

Interlaced: Fully decentralized churn stabilization for Skip Graph-based DHTs

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

March 19, 2019

Abstract

As a distributed hash table (DHT) routing overlay, Skip Graph is used in a variety of peer-to-peer (P2P) systems including cloud storage, social networks, and search engines. The overlay connectivity of P2P systems is negatively affected by the arrivals and departures of nodes to and from the system that is known as *churn*. Preserving connectivity of the overlay network (i.e., the reachability of every pair of nodes) under churn is a performance challenge in every P2P system including the Skip Graph-based ones. The existing decentralized churn stabilization solutions that are applicable on Skip Graphs have intensive communication complexities, which leave them unable to provide a strong overlay connectivity, especially under high rates of churn.

In this paper, we propose *Interlaced*, a fully decentralized churn stabilization mechanism for Skip Graphs that provides drastically stronger overlay connectivity without changing the asymptotic complexity of the Skip Graph in terms of storage, computation, and communication. We also propose the Sliding Window De Bruijn Graph (*SW-DBG*) as a tool to predict the availability of nodes with high accuracy. Our simulation results show that in comparison to the best existing DHT-based solutions, *Interlaced* improves the overlay connectivity of Skip Graph under churn with the gain of about 1.81 times. A Skip Graph that benefits from *Interlaced* and *SW-DBG* is about 2.47 times faster on average in routing the queries under churn compared to the best existing solutions. We also present an adaptive extension of *Interlaced* to be applied on other DHTs, for example Kademlia.

1 Introduction

Skip Graph [1] is a structured overlay where nodes can efficiently perform searches for other nodes or their data objects in a fully decentralized manner. To perform the searches, each node needs to know only a few other nodes

of the system, namely the node’s neighbors. Using the neighboring knowledge, nodes exploit a Skip Graph overlay as a distributed routing infrastructure to initiate or route a search message. Each Skip Graph node keeps its neighbors as $\{identifier, address\}$ pairs in its lookup table. Neighboring relationships in Skip Graphs hence resemble the general idea of the distributed hash tables (DHTs) and yield Skip Graph being known as a DHT-based routing overlay. Because of fast searching, load balancing, and scalability, the Skip Graph is considered as a suitably structured DHT overlay for distributed services such as P2P cloud storage [2–9], and likewise, it can be applied as an alternative overlay in DHT-based applications.

Nodes in a P2P system switch between offline and online states intermittently. Switching to an offline state is considered as a departure from the system. A departed node may come back at a later time and start another online session or may leave the system permanently. Such behavior of dynamic arrivals and departures of the nodes to and from the P2P system, respectively, is referred as *churn*. Churn jeopardizes the connectivity of the overlay network, which we define as reachability of every pair of nodes through the overlay. The compromised overlay connectivity results in search failure, inconsistent search results, and out-dated lookup table entries.

The existing churn stabilization solutions that are applicable on a Skip Graph aim to augment the overlay network by increasing the communication complexity of Skip Graph from $O(\log n)$ to $O(\log^2 n)$ [10], distorting the Skip Graph topology [11] that makes it inapplicable on many scenarios (e.g., locality-awareness [12]), tweaking the size of lookup tables based on the churn rate of the underlying system with minimum consideration of nodes’ availability [13], frequently probing the online status of each neighbor [14–16] that applies a constant communication complexity to the system, or allocating a set of backup neighbors that are contacted alternatively in the event of an unnoticed departure of a neighbor (i.e., the neighbor goes offline without informing the others) [10, 17, 18]. The common downside of all the existing applicable churn stabilization solutions on Skip Graph overlay is that their objective function does not consider node’s position in the overlay network, query latency, communication cost, and node’s availability all together, and sacrifices at least one of them in favor of the rest, which negatively affects the query processing and response time of the system.

To preserve the structural integrity, as well as the routing functionality of the Skip Graph-based P2P overlays under churn, **we propose *Interlaced*, a fully decentralized churn stabilization mechanism for the Skip Graph-based P2P overlays.** *Interlaced* is a backup-based churn stabilization solution that utilizes backup neighbors, and provides scoring mechanisms based on their positioning in the overlay, routing latency, communication cost, as well as neighbor’s availability probability. *Interlaced* does not change the asymptotic complexity of the Skip Graph in terms of communication, computation, and storage. As an independent contribution, we also propose Sliding Window De Bruijn Graph (*SW-DBG*), a fine-grained mechanism to predict the availability probability of the nodes under churn. *Interlaced* uses *SW-DBG* as a tool to

predict the availability of the nodes. Compared to the existing solutions, by benefiting from *Interlaced* and *SW-DBG*, a node can efficiently route the search messages in the absence of its online lookup table’s neighbors with the maximized average success ratio as well as the minimized average search latency. We define the average success ratio of the searches as the ratio of successfully completed searches over all the initiated searches in the system, and define the average search latency as the time it takes for the searches to be routed to the search targets or to be declared as failures. Since Skip Graph can be utilized as a DHT alternative, any DHT-based application can benefit from *Interlaced* and *SW-DBG*.

We consider two main goals in the design of *Interlaced*; maintaining both the connectivity of overlay (i.e., the success ratio of searches) and overlay’s speed (i.e., search latency) for a Skip Graph that undergoes churn. To increase the probability of successful searches under churn, our proposed *Interlaced* employs backup neighbors that are contacted alternatively upon detection of an unnoticed departure of an overlay neighbor (i.e., a lookup neighbor). Using *Interlaced*, each node keeps its backup neighbors in a memory space, named *backup table*, with the same asymptotic space complexity as the lookup table (i.e., $O(\log n)$). Although larger than $O(\log n)$ backup tables seem more successful on routing the searches by providing more alternatives, they increase the communication complexity needed for routing the search queries. The increased communication complexity increases the overall search time and applies additional communication overhead for maintenance that is not bandwidth friendly and congests the system in larger scales. As a general design strategy, *Interlaced* gives more priority on minimizing the overall routing time under churn than maximizing the overlay connectivity. This is in contrast to the existing solutions that solely aim to maximize the overlay connectivity with the minimum attention to the routing time. As a supporting example, a system with the search success ratio of 0.7 but average search latency of 10 seconds is preferable over its counterpart with success ratio of 0.9 but average search latency of 30 seconds. As a failed search in the former system is highly probable to be successful at the second trial, resulting the overall average search time of 20 seconds, which is still 1.5 times faster in terms of query processing time than the latter.

Compared to the existing DHT-based solutions using backup tables [10, 14, 16, 17, 19], *Interlaced* offers a more delicate heuristic in terms of the search path length and search path latency that improves both the overlay connectivity and response time of the system, respectively. Additionally, *Interlaced* operates without any maintenance communication overhead required. To improve the search latency, *Interlaced* works on top of an availability prediction scheme that helps to consider a precedence for backup neighbors based on their availability probability, and contact the most likely online ones first. Our proposed *SW-DBG* executed by each node predicts the node’s availability probability, and serves *Interlaced* with this need.

The contributions of this paper are as follows:

- We propose *Interlaced*, which is a fully decentralized churn stabilization protocol for the Skip Graph-based P2P overlays.
- As an independent contribution, we propose the Sliding Window De Bruijn Graph (*SW-DBG*) that provides an accurate estimate of the availability probability of the nodes.
- We provide an analytical model to predict the behavior of *Interlaced* under uniform failure model, and to estimate the proper backup size that maximizes the performance of the system.
- We extended the Skip Graph simulator, SkipSim [20], implemented, and simulated the best known decentralized churn stabilization and availability prediction mechanisms under churn, and compared with our *Interlaced* and *SW-DBG*.
- Our simulation results show that compared to the best existing solutions that are applicable to the Skip Graph, *Interlaced* improves the search success ratio of the Skip Graph overlay with the gain of about 1.81 times. Likewise, *SW-DBG* improves the availability prediction of the nodes with the gain of about 1.11. The search process of a Skip Graph-based P2P system that benefits from *Interlaced* and *SW-DBG* is about 2.47 times faster on average compared to the best existing solutions.

In Section 2 we describe the structure of Skip Graph, its typical search for numerical ID protocol, and preliminaries such as De Bruijn Graph and churn model. In Section 3 we state our system model. Our proposed *SW-DBG* and *Interlaced* are presented in Sections 4 and 5, respectively. The related works are surveyed in Section 6. Our simulation setup followed by analytical and performance results are presented in Sections 7 and 8. We conclude the paper in Section 9.

2 Preliminaries

2.1 Skip Graph

Structure: An example Skip Graph [1] with 10 nodes and 4 levels is represented by Figure 1. In general, a Skip Graph with n nodes has $O(\log n)$ levels that are numbered starting from 0 in a bottom-up manner. Each Skip Graph node has exactly one element in each level, and is identified with a name ID and a numerical ID. Name IDs are binary strings of size $O(\log n)$ bits, and numerical IDs are non-negative integers. In Figure 1, elements of a node are represented by squares, with numerical IDs enclosed and name IDs are located beneath each element.

Level 0 of a Skip Graph has exactly one distributed list with nodes that are sorted in ascending order. Distributed list means that there is no central entity

(e.g., server) that is supposed to keep the list partially or as the whole; instead, each list's element keeps the address of its successor and predecessor. In level $i > 0$, there exists 2^i lists, where nodes in each list have a common prefix of at least i bits long in their name IDs. For example, in Figure 1 name IDs of 0010, 0110, 0001, 0111, 0000, and 0011, all coexist in the same list at level 1 of the Skip Graph since all their name IDs start with 0 prefix. Likewise, name IDs of 0010, 0001, 0000, and 0011 are located in the same list at level 2 due to their 2-bit common prefix of 00. However, there exists no name ID in the Skip Graph with the prefix of 11, which makes the corresponding list with the prefix of 11 on level 2 empty. Without loss of generality, in this study, we assume the uniqueness of name IDs and numerical IDs and hence we identify a node with either its name ID or numerical ID e.g., by node 43 we mean the node that holds the numerical ID of 43 and the name ID of 1001.

In a Skip Graph-based P2P overlay, each peer from the real world is represented by a Skip Graph node. The numerical ID of each node is the hash value of its corresponding peer's IP address. We assume the name IDs of nodes are generated by a locality-aware name ID strategy e.g., LANS [8]. With locality-aware name IDs, the latency between two nodes in the underlying network is an inverse function of the length of their name IDs' common prefix in the Skip Graph overlay, i.e., longer common prefix conveys lower latency. In a Skip Graph-based P2P overlay, a node is supposed to only know its directly connected predecessor and successor at each level, which are called its left and right neighbors on that level, respectively. A Skip Graph node keeps its neighboring information locally as (*address*, *numerical ID*, *name ID*) tuples in a table, which is called the lookup table of that node as its local view of the system. The lookup table of node 43 from Figure 1 is illustrated in Figure 2 where *Addr* is the (IP) address of the node with numerical ID of *x*.

Search for numerical ID: Search for numerical ID is a fully decentralized search protocol of a Skip Graph, where a node named the *search initiator* starts the search for a *target numerical ID*. Considering a Skip Graph-based P2P system as a distributed database of (*address*, *numerical ID*, *name ID*) records, the search for numerical ID is analogous to a distributed *get()* that retrieves the address of the node with the closest numerical ID to the target numerical ID. As a convention, the search for numerical ID always returns the address of the node with the greatest numerical ID that is less than or equal to the target numerical ID. In the exceptional case where the target numerical ID is less than all the numerical IDs of the Skip Graph, the address of the lowest numerical ID that is greater than the target numerical ID is returned. Nodes route a search message based on their lookup table information. Although nodes may leverage their cached information from previous routings to expedite the ongoing routing tasks, nevertheless, the existence of lookup table guarantees an asymptotic communication complexity of $O(\log n)$ in expectation with high probability for processing a single search for a numerical ID in a fully decentralized manner [1]. It is worth mentioning that the lookup table of each node is as big as the number of Skip Graph's levels i.e., $O(\log n)$.

The thick arrows of Figure 1 show an example search for numerical ID with

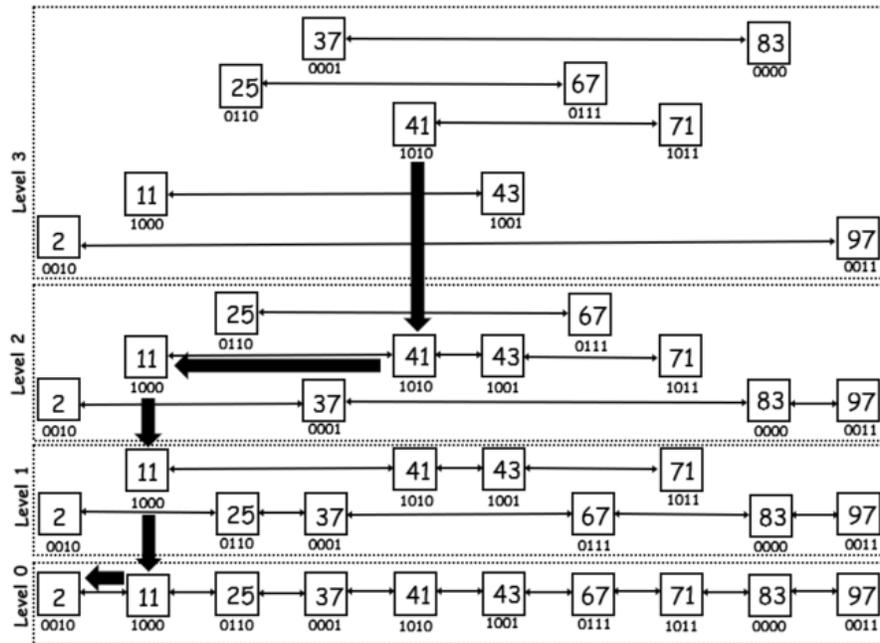


Figure 1: An example Skip Graph with 10 nodes and 4 levels. Nodes are visualized by squares with numerical IDs enclosed and name IDs beneath. Example search for numerical ID of 2 that is initiated by node 41 is depicted by the thick arrows.

Level	Left Lookup Neighbor	Right Lookup Neighbor
3	(A11, 11, 1000)	null
2	(A41, 41, 1010)	(A71, 71, 1011)
1	(A41, 41, 1010)	(A71, 71, 1011)
0	(A41, 41, 1010)	(A67, 67, 0111)

Figure 2: Lookup table of node 43 from Figure 1.

node 41 as the search initiator and 2 as the target numerical ID. A search for numerical ID always starts from the top-most level of the search initiator. In this example, since the target numerical ID of 2 is less than the search initiator's numerical ID (i.e., 41), the search messages are always routed towards the left. The horizontal thick arrows correspond to the routed search messages to the left neighbors, while the vertical ones represent the internal computation of nodes stepping down in their lookup table. On a certain level, the search message is forwarded repeatedly leftward while the left neighbor's numerical ID is greater than or equal the search target. If there is no such neighbor on the left, the node jumps down one level in its lookup table and checks the eligibility of its lower level's neighbor in the search direction. On receiving the search message at level 0, node 2 realizes that it holds the exact target numerical ID as the search message and hence introduces itself as the search result to the search initiator node 41.

2.2 De Bruijn Graph (DBG):

De Bruijn Graph [21] (DBG) is a pattern recognition data structure that is also used as a tool to detect and extract the availability pattern of a single node [22, 23]. In this paper we utilize a slightly modified version of DBG as follows. We identify a DBG by a *state size* of x bits and 2^x vertices that are labeled by the x -bit binary representation of all integers in $[0, 2^x - 1]$ range. A vertex with the binary label of $b_1b_2\dots b_x$ has exactly two outgoing edges to the vertices associated with $\underline{b_2}b_3\dots 0$ and $\underline{b_2}b_3\dots 1$. In representation of the availability behaviour of a node with a DBG, the time is divided into time slots with fixed identical size. Each DBG vertex represents the availability history of the node within the last x time slots, with 1 corresponding to an online state, and 0 corresponding to an offline state. The rightmost and leftmost bits of each state represent the newest and oldest availability status of the node within a window of x time slots, respectively. Having the availability history of a node within the last x time slots as $b_1b_2\dots b_x$, the outgoing edges to $\underline{b_2}b_3\dots 0$ and $\underline{b_2}b_3\dots 1$ denote the availability status of node in the $x + 1^{st}$ slot, which is represented by the rightmost bit (i.e., the underlined ones). The outgoing edge $b_1b_2\dots b_x \rightarrow \underline{b_2}b_3\dots 1$ is associated with $p_{b_1b_2\dots b_x}^1$ i.e., the probability of being online in the $x + 1^{th}$ time slot given the history of $b_1b_2\dots b_x$ in the last x time slots. Similarly, $p_{b_1b_2\dots b_x}^0$ relates to the edge $b_1b_2\dots b_x \rightarrow \underline{b_2}b_3\dots 0$, and represents the probability of being offline in the $x + 1^{th}$ time slot given the availability history of $b_1b_2\dots b_x$ for the node in the last x time slots. The probabilities are taken over all the time slots a specific state is visited, which yields $p_{state}^0 + p_{state}^1 = 1$ for every *state* of a DBG.

2.3 Churn Model

P2P overlays are dynamic with respect to the time i.e., nodes frequently switch between online and offline states. Such dynamic aspect of the system is described by a churn model [24]. A churn model is identified with a session length and

an inter-arrival time distribution. Session length distribution characterizes the online duration of the nodes in the system. The inter-arrival time distribution characterizes time between the start of two consecutive online sessions of nodes in the system.

3 System Model and Scenario

Model: We consider the Skip Graph as an application layer protocol that is executed independently by the peers in an honest manner i.e., each peer follows the exact protocol without deviation and represents a Skip Graph node. The independent executions of Skip Graph protocol by the participating peers shapes a P2P overlay that is constructed by joining the first peer to the system and grows over the time by joining other peers. In particular, peers use the insertion algorithm of the Skip Graph to join the system as Skip Graph nodes [1]. After joining the system, nodes frequently perform the search for numerical IDs to find each other and resources, help other nodes joining the system, or to perform other P2P tasks e.g., replication [5, 7, 25]. We define the *system capacity* n as the number of registered users to the Skip Graph-based P2P system which is constant despite the churn of nodes. In this paper, we consider the system capacity as the smallest power of two that is greater than or equal to the total number of registered nodes in the system. We define the *timeout failure* as the situation where a node does not hear from its offline neighbor within a certain time interval after routing a search message to it. We consider the time interval duration that can trigger a timeout failure as the function of the round trip time between the node and its neighbor.

Scenario: Using our proposed *SW-DBG* (see Section 4), each online node computes its own availability probability at the end of each time slot and piggybacks it alongside its address, name ID, and numerical ID on all the search messages it routes or initiates. On receiving a search message to route, the node updates its backup table with the piggybacked availability information of other nodes. To do such an update, upon a message reception, the node invokes the *backupUpdate* event handler of our proposed *Interlaced*. As described in Section 2, a node routes a search message by forwarding it to one of the lookup neighbors that is placed in the level and direction of the search. However, due to the churn, the selected lookup neighbor may be offline, and unable to receive the search message. We presume an associated timer for every forwarded message with the expiration time as a function of the round trip time between the node and its corresponding neighbor. If no (TCP) acknowledgment is received from the neighbor before the timer expires, that neighbor is considered as offline, and a timeout failure happens. On timeout failures, the node invokes the *backupResolve* event handler of *Interlaced*, which returns back an online candidate node from the backup table that is consistent with the search path, and the search message is redirected to this online candidate.

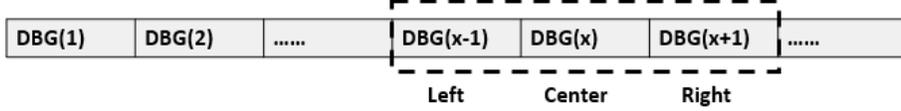


Figure 3: The general form of Sliding Window De Bruijn Graphs (*SW-DBG*) with the current state window adjusted on $\text{DBG}(x-1)$, $\text{DBG}(x)$, and $\text{DBG}(x+1)$, which are denoted by Left, Center, and Right DBGs, respectively.

4 Sliding Window De Bruijn Graph (*SW-DBG*)

4.1 Overview

Choosing a proper state size for DBG predictors is a challenge as large state sizes enforce a noise over the DBG that increases the prediction error [22], while small state sizes fail DBG to completely capture the availability behavior of the nodes. Furthermore, DBG has exponential asymptotic space and time complexities as a function of its state size. Therefore, finding the smallest state size that provides an acceptable level of prediction accuracy lets the nodes to operate efficiently in both space and time. In order to find the proper state size of DBGs and predict the availability probability of a node in an adaptive manner, we propose the Sliding Window De Bruijn Graph (*SW-DBG*). By adaptive we mean that in contrast to the existing DBG-based predictors [22, 23], which have a fixed state size, *SW-DBG* continuously moves towards the state size that describes the node’s instantaneous availability status the best. As shown by Figure 3, *SW-DBG* is a list of DBGs that are represented by $\text{DBG}(x)$ where x is the state size of the corresponding DBG. The list is started by $\text{DBG}(1)$, and the state size of DBGs increases by one moving from left to the right. Despite this long list of DBGs, however, an *SW-DBG* is required to only keep three consecutive DBG instances on its memory space, and operate on those accordingly. This set of three consecutive DBGs is called as the **current state window** that is represented by a dashed rectangle in Figure 3. Each node updates the current state window’s DBGs of its *SW-DBG* once within each time slot, and computes and piggybacks its stationary online probability on all the search messages it routes or initiates. We define the **stationary online probability** of a node as the probability of the node being online after infinitely many time slots elapsed (i.e., at the time $t \rightarrow \infty$). The stationary online probability of a node is updated frequently (i.e., once every time slot) to reflect the possible changes in the availability behavior. After each update, *SW-DBG* may move the current state window to the left or right if it realizes that a smaller or bigger state size may describe the availability status of the node with lower prediction error. We assume that an offline node updates its *SW-DBG* for all its offline time slots right at the end of its first new online time slot at the system upon arrival.

4.2 Stationary Online Probability

We model each $\text{DBG}(x)$ of a *SW-DBG* with a single Markov Chain process [26], by representing each vertex i of DBG (i.e., $i = b_1b_2\dots b_x$) with a Markov Chain state where x is the state size of DBG . The transition matrix of each DBG is represented by P , and shown by Equation 1. $p_{i,j}$ is the transition probability from state i to state j , and is determined by Equation 2. \mathcal{S}_x denotes the set of $\text{DBG}(x)$'s states, and constitute of all x bits binary strings. As presented in Section 2, for state $i = b_1b_2\dots b_x$, p_i^0 and p_i^1 denote the transition probabilities of $b_1b_2\dots b_x \rightarrow b_2b_3\dots 0$ and $b_1b_2\dots b_x \rightarrow b_2b_3\dots 1$, respectively.

$$P = \{p_{i,j} | 1 \leq i, j \leq x\} \quad (1)$$

$$p_{i,j} = \begin{cases} p_i^0, & \text{if } j = b_2\dots b_x 0 \\ p_i^1, & \text{if } j = b_2\dots b_x 1 \\ 0, & \text{Otherwise} \end{cases} \quad (2)$$

When the Markov Chain analogy of a $\text{DBG}(x)$ is Ergodic (i.e., each state is accessible from every other state), the system that is illustrated by Equation 3 comes to a unique answer in the form of $\{\pi_1, \pi_2, \dots, \pi_x\}$. π_i denotes the stationary probability of state i , and represents the probability of visiting state i after infinitely many state transitions independent of the initial state.

$$\begin{cases} 1 &= \sum_{i \in \mathcal{S}_x} \pi_i \\ \pi_i &= \sum_{j \in \mathcal{S}_x} \pi_j \times p_{j,i}, \forall j \in \mathcal{S}_x \end{cases} \quad (3)$$

We represent the stationary online probability of an Ergodic $\text{DBG}(x)$ ($x \geq 1$) by sop_x . As shown by Equation 4, sop_x corresponds to the aggregated stationary probabilities of all DBG 's states i that end with 1. Since the rightmost bit of each state corresponds to the most recent availability status, the states that end with 1 represent an online status of the node in its availability history. Summing up the stationary probabilities of all those states results in the stationary online probability of the node.

$$\text{sop}_x = \sum_{\{i \in \mathcal{S}_x | i = b_1b_2\dots 1\}} \pi_i \quad (4)$$

4.3 Algorithm Description

As shown by Figure 3, at any point in time, the current state window has three DBG s that are denoted by $\text{DBG}(x-1)$, $\text{DBG}(x)$, and $\text{DBG}(x+1)$, $x \geq 2$, and called the Left, Center, and Right DBG s, respectively. A node initializes its instance of *SW-DBG* by adjusting the Left DBG of the current state window on $\text{DBG}(1)$. While the node is online, it updates the current state window

with its own availability status, by invoking *stateUpdate* algorithm at the end of each time slot. As shown by Algorithm 1, the inputs to *stateUpdate* are the current state window *cw* (i.e., the collection of 3 DBGs; Left, Center, and Right), as well as the *status* of the node, which is 1 if the node is online, and 0 otherwise. On receiving the inputs, *stateUpdate* updates the DBGs inside the current state window by a call to their *update* routine that updates the current state of DBG with the *status* bit, updates the proper state transitions' probabilities, and returns the stationary online probability upon existence. If the modeled Markov Chain of a DBG does not show *ergodicity* [26], *update* returns either 0 or 1 depending on the existence of an offline or online absorbing state, respectively (Algorithm 1, Lines 1-3). In other words, in our system scenario, non-ergodicity happens when there exists an absorbing online or offline state, and hence there exists no stationary distribution for the modeled Markov Chain. By an absorbing offline or online state, we mean an absorbing state that ends with 0 or 1, respectively.

updateStatus computes the prediction error of the sliding window's DBGs that is determined as the absolute value of the difference between their *sop* probability and their availability fraction within a state size window of last time slots. For example, if one DBG(3) predicts the stationary online probability as 0.2 while the availability status of the node in the window of last 3 time slots is 101, the prediction error is computed as $|0.2 - \frac{2}{3}| = 0.44$, which is a noticeable prediction error. The strictly decreasing prediction errors of current state window's DBGs towards either the right (i.e., $predErr_{Left} > predErr_{Center} > predErr_{Right}$), or the left (i.e., $predErr_{Left} < predErr_{Center} < predErr_{Right}$) implies a prediction accuracy improvement on moving the current state window towards the right or left direction, respectively. A right movement is done by dropping the *Left* DBG out of the current state window and replacing its pointer with *Center* DBG. The *Center* DBG is also replaced by the *Right* DBG, and finally the *Right* DBG's state size is increased by one bit via performing a call on its *enlarge* function that does the enlargement by mapping each state $b_1b_2\dots b_x$ to the two states $b_1b_2\dots b_x0$ and $b_1b_2\dots b_x1$ while preserving the associated probabilities of the states. A left movement is done similar to the right movement except that the function *shrink* of the *Left* DBG is called that maps each two states $b_1b_2\dots b_{x-1}0$ and $b_1b_2\dots b_{x-1}1$ to a single $b_1b_2\dots b_{x-1}$ state in the new shrunk *Left* DBG. We compute the probability of each shrunk state as the average probability of the states that are mapped to it (Algorithm 1, Lines 4-14). The *updateStatus* function terminates its task by returning back the stationary online probability of the current state window that is defined as the stationary online probability of its DBG with minimum prediction error (Algorithm 1, Line 15). With this approach of shrinking and enlarging the current state window's DBGs, our proposed *SW-DBG* adaptively chooses the DBG size that best describes the availability behavior of the node, which is in contrast to the conventional strategy of traditional DBG on having a fixed-size state size independent of the node's availability behavior.

Algorithm 1: stateUpdate

Input: DBG[] cw , bit $status$
// updating the current state window with input status
1 **for** $i \in \{Left, Center, Right\}$ **do**
2 | $sop_i = cw[i].update(status)$;
3 **end**
// computing the prediction errors
4 **for** $i \in \{Left, Center, Right\}$ **do**
5 | $predErr_i = |status - sop_i|$;
6 **end**
// checking for the enlarge or shrink
7 **while** $predErr_{Left} > predErr_{Center} > predErr_{Right}$ **do**
8 | Slide the current window to the right;
9 | $cw[Right] = cw[Right].enlarge()$;
10 **end**
11 **while** $predErr_{Left} < predErr_{Center} < predErr_{Right}$ **do**
12 | Slide the current window to the left;
13 | $cw[Left] = cw[Left].shrink()$;
14 **end**
15 **return** $cw[\operatorname{argmin}_{predErr}\{cw\}].sop$;

5 Interlaced

5.1 Overview

Interlaced is a fully decentralized application layer churn stabilization protocol that is executed independently by every node of a Skip Graph-based P2P overlay. A node executes *Interlaced* upon facing a timeout failure (see Section 3) on routing a search message based on its lookup table. We denote the node that executes an instance of *Interlaced* by *executor*. In other words, any node running *Interlaced* is named as an executor, and this does not imply the restriction to any special node like a super node. The main goal of *Interlaced* is to recover a search query from a timeout failure on its current path. *Interlaced* assumes an additional $O(\log n)$ storage of the neighboring information for the executor that is called *backup table*, which is a common trait among the existing solutions [14, 16, 17, 19]. The executor holds the collected availability information of some other nodes of Skip Graph on its backup table in a level-wise manner similar to the lookup table. On referring to the backup table, *Interlaced* finds the best routing candidate and redirects the search message to it. *Interlaced* chooses the best routing candidate based on the numerical ID distance to the search target (i.e., number of intermediate nodes on the path), the locality-aware name ID similarity with the executor (i.e., common prefix) that corresponds to a expected latency in the underlying network, and the stationary online probability of the candidate. On redirecting the search message to the best candidate, if no acknowledgement is received within a certain time interval, the best candidate

Level	Left Lookup Neighbor
0	(A41, 41, 1010)

(a) Left lookup table entry at level 0

Level	Left Backup Neighbors
0	(A25, 25, 0110, (A25).sop, (A25).score), (A37, 37, 0110, (A37).sop, (A37).score)

(b) Left backup table entry set at level 0

Figure 4: A comparison between a lookup table entry of node 43 in the sample Skip Graph of Figure 1, and its corresponding potential backup table entry set at the same level and direction.

is presumed offline, and *Interlaced* moves to the next best routing candidate.

5.2 Backup table

The backup table resembles the lookup table in structure, except that instead of storing a single node information at every entry (i.e., cell), each entry of a backup table represents a set of nodes' information. Hence, each level of a backup table constitutes of two **entry sets** (i.e., one at the left and one at the right), each representing a set of nodes' information.

Interlaced changes the size of each set adaptively with the overall number of backup neighbors being within the range of $[0, b]$ where b is a system-wide constant named as the *backup size*. This implies that backup table has the same memory complexity of $O(\log n)$ as the lookup table. Each entry holds the information of a single node in the form of $(address, numID, nameID, sop, score)$ tuple where *address* represents the (IP) address of the node in the underlying network. *numID* and *nameID* represent the numerical and name IDs of the node, respectively. The stationary online probability of the node (see Section 4) is denoted by *sop*, and *score* is a real number that is frequently updated and used by *Interlaced* to select the best routing candidate. Figure 4 shows the left lookup table entry of node 43 at level 0 from the sample Skip Graph of Figure 1, and a potential corresponding backup table entry set to it at the same level and direction. Note that although backup table entry set of Figure 4 holds two neighbors, nevertheless, these two neighbors are solely for the sake of clarification, and in practice a backup table entry set is able to hold an arbitrary number of neighbors limited by the overall size b of the backup table.

5.3 Algorithm Description

Interlaced consists of two event handlers: *backupUpdate* and *backupResolve* that are called by the executor on the events of receiving a search message to route, and timeout failure, respectively.

5.3.1 Updating backup table (*backupUpdate*):

Identifying the proper entry set: In our system model, we assume that while a node is online it computes its stationary online probability by applying *SW-DBG* on its availability history, and piggybacks its (IP) address, numerical ID, name ID, and stationary online probability on the search messages it routes or initiates (as described in Section 4). On reception of a search message that

contains the piggybacked information of other nodes, the executor invokes the *backupUpdate* event handler of *Interlaced* on the set of piggybacked information. On receiving the inputs, *backupUpdate* inserts each piggybacked element e of the message that does not correspond to a lookup table neighbor, into a proper entry set of the executor’s backup table. The proper set is identified by the direction and level. The direction is determined based on the numerical ID position of the executor with respect to the piggybacked element’s numerical ID i.e., right direction if $e.numID > executor.numID$, and left direction otherwise. The level corresponds to the common prefix length between the executor’s and element e ’s name IDs. If an older version of e (i.e., a version with possibly different stationary online probability) resides in the backup table, *backupUpdate* overwrites it with the new element. By an older version, we mean an element e that exists in the backup table due to the *backupUpdate* invocation on previous messages that contained e . Otherwise, it adds the piggybacked element into the proper backup table’s entry set that is identified by the level and direction.

Scoring-based element replacement: Before inserting a new piggybacked element into the proper backup table entry set, *backupUpdate* evaluates the size of the backup table against the maximum permissible size b . In contrast to the existing solutions that discard the oldest entry in the case that backup table is full, *Interlaced* provides a scoring mechanism to distinguish the backup table neighbors and replaces the element with the minimum score by the new piggybacked element. The scoring-based element replacement mechanism of *backupUpdate* is enabled upon a backup table size violation, sorts all the entries of backup table based on their updated score value as represented by Equation 5, and drops the element with the minimum score value out of the backup table. This squeezes the size of backup table down to $b - 1$ element, which can accommodate the new piggybacked element.

$$e.score = \frac{e.sop \times commonPrefixLength}{|e.numID - numID|} \quad (5)$$

In Equation 5, $e.score$ denotes the score that *backupUpdate* assigns to element e , which has a direct relation with its stationary online probability (i.e., $e.sop$) as well as the length of common prefix between the name IDs of executor and element e (i.e., $commonPrefixLength$). We utilize a locality-aware Skip Graph [6] where name IDs’ common prefix length reversely reflects the underlying latency between two nodes i.e., a longer common prefix length reflects a lower latency. Hence, the score of an element is a direct function of its availability probability that is projected by its latency towards the executor node. Moreover, the score of an element is an inverse function of its numerical ID distance with respect to the executor. Following the search for numerical ID protocol (see Section 2), the numerical ID distance to the search target keeps decreasing as the search message proceeds towards the target. This hints that as the search path lengthens, the probability that an executor node on the path receives a search message for a target in its own numerical ID vicinity increases. Consequently, for the routing to be successfully conducted under churn, *backupUpdate* aims to keep the backup table elements in the numerical ID proximity of the

executor node by assigning a higher score to the backup neighbors in shorter numerical ID distances than the others. This increases the chance of having routing candidates inline with the search direction and in the proximity of the executor’s numerical ID. Once the size reduction is done, *backupUpdate* function adds the new element to the identified set by the level and direction, and returns back the updated backup table once all elements in the input search message were processed.

5.3.2 Using backup table (*backupResolve*):

As the timeout failure event handler on routing a search message, the executor invokes *backupResolve* algorithm that is shown by Algorithm 2. Inputs to *backupResolve* are the backup table and name ID of the executor (i.e., *backup* and *nameID*, respectively), the search target numerical ID (i.e., *target*), the current ongoing level and direction of the search (i.e., *level* and *dir*, respectively), and the search message itself (i.e., *msg*). The direction *dir* is either LEFT or RIGHT. As the output, *backupResolve* returns an online routing candidate from the backup table if such online candidate exists. Otherwise, it returns *NULL*. If an online candidate is returned by *backupResolve*, the search message is redirected to that candidate by the executor.

On receiving the inputs, *backupResolve* iterates over the *backup[level][dir]* set, and evaluates the eligibility of each element as a routing candidate. This is done by invoking *candCheck(e, target, dir, msg)* that evaluates each element from two aspects; being on the search direction, and not yet being visited by this search message (i.e., $e \notin msg$). In checking the consistency with the search direction, *candCheck* returns *false* if the direction is RIGHT (or LEFT) but the *e.numID* is greater (or less) than the *target*. Moreover, to avoid looping on the search path, *candCheck* returns *false* if there exists a piggybacked node information on *msg* that corresponds to element *e* (i.e., $e \in msg$). If none of these violations happen, *candCheck* returns *true*. All elements with the *candCheck* value of *true* are evaluated as tentative routing candidates and added to the *candidatesList*, which denotes the list of candidates. Before adding each candidate *e* to the list of candidates, *backupResolve* scores it similar to the scoring strategy of *backupUpdate* (Equation 5), except that the numerical ID distance is evaluated with respect to the *target*, and not the executor node. In other words, to compute the score of each backup table element *e*, *backupUpdate* follows Equation 5 by replacing *e.sop* with the stationary online probability of element *e*, *commonPrefixLength* with the common prefix length between the name IDs of the executor and element *e*, and the denominator with the numerical ID distance of element *e* to the search target. This scoring is to discriminate the candidates based on their stationary online probability, search path length to the *target*, and latency between them and executor. By the search path length, we mean the number of nodes on the path between the candidate and *target*. A lower numerical ID distance to the search target implies the lower number of intermediate nodes on the search path (i.e., a shorter search path) in expectation with high probability, which increases the success probability of

search under churn as the search is more likely to hit the *target* in the immediate subsequent steps. Likewise, under the locality-awareness assumption of the overlay, routing to a more similar name ID with respect to the executor, decreases the expected routing latency, which compensates the delay caused by timeout failure, partially. A higher score corresponds to a better candidate in terms of availability probability, closeness to the *target* in the overlay network, and lower latency to the executor on receiving the redirected search message in the underlying network (Algorithm 2, Lines 1-10).

After creating the list of candidates, *backupResolve* selects the candidate with the maximum score as the best routing candidate, checks its online status (e.g., by pinging it), and returns its address to the executor in the case that it is online. If the best candidate is offline, *backupResolve* removes it from both the backup table and candidates list and moves to the next best candidate. Removing offline candidates from the backup table is to address the permanent departure of nodes that have resided long enough in the system to show a good stationary online probability, but are no longer available after their departure. *candidatesList* running out of candidates implies that there is no other online alternative to redirect the search message. This makes *backupResolve* to return *NULL*, which hints the termination of the search at that level to the executor (Algorithm 2, Lines 11-21). Termination at non-zero levels makes the executor to continue the search at lower levels (see Section 2). However, termination at level zero corresponds to the termination of the whole search, and executor returns back its own address as the search result to the search initiator. Moreover, following the *Interlaced*'s preference on fast response under churn than full connectivity, it only refers the part of the backup table that is consistent with the search.

5.4 Applying on other DHTs

Interlaced is capable of being efficiently applied on other DHTs especially the prefix-based ones like Kademlia [17], which resembles Skip Graph in prefix-based binary string identifiers. To apply *Interlaced* on such DHTs, each node needs to allocate a backup table and apply some minor modifications to the event handlers of *Interlaced* i.e., *backupUpdate* and *backupResolve*. For *backupUpdate*, determining the level and direction of a new element to be inserted in the backup table needs to be changed based on the architecture of the DHT. For example, in Kademlia where the search messages are routed based on the XOR distance between the executor's and the target's identifiers, identifying the proper backup set is done by the XOR distances, and without the need to determine any direction. As each backup set is identified by an identifier prefix, the proper backup set for a new element is the one with the minimum XOR distance to its identifier. Also, the scoring-based element replacement part of *backupUpdate* is modified as shown by Equation 6 where *e.id* and *id* are the identifiers of the piggybacked element *e* and executor, respectively, and \oplus denotes the XOR operation. Nodes in Kademlia are solely identified by a single binary string. Hence, the common prefix length concern of *backupUpdate* is addressed by the XOR distance. Upon the locality-awareness of the Kademlia's identifiers, the

Algorithm 2: backupResolve

Input: backup table $backup$, executor name ID $nameID$, search target numerical ID $target$, level of search $level$, direction of search dir , Message msg

// initializing the list of candidates to contact

```
1 candidatesList =  $\emptyset$ ;  
2 for element  $e \in backup[level][dir]$  do  
3   if  $e.numID == target$  then  
4     // the search target has been found  
5     return  $e.address$ ;  
6   end  
7   if  $candCheck(e, target, dir, msg)$  then  
8     // assigning the score to the candidate  
9      $e.score = \frac{e.sop \times commonPrefixLength}{|e.numID - target|}$ ;  
10    // adding the candidate to the list  
11    candidatesList.add( $e$ );  
12  end  
13 end  
14 while !candidatesList.isEmpty() do  
15   // picking the best candidate from list  
16    $bestCandidate = \operatorname{argmax}_{score}\{candidatesList\}$ ;  
17   // redirecting the search message to the best routing candidate  
18   send( $msg, bestCandidate.address$ );  
19   if  $bestCandidate.isonline()$  then  
20     break;  
21   else  
22     // removing the offline best candidate from the candidates list and  
23     backup table  
24     candidatesList.remove( $bestCandidate$ );  
25     backup.remove( $bestCandidate$ );  
26   end  
27 end  
28 return NULL;
```

scoring-based element replacement of *backupUpdate* also considers the latency constraint i.e., a lower XOR distance implies higher common prefix, and hence a lower underlying latency, which results in a higher score.

$$e.score = \frac{e.sop}{e.id \oplus id} \quad (6)$$

For *backupResolve* event handler, the node needs to change the *candCheck* implementation (Algorithm 2, Line 6) based on the architecture of DHT. For example, in Kademia a (backup) neighbor is eligible to be contacted by the search protocol upon a lower XOR distance with the target identifier than the executor’s one. Likewise, the DHT node needs to change the scoring mechanism of *backupResolve* event handler based on the identifiers’ distance metric of the DHT overlay. In Kademia’s case that operates on XOR distances, the scoring mechanism needs to consider the XOR distance of each node to the search target as shown by Equation 7, which is the updated scoring formula of *backupResolve* where *target* is the search target’s identifier, and *commonPrefixLength* is the length of common prefix between the identifiers of element *e* and executor.

$$e.score = \frac{e.sop \times commonPrefixLength}{e.id \oplus target} \quad (7)$$

6 Related Works

6.1 Churn Stabilization

As a broad taxonomy, the P2P churn stabilization mechanisms for structured overlays are divided into static and dynamic approaches as described next.

Static Churn Stabilization: The aim of static approaches is to enrich the structural connectivity of the P2P overlay. D1HT [27], Skip+ [10], and Epi-Chord [28] augment the overlay connectivity by providing *larger routing tables*. These approaches need high maintenance overhead in terms of the number of update rounds and exchanged messages e.g., $O(\log^2 n)$ communication complexity for Skip+ [10], which makes these approaches inefficient under continuous high churn rates where the overlay connectivity changes faster than the update rate of the nodes [29]. As another example, D1HT needs $O(n)$ lookup table memory complexity, and epidemic dissemination of queries for the sake of maintenance. *Hierarchical overlays* [30, 31] cause unbalanced load on the upper tier nodes, and direct the P2P system towards centralization. Rainbow Skip Graph [11] and Tiara [32] follow the *virtual nodes* approach where two or more physical peers are coupled into one single node of the P2P overlay to reduce the negative effect of individual departures on the overlay connectivity [33]. Virtual nodes, however, distort the structure of overlay and make it inappropriate for many applications including locality-aware replication [7] that requires the nodes to be orchestrated based on their locality information [6] rather than the availability.

Dynamic Churn Stabilization: In contrast to the static approaches that reinforce the structured overlay uniformly regardless of the churn, the dynamic

churn stabilization approaches aim to maintain the connectivity based on the local knowledge of nodes about the underlying churn. The knowledge varies from a rigid assumption to intermittent perception from the underlying churn. The dynamic churn stabilization approaches are further classified into proactive, reactive, and predictive. In *proactive* dynamic stabilization, nodes frequently check their neighbors' availability by either pinging them, or conducting a search for them [14–16, 34–37]. The main disadvantage of proactive approaches is their dependency on active exploration of the overlay network, which leads a persistent communication overhead.

In *reactive* churn stabilization approaches, the maintenance of routing tables is done only upon the detection of an entry's failure [38]. The threshold of failure at which the reactive stabilization starts is a function of the underlying churn rate. Reactive churn stabilization approaches are not bandwidth friendly. They congest the underlying network in the events where the churn rate is not well-predicted, or the threshold is not well-chosen. DKS [38] provides a structured P2P overlay construction that is in compliance with circular DHTs like Chord [39] as well as those based on XOR distances e.g., Kademia [17]. To handle churn, each DKS node holds a fixed number of pointers to the nodes that immediately follow it in the identifier space. The list is updated upon failure detection of any of the successors in a "correction-on-used" manner i.e., the failed node is replaced by a new successor. DKS's success on maintaining connectivity is correlated with the failure pattern of the successors i.e., concurrent failures of the successors keep a node away from finding new successors and recovering a search from failure. 1-backtracking [40] is another reactive solution where a message is back-tracked one step upon detection of a failure on the path, and re-routed again. In systems with high churn rate or low availability, it is very likely for the alternative backtracked neighbors to be offline, which causes the entire search to be dropped but at a longer response time compared to no backtracking scenario. Also, increasing the degree of backtracking (e.g., 2-backtracking) results in high search delay due to the exponential growth of the alternatives that are contacted blindly without any availability perception. In contrast to *Interlaced*, both DKS and backtracking approaches do not consider the search latency, search path length, and availability of the alternative neighbors for recovering a search message from failure.

Predictive churn stabilization is another dynamic approach where nodes aim to predict the failure of their neighbors ahead in time and redirect the search queries to the highly likely online neighbors. The general idea of predictive approaches is similar to our proposed *Interlaced*. For example, Kademia [17] resembles *Interlaced* in holding backup neighbors. However, in contrast to the *Interlaced*, the backup neighbors in Kademia are solely scored based on their last-seen time [16, 18, 41]. Kademia's approach on replacing the oldest entry with the newest piggybacked element increases the lack of routing candidates especially under high churn rates where the buckets are updated more frequently, which degenerates the connectivity of the system. Also, scoring the backup neighbors solely on a least-recently-seen basis increases the expected number of trials takes to find an online routing candidate, which increases the communica-

tion overhead as well as the query processing time, and exposes the underlying network to congestion in larger scales. Moreover, in contrast to *Interlaced*, Kademia does consider the latency and search path length in its scoring mechanism.

6.2 Availability Prediction

For availability prediction, *regularity-based* solutions [42–44] aim to extract the long-term regular behavior of nodes, and do not exploit irregular nodes, which constitute a large chunk of the system. Despite not showing a regular availability pattern, irregular nodes may be online for a long enough while to participate in a churn stabilization protocol. *Vector-based* solutions [5, 25, 45] predict the availability probability of the nodes within the slots of a fixed-periodic time interval e.g., availability probability in each hour of a day. *Hidden Markov Model (HMM)* [46] approaches predict the availability of a node qualitatively within the four states of born, young, aged, and offline. Both the vector-based and HMM-based approaches are long-term solutions that necessitate long learning phases, and are not applicable to instantaneous prediction cases like churn stabilization. Accordion [37] utilizes a variation of Lifetime predictor [47], where the availability probability of a node is computed as a function of its accumulative sessions’ lengths. LUDP [48] predicts the availability of nodes based on their number of incoming connections where a higher number of incoming connections as well as age corresponds to a higher availability probability. The age of a node is determined by its accumulative online sessions’ lengths.

6.3 Algorithms used for comparison:

We selected Kademia [17] and DKS [38] for the sake of implementation and comparison with our proposed *Interlaced* as they are the only ones that resemble *Interlaced* in keeping the communication complexity of Skip Graph intact, and being needless of frequent probing. Among the availability prediction solutions, we selected DBG [22], Lifetime [47], and LUDP [48] for the sake of implementation and comparison with our proposed *SW-DBG* as they are the only ones that resemble *SW-DBG* in providing an instantaneous and fine-grained availability prediction. The implementation details of these algorithms are described next.

6.3.1 Churn Stabilization

Kademia [17]: We follow the same implementation as *Interlaced*, except, we replace the backup table’s sets with double linked-lists. We distribute the backup size, b , uniformly among the buckets at every level and direction. In case the total number of levels is not a divisor of b , the remainder is equally distributed at each level by a value of two (i.e., one extra pointer at each direction) starting from level zero. The bottom-up distribution of the surplus backup is due to the importance of the bottom-most levels on the success of the search. Essentially, the level zero of Skip Graph is where an unsuccessful search terminates. Hence

having a bigger backup size at this level increases the chance of the search being rescued from failure. We modify the *backupUpdate* to simply insert the piggy-backed elements to the head of the proper linked-list, and remove linked-list’s tail if the size goes beyond the permissible, b . We modify the *backupResolve* such that after identifying the proper bucket, it checks the online status of the routing candidates inside the bucket starting from the head until it finds an online routing candidate to return. Similar to *Interlaced*, *backupResolve* returns NULL if the list runs out of routing candidate.

DKS [38]: In our implementation of DKS, we distribute the backup size, b , among the levels similar to the Kademia’s case. Each Skip Graph node then continuously holds a list of pointers to its immediate neighbors at each level and direction. DKS does not piggyback any availability information on the search messages. Rather, upon joining the system, each node contacts its immediate neighbors and initializes its pointers list by invoking the *backupUpdate*. Also, in our implementation of the *backupResolve* a node selects the proper routing candidate from its pointer list that is identified by the level and direction of the search. If the selected routing candidate is offline, it is removed from the pointer list and the list’s tail is updated by appending the immediate neighbor of the current tail. The *backupResolve* is invoked repeatedly in this manner until an online routing candidate is found, or no more immediate neighbor of the tail is available to be alternatively appended to the list of pointers.

6.3.2 Availability Prediction

DBG [22]: We implement DGB in similar way as *SW-DBG* with the current state window is being removed and only one DBG with fixed state size is employed.

Lifetime [47]: We compute the availability probability of each node at each time slot as the fraction of its accumulative session lengths to the total number of elapsed time slots since the beginning of the simulation.

LUDP [48]: We quantified the availability probability of each node as a function of its age (i.e., the overall number of time slots it has been online) as well as its total number of incoming connections as shown by Equation 8 where op_t is the online probability of the node at the t^{th} time slot, T_o is the age of the node, and Num_{in} denotes the total number of incoming connections of the node.

$$op_t = \frac{T_o \times Num_{in}}{t \times n} \quad (8)$$

7 Simulation Setup

To simulate and evaluate the churn stabilization solutions, we extended the Skip Graph simulator SkipSim [20] by enabling arrival, searching, and departure under the crash-failure model where nodes depart the system without notifying their overlay neighbors. Among the existing churn models of P2P systems,

we found the BitTorrent-based models in [24] stronger, more realistic, reliable, parametrically clearer than the others. Therefore, we applied the same Debian churn distribution as [24] on SkipSim. Debian churn model follows a Weibull distribution with the average session length of 2.71 hours, and the average interarrival time of 39.86 seconds. We implemented *Interlaced*, *SW-DBG* as well as the best existing churn stabilization and availability prediction approaches that are applicable to our system model, and simulated all approaches under the specified Debian churn model. The average downtime of the nodes (i.e., average offline time) is correlated with the size of the system. We provide analysis over downtime in Section 8.

We consider the time as discrete with fixed time slots of one hour. The Skip Graph is initially empty of nodes. At the beginning of each time slot, some nodes arrive the system, join the Skip Graph overlay, and start interacting with others via the search for numerical ID protocol. Such interaction is the basic operation of our system model that is required for example by the replication [5, 7, 8], or aggregation [49] protocols. The number of arrivals to the system is determined by the interarrival time distribution. Likewise, in each time slot, SkipSim selects the number initiated searches uniformly between $[0, \binom{n_o}{2}]$ where n_o denotes the number of online nodes in that time slot. The initiator and target numerical IDs of each search are uniformly selected from the set of online nodes in that time slot. Each node is only available for a limited number of time slots that follows the session length distribution of the churn model and leaves the system once its session length is over. In SkipSim a node departs the system at the end of the time slot that its session length terminates, and becomes offline.

We simulated each algorithm for 100 randomly generated topologies, each with the system capacity of 1024 nodes. Each topology was simulated for one week (i.e., 168 time slots). In order to simulate pinging, each time during a search a node checks the online status of another node, SkipSim adds the corresponding round trip time (RTT) to the total search time.

8 Analytical and Performance Results

8.1 Analytical Framework

We present a framework that analyzes the success probability of *Interlaced* as a function of the backup size b on routing search messages under uniform churn model. In the uniform churn model, at any time, the probability that a given node goes offline is a constant denoted by q , which is independent of the failure of other nodes. With this framework, we provide a conservative estimation of backup size to achieve the maximum success ratio of searches, and support it with experimental results that are presented in Section 8. In our framework, we assume a P2P system of size n i.e., n nodes with unique numerical IDs from 1 to n . For each node, we model the sample space for choosing a random neighbor as the set of all piggybacked identifiers of other nodes that the node receives upon routing search requests. Moreover, to generalize the sample space for

every chosen node, we assume that the node routes enough search requests for the sample space size to converge to n . Let X be a random variable denoting a uniformly chosen numerical ID from the set $[1, n]$. The probability of X has a certain numerical ID of x is denoted by Equation 9.

$$Pr(X = x) = \frac{1}{n} \quad (9)$$

Let T be another random variable denoting a uniformly chosen search for numerical ID target from the set of numerical IDs of all the nodes in system. Once node x is determined, the probability that a search for a randomly chosen numerical ID of t in the right direction reaches x is denoted by Equation 10. Right direction of the search corresponds to $x \leq t$ in the conditional probability of Equation 10. The probability is taken over the set of all numerical IDs that are greater than x and less than t . The right search direction is assumed without loss of generality, and a similar analysis is applicable to the left direction.

$$Pr(T = t \wedge x \leq t | X = x) = \frac{1}{n - x + 1} \quad (10)$$

We start by the simplest scenario where a node x holds only one right neighbor address. Let Y be another random variable that corresponds to the numerical ID of the sole right neighbor of node x . We define $p_{x,t}$ that is shown by Equation 11 as the probability of node Y taking a numerical ID value in $(x, t]$, and hence being a proper routing candidate for node x to forward the search for numerical ID of t to it. In other words, $p_{x,t}$ is the probability of the search successfully passing x and proceeding on right direction towards t via node Y .

$$p_{x,t} = Pr(x \leq Y \leq t | X = x, T = t, x \leq t) = \frac{t - x}{n} \quad (11)$$

We define p as the marginal probability of $p_{x,t}$ with respect to both x and t i.e., the probability of a uniformly chosen node as the right neighbor of a node on a search path for a target numerical ID being a proper routing candidate. By being a routing candidate, we mean that the numerical ID of the chosen neighbor lays between the node's and the target's numerical IDs. As shown by Equations 12-14, p is directly derived by taking the sum of products of Equations 9, 10, and 11 over all possible values of x and t . As x moves over the numerical ID domain of $[1, n]$, t varies between $[x, n]$, which implies that the search target t should be greater than or equal to x .

$$p = \sum_{x=0}^n \sum_{t=x}^n p_{x,t} \times Pr(T = t | X = x, x \leq t) \times Pr(X = x) \quad (12)$$

$$= \sum_{x=0}^n \sum_{t=x}^n \frac{t - x}{n} \times \frac{1}{n - x + 1} \times \frac{1}{n} \quad (13)$$

$$= \frac{1}{n^2} \sum_{x=0}^n \sum_{t=x}^n \frac{t - x}{n - x + 1} \quad (14)$$

We skip the intermediate computations of summation for the sake of space, which yields that the probability p converges to $\frac{1}{4}$ as the system size approaches infinity. This conveys that each node of the system choosing only one of the other nodes uniformly as its right neighbor, the probability that the chosen right neighbor is a routing candidate for a randomized search target is about $\frac{1}{4}$. Assuming the uniform churn model with the failure probability of q turns the probability p to the more realistic $p' = p \times (1 - q) \approx \frac{1-q}{4}$ i.e., the probability in which a uniformly chosen right neighbor for a uniformly chosen node on a search path for a uniformly chosen target is an online routing candidate.

Let random variable Z denotes the number of online routing candidates for a uniformly chosen node on the search path of a uniformly chosen numerical ID among the b many backup neighbors that are chosen uniformly. The uniformly chosen backup neighbors is the weakened version of *Interlaced* that we utilize as a conservative analytical baseline. Z is a Binomial random variable with success probability of p' [26]. We define the failure probability p_f as the probability of having no online routing candidate for a node on a search path of a search for numerical ID as shown by Equation 15.

$$p_f = \binom{b}{0} \times (p')^0 \times (1 - p')^b = (1 - p')^b \quad (15)$$

Treating p_f as a Bernoulli probability, we denote the expected path length of a search for a uniformly chosen numerical ID that leads to failure with E_f as shown by Equation 16. In other words, E_f corresponds to the expected number of nodes that a search message should traverse to face a failure with high probability.

$$E_f = \frac{1}{p_f} = \frac{1}{(1 - p')^b} = \frac{1}{(1 - \frac{1-q}{4})^b} \quad (16)$$

Considering the failure probability of nodes (i.e., q) as a system-wide constant bound, E_f is tweakable with the size of backup table i.e., b . Equation 16 implies that in order to achieve no failure in expectation with high probability, b should be chosen large enough that E_f stays beyond the average search path length of the system.

For example, in the specified Debian churn model (see Section 7), each node has an average session length of about 2.71 hours and then goes offline. Having an average inter-arrival time of 39.86 seconds results in an average of 90 hourly arrivals, which causes an offline node to return back to the system in about 12 hours in expectation. Modeling this behavior with a uniform churn model results in a uniform failure probability of about 0.82 that is analogous to q in our proposed framework. In a system with $n = 1024$ nodes under this uniform churn model, the expected number of online nodes at each time slot is about 184 that is obtained from Equation 17, and is denoted by E_{online} . During the simulation, we consider a node as online within a (one hour) time slot, if it arrives at the system at that time slot.

Prediction Strategy	<i>SW-DBG</i>	DBG(1)	DBG(2)	DBG(3)	DBG(4)	LUDP	Lifetime
Average Prediction Error	0.18	0.28	0.26	0.23	0.21	0.51	0.34

Table 1: Average prediction error of availability prediction strategies under Debian churn model.

$$E_{online} = (1 - q) \times n = 0.18 \times 1024 \approx 184 \quad (17)$$

The search path length in Skip Graph is asymptotically logarithmic in the number of nodes i.e., a Skip Graph with n nodes experiences search paths' length of $O(\log n)$ nodes. Having an average of about 184 online nodes in the Skip Graph at every time slot, we approximate the lower bound on the average search path length as $\lceil \log 184 \rceil = 8$ nodes. Assuming $E_f = 8$ and $q = 0.82$, we obtain $b \approx 40$ from Equation 16 as an estimate on the backup size that maximizes the average success ratio of searches under the specified Debian churn model.

8.2 Performance Results

Availability Prediction: Table 1 represents a comparison between the average prediction error of our proposed *SW-DBG* and the existing availability prediction solutions. For the sake of comparison, we measure the availability prediction error as the average difference between the nodes' predicted availability probability, and their availability status. When a node is offline, its availability status is equal to 0, and when it becomes online, its availability status is 1. The average is taken over all the simulation's time slots. $DBG(x)$ denotes the DBG implementation with the state size of x . Compared to $DBG(4)$ that performs as the best existing availability predictor, ***SW-DBG predicts the availability of nodes with about 1.11 times more accuracy*** under the Debian churn model. Growing the state size of DBGs beyond 4-bits increases their state update running time exponentially in their state size, and is not applicable to our simulation scale.

Average Success Ratio: Consistent with existing churn stabilization studies like [13], we consider the average success ratio of searches as the connectivity performance of Skip Graph overlay under churn. Figure 5.a shows connectivity performance of churn stabilization approaches in the specified Debian churn model as the backup size (i.e., the b parameter) increases. *Interlaced- x* represents the setup with *Interlaced* as the churn stabilization approach, and the availability prediction strategy denoted by x . As illustrated in Figure 5.a, there is a similar connectivity performance pattern among almost all the *Interlaced- x* approaches i.e., the success ratios start by a fast growth for smaller backup sizes up to a common breakpoint in which the growth rate slows down and gets steady. For example, *Interlaced-SW-DBG* experiences a drastic increase of success ratio in moving from the backup size of 10 to 20 (nodes) followed by a narrow increase after passing the breakpoint of 20, and converges to a steady state value of success ratio of about 0.9. As detailed earlier, we obtain $b \approx 40$ from our analytical framework as an estimate on the backup size that maximizes the average success ratio of searches under the specified Debian churn model.

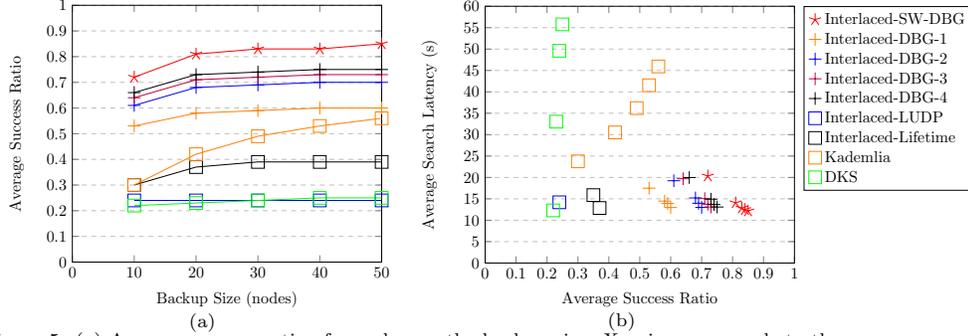


Figure 5: (a) Average success ratio of searches vs the backup size. X-axis corresponds to the backup size (i.e., b), and Y-axis corresponds to the average success ratio of the searches. The desired data points are the ones towards the top right corner, which correspond to a higher connectivity of overlay under churn, compared to the rest. (b) Average search latency vs average success ratio of the searches. X-axis corresponds to the average success ratio, and Y-axis corresponds to the average search latency in seconds. The desired data points are the ones towards the bottom right corner, which correspond to a higher connectivity of overlay under churn at a lower latency, compared to the rest.

As shown in Figure 5.a, for almost all of the *Interlaced- x* approaches, backup size of 40 yields a success ratio in the steady-state part of the graph that converges to the maximum value. $b = 40$ is a conservative estimate on backup size, and similar connectivity performance is obtainable with even lower backup sizes (e.g., $b = 20$). However, it is yet a proper estimator to maximize the connectivity performance of the *Interlaced- x* approaches under the characterized churn model.

The scoring strategy of *Interlaced* on the backup neighbors is directly affected by the underlying availability prediction accuracy. Supported by Table 1, the prediction error of DBGs narrows down as their state size increases, which results in a direct relationship between the average success ratio of *Interlaced-DBG* approaches and the state size of DBG. *Interlaced-Lifetime* follows the breakpoint pattern but at a lower success ratio rate compared to the DBG-based approaches. This follows from the Lifetime prediction mechanism’s inferiority compared to the DBG-based approaches in predicting the availability of nodes. *Interlaced-LUDP* is the only *Interlaced- x* approach that does not follow the breakpoint pattern. This is due to the 0.51 prediction error margin of LUDP, which provides a completely randomized prediction of availability, and results in *Interlaced-LUDP* to perform as the weakest compared to the rest. DKS acts closely to LUDP as the second weakest, which is due to failure on recovering from the concurrent departures of consecutive neighbors. Higher backup sizes increase the success ratio of Kademlia with more routing candidates provided. However, the coarse-grained approach of Kademlia on replacing the oldest entry with the new candidate as well as keeping fixed size backup list at each level results in the lack of routing candidates within proximity of the nodes in the identifier space. This makes Kademlia perform less efficiently in the lower backup sizes, and converge to *Interlaced-DBG(1)* in the higher backup sizes. Benefiting from *SW-DBG* that adaptively chooses the best state size for

each node, as well as scoring the backup table elements based on both their numerical ID distances and availability behavior features *Interlaced-SW-DBG* the best among the others. Compared to the Kademia that acts as the best existing solution applicable on Skip Graph, ***Interlaced-SW-DBG* improves the average success ratio of the searches with the gain of about 1.81 times on average.**

Average Search Latency: Figure 5.b shows the the average search latency of Skip Graph under churn versus the average success ratio of the search for different churn stabilization approaches. A point (x, y) on this figure is interpreted as *y is the best average search latency that is provided by the corresponding churn stabilization approach to maintain the average search success ratio of x.* The desired data points in Figure 5.b are the ones towards the bottom right corner, which correspond to a higher connectivity of overlay under churn at a lower latency, compared to the rest. There exists a correlation between the prediction error of the availability predictors (Table 1) and the average search latency (Figure 5.b). A higher accuracy of availability prediction yields in maintaining a higher number of likely available backup neighbors. This lowers the expected number of timeouts that contribute to the overall search latency, which reduces the average search latency. Ignoring the availability of neighbors in DKS significantly increases its search latency as the backup size grows (i.e., larger success ratios of search). A similar pattern occurs for Kademia with loosened availability constraint on the selection of routing candidates. Affected by the immense availability prediction error of LUDP, the *Interlaced-LUDP* approach initially keeps likely unavailable backup neighbors that are later dropped by *Interlaced* due to their timeout failures. Therefore, as the time goes on, the majority of nodes are with depleted back up tables, which conclude the search faster but with the wrong result. This is why *Interlaced-LUDP* seems faster than the rest but at a miserably lower average success ratio of the search. Similar to the connectivity performance, our proposed ***Interlaced-SW-DBG* outperforms Kademia by performing the searches 1.81 times more successful, and 2.47 times faster on average, and acts as the best among all the existing counterparts.**

8.3 Space, Time, and Communication Complexities

SW-DBG: As DBGs are sorted in ascending order based on their state size in the current state window, the size of *SW-DBG* is bound by the Right DBG’s size. Having the state size of k for the Right DBG in the current state window, the asymptotic time and memory complexity of *statusUpdate* (Algorithm 1) is $O(2^k)$. Despite this exponential asymptotic upper bound, based on our simulation, the average state size of the Right DBG is about 3.6 bits and does not cross the 5 bits. The standard deviation of average Right DBG state size of *SW-DBG* is about 0.2. Representing each vertex by a k -bit state string and an integer probability value between 0 and 100 applies an average and maximum memory overhead of about 64 and 256 bytes on each node that utilizes *SW-*

DBG. Running on Intel i5 2.60 GHz CPU and 8 GB of RAM, a single execution of *statusUpdate* takes the average running time of 1.45 milliseconds. *SW-DBG* does not impose any communication overhead on the system.

Interlaced: The only memory overhead of *Interlaced* on a node is a backup table size of $O(b)$ where b is a system wide constant that denotes the maximum backup size. As each of the *Interlaced* event handlers iterates over the backup table either partially (i.e., one entry set) or entirely, the worst-case asymptotic running time of *Interlaced* is $O(b)$. The worst-case communication complexity of *backupUpdate* and *backupResolve* are $O(1)$ and $O(b)$, respectively. The asymptotic $O(b)$ worst case communication complexity of *backupResolve* occurs when all the backup neighbors are placed in a single entry set, and are contacted one by one on resolving a failure. However, the expected number of backup neighbors for each level of a node is $\frac{b}{\log n}$, which applies the expected communication complexity of $O(\frac{b}{\log n})$. As long as $b = O(\log^2 n)$, *Interlaced* does not change the communication complexity of the Skip Graph overlay. Based on the simulation results, for the backup size of $b = 50$, the average number of backup neighbors for each level of Skip Graph remains close to its expected value (i.e., an average of about 3.32), that imposes the average communication complexity of 1.55 messages per invocation of *backupResolve* by each node on a search path.

9 Conclusion

To maximize the connectivity of Skip Graph-based DHT overlays under churn we proposed *Interlaced*, a fully decentralized predictive churn stabilization algorithm that provides fine-grained scoring mechanisms based on the distribution of nodes in both the overlay and identifier space, as well as their availability probability. *Interlaced* does not change the asymptotic communication complexity. As an independent contribution, we proposed *SW-DBG*, a tool to predict the availability probability of the nodes.

We extended the Skip Graph simulator, SkipSim [20], implemented and simulated the state-of-the-art availability prediction methods as well as churn stabilization approaches that are applicable on a Skip Graph overlay. Our simulation results show that compared to the best existing solutions that are applicable on a Skip Graph overlay, *Interlaced* improves the connectivity of the Skip Graph overlay under churn with the gain of about 1.81 times. Likewise, compared to the existing availability prediction approaches for P2P systems, *SW-DBG* is about 1.11 times more accurate. A Skip Graph that benefits from *Interlaced* and *SW-DBG* is about 2.47 times faster on average in processing the search queries under churn compared to the best existing solutions.

Acknowledgement

The authors thank Muharrem Salel for his contributions to the SkipSim implementation.

References

- [1] J. Aspnes and G. Shah, “Skip graphs,” in *ACM TALG*, 2007.
- [2] E. Udoh, *Cloud, grid and high performance computing: emerging applications*. Information Science Reference, 2011.
- [3] S. Batra and A. Singh, “A short survey of advantages and applications of skip graphs,” 2013.
- [4] T. Shabeera, P. Chandran, and S. Kumar, “Authenticated and persistent skip graph: a data structure for cloud based data-centric applications,” in *ACM CSS 2012*.
- [5] Y. Hassanzadeh-Nazarabadi, A. K p c , and  .  zkasap, “Awake: decentralized and availability aware replication for p2p cloud storage,” in *Smart Cloud*. IEEE, 2016.
- [6] —, “Locality aware skip graph,” in *IEEE ICDCSW, 2015*.
- [7] —, “Laras: Locality aware replication algorithm for the skip graph,” in *IEEE NOMS*, 2016.
- [8] —, “Decentralized and locality aware replication method for dht-based p2p storage systems,” in *Future Generation Computer Systems*. Elsevier, 2018.
- [9] S. Taheri Boshrooyeh and  .  zkasap, “Guard: Secure routing in skip graph,” in *IFIP Networking*, 2017.
- [10] R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. T ubig, “Skip+: A self-stabilizing skip graph,” in *Journal of the ACM*. ACM, 2014.
- [11] M. T. Goodrich, M. J. Nelson, and J. Z. Sun, “The rainbow skip graph: a fault-tolerant constant-degree distributed data structure,” in *ACM-SIAM SODA*. ACM, 2006.
- [12] T. Toda, Y. Tanigawa, and H. Tode, “Autonomous and distributed construction of locality aware skip graph,” in *CCNC*. IEEE, 2017.
- [13] O. Herrera and T. Znati, “Modeling churn in p2p networks,” in *ANSS*. IEEE, 2007.
- [14] A. G. Medrano-Ch vez, E. P rez-Cort s, and M. Lopez-Guerrero, “A performance comparison of chord and kademia dhTs in high churn scenarios,” in *Peer-to-Peer Networking and Applications*. Springer, 2015.
- [15] S. Rhea, D. Geels, T. Roscoe, J. Kubiawicz *et al.*, “Handling churn in a dht,” in *ATC*. USENIX, 2004.

- [16] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek, “Comparing the performance of distributed hash tables under churn,” in *Third international conference on Peer-to-Peer Systems*. Springer-Verlag, 2004.
- [17] P. Maymounkov and D. Mazieres, “Kademlia: A peer-to-peer information system based on the xor metric,” in *International Workshop on Peer-to-Peer Systems*. Springer, 2002.
- [18] H. Heck, O. Kieselmann, and A. Wacker, “Evaluating connection resilience for the overlay network kademlia,” in *arXiv preprint arXiv:1703.09171*, 2017.
- [19] Z. Trifa and M. Khemakhem, “Effects of churn on structured p2p overlay networks,” in *ACECS*, 2014.
- [20] “Skipsim: <https://github.com/yaahaanaa/skipsim>.”
- [21] F. de Bruijn, “A combinatorial problem,” in *The Section of Sciences*, 1946.
- [22] J. W. Mickens and B. D. Noble, “Exploiting availability prediction in distributed systems,” in *Ann Arbor*, 2006.
- [23] —, “Improving distributed system performance using machine availability prediction,” in *ACM SIGMETRICS Performance Evaluation Review*, 2006.
- [24] D. Stutzbach and R. Rejaie, “Understanding churn in peer-to-peer networks,” in *SIGCOMM*. ACM, 2006.
- [25] K. Ramachandran, H. Lutfiyya, and M. Perry, “Decentralized approach to resource availability prediction using group availability in a p2p desktop grid,” in *Future Generation Computer Systems*. Elsevier, 2012.
- [26] D. P. Bertsekas and J. N. Tsitsiklis, *Introduction to probability*. Athena Scientific Belmont, MA, 2002.
- [27] L. R. Monnerat and C. L. Amorim, “D1ht: a distributed one hop hash table,” in *IPDPS*. IEEE, 2006.
- [28] B. Leong, B. Liskov, and E. D. Demaine, “Epichord: Parallelizing the chord lookup algorithm with reactive routing state management,” in *Computer Communications*. Elsevier, 2006.
- [29] R. Kaur, A. L. Sangal, and K. Kumar, “Churn handling strategies for structured overlay networks: A survey,” in *Multiagent and Grid Systems*. IOS Press, 2017.
- [30] —, “A persistent structured hierarchical overlay network to counter intentional churn attack,” in *Journal of Computer Networks and Communications*. Hindawi Publishing Corp., 2016.

- [31] J. Sacha, J. Dowling, R. Cunningham, and R. Meier, "Discovery of stable peers in a self-organising peer-to-peer gradient topology," in *IFIP DIAS*. Springer, 2006.
- [32] T. Clouser, M. Nesterenko, and C. Scheideler, "Tiara: A self-stabilizing deterministic skip list and skip graph," in *Theoretical Computer Science*. Elsevier, 2012.
- [33] X. Meng, X. Chen, and Y. Ding, "Using the complementary nature of node joining and leaving to handle churn problem in p2p networks," in *Computers & Electrical Engineering*. Elsevier, 2013.
- [34] G. Ghinita and Y. M. Teo, "An adaptive stabilization framework for distributed hash tables," in *IPDPS*. IEEE, 2006.
- [35] 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," in *IEEE Journal on selected areas in communications*, 2004.
- [36] R. R. Paul, P. Van Roy, and V. Vlassov, "Interaction between network partitioning and churn in a self-healing structured overlay network," in *IEEE ICPADS*, 2015.
- [37] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek, "Bandwidth-efficient management of dht routing tables," in *NSDI*. USENIX Association, 2005.
- [38] L. Onana Alima, S. El-Ansary, P. Brand, and S. Haridi, "Dks (n, k, f): a family of low communication, scalable and fault-tolerant infrastructures for p2p applications," in *CCGrid*. IEEE COMPUTER SOC, 2003.
- [39] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM Computer Communication Review*, 2001.
- [40] J. S. Kong, J. S. Bridgewater, and V. P. Roychowdhury, "Resilience of structured p2p systems under churn: The reachable component method," in *Computer Communications*. Elsevier, 2008.
- [41] M. Hojo, R. Banno, and K. Shudo, "Frt-skip graph: A skip graph-style structured overlay based on flexible routing tables," in *ISCC*. IEEE, 2016.
- [42] A. Pace, V. Quema, and V. Schiavoni, "Exploiting node connection regularity for dht replication," in *SRDS*. IEEE, 2011.
- [43] G. Song, S. Kim, and D. Seo, "Replica placement algorithm for highly available peer-to-peer storage systems," in *AP2PS*. IEEE, 2009.
- [44] R. Kaur, A. L. Sangal, and K. Kumar, "Modeling and simulation of adaptive neuro-fuzzy based intelligent system for predictive stabilization in structured overlay networks," in *Engineering Science and Technology, an International Journal*. Elsevier, 2017.

- [45] D. Lázaro, D. Kondo, and J. M. Marquès, “Long-term availability prediction for groups of volunteer resources,” in *Journal of Parallel and Distributed Computing*. Elsevier, 2012.
- [46] R. Kaur, A. L. Sangal, and K. Kumar, “Performance analysis of predictive stabilization for churn handling in structured overlay networks,” in *BDAW*. ACM, 2016.
- [47] D. Leonard, Z. Yao, V. Rai, and D. Loguinov, “On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks,” in *IEEE/ACM TON (Transactions on Networking)*. IEEE Press, 2007.
- [48] F. E. Bustamante and Y. Qiao, “Designing less-structured p2p systems for the expected high churn,” in *IEEE/ACM TON (Transactions on Networking)*, 2008.
- [49] Y. Hassanzadeh-Nazarabadi and Ö. Özkasap, “Elats: Energy and locality aware aggregation tree for skip graph,” in *BlackSeaCom*. IEEE, 2017.