As the system capacity scales up exponentially the average number of children assigned to non-leaf nodes in prefix-based aggregation tree increases exponentially. This results in an exponentially increasing energy cost on non-leaf nodes of the prefix-based tree. The degree of each node in the broadcast tree is bounded by the number of its neighbors. Having a system capacity of n, the number of neighbors of a node in the Skip Graph is $O(\log n)$. Hence the exponentially increasing system capacity does not have a noticeable effect on the energy cost of the broadcast tree. Similar to the broadcast tree, *ELATS* acts independent of the system capacity in terms of energy cost and provides the similar energy cost as the broadcast tree which is the most energy efficient existing solution. In

Figure 7.1, the data points of *ELATS* have the constant standard deviation of about $0.82J$, while the data points of the prefix-based tree and the broadcast tree have the increasing standard deviations with the average of $221.5J$ and $0.83J$, respectively.

7.6.2 Locality Awareness

Figure 7.2 shows the average of the maximum latency of root to leaves' path in aggregation trees as the system capacity scales up. In a locality aware Skip Graph as the system capacity increases, the system is divided into more condensed regions and more accurate locality aware name IDs are achieved [39]. This results in an improvement in the latencies of prefix-based and *ELATS*. Since the broadcast tree, which performs as the best energy efficient existing solution, is not locality aware, it performs the worst in terms of latency when compared to *ELATS* and prefix-based tree. As shown in Figure 7.2, in comparison to the prefix-based which acts as the best existing locality aware solution, *ELATS* improves the average value of the maximum root to leaves' path latency with the gain of about 8%. Likewise, in Figure 7.2, the data points have the decreasing standard deviation for *ELATS* with an average of $0.58ms$, and the constant standard deviations of about $0.77ms$ and $199ms$ for the prefix-based tree and the broadcast tree, respectively.

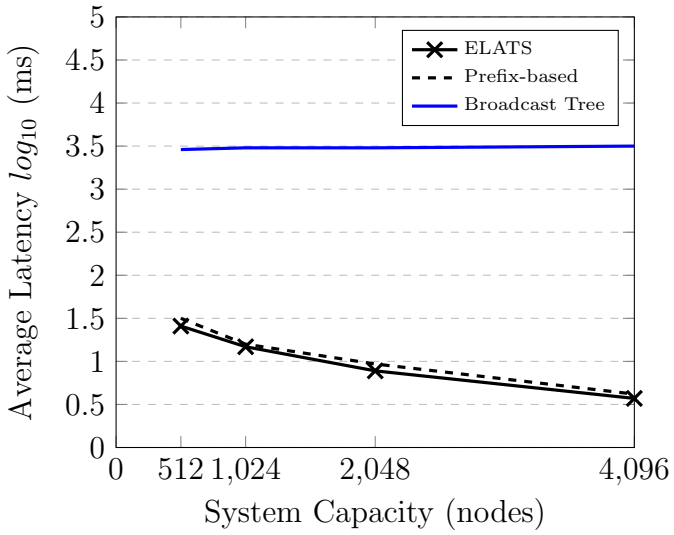

Figure 7.2: The average of maximum root to leaves' path latency in the aggregation tree vs System size. Y-axis shows the base 10 logarithms of average path latency.

7.7 Conclusion

To improve the energy efficiency and response time of the aggregation procedure in a Skip Graph, we propose *ELATS*, the first fully decentralized energy and locality aware aggregation tree. We simulated and compared *ELATS* with state-of-the-art aggregation tree algorithms for DHTs. Compared to the best existing solutions which are either locality aware or energy aware, *ELATS* improves aggregation latency with a gain of about 8% while simultaneously providing similar energy efficiency to the best energy aware existing solution.

Chapter 8

Future Works

8.1 Multi-objective replication algorithm

The locality and availability awareness of replication in *SEALA* are currently provided by *LARAS* [36] and *Awake* [111], respectively. These two algorithms are working independently of each other, which results in providing only one of features (i.e., either locality or availability). A locality aware replication does not necessarily preserve the availability awareness of replication and vice versa. Likewise, the load balancing of replicas is the missing feature of both algorithms. Ignoring the load of nodes (i.e., the number of currently assigned replication duties to the nodes), results in almost the same set of replicas to be selected in the case where different data owners try to replicate for the same or similar sets of data requesters with noticeable overlap. Holding many replication duties by a few nodes increases the rate by which those nodes are queried. This, in turn, results in higher query processing, response time, and failure chance of them, which degrades the performance of the system in terms of response time and data availability.

To the best of our knowledge, there exist no decentralized multi-objective replication algorithm for P2P cloud storage system, considering the locality and availability awareness of replication as well as the load of replicas. As our future work, we aim to propose the first fully decentralized multi-objective replication algorithm, which provides the locality and availability awareness

of replication, while minimizes the load of replication duties on the nodes. The proposed algorithm will result in the uniform distribution of replicas among high available nodes with minimum access latency to the data requesters.

8.2 Churn resilient Skip Graph

As was described in Section 6.2, churn is defined as the dynamic arrival and departures of nodes to and from the system, respectively. A node may fail or depart the system without notifying its neighbors. By falsely assuming that the node is online, neighbors may try to route the search queries via that offline node. This results in the forwarded search queries to the offline nodes to be discarded and failed. We define this problem as the query path failure which arises by the failure of at least one node in a search path. The underlying routing infrastructure of *SEALA* is a Skip Graph in which with the maximum registered users of n, a search path has the average length of $O(\log n)$. With the rate of q queries per second, the expected number of search paths that cross a certain node is $\frac{q}{O(\log n)}$. This means that by failure or departure of a single node, $\frac{q}{O(\log n)}$ search queries are failed per second on the average. This rate is increased linearly as the number of offline nodes increases. The continuation of this process results in a partitioned Skip Graph into several disjoint sub-graphs, which negatively affects the functionality of the whole system.

To the best of our knowledge, there exist no churn resilient algorithm aiming in recovering path failures of Skip Graph. The traditional churn resilient DHT-based solutions applicable on Skip Graph mainly either provide fault tolerance by redundancy [109], or are limited to certain assumptions about the underlying churn behavior of system [126]. Loosing the constraint on redundancy comes with an energy cost to the system. Likewise, being dependant to the underlying churn behavior of the system results in high rate of search failure when the churn behavior changes or is mispredicted. As our future work, we intend to propose the first energy aware churn resilient approach for the Skip Graph, which tries to recover failed path with

an improved energy cost. Our proposed method will be independent of the underlying churn behavior of the system and provide K-fault tolerant connectivity of Skip Graph nodes under churn. By K-fault tolerant we mean that every pair of nodes can still reach each other from the underlying Skip Graph even after failure or departure of K randomly chosen nodes.

8.3 Authenticated Skip Graph

In the current structure of *SEALA*, all nodes are assumed, to be honest. This includes the honest behavior of nodes in computing their name IDs, routing the search queries, and reporting their locality, availability, and load information. This assumption is far from reality where there are some malicious nodes, which do not follow the system protocols in an individual or collaborative manner. The malicious set of nodes can hurt the functionality of Skip Graph from several aspects.

One vulnerability of *SEALA* comes from the routing attack [127] on its underlying Skip Graph. In the routing attack, the malicious nodes try to drop, manipulate, misdirect, or even falsify the answer of the search query. This results in partitioning the system into disjoint sub-graph where the nodes inside one sub-graph are unreachable from the other sub-graphs. To the best of our knowledge, there exist no decentralized authenticated search query strategy for Skip Graph. As our future work, we address the authentication of the search for name ID and the search for numerical ID queries of Skip Graph in a fully decentralized manner. Using our proposed method, each node will be able to verify the result of a search query in a fully decentralized manner, and in the presence of colluding malicious adversaries.

Another vulnerability of *SEALA* arises from the dishonest behavior of nodes. To prevent from being assigned a replication duty, nodes may try to be dishonest in computing and reporting their name IDs, availability information, and their loads. Once a node realizes that its availability, locality, or load features put it at the risk of a replication duty assignment, the node may try to compute or report its locality or availability features in a way that its chance of being abandoned from replication duty increases. These dishonest

behaviors result in far from reality inputs to the corresponding replication algorithms, which makes them unable to perform efficiently. This weakens the performance of the system in terms of query processing, response time, and data availability. As our future work, we try to come up with fully decentralized authentication schemes, which enables each node to authenticate the honesty of others in computing and reporting their locality, availability, and load information in the presence of colluding malicious adversaries.

8.4 Research Timeline

Title	Deliverable Tasks	Expected Date
1^{st} Progress	Multi-objective replication algorithm	November 2017
2^{nd} Progress	Churn-resilient strategy	May 2018
3^{rd} Progress	Authentication mechanisms	November 2018
Thesis Presentation	Finalized Thesis	December 2018

References

[1] S. Legtchenko, S. Monnet, P. Sens, and G. Muller, "Relaxdht: A churn-resilient replication strategy for peer-to-peer distributed hash-tables," *ACM TAAS*, 2012.

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

[3] S. Le Blond, F. Le Fessant, and E. Le Merrer, "Finding good partners in availability-aware p2p networks," in *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2009.

[4] J. Paiva, J. Leitao, and L. Rodrigues, "Rollerchain: A dht for efficient replication," in *NCA*. IEEE, 2013.

[5] M. Rahmani and M. Benchaiba, "A comparative study of replication schemes for structured p2p networks," in *9th International Conference on Internet and Web Applications and Services*, 2014.

[6] H. Shen, "An efficient and adaptive decentralized file replication algorithm in p2p file sharing systems," *Parallel and Distributed Systems, IEEE Transactions on*, 2010.

[7] M. Almashor, I. Khalil, Z. Tari, A. Y. Zomaya, and S. Sahni, "Enhancing availability in content delivery networks for mobile platforms," *Parallel and Distributed Systems, IEEE Transactions on*, 2015.

[8] A. Kermarrec, E. L. Merrer, G. Straub, and A. Van Kempen, "Availability-based methods for distributed storage systems," in *IEEE SRDS 2012*.

[9] L. Jia, R. Rajaraman, and T. Suel, "An efficient distributed algorithm for constructing small dominating sets," *Distributed Computing*, 2002.

[10] M. Dam and R. Stadler, "A generic protocol for network state aggregation," 2005.

[11] D. Dolev, O. Mokryn, and Y. Shavitt, "On multicast trees: structure and size estimation," *IEEE/ACM Transactions on Networking*, 2006.

[12] A. D. Peris, J. M. Hernández, and E. Huedo, "Evaluation of alternatives for the broadcast operation in kademlia under churn," *Peer-to-Peer Networking and Applications*, 2016.

[13] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi, "Efficient broadcast in structured p2p networks," in *International workshop on Peer-to-Peer systems*. Springer, 2003.

[14] J. Li, K. Sollins, and D.-Y. Lim, "Implementing aggregation and broadcast over distributed hash tables," *ACM SIGCOMM*, 2005.

[15] K. Huang and D. Zhang, "Dht-based lightweight broadcast algorithms in large-scale computing infrastructures," *Future Generation Computer Systems*, 2010.

[16] M. Wählisch, T. C. Schmidt, and G. Wittenburg, "Broadcasting in prefix space: P2p data dissemination with predictable performance," in *IEEE ICIW 2009*.

[17] A. D. Peris, J. M. Hern'ndez, and E. Huedo, "Evaluation of the broadcast operation in kademlia," in *2012 IEEE HPCC-ICESS, 2012*.

[18] "TONIC," http://www.r2.com.au/page/products/show/tonic/, accessed: 2017-05-01.

[19] "Squiggle," https://github.com/hasankhan/Squiggle, accessed: 2017-05-01.

[20] "GNU Social," https://gnu.io/social/, accessed: 2017-05-01.

[21] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 2012, pp. 481–494.

[22] J. S. Hale, "Amazon cloud drive forensic analysis," *Digital Investigation*, vol. 10, no. 3, pp. 259–265, 2013.

[23] W. Hu, T. Yang, and J. N. Matthews, "The good, the bad and the ugly of consumer cloud storage," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 110–115, 2010.

[24] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," in *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*. IEEE, 2001, pp. 99–100.

[25] A. R. Bharambe, C. Herley, and V. N. Padmanabhan, "Analyzing and improving bittorrent performance," *Microsoft Research, Microsoft Corporation One Microsoft Way Redmond, WA*, vol. 98052, pp. 2005–03, 2005.

[26] O. Babaoglu and M. Marzolla, "Peer-to-peer cloud computing," *Online Accessed-26 January*, vol. 2011, pp. 1–9, 2017.

[27] T. Prasath and S. Karthikeyan, "Cloud storage vendors wide support and security key features for shifting towards business perspective," *International Journal of Cloud Computing and Services Science*, vol. 2, no. 6, p. 421, 2013.

[28] L. Martinelli. (2014) Cuckoodrive. [Online]. Available: https://github.com/lukasmartinelli/cuckoodrive

[29] K. Graffi and A. Ippisch, "Accelerating data synchronization between smartphones and tablets using powerfolder in ieee 802.11 infrastructure-based mesh networks," in *Qatar Foundation Annual Research Conference Proceedings*, vol. 2016, no. 1. HBKU Press Qatar, 2016, p. ICTPP2257.

[30] M. Scanlon, J. Farina, and M.-T. Kechadi, "Network investigation methodology for bittorrent sync: A peer-to-peer based file synchronisation service," *Computers & Security*, vol. 54, pp. 27–43, 2015.

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

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

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

[34] 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*.

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

[36] Y. Hassanzadeh-Nazarabadi, A. Küpçü, and Ö. Özkasap, "Laras: Locality aware replication algorithm for the skip graph," in *IEEE NOMS 2016*.

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

[38] "Skipsim: https://crypto.ku.edu.tr/downloads."

[39] Y. Hassanzadeh-Nazarabadi, A. Küpçü, and Ö. Özkasap, "Locality aware skip graph," in *IEEE ICDCSW, 2015*.

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

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

[42] W. Galuba and S. Girdzijauskas, "Distributed hash table," in *Encyclopedia of Database Systems*. Springer, 2009, pp. 903–904.

[43] 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.

[44] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*. ACM, 2001.

[45] S. Buchegger, D. Schiöberg, L.-H. Vu, and A. Datta, "Peerson: P2p social networking: early experiences and insights," in *2nd ACM EuroSys*, 2009.

[46] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of P2P systems*, 2003.

[47] B. Van Schewick, *Internet architecture and innovation*. MIT Press, 2010.

[48] K. D. Bowers, A. Juels, and A. Oprea, "Hail: a high-availability and integrity layer for cloud storage," in *16th ACM CSS, 2009*.

[49] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *14th ACM CSS, 2007*.

[50] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *16th ACM CSS, 2009*.

[51] E. Shi, E. Stefanov, and C. Papamanthou, "Practical dynamic proofs of retrievability," in *2013 ACM SIGSAC*.

[52] E. Esiner, A. Kachkeev, S. Braunfeld, A. Küpçü, and Ö. Özkasap, "Flexdpdp: Flexlist-based optimized dynamic provable data possession." *IACR Cryptology*, 2015.

[53] A. Küpçü, "Official arbitration with secure cloud storage application," *The Computer Journal*, p. bxt138, 2013.

[54] A. Kachkeev, E. Esiner, A. Küpçü, and Ö. Özkasap, "Energy efficiency in secure and dynamic cloud storage," in *EE-LSDS*. Springer, 2013.

[55] D. Cash, A. Küpçü, and D. Wichs, "Dynamic proofs of retrievability via oblivious ram," in *EUROCRYPT 2013*. Springer.

[56] M. Etemad and A. Küpçü, "Transparent, distributed, and replicated dynamic provable data possession," in *Applied Cryptography and Network Security*. Springer, 2013.

[57] E. Esiner, A. Küpçü, and Ö. Özkasap, "Analysis and optimizations on flexdpdp: A practical solution for dynamic provable data possession," in *ICC 2014, Muscat*.

[58] D. A. Huffman *et al.*, "A method for the construction of minimum redundancy codes," *IRE*, 1952.

[59] I. Abraham, D. Malkhi, and O. Dobzinski, "Land: Locality aware networks for distributed hash tables," TR 2003-75, Leibnitz Center, The Hebrew University, Tech. Rep.

[60] W. Wu, Y. Chen, X. Zhang, X. Shi, L. Cong, B. Deng, and X. Li, "Ldht: locality-aware distributed hash tables," in *IEEE ICOIN 2008*.

[61] G. Huston, "Exploring autonomous system numbers," *The Internet Protocol Journal*, 2006.

[62] S. Zhou, G. R. Ganger, and P. A. Steenkiste, "Location-based node ids: Enabling explicit locality in dhts," *Technical Report, Carnegie Mellon University*, 2003.

[63] M. J. Freedman, M. Vutukuru, N. Feamster, and H. Balakrishnan, "Geographic locality of ip prefixes," in *5th ACM SIGCOMM*. USENIX, 2005.

[64] V. De Silva and J. B. Tenenbaum, "Sparse multidimensional scaling using landmark points," Stanford, Tech. Rep., 2004.

[65] U. Brandes and C. Pich, "Eigensolver methods for progressive multidimensional scaling of large data," in *Graph Drawing*. Springer, 2007.

[66] A. Allan, R. Humphrey, and G. D. Fatta, "Non-euclidean internet coordinates embedding," in *13th ICDMW, 2013*. IEEE.

[67] S. Lee and S. Choi, "Landmark mds ensemble," *Pattern Recognition*, 2009.

[68] F. Araújo and L. Rodrigues, "Geopeer: A location-aware peer-to-peer system," in *3rd IEEE NCA, 2004*.

[69] D.-T. Lee and B. J. Schachter, "Two algorithms for constructing a delaunay triangulation," *International Journal of Computer & Information Sciences*, 1980.

[70] F. Aurenhammer, "Voronoi diagramsa survey of a fundamental geometric data structure," *ACM CSUR*, 1991.

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

[72] 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.

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

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

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

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

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

[78] 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.

[79] 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.

[80] 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.

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

[82] 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.

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

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

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

[86] 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.

[87] 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.

[88] Y. Chen, R. H. Katz, and J. D. Kubiatowicz, "Dynamic replica placement for scalable content delivery," in *Peer-to-peer systems*. Springer, 2002.

[89] 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.

[90] 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.

[91] A. Pace, V. Quéma, and V. Schiavoni, "Exploiting node connection regularity for dht replication," in *IEEE SRDS, 2011*.

[92] R. Rodrigues and B. Liskov, "High availability in dhts: Erasure coding vs. replication," in *Peer-to-Peer Systems IV*. Springer, 2005.

[93] N. Bartolini, F. L. Presti, and C. Petrioli, "Optimal dynamic replica placement in content delivery networks," in *ICON2003*.

[94] "lpsolve5.5: http://lpsolve.sourceforge.net/5.5/."

[95] D. Yang, Y.-x. Zhang, H.-k. Zhang, T.-Y. Wu, and H.-C. Chao, "Multi-factors oriented study of p2p churn," *International Journal of Communication Systems*, 2009.

[96] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability." in *OSDI*, 2004.

[97] C. Blake and R. Rodrigues, "High availability, scalable storage, dynamic peer networks: Pick two." in *HotOS*, 2003.

[98] V. Simon, S. Monnet, M. Feuillet, P. Robert, and P. Sens, "Splad: scattering and placing data replicas to enhance long-term durability," *Technical Document, Inria*, 2014.

[99] H. Shen and C.-Z. Xu, "Locality-aware and churn-resilient load-balancing algorithms in structured peer-to-peer networks," *Parallel and Distributed Systems, IEEE Transactions on*, 2007.

[100] A. Datta and K. Aberer, "Internet-scale storage systems under churn–a study of the steady-state using markov models," in *P2P Computing*. IEEE, 2006.

[101] A. Lawrance and P. Lewis, "An exponential moving-average sequence and point process (ema1)," *Journal of Applied Probability*, 1977.

[102] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," in *SIGCOMM*. ACM, 2006.

[103] R. Jiménez, F. Osmani, and B. Knutsson, "Connectivity properties of mainline bittorrent dht nodes," in *P2P Computing*. IEEE, 2009.

[104] J. L. J. Laredo, P. A. Castillo, A. M. Mora, J. J. Merelo, and C. Fernandes, "Resilience to churn of a peer-to-peer evolutionary algorithm," *International Journal of High Performance Systems Architecture*, 2008.

[105] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, and X. Zhang, "Measurements, analysis, and modeling of bittorrent-like systems," in *ACM SIGCOMM*, 2005.

[106] D. Wu, Y. Tian, K.-W. Ng, and A. Datta, "Stochastic analysis of the interplay between object maintenance and churn," *Computer communications, Elsevier*, 2008.

[107] Z. Ou, E. Harjula, and M. Ylianttila, "Effects of different churn models on the performance of structured peer-to-peer networks," in *Personal, Indoor and Mobile Radio Communications*. IEEE, 2009.

[108] Z. Yang, Y. Xing, F. Xiao, Z. Qu, X. Li, and Y. Dai, "Exploring peer heterogeneity: Towards understanding and application," in *P2P Computing*. IEEE, 2011.

[109] S. Wakayama, S. Ozaki *et al.*, "A design for distributed backup and migration of distributed hash tables," in *Applications and the Internet, 2008. SAINT 2008. International Symposium on*. IEEE.

[110] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 2002.

[111] Y. Hassanzadeh-Nazarabadi, A. Küpçü, and Ö. Özkasap, "Awake: decentralized and availability aware replication for p2p cloud storage," in *IEEE SmartCloud, 2016*.

[112] S. Taheri-Boshrooyeh, A. Küpçü, and Ö. Özkasap, "Security and privacy of distributed online social networks," in *IEEE ICDCSW, 2015*.

[113] T. Jurdziński, M. Kutyłowski, and J. Zatopiański, "Energy-efficient size approximation of radio networks with no collision detection," *Computing and Combinatorics*, 2002.

[114] R. Zhou and K. Hwang, "Gossip-based reputation aggregation for unstructured peer-to-peer networks," in *2007 IEEE IPDPS*.

[115] M. Bagaa, A. Derhab, N. Lasla, A. Ouadjaout, and N. Badache, "Semi-structured and unstructured data aggregation scheduling in wireless sensor networks," in *INFOCOM, 2012 Proceedings IEEE*.

[116] M. Haridasan and R. Van Renesse, "Gossip-based distribution estimation in peer-to-peer networks." in *IPTPS*. Citeseer, 2008.

[117] C. Gkantsidis, M. Mihail, and A. Saberi, "Random walks in peer-to-peer networks: algorithms and evaluation," *Performance Evaluation*, 2006.

[118] Ö. Özkasap, E. Cem, S. E. Cebeci, and T. Koc, "Flat and hierarchical epidemics in p2p systems: Energy cost models and analysis," *Future Generation Computer Systems*, vol. 36, pp. 257–266, 2014.

[119] N. Bisnik and A. Abouzeid, "Modeling and analysis of random walk search algorithms in p2p networks," in *IEEE IPTPS 2005*.

[120] R. Zhou and K. Hwang, "Trust overlay networks for global reputation aggregation in p2p grid computing," in *IPDPS*. IEEE, 2006.

[121] E. Cem and Ö. Özkasap, "Profid: Practical frequent items discovery in peer-to-peer networks," *Future Generation Computer Systems*, 2013.

[122] A.-M. Kermarrec and M. Van Steen, "Gossiping in distributed systems," *ACM SIGOPS Operating Systems Review*.

[123] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized gossip algorithms," *IEEE/ACM Transactions on Networking (TON)*, 2006.

[124] F. Kuhn and R. Wattenhofer, "Constant-time distributed dominating set approximation," *Distributed Computing*, 2005.

[125] Q.-S. Hua, H. Fan, L. Qian, M. Ai, Y. Li, X. Shi, and H. Jin, "Brief announcement: A tight distributed algorithm for all pairs shortest paths and applications," in *ACM Symposium on Parallelism in Algorithms and Architectures, 2016*.

[126] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a dht." Report No. UCB/CSD-03-1299, University of California, also Proc. USENIX Annual Technical Conference, 2003.

[127] P. Yi, Z. Dai, S. Zhang, Y. Zhong *et al.*, "A new routing attack in mobile ad hoc networks," *International Journal of Information Technology*, vol. 11, no. 2, pp. 83–94, 2005.