Locally self-adjusting distributed algorithms

Sikder Rezwanul Huq
University of Iowa

Copyright © 2018 Sikder Rezwanul Huq

This dissertation is available at Iowa Research Online: https://ir.uiowa.edu/etd/6594

Recommended Citation

Follow this and additional works at: https://ir.uiowa.edu/etd
Part of the Computer Sciences Commons
LOCALLY SELF-ADJUSTING DISTRIBUTED ALGORITHMS

by

Sikder Rezwanul Huq

A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Computer Science in the Graduate College of The University of Iowa

December 2018

Thesis Supervisor: Professor Sukumar Ghosh
This is to certify that the Ph.D. thesis of

Sikder Rezwanul Huq

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the December 2018 graduation.

Thesis committee:

Sukumar Ghosh, Thesis Supervisor

Ted Herman

Kasturi Varadarajan

Octav Chipara

Zubair Shafiq
ACKNOWLEDGEMENTS

First and foremost, I want to express my sincere gratitude to my advisor and mentor, Professor Sukumar Ghosh, for all his supports and encouragement throughout my Ph.D. studies. Without his endless patience and guidance, this thesis would not have been possible. He was always there to help me through thick and thin. It has been an honor to have him as my advisor.

Besides my advisor, I would like to thank the rest of my thesis committee members for their encouragement, insightful comments and critical questions which motivated me to widen and deepen my research from different perspectives.

I would also like to thank Professor Kasturi Varadarajan, Professor Sriram Pemmaraju, Professor Sukumar Ghosh, Professor Ted Herman, and Professor Octav Chipara for their various graduate level courses that gave me the foundation of doing research in this area.

I owe a big thanks to Professor Zubair Shafiq. It was an amazing and valuable experience to work and coauthor a paper with him.

My heartiest thanks goes to all the members of the Distributed Computing Reading Group and the Algorithms Reading Group at the Computer Science department. The presentations and discussions in these group meetings helped me to shape my perspective of research in distributed computing. I also want to thank all the members of the Computer Science department, including faculty members, program administrators, students and my fellow classmates. I am grateful to this department for giving me the opportunity to teach
various undergraduate courses as a primary instructor. These experiences would be invaluable for my future career.

Finally, I am indebted to my parents, Justice Sikder Maqbul Huq and Begum Qumrunnessa Huq and to all my family members and close friends for their constant encouragement. Special thanks to my lovely wife Tunajjina Jalil for her love, patience and endless support.
ABSTRACT

In this dissertation, we study self-adjusting algorithms for large-scale distributed systems. Self-adjusting algorithms enable distributed systems to adjust their properties dynamically as the input pattern changes. Self-adjustment is an attractive tool as it has the potential to significantly improve the performance of distributed systems, especially when the input patterns are skewed.

We start with a distributed self-adjusting algorithm for skip graphs that minimizes the average routing costs between arbitrary communication pairs by performing topological adaptation to the communication pattern. Our algorithm is fully decentralized, conforms to the $CONGEST$ model (i.e. uses $O(\log n)$ bit messages), and requires $O(\log n)$ bits of memory for each node, where $n$ is the total number of nodes. Upon each communication request, our algorithm first establishes communication by using the standard skip graph routing, and then locally and partially reconstructs the skip graph topology to perform topological adaptation. We propose a computational model for such algorithms, as well as a yardstick (working set property) to evaluate them. Our working set property can also be used to evaluate self-adjusting algorithms for other graph classes where multiple tree-like subgraphs overlap (e.g. hypercube networks). We derive a lower bound of the amortized routing cost for any algorithm that follows our model and serves an unknown sequence of communication requests. We show that the routing cost of our algorithm is at most a constant factor more than the amortized routing cost of any algorithm conforming to our computational model. We also show that the expected transformation cost for our
algorithm is at most a logarithmic factor more than the amortized routing cost of any algorithm conforming to our computational model.

As a follow-up work, we present a distributed self-adjusting algorithm (referred to as DyHypes) for topological adaption in hypercubic networks. One of the major differences between hypercubic networks and skip graphs is that hypercubic networks are more rigid in structure than that of skip graphs. This property of hypercubic networks makes self-adjustment significantly different compared to skip graphs. Upon a communication between an arbitrary pair of nodes, DyHypes transforms the network to place frequently communicating nodes closer to each other to maximize communication efficiency, and uses randomization in the transformation process to speed up the transformation and reduce message complexity. We show that, as compared to DSG, DyHypes reduces the transformation cost by a factor of $O(\log n)$, where $n$ is the number of nodes involved in the transformation. Moreover, despite achieving faster transformation with lower message complexity, the combined cost (routing and transformation) of DyHypes is at most a log log $n$ factor more than that of any algorithm that conforms to the computational model adopted for this work. Similar to DSG, DyHypes is fully decentralized, conforms to the $CONGEST$ model, and requires $O(\log n)$ bits of memory for each node, where $N$ is the total number of nodes.

Finally, we present a novel distributed load balancing algorithm called Meezan to address the load imbalance among large-scale networked cache servers. Modern web services rely on a network of distributed cache servers to efficiently deliver content to users. Load imbalance among cache servers can substantially degrade content delivery perfor-
mance. Due to the skewed and dynamic nature of real-world workloads, cache servers that serve viral content experience higher load as compared to other cache servers. Our algorithm Meezan replicates popular objects to mitigate skewness and adjusts hash space boundaries in response to load dynamics in a novel way. Our theoretical analysis shows that Meezan achieves near perfect load balancing for a wide range of operating parameters. Our trace driven simulations shows that Meezan reduces load imbalance by up to 52% as compared to prior solutions.
In this dissertation, we study self-adjusting algorithms for large-scale distributed systems. Self-adjusting algorithms enable distributed systems to adjust their properties dynamically as the input pattern changes. Self-adjustment is an attractive tool as it has the potential to significantly improve the performance of distributed systems, especially when the input patterns are skewed.

We start with a distributed self-adjusting algorithm for skip graphs that minimizes the average routing costs between arbitrary communication pairs by performing topological adaptation to the communication pattern. Our algorithm is fully decentralized, conforms to the $CONGEST$ model (i.e. uses $O(\log n)$ bit messages), and requires $O(\log n)$ bits of memory for each node, where $n$ is the total number of nodes. Upon each communication request, our algorithm first establishes communication by using the standard skip graph routing, and then locally and partially reconstructs the skip graph topology to perform topological adaptation. We propose a computational model for such algorithms, as well as a yardstick (working set property) to evaluate them. Our working set property can also be used to evaluate self-adjusting algorithms for other graph classes where multiple tree-like subgraphs overlap (e.g. hypercube networks). We derive a lower bound of the amortized routing cost for any algorithm that follows our model and serves an unknown sequence of communication requests. We show that the routing cost of our algorithm is at most a constant factor more than the amortized routing cost of any algorithm conforming to our computational model. We also show that the expected transformation cost for our
algorithm is at most a logarithmic factor more than the amortized routing cost of any algorithm conforming to our computational model.

As a follow-up work, we present a distributed self-adjusting algorithm (referred to as DyHypes) for topological adaption in hypercubic networks. One of the major differences between hypercubic networks and skip graphs is that hypercubic networks are more rigid in structure than that of skip graphs. This property of hypercubic networks makes self-adjustment significantly different compared to skip graphs. Upon a communication between an arbitrary pair of nodes, DyHypes transforms the network to place frequently communicating nodes closer to each other to maximize communication efficiency, and uses randomization in the transformation process to speed up the transformation and reduce message complexity. We show that, as compared to DSG, DyHypes reduces the transformation cost by a factor of $O(\log n)$, where $n$ is the number of nodes involved in the transformation. Moreover, despite achieving faster transformation with lower message complexity, the combined cost (routing and transformation) of DyHypes is at most a $\log \log n$ factor more than that of any algorithm that conforms to the computational model adopted for this work. Similar to DSG, DyHypes is fully decentralized, conforms to the $CONGEST$ model, and requires $O(\log n)$ bits of memory for each node, where $N$ is the total number of nodes.

Finally, we present a novel distributed load balancing algorithm called Meezan to address the load imbalance among large-scale networked cache servers. Modern web services rely on a network of distributed cache servers to efficiently deliver content to users. Load imbalance among cache servers can substantially degrade content delivery perfor-
mance. Due to the skewed and dynamic nature of real-world workloads, cache servers that serve viral content experience higher load as compared to other cache servers. Our algorithm Meezan replicates popular objects to mitigate skewness and adjusts hash space boundaries in response to load dynamics in a novel way. Our theoretical analysis shows that Meezan achieves near perfect load balancing for a wide range of operating parameters. Our trace driven simulations shows that Meezan reduces load imbalance by up to 52% as compared to prior solutions.
# TABLE OF CONTENTS

LIST OF TABLES ................................................. xiii

LIST OF FIGURES ............................................... xiv

CHAPTER

1 INTRODUCTION ............................................... 1

1.1 Background .................................................. 1

1.2 Topological Self-Adjustment ............................... 2

1.2.1 Motivation ............................................... 2

1.2.2 Related Work ............................................. 4

1.2.3 Our Contribution ........................................ 5

1.2.3.1 Locally Self-Adjusting Skip Graphs................ 5

1.2.3.2 Locally Self-Adjusting Hypercubic Networks ....... 7

1.3 Dynamic Hash Space Adjustment in Networked Caches ... 7

1.3.1 Motivation ............................................... 7

1.3.2 Related Work ............................................. 9

1.3.3 Our Contribution ........................................ 11

1.4 Thesis Structure ............................................ 11

2 SELF-ADJUSTING MODEL FOR TOPOLOGICAL ADAPTATION AND
THE WORKING SET PROPERTY ............................... 12

2.1 Skip Graph Model ......................................... 12

2.2 Hypercube Model .......................................... 15

2.3 Self-Adjusting Model for Skip Graphs and Hypercubic Networks .... 16

2.4 Working Set Property ...................................... 17

3 LOCALLY SELF-ADJUSTING SKIP GRAPHS ................. 21

3.1 Dynamic Skip Graphs (DSG) .............................. 21

3.1.1 Overview ............................................... 21

3.1.2 Setup and Notations .................................... 23

3.1.3 Transformation From \( S_t \) to \( S_{t+1} \) ............. 25

3.1.4 Assignment of New Group-ids ......................... 30

3.1.5 Assignment of New Timestamps ....................... 31

3.1.6 Maintaining the a-balance Property .................. 33

3.1.7 Node Addition/Removal ............................... 35

3.2 Approximate Median Finding for Skip Graphs (AMF) ...... 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3 Analysis</td>
<td>40</td>
</tr>
<tr>
<td>3.4 Conclusion</td>
<td>46</td>
</tr>
<tr>
<td>4 LOCALLY SELF-ADJUSTING HYPERCUBIC NETWORKS</td>
<td>48</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>48</td>
</tr>
<tr>
<td>4.2 Proposed Algorithm</td>
<td>49</td>
</tr>
<tr>
<td>4.2.1 Algorithm Overview and Notations</td>
<td>49</td>
</tr>
<tr>
<td>4.2.2 Major Algorithmic Challenges</td>
<td>52</td>
</tr>
<tr>
<td>4.2.3 Our Key Idea to Address the Challenges</td>
<td>53</td>
</tr>
<tr>
<td>4.2.4 The Algorithm: Dynamic Hypercubic Networks (DYHYPES)</td>
<td>54</td>
</tr>
<tr>
<td>4.2.4.1 Subtree Rotation</td>
<td>56</td>
</tr>
<tr>
<td>4.2.4.2 Inter-Group Transformation</td>
<td>58</td>
</tr>
<tr>
<td>4.2.4.3 Intra-Group Transformation</td>
<td>59</td>
</tr>
<tr>
<td>4.3 Analysis</td>
<td>64</td>
</tr>
<tr>
<td>4.4 Conclusion</td>
<td>76</td>
</tr>
<tr>
<td>5 DYNAMIC HASH SPACE ADJUSTMENT IN NETWORKED CACHES</td>
<td>77</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>77</td>
</tr>
<tr>
<td>5.2 Preliminaries</td>
<td>80</td>
</tr>
<tr>
<td>5.2.1 Design Goal</td>
<td>80</td>
</tr>
<tr>
<td>5.2.2 Background &amp; Motivation</td>
<td>80</td>
</tr>
<tr>
<td>5.2.2.1 Consistent Hashing</td>
<td>80</td>
</tr>
<tr>
<td>5.2.2.2 Data Replication</td>
<td>82</td>
</tr>
<tr>
<td>5.2.2.3 Hash Space Adjustment</td>
<td>83</td>
</tr>
<tr>
<td>5.2.2.4 Dynamic server addition/removal</td>
<td>86</td>
</tr>
<tr>
<td>5.3 Proposed Approach</td>
<td>87</td>
</tr>
<tr>
<td>5.3.1 Architectural Overview</td>
<td>87</td>
</tr>
<tr>
<td>5.3.2 Solution Overview</td>
<td>88</td>
</tr>
<tr>
<td>5.3.3 Popular Object Replication</td>
<td>89</td>
</tr>
<tr>
<td>5.3.4 Hash Space Adjustment</td>
<td>90</td>
</tr>
<tr>
<td>5.3.4.1 Formation of Random Localities</td>
<td>91</td>
</tr>
<tr>
<td>5.3.4.2 Addition or Removal of Servers</td>
<td>91</td>
</tr>
<tr>
<td>5.3.4.3 Server Positioning and Key Transfer</td>
<td>91</td>
</tr>
<tr>
<td>5.3.5 Implementation Details</td>
<td>94</td>
</tr>
<tr>
<td>5.4 Experimental Results</td>
<td>96</td>
</tr>
<tr>
<td>5.4.1 Data</td>
<td>96</td>
</tr>
<tr>
<td>5.4.2 Performance Metrics</td>
<td>96</td>
</tr>
<tr>
<td>5.4.3 Baselines for Comparison</td>
<td>97</td>
</tr>
<tr>
<td>5.4.4 Experimental Setup</td>
<td>98</td>
</tr>
<tr>
<td>5.4.5 Performance Comparison</td>
<td>99</td>
</tr>
<tr>
<td>5.4.6 Discussions</td>
<td>102</td>
</tr>
<tr>
<td>5.4.7 Summary</td>
<td>102</td>
</tr>
</tbody>
</table>
5.5 Conclusion ................................................................. 103

6 FUTURE WORK ........................................................... 105

   6.1 Distributed Online Balanced Repartitioning ................. 105
   6.2 Topological Self-Adjustment in the Internet of Things (IoT) .. 106

APPENDIX ................................................................. 109

   A ................................................................. 110

      A.1 Standard Skip Graph Routing ......................... 110
      A.2 Updating Group-ids for Levels Below $\alpha$ ............. 110
      A.2.1 Distributed Sum Using a Skip List ................. 111
      A.3 Base Case Calculations for Lemma 6 ............... 112
LIST OF TABLES

Table

3.1 Notations ......................................................... 40
4.1 Results summary .............................................. 49
4.2 Notations .......................................................... 55
LIST OF FIGURES

Figure

2.1 The lowest 3 levels of a skip graph shown in (a) is mapped to an equivalent tree structure in (b). ......................................................... 13

2.2 Tree representation of a hypercubic networks of 8 nodes. ....................... 15

2.3 For the access pattern shown in (a), the working set number for the last communication between \( u \) and \( v \) is 5, as the number of distinct nodes in \( G \) that has a path from either \( u \) or \( v \) is 5 (e,a,k,u and v). ........................................ 19

2.4 A communication graph. Each edge is labeled with the timestamp of the most recent communication between the end nodes. ............................... 20

3.1 For the communication graph \( G \) shown in (a), possible skip graph representations at time 8 and 9 (\( S_8 \) and \( S_9 \)), obtained by DSG, are shown in (b) and (c), respectively. The rounded rectangles show the groups of nodes at different levels, and the number below each node is the timestamp for the node at the corresponding level. For example, in \( S_9 \) (figure c), the group of node \( B \) at level 2 has 3 nodes (\( B \), \( G \), and \( D \)), and the timestamp of node \( B \) for level 2 (i.e. \( T^B_2 \)) is 4. ................................................................. 24

4.1 Tree representation of a hypercubic networks of 8 nodes. Green boxes indicate groups at different levels. Nodes are placed in the incremental order of their membership vectors from left to right. ................................. 51

4.2 Subtree rotation to move nodes A and E in a subtree of size 2. ................. 57

4.3 Intergroup transformation example: group \( B \) moves next to group \( A \), and the size of group \( B \) is larger than the size of the \( \alpha \)-relative group \( C_1 \). ......................... 60

4.4 Intergroup transformation example: group \( B \) moves next to group \( A \), and the size of group \( B \) is smaller than the size of the \( \alpha \)-relative group \( C_1 \). ......................... 61

4.5 Intragroup transformation example for communication \((A, E)\). The green boxes indicate groups at different levels and the numbers below the nodes indicate the node’s timestamp for the corresponding level. ....................... 63
5.1 Front-end web server requests for a data objects to the appropriate cache server. The cache server replies with object if the object is present in cache. Otherwise, the object is copied from the origin storage. .......................... 77

5.2 A ring based hash space for 3 servers, each with 4 virtual nodes. ............... 81

5.3 An example of data replication. Incremental numbers are added to the object’s identifier to distribute the load associated with popular objects across multiple servers. ......................................................... 82

5.4 Performance of different consistent hashing based approaches for an one hour long real-world access log. ............................................................... 84

5.5 An example of hash space adjustment. ......................................................... 85

5.6 An example of server addition/removal. ..................................................... 86

5.7 Illustration of hash space adjustment. ....................................................... 88

5.8 Adjusted hash space in a locality. ............................................................ 93

5.9 Average $max/avg$. ................................................................................... 99

5.10 Average number of servers. ................................................................. 100

5.11 Average data migration (in gigabytes). .................................................. 100

5.12 Average replication overhead (%). ......................................................... 100

5.13 Average number of counters (% of load threshold) per server per time interval for Meezan. ................................................................. 103
CHAPTER 1
INTRODUCTION

1.1 Background

Locally self-adjusting algorithms allow distributed systems to adjust their properties automatically to respond to changes in input patterns. For example, a network of virtual machines (VM) may migrate frequently communicating VMs physically close to each other to maximize communication efficiency. Given that many real world input patterns are highly skewed, self-adjustment in distributed systems has the potential to significantly improve overall performance by treating frequently repeating input patterns differently.

This dissertation focuses on designing, analyzing, and evaluating self-adjusting algorithms for certain distributed systems where input patterns are often skewed. In a broader sense, these algorithms respond to the change in input pattern by dynamically adjusting some system properties (e.g. topology, resource allocation, hash space boundaries etc.) to optimize performance in terms of one or more specified metrics.

For large distributed systems, it is crucial that adjustments take place locally without global knowledge. This can be challenging due to the following reasons. First, a local adjustment must not violate any global consistency required by the system. For example, a self-adjusting binary search tree (BST) must ensure that the overall structure remains a BST after any topological transformation performed locally. Second, a local adjustment must contribute to achieving a global performance improvement.

In this dissertation, we study self-adjustment in distributed systems in two different
directions: (1) topological self-adjustment and (2) dynamic hash space adjustment. We present the motivation, related work and our contributions in both of the directions in the following sections.

### 1.2 Topological Self-Adjustment

Many peer-to-peer communication topologies and distributed data structures are designed to reduce the worst-case time per operation and do not take advantage of the skew in communication patterns. Given that most real-world communication patterns are skewed, self-adjustment is an attractive tool that can significantly reduce the average communication cost for a sequence of communications. For an unknown sequence of communications, self-adjusting algorithms minimize the average communication cost by performing topological adaptation to the communication pattern.

#### 1.2.1 Motivation

In the 1980s, Sleator and Tarjan published their seminal work on *Splay Tree* [56], which has been the inspiration of subsequent studies in self-adjusting algorithms, for example *Tango BSTs* [26], *multi-splay trees* [59], *CBTree* [4], *dynamic skip list* [18] etc. Instead of optimizing traditional metrics such as the worst case search depth, all these data structures are designed *self-adjust* to their usage pattern by moving frequently accessed elements closer to the root. Meaning, these data structures support centralized lookup operations and rely on a tree or tree-like structure.

However, today’s distributed data structures and networks are still optimized toward static metrics such as the longest route between communicating nodes. As many communi-
cation patterns are highly skewed and dynamic, self-adjusting networks have the potential to significantly reduce average communication distances by performing topological adaptation to the communication pattern. To address this, Avin et al. proposed SplayNets [11], which initiated the study of self-adjustment for distributed data structures and networks, where each communication involves a source-destination pair. However, splayNet relies on a single BST structure, and we are not aware of any study of self-adjustment for more complex data structures that relies on the interaction of multiple overlapping tree-like structures (e.g. skip graphs, hypercubic networks). This motivates our work on self-adjustment for such structures.

In addition to minimizing the average/amortized communication distance between communicating nodes, reducing energy waste is another motivation for self-adjustment in large networks. As the modern networks are rapidly growing in size, keeping the cost of power consumption is increasingly becoming a major challenge. The cost of power consumption in data centers in the United States alone was estimated to be 50 billion dollars [52] and this cost is estimated to double in every five years [20]. It is also estimated that routing consumes about 20%-30% of the total energy in data centers [28]. Moreover, estimated energy consumption by routing in network-in-chip (NoC) is even up to 50% of the total energy [49]. These numbers indicate that clever algorithms are necessary to reduce energy consumption caused by routing in large networks. The algorithms presented in this dissertation may be used to reduce energy consumption as they reduce the routing distance between nodes in large networks.
1.2.2 Related Work

Interest in self-adjusting data structures grew out of Sleator and Tarjan’s seminal work on *splay tree* [56] that emphasized the importance of amortized cost and proposed a restructuring heuristics to attain the amortized time bound of $O(\log n)$ per operation. Prior to that, Allen and Munro [5], and Bitner [16] proposed two restructuring heuristics for search trees, but none were efficient in the amortized sense. In [12], Bagchi et al. presented an algorithm for efficient access in *biased skip lists* where non-uniform access patterns are biased according to their weights, and the weights are known. Bose et al. [18] investigated the efficiency of access in skip lists when the access pattern is unknown, and developed a deterministic self-adjusting skip list whose running time matches the working set bound, thereby achieving dynamic optimality. Afek et al. [4] presented a version of self-adjusting search trees, called CBTree, that promote a high degree of concurrency by reducing the frequency of “tree rotation.” Avin et al. [11] presented *SplayNet*, a generalization of Splay tree, where, unlike the splay trees, communication is allowed between any pair of nodes in the tree. In [10], Avin, et al. extended the concept of splay trees to P2P overlay networks of multiple binary search trees (OBST). However, their work addresses a routing variant of the classical splay trees that focuses on the lookup operation only. In [9], Avin et al. presented a greedy policy for self-adjusting grid networks that locally minimizes an objective function by switching positions between neighboring nodes. *SKIP* [37] presented a self-stabilization (not self-adjusting) algorithm for skip graphs. [27] presents some of the early ideas of our work.
1.2.3 Our Contribution

The topologies of both skip graph and hypercubic network relies on many overlapping tree-like structures that interact among each other. We present locally self-adjusting algorithms for both of these networks.

1.2.3.1 Locally Self-Adjusting Skip Graphs

We present a self-adjusting algorithm *Dynamic Skip Graphs* [32], [33] (DSG) for skip graph topologies. A skip graph [7] \( G = (V, E) \) is a distributed data structure and a well-known peer-to-peer communication topology that guarantees \( O(\log n) \) worst-case communication time between arbitrary pairs of nodes, where \( n = |V| \). The major advantage of skip graphs over BSTs is that skip graphs are highly resilient and capable of tolerating a large fraction of node failures. In order to achieve such resilience, skip graphs rely on interactions among \( n \) overlapping skip list structures. In general, topological rearrangement of nodes of one skip list affects the structure of multiple other skip lists. Moreover, since the access pattern in unknown, an adversarial access sequence may incur the worst case communication cost for each of the communication requests. Thus it is important to keep all the skip lists balanced so that the worst case communication cost for any pair of nodes remains logarithmic. Self-adjusting algorithms generally attempt to move frequently communicating nodes closer to each other. However, for skip graphs, such an attempt may result in an imbalance situation and drive other uninvolved nodes away from each other, which makes it challenging to design a self-adjusting algorithm for skip graphs.

Our algorithm DSG is designed for the \( CONGEST \) model (i.e. allowed message
size per link per round is up to $O(\log n)$ bits, and requires $O(\log n)$ bits of memory for each node. We show that, for an unknown communication sequence, the routing cost for DSG is at most a constant factor more than the optimal amortized routing cost, and the expected transformation cost is at most a logarithmic factor more than the amortized cost of any algorithm conforming to our computational model.

Major contributions of this work are listed below:

1. We propose a computational model for self-adjusting skip graphs. Upon each communication request, our model requires any algorithm to transform the topology such that communicating nodes (the source-destination pair) get connected via a direct link. Our model also limits the memory of each node to $O(\log n)$ bits and conforms to the $CONGEST$ model.

2. We propose a working set property to evaluate self-adjusting algorithms for skip graphs or similar distributed data structures.

3. We propose a self-adjusting algorithm Dynamic Skip Graphs (DSG) conforming to our model and analyze its performance.

4. Our algorithm uses a distributed and randomized approximate median finding algorithm (AMF) designed for skip graphs. We show that AMF finds an approximate median in expected $O(\log n)$ rounds.
1.2.3.2 Locally Self-Adjusting Hypercubic Networks

We present a self-adjusting algorithm for hypercubic networks. Hypercubic networks have various applications including parallel computing, computer architecture, routing, peer-to-peer networks etc. In this work, we use the same self-adjusting model that we used for skip graphs. The major contributions are listed below:

1. We show a relationship between our proposed working set property and optimality in hypercubic networks.

2. We propose a self-adjusting algorithm DYHYPES conforming to our model and analyze its performance. We also show that DYHYPES runs faster than DSG and requires less messages for transformation.

1.3 Dynamic Hash Space Adjustment in Networked Caches

1.3.1 Motivation

Distributed key-value networked caches (e.g., memcached [2], redis [3]) are widely used by modern web services. Networked cache systems decrease latencies by caching popular content and also mitigate performance bottlenecks at backend storage. Memcached is used by many large-scale web services such as Facebook, YouTube, Twitter, and Wikipedia. Facebook, for example, deploys more than ten thousand memcached servers which lead up to 10X performance improvement [50, 42].

A key challenge faced by large-scale key-value networked caches is skewed and dynamic workload which can result in significant load imbalance. For instance, viral content in online social media represents a disproportionate fraction of total cache workload
Moreover, peak and trough workload of real-world caches often differ by several orders of magnitude [8, 54]. The load imbalance caused by skewed and dynamic workload can nullify the performance benefit of using cache systems and may even result in performance degradation [31]. Prior research has demonstrated that load imbalance can result in more than 60% degradation in throughput and $3 \times$ degradation in latency [22, 29].

Given that load imbalance caused by skewed and dynamic workload may severely degrade overall system performance, it is crucial to distribute load evenly among cache servers. There are three key technical challenges. First, the time-varying nature of workload requires addition or removal of servers to ensure high resource utilization. Second, such addition or removal of servers involves data migration among cache servers that also needs to be minimized. Finally, centralized solutions to sufficiently address the load imbalance problem are not scalable because the centralized point may cause performance bottlenecks.

Prior work commonly employs consistent hashing [38] for look-up operations, data partitioning, and load balancing. However, consistent hashing does not guarantee load balancing because it simply maps cache servers to random positions in the hash space and it is not sensitive to server load. Poor load balancing results in more than necessary server additions or removals, which in turn results in more data migration.

Two techniques are typically used to mitigate this problem: data replication and hash space adjustment. Data replication aims to replicate popular objects at multiple cache servers [29]. Hash space adjustment aims to adaptively adjust the hash space boundaries between neighboring cache servers [35]. We show that existing data replication and hash
space adjustment techniques do not achieve optimal load balancing. We also show that while data replication and hash space adjustment individually improve load balancing, they still leave much scope for further improvement when not carefully applied together. To the best of our knowledge, prior work lacks distributed data replication and hash space adjustment approaches to address the load imbalance problem.

1.3.2 Related Work

**Consistent Hashing.** Distributed hash tables (DHT) have been widely used in peer-to-peer (P2P) systems (e.g. [57, 40, 53, 24, 25]) for efficient lookup and load balancing. These systems use consistent hashing [38, 39] for data partition. Prior work [31, 8, 35, 29] as well as this paper, show that consistent hashing is not effective in optimizing load balancing for skewed and dynamic workloads. Although our work uses a ring based hash space similar to consistent hashing, the focus of this work is to study additional data replication and hash space adjustment techniques to mitigate load imbalance in networked caches.

**Data Replication.** Data replication [50, 13, 48, 60] is widely used to mitigate the impact of workload skewness. Hong et al. [29] proposed an adaptive data replication technique, which is, to the best knowledge, the most recent work on adaptive data replication designed for distributed key-value cache systems. In contrast to [29], our data replication policy is different because we consider the request frequency of current time interval along with EWMA count to compute salt. Huang et al. [31] investigated the reasons of load imbalance across cache servers using real world traces from Facebook’s Tao cluster [19]. They concluded that existing techniques such as consistent hashing and replication
approaches do not completely address the load skewness. Our work attempts to fill this gap by combining data replication with dynamic hash space adjustment.

**Hash Space Adjustment.** Hwang et al. [35] proposed an adaptive load balancer that allows adjustment in hash space boundaries. Their approach relies on initial positioning of virtual nodes and centralized coordination. In contrast, our hash space adjustment technique is distributed, more aggressive compared to [35], and when combined with our data replication policy outperforms both [29] and [35]. Li et al. [44] proposed a dynamic server provisioning based protocol for memory cache clusters to optimize load balancing and minimize data migration under dynamics. Our approach is different because: (1) we do not use virtual nodes; (2) we allow dynamic shift of hash space boundaries in addition to adding or removing servers; and (3) our approach is fully distributed whereas Proteus is not. Cheng et al. [22] proposed a protocol, called MBal, for multi-threaded servers. MBal uses a configurable resource container called *cachelet* that encapsulates multiple virtual nodes. For server level load balancing, MBal migrates cachelets from one server to another along with other load balancing techniques. Our work differs as we dynamically adjust the hash space boundaries and always migrate data from a server to its neighboring servers. Blink [55] and Couchbase [1] use centralized coordination for remapping keys or buckets to the servers. In contrast, our approach adjusts hash space boundaries in a distributed manner without requiring global coordination.

**Miscellaneous.** LAMA [30], CircularCache [45], and DynaCache [23] proposed adaptive cache controllers that focus on cache eviction policy and/or memory management. Although Meezan results in efficient resource management by reducing replication overhead
and average number of servers, our focus is on reducing load imbalance rather than improving cache hit ratio. AdaptCache [6] proposed an adaptive load balancing technique for distributed caches, however their work is designed for a different architecture, and not applicable to networked caches such as Memcached. Finally, RackOut [51] proposes a rack-scale memory pooling technique to mitigate load imbalance. In contrast, our approach targets architectures where servers cannot directly access other servers’ memory.

1.3.3 Our Contribution

We present our solution Meezan [34], a novel distributed protocol for optimizing load balancing in large-scale networked key-value caches. Meezan dynamically adjusts the hash space boundaries of the cache server and performs data replication to facilitate effective hash space adjustment. Our theoretical analysis shows that Meezan achieves near perfect load balancing for a wide range of operating parameters. Our trace driven simulations show that Meezan reduces load imbalance by up to 52% as compared to prior solutions.

1.4 Thesis Structure

Chapter 2 presents our self-adjusting model for topological adjustment, along with our proposed working set property. Chapter 3 presents a locally self-adjusting algorithm for skip graphs. In Chapter 4, we present a locally self-adjusting algorithm for hypercubic networks. We present a distributed algorithm for dynamic load balancing in networked caches in Chapter 5. Finally, we present some ideas for possible direction of research in Chapter 6.
CHAPTER 2
SELF-ADJUSTING MODEL FOR TOPOLOGICAL ADAPTATION AND THE
WORKING SET PROPERTY

2.1 Skip Graph Model

We begin with a quick introduction of Skip Graphs [7]. A Skip Graph consists of nodes positioned in the ascending order of their identifiers (often called keys) in multiple levels. Level 0 consists of a doubly linked list containing all the nodes. The linked list at level 0 is split into 2 distinct doubly linked lists at level 1. Similarly, each of the linked lists at level 1 is split into 2 distinct linked lists at level 2, and this continues recursively for upper levels until all nodes become singleton. In other words, every linked list with at least 2 nodes at any level \( i \) is split into 2 distinct linked lists at level \( i + 1 \). The number of levels in a skip graph is called the *height* of the skip graph. When a linked list splits into 2 linked list at the next upper level, we denote the split linked lists as "0-sublist" (or 0-subgraph) and "1-sublist" (or 1-subgraph). Note that we refer the base (lowest) level as level 0. We denote the height of a skip graph as \( H \).

For simpler representation, we map a skip graph into a binary tree of linked lists. To this end, the linked list at level 0 is represented by the *root* node of the tree, and the 0-sublist and the 1-sublist at level 1 are represented by the *left child* and *right child* of the root, respectively. Similarly, every linked list of the skip graph is represented by a node in the binary tree, mimicking the parent child relationship of the skip graph in that of its equivalent binary tree. Figure 2.1(a) shows a skip graph with 3 levels, and figure 2.1(b) shows the corresponding binary tree representation.
Each node $x$ in a skip graph has a membership vector $m(x)$ of size $H - 1$. The $i^{th}$ bit of $m(x)$ represents the sublist (0 or 1) that contains node $x$ at level $i$. For example, the
membership vector of node $M$ in figure 2.1(b) is 01, as $M$ belongs to the 0-sublist at level 1, and 1-sublist at level 2.

Every node in the binary tree is the root of a subtree that represents a sub(skip)graph (or sub graph) of the skip graph. We refer the subgraph rooted by a 0-sublist and 1-sublist as 0-subgraph and 1-subgraph, respectively. Since the construction is recursive, we can also designate a subgraph as $b$-subgraph, where $b$ is a bit string containing the common prefix bits of the membership vectors for all nodes in the subgraph. For example, the subgraph containing nodes $G$ and $W$ in figure 2.1(b) is designated by 10-subgraph.

Let $V = \{1, ..., n\}$ be a set of nodes (or peers). Let $S$ be the family of all Skip Graphs of $n$ nodes, where each topology $G(V, E) \in S$ is a skip graph with $O(\log n)$ levels. For any skip graph $S \in S$, $L_i$ denotes the set of all linked lists at level $i$ of $S$. We define the following balance property that must hold for the family of skip graphs $S$:

**Definition (a-balance Property).** A Skip Graph satisfies the $a$-balance property if there exists a positive integer $a$, such that among any $a + 1$ consecutive nodes in any linked list $l \in L_i$, at most $a$ nodes can be in a single linked list in $L_{i+1}$. The $a$-balance property ensures that the length of the search path between any pair of nodes is at most $a \cdot \log n$.

**Definition (Sub Skip Graph).** A sub skip graph (often called subgraph in this paper) is a skip graph that is a part of another skip graph. In other words, given a skip graph $S(V, E)$, a sub skip graph $S'(V', E')$ is a skip graph in $S$ such that $V' \subseteq V$ and $E'$ is the set of links from $S$ induced by the nodes in $V'$. We call a sub skip graph is at level $d$ when all nodes in the sub skip graph share a common membership vector prefix of size $d$. We call the lowest level of a sub skip graph as the base level, and the linked list that contains the nodes of a
sub skip graph as the base linked list for that sub skip graph. Observe that there exists a sub skip graph for all linked lists in any skip graph.

### 2.2 Hypercube Model

Similar to skip graphs, we use a tree representation for hypercubic network as shown in Figure 2.2. Observe that, because of the structure of hypercubic networks, even a hypercubic network with eight nodes can have many possible tree representations. Also, similar to skip graphs, our model allows a hypercubic network $O(\log n)$ bit memory per node and assumes a $CONGEST$ model of communication.

![Figure 2.2: Tree representation of a hypercubic networks of 8 nodes.](image)

To be consistent with the skip graphs, we use the term membership vector for hyper-
cubic networks to refer to coordinates. Also, we often write *subtree* or *subnetwork* instead of subgraph when we refer to a subtree in the tree modeling of the network.

### 2.3 Self-Adjusting Model for Skip Graphs and Hypercubic Networks.

We consider a synchronous computation model, where communications occur in *rounds*. A node can send and receive at most 1 message through a link in a round (i.e. *CONGEST* model). Our model limits the memory of each node to $O(\log n)$ bits. Let $\mathcal{N}$ be the family of permissible network topologies. Given a network $N \in \mathcal{N}$, and a pair of communicating nodes $(u, v) \in V \times V$, a self-adjusting algorithm performs the followings:

1. Establishes communication between nodes $u$ and $v$ in $N$.
2. Transforms the network $N$ to another network $N' \in \mathcal{N}$, such that nodes $u$ and $v$ move to subtree (in the tree that represents the network) of size two in $N'$. This implies that a direct link needs to be established between nodes $u$ and $v$.

Note that the height of $N'$ must be $O(\log n)$ since $N' \in \mathcal{N}$. Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ be an unknown access sequence consisting of $m$ sequential communication requests, $\sigma_t = (u, v) \in V \times V$ denotes a routing request from source $u$ to destination $v$. Given a skip graph $S$, we define the distance $d_S(\sigma_t)$ as the number of intermediate nodes in the communication path from the source to destination associated with request $\sigma_t$, where the communication path is obtained by standard routing algorithm of the network. An overview of the standard skip graph routing [7] is presented in Appendix A.1.

Given a request $\sigma_t$ at time $t$, let an algorithm $\mathcal{A}$ transforms the current network $N_t \in \mathcal{N}$ to $N_{t+1} \in \mathcal{N}$. We define the cost for network transformation as the number of
rounds needed to transform the topology. We denote this transformation cost at time $t$ as $\rho(\mathcal{A}, S_t, \sigma_t)$. Similar to a prior work [11], we define the cost of serving request $\sigma_i$ as the distance from source to destination plus the cost of transformation performed by $\mathcal{A}$ plus one, i.e., $d_{S_t}(\sigma_t) + \rho(\mathcal{A}, S_t, \sigma_t) + 1$.

**Definition (Average and Amortized Cost).** We used definitions similar to [11] here. Given an initial network $N_0$, the average cost for algorithm $\mathcal{A}$ to serve a sequence of communication requests $\sigma = (\sigma_1, \sigma_2, \cdots, \sigma_m)$ is:

$$\text{Cost}(\mathcal{A}, N_t, \sigma) = \frac{1}{m} \sum_{i=1}^{m} (d_{S_t}(\sigma_t) + \rho(\mathcal{A}, S_t, \sigma_t) + 1)$$ \hspace{1cm} (2.1)

The amortized cost of $\mathcal{A}$ is defined as the worst possible cost to serve a communication sequence $\sigma$, i.e.

$$\max_{N_0, \sigma} \text{Cost}(\mathcal{A}, N_0, \sigma).$$

### 2.4 Working Set Property

**Definition (Working Set Number).** We denote the working set number for request $\sigma_i$ as $T_i(\sigma_i)$. Let $u_i$ be the source and $v_i$ be the destination specified by communication request $\sigma_i$. To define $T_i(\sigma_i)$, we construct a communication graph $G$ with the nodes that communicated (either as source or destination) during the time period starting from the last time $u_i$ and $v_i$ communicated, and ending at time $i$. We draw an edge between any two nodes in $G$ if they communicated in this time duration. Now, we define the working set number for request $\sigma_i$, $T_i(\sigma_i)$, as the number of distinct nodes in $G$ that have a path from either $u_i$ or $v_i$. In case $u_i$ and $v_i$ are communicating for the first time, $T_i(\sigma_i) = n$ by default, where $n$ is the
number of nodes in the skip graph.

As an example, for the latest communication request \((u, v)\) shown in figure 2.3(a), the corresponding communication graph \(G\) is shown in figure 2.3(b). The number of distinct nodes in \(G\) that have a path from either \(u\) or \(v\) is 5; therefore the working set number for the communication request is 5.

**Definition (Working Set Property).** For a skip graph \(S\) at time \(i\), the working set property for any node pair \((x, y)\) ∈ \(S\) holds iff \(d_S(x, y) \leq \log T_i(x, y)\). Note that, the log in the definition is to address the tree-like structure of the skip graph topology.

**Definition (Working Set Bound).** We define the working set bound as \(WS(\sigma) = \sum_{i=1}^{m} \log(T_i(\sigma_i))\).

**Theorem 1.** For an unknown communication sequence \(\sigma = \sigma_1, \sigma_2, \ldots, \sigma_m\), the amortized routing cost for any self-adjusting algorithm conforming to our model is at least \(WS(\sigma)\) rounds.

**Proof.** Let us assume that the theorem does not hold. Then there must be at least one request \(\sigma_i = (u_i, v_i)\) such that \(d_{S_i}(u_i, v_i) < T_i(u_i, v_i)\). However, due to the construction of the skip graph, this results in existence of a node \(w_i\) such that \(d_{S_i}(u_i, w_i) > T_i(u_i, w_i)\). An example to illustrate this idea is presented right after this proof.

Since the communication sequence is unknown, it is possible that \((u_i, w_i)\) is chosen as request \(\sigma_i\) instead of \((u_i, v_i)\). Since this argument is applicable for any \(\sigma_i \in \sigma\), the theorem holds.

**Example of Working Set Bound.** Consider the communication graph in Figure 2.4. Let \(l\)
(a) An access pattern showing a repeating communications between \( u \) and \( v \).

(b) Communication graph \( G \) for the time duration shown in (a).

Figure 2.3: For the access pattern shown in (a), the working set number for the last communication between \( u \) and \( v \) is 5, as the the number of distinct nodes in \( G \) that has a path from either \( u \) or \( v \) is 5 (e,a,k,u and v).

be a linked list of size \( 2k \) in a skip graph. Let nodes \( U, V, A, A_1, A_2, \ldots, A_{n-2} \) belong to the linked list \( l \). Let the linked list \( l \) split into 2 subgraphs (i.e. sublists) each with size \( k \) at the next (upper) level. Suppose node \( A \) moves to the 0-subgraph. Now, we need to choose other \( k - 1 \) nodes to accompany node \( A \) in the 0-subgraph.

Let we move nodes \( U \) and \( V \) to the 0-subgraph. Then there exists a node \( A_i, 1 \leq i \leq k - 2 \) that moves to the 1-subgraph. Clearly this violates the working set property for the pair \((A, A_i)\). However, if we move nodes \( U \) and \( V \) to the 1-subgraph, we violate the working set property for the pair \((A, U)\). Thus, \( U \) must move to the 0-subgraph and \( V \)
must move to the 1-subgraph. Note that, at time $t' + k$, the working set number for pair $(U, V)$, $T_{t' + k}(U, V) = k + 1$; and the routing distance for pair $(U, V)$ is $\lceil T_{t' + k}(U, V) \rceil = \lceil \log(k + 1) \rceil = \log_2(2k)$.

![A communication graph. Each edge is labeled with the timestamp of the most recent communication between the end nodes.](image)
3.1 Dynamic Skip Graphs (DSG)

3.1.1 Overview

Upon a communication request, our algorithm DSG first establishes the communication using the standard skip graph routing, then performs atomic topological transformation conforming to the self-adjusting model. The key idea behind DSG is that frequently communicating nodes form groups at different levels and a node’s attachment to a group is determined by a timestamp. Each node has a group-id and a timestamp associated with each level. A node is a member of a group at each level, and the group-id is an identifier that represents a group. All nodes belong to the same group at a level hold the common group-id. The timestamp associated with a level is used by DSG to identify how attached a node is with its group at that level.

When two nodes from two different groups communicate, DSG merges the communicating groups to a single group. However, when a group grows too big to be accommodated in a single linked list at the corresponding level (due to the structural constraint of the skip graph), DSG splits the group into two smaller groups. There are three challenges here. First, since the goal of DSG is to ensure that the working set property always holds for any node pair in any group, distance between any two nodes from any non-communicating group should not increase. In other worlds, routing distances among nodes of any non-communicating groups should not be affected due to a transformation. Second,
the working set property must hold for any node pair in the merged and split groups after a transformation. Third, a transformation requires a partial reconstruction of the skip graph structure. According to our self-adjusting model, the height of the skip graph must remain $O(\log n)$ after any reconstruction.

Transformation starts from the highest level at which communicating nodes share a common linked list. For example, the highest level with a common linked list for nodes $A$ and $M$ in the skip graph in Figure 2.1 is level 1 and the common linked list is the linked list that contains only nodes $A$, $J$ and $M$. Starting from the highest level with common linked list, transformation continues recursively and parallelly in the upper levels until all the involved nodes become singleton (i.e. move to a linked list of size 1). For each of the newly created linked lists of size $> 1$, transformation takes place as follows:

- Each node of the linked list computes a priority using certain priority rules. Each priority is a function of node’s group-id and timestamp for the corresponding level. The priorities are computed in a way such that all groups have a distinct range of priorities. The communicating nodes have the highest priority, each node of the merged group has a positive priority, and all other nodes have a negative priority.

- All nodes of the linked list compute an approximate median priority using the algorithm AMF. In general, at the next upper level after transformation, any node with a priority higher or equal to the approximate median priority moves to the 0-subgraph, and any node with a priority lower than the approximate median priority moves to the 1-subgraph. DSG uses priorities to ensure that nodes from the same group remain together after a transformation. However, a transformation technique based on com-
paring priority with approximate median priority may split a non-communicating group. Such cases are handled carefully by DSG. Note that, the approximate median priority is used to ensure that the sizes of the 0-subgraph and 1-subgraph are roughly the same after a transformation, keeping the height of the skip graph always \( O(\log n) \).

Since communicating nodes always move to the 0-subgraph, after transformation in all levels, communicating nodes are guaranteed to move to a linked list of size 2. Each node involved in the transformation reassigns its group-ids and timestamps such that DSG can work consistently for future communication requests.

### 3.1.2 Setup and Notations

Let \( H_t \) be the height of the skip graph at time \( t \). DSG requires every node to hold \( H_t \) bits to store its membership vector. In addition, each node stores a timestamp and a group-id for each of the levels. For node \( i \) and level \( j \), we use the notations \( V^i_j \), \( T^i_j \) and \( G^i_j \) to denote the stored membership vector bit, timestamp, and group-id respectively. Initially, all timestamps are set to zero and all group-ids are set to the corresponding node’s identifier.

Figure 3.1(a) shows a communication graph \( G \), where each edge is labeled with the time associated with the most recent communication. Observe that nodes \( U \) and \( V \) communicate at time 8 as shown in \( G \). For the communication graph \( G \), Figure 3.1(b) shows a binary tree representation of a valid skip graph \( (S_8 \in \mathcal{S}) \) at time 8 obtained by DSG. Figure 3.1(c) shows a valid skip graph \( (S_9 \in \mathcal{S}) \) representation, where \( S_9 \) is transformed from \( S_8 \) using DSG, as a result of the communication between nodes \( U \) and \( V \) at time 8.
Figure 3.1: For the communication graph $G$ shown in (a), possible skip graph representations at time 8 and 9 ($S_8$ and $S_9$), obtained by DSG, are shown in (b) and (c), respectively. The rounded rectangles show the groups of nodes at different levels, and the number below each node is the timestamp for the node at the corresponding level. For example, in $S_9$ (figure c), the group of node $B$ at level 2 has 3 nodes ($B$, $G$, and $D$), and the timestamp of node $B$ for level 2 (i.e. $T^B_2$) is 4.
The numbers below each node in Figure 3.1(b) and (c) give the timestamp of the node at the corresponding level. We use this transformation from $S_8$ to $S_9$ as an example for the description of our algorithm for the remaining paper.

For a communication request from node $u$ to node $v$, let $\alpha$ be the highest common level of the current skip graph with a linked list containing both nodes $u$ and $v$. Let $l_\alpha$ denote that linked list, then $u, v \in l_\alpha$. Let $t$ be the time when the request is originated, and $S_t$ be the skip graph at time $t$. In Figure 3.1(b), for nodes $U$ and $V$, $\alpha = 0$, $l_\alpha$ is the linked list represented by the root of the binary tree, and $t = 8$.

We explain different parts of our algorithm in detail in following subsections.

3.1.3 Transformation From $S_t$ to $S_{t+1}$

Upon a routing from node $u$ to node $v$, node $v$ records the highest common level number $\alpha$ and shares $\alpha$ with node $u$. Then both nodes $u$ and $v$ broadcast a transformation notification to all nodes in $l_\alpha$. The notification message includes all $H_t$ timestamps, group-ids and membership vectors of nodes $u$ and $v$. All nodes $x \in l_\alpha$ compute a priority $P(x)$ by using following priority-rules:

P1: (Rule for communicating nodes). Nodes $u$ and $v$ set $\infty$ as their priority. In other words, set $P(u) = P(v) = \infty$.

P2: (Rule for nodes in the same group of either communicating nodes at level $\alpha$). All nodes $x \in l_\alpha, x \neq u, x \neq v, G^x_\alpha = G^u_\alpha$ assign their priority $P(x) = \min(T^x_c, T^u_c)$ where $c$ is the highest level in $S_t$ such that $G^x_c = G^u_c$. Similarly, all nodes $x \in l_\alpha, x \neq u, x \neq v, G^x_\alpha = G^v_\alpha$ assign $P(x) = \min(T^x_c, T^v_c)$ where $c$ is the highest level in $S_t$. 

such that $G^x_c = G^u_c$.

P3: (Rule for other nodes). Each node $x \in l_\alpha, x \neq u, x \neq v, G^x_\alpha \neq G^u_\alpha, G^x_\alpha \neq G^v_\alpha$ (i.e. neither in $u$’s nor in $v$’s group at level $\alpha$) set $P(x) = -(G^x_\alpha \cdot t) + T_{x,\alpha+1}^x$.

We require that group identifiers are non-negative integers (possibly an IP address of a node). Observe that, all nodes of the communicating group at level $\alpha$ have a positive priority as a timestamp is always positive, and rest of the nodes have a negative priority as DSG ensures that $t > T_{x,\alpha+1}^x$. Also, according to P3, priorities assigned to nodes from a non-communicating group range between $-(G^x_\alpha \cdot t)$ and $-(G^x_\alpha + 1) \cdot t$, where $G^x_\alpha$ is the group-id.

In our example in Figure 3.1, as nodes U and V communicate at time 8, the $\alpha$ is 0 ($S_8$ in Figure 3.1(b)). Let us assume that the nodes’ numerical identifiers are determined by their positions in the English alphabet, e.g. identifier for node A is 1, identifier for node B is 2, and so on. According to the priority rule P1, $P(U) = P(V) = \infty$; and according to the priority rule P2, $P(D) = 2, P(G) = 2, P(B) = 2$ and $P(E) = 5$. Let us assume that the group-ids $G^H_0 = G^I_0 = 10$ (as H is the tenth letter in alphabet) and $G^F_0 = G^I_0 = 6$. Hence, according to the priority rule P3, $P(H) = P(J) = -(10 \times 7) + 2 = -68$, and $P(F) = P(I) = -(6 \times 7) + 2 = -40$.

At this point, node $u$’s group at level $\alpha$ merges with node $v$’s group at the same level by updating their group-ids. To this end, all nodes $x \in l_\alpha$ with $G^x_\alpha = G^u_\alpha$ or $G^x_\alpha = G^v_\alpha$ set $G^x_\alpha = u$. Note that, by $u$, we mean the identifier of node $u$.

Transformation begins at level $\alpha + 1$ and recursively continues at upper levels. Only the nodes in linked list $l_\alpha$ take part in the transformation. For the remaining section, we
write $d$ to refer the current level of transformation and $l_{d-1}$ to refer the linked list that is involved in the transformation. Initially $d = \alpha + 1$ and $l_{d-1} = l_{\alpha}$.

All nodes in $l_{d-1}$ find an approximate median priority to decide whether to move to the 0-subgraph or to the 1-subgraph (i.e. determine new membership vector bit $V_d^x$ in the new skip graph $S_{t+1}$). We propose a distributed approximate median finding algorithm (AMF) for skip graphs to find the approximate median in expected $O(\log_\alpha n)$ rounds, where $\alpha$ is a constant. The algorithm AMF is described in section 3.2. For now let us consider AMF as a black box that finds an approximate median priority and broadcasts it to all nodes in linked list $l_{d-1}$.

However, to utilize AMF, in some cases we require to identify the nodes that moved to the 0-subgraph by receiving a positive approximate median priority. To this end, we introduce a set of boolean variables referred to as is-dominating-groups, held by each node of the skip graph. Each node holds $H_t$ is-dominating-group variables, one for each level. Let $D_d^x$ denote the is-dominating-group of node $x$ for level $d$. The goal is to ensure that any node $x$ with $D_d^x = True$ moved to the 0-subgraph at level $d + 1$ in past when it received a positive approximate median priority at level $d$ for the last time.

Let the approximate median priority be $M$. One of the following cases must follow:

**Case 1 (M is positive).** Each node $x$ with $P(x) \geq M$ moves to the 0-subgraph at level $d$ and sets is-dominating-group $D_d^x = True$. Each node $x$ with $P(x) < M$ moves to the 1-subgraph at level $d$ and sets is-dominating-group $D_d^x = False$. According to the priority rules P1 and P2, this case splits the merged group of nodes $u$ and $v$.

**Case 2 (M is Negative).** When $M$ is negative, there may exist a group $g_s$ such that
all nodes $x \in g_s$ finds the following condition true.

$$-G_{d-1}^x \cdot t \geq M \geq -(G_{d-1}^x + 1) \cdot t$$  \hspace{1cm} (3.1)

If that happens, splitting group by comparing $P(x)$ and $M$ (as we do for case 1) may split the group $g_s$ at level $d$ as its nodes may move to different subgraphs. Given that $M$ is negative, the priority rule P3 confirms that group $g_s$ is a non-communicating group at level $d$ (i.e. contains neither $u$ nor $v$). Clearly the working set property will be violated if the group $g_s$ is split at level $d$ since it will increase the distance between some node pairs in group $g_s$. To fix this issue, nodes in $l_d$ perform a distributed count to compute $|l_d|$ and $|g_s|$. Then nodes decide which subgraph to move to as follows:

- If $|g_s| > \frac{2}{3}|l_d|$ ($|g_s|$ is too big, thus we need to split $g_s$)
  - any node $x \in g_s$ moves to the 1-subgraph if $D_d^x = True$; $x$ moves to the 0-subgraph otherwise.
  - any node $x \in l_{d-1}$ and $x \notin g_s$ moves to the 0-subgraph.

- If $|g_s| < \frac{1}{3}|l_d|$ ($|g_s|$ is sufficiently small, thus we move all nodes of $g_s$ either to the 0-subgraph or to the 1-subgraph)
  - any node $x$ such that $x \in l_{d-1}$ and $x \notin g_s$ moves to the 0-subgraph if $P(x) \geq M$; $x$ moves to the 1-subgraph otherwise.

- Let $L_{low} = \{x \in l_d|P(x) < M\}$ and $L_{high} = \{x \in l_d|P(x) \geq M\}$. Clearly, $|l_d| = L_{low} + L_{high}$. Any node $x \in g_s$ moves to the 0-subgraph if $L_{high} < L_{low}$; $x$ moves to the 1-subgraph otherwise.
- If $\frac{1}{3}|l_d| \leq |g_s| \leq \frac{2}{3}|l_d|$ (Move all nodes of $g_s$ to the 1-subgraph and rest of the nodes to the 0-subgraph)

  - any node $x$ such that $x \in l_{d-1}$ and $x \notin g_s$ moves to the 0-subgraph.

  - any node $x \in g_s$ moves to the 1-subgraph.

Note that $|g_s|$, $L_{low}$ and $L_{high}$ can be computed in $O(\log n)$ rounds by computing distributed sum using a balanced skip list. Algorithm AMF constructs a balanced skip list to compute the median priority. The balanced skip list created by AMF can be reused to compute $|g_s|$, $L_{low}$ and $L_{high}$. To avoid distraction, we present the distributed sum algorithm in Appendix A.2.1.

As nodes decide which subgraph to move to, two new linked lists are formed at level $d$. To find the left and right neighbors at level $d$, nodes linearly search for neighbors at level $d-1$. Because of the a-balance property, it is guaranteed that a node finds both of its left and right neighbors in at most $a$ rounds (end-nodes have just one neighbor). All nodes $x$ in the new linked lists that do not contain communicating nodes $u$ and $v$ recompute $P(x)$ (for upcoming transformation at level $d$) using the priority rule P4. Computation of $P(x)$ with P4 is similar to P3 except that $\alpha$ is replaced by $d$.

P4: (Rule for nodes moved to a linked list that does not contain nodes $u$ and $v$). Each node $x$ moves to a linked list $l_d$ such that $u, v \not\in l_{d-1}$ sets $P(x) = -(G^x_d \cdot t) + T^x_{d+1}$.

Since the communicating nodes $u$ and $v$ always move to the 0-subgraph, a linked list $l_d$ contains nodes $u$ and $v$ only if the following condition is true for all nodes $x \in l_d$.

$$V^x_{\alpha+1} = V^x_{\alpha+2} = \ldots = V^x_d = 0$$ (3.2)
The transformation procedure described above is performed recursively and parallelly by each new linked list until all nodes become members of a singleton list. Clearly this ensures that node \( u \) and \( v \) get connected directly, and all nodes \( x \in l_\alpha \) finds their new and complete membership vectors.

Getting back to our example in Figure 3.1, let the approximate median priority \( M \) be 2 for \( d = 1 \). Then nodes U, V, E, B, G and D move to the 0-subgraph and nodes F, I, H and J move to the 1-subgraph at level 1. Now nodes in the 1-subgraph recomputes their priority by using P4. Then both linked lists at level 1 performs the transformation recursively and parallelly, and this continues at upper levels.

In the following two subsections, we discuss how nodes involved with a communication reassigns their group-ids and timestamps.

### 3.1.4 Assignment of New Group-ids

For each level \( d = \alpha, \alpha + 1, \cdots, H_{t+1} \), each node \( x \in l_{d-1} \) checks if nodes \( u \) and \( v \) belong to their own linked list at level \( d \) by checking the condition in equation 3.2. If a node finds the condition in equation 3.2 to be true, the node sets its group-id \( G^x_d \) to the identifier of node \( u \). If the condition in equation 3.2 is found to be false by any node, it checks if its group at level \( d \) is getting split due to the transformation. A group without communicating nodes \( u \) and \( v \) can split at level \( d \) only if the group is \( g_s \) and \( |g_s| > \frac{2}{3} |l_d| \). In case of such a split, the identifier of the left-most node in the split group is used as the new group-id. Observe that it is easy for the left-most node to detect itself since it does not find a left neighbor. The balanced skip list structure created by algorithm AMF is then reused to
propagate the new group-id to all the nodes of the group. The left-most node first sends its identifier to the head node of the balanced skip list, and then the identifier is broadcasted to all nodes at the base level of the skip list. The balanced skip list is destroyed once the new group-id is sent to all members of the group.

As nodes $u$ and $v$ move to the same group, for correctness, it is necessary that all nodes in node $u$’s and node $v$’s group at any level $d < \alpha$ in $S_t$ must have the same group-id at level $d$ in $S_{t+1}$. All such nodes update their group-ids for levels below $\alpha$ by using the procedure presented in Appendix A.2, only if $G_u^{\alpha-1} \neq G_v^{\alpha-1}$.

3.1.5 Assignment of New Timestamps

Each node $x \in l_\alpha$ updates its timestamps by using the following timestamp-rules (executes in the order given below):

**T1:** Let $d'$ be the level at which nodes $u$ and $v$ form a linked list of size 2. Nodes $u$ and $v$ set $T_u^{d'} = T_u^{d'+1} = t$ and $T_v^{d'} = T_v^{d'+1} = t$. Note that both nodes $u$ and $v$ become singleton in level $d'+1$. Nodes $u$ and $v$ also set $T_i^u = T_i^v = \max(T_i^u, T_i^v)$ for $i = d' - 1, d' - 2, \cdots, B_u$. For an example, check the timestamps of nodes $U$ and $V$ in $S_9$ (Figure 3.1(c)).

**T2:** For a node $x$, let $c'$ be the size of the longest common postfix between membership vectors $m(u')$ and $m(x)$ in $S_t$, where $u'$ is the nearest communicating node (either $u$ or $v$) to $x$ in $S_t$. For example, in $S_8$ (Figure 3.1(b)), $c'$s for nodes $E$ and $G$ are 2 and 1, respectively. Let $M_x^d$ be the approximate median priority received by node $x$ at level $d$. For each level $d > \alpha$, each node $x$ with $G_x^d = G_u^d$ in $S_{t+1}$ (i.e. after group
reassignment) sets $T_{d+1}^x = T_c^x$, where $c$ is the lowest level in $S_t$ with $\alpha \leq c < c'$ such that $T_c^x > M_d^x$. If no such $T_c^x$ exists, node $x$ sets $T_{d+1}^x = M_d^x$. For an example, check the timestamps of node $E$ in $S_{t+1}$ (Figure 3.1(c)) at levels 1 and 2, assuming $M_0^E = 2$ and $M_1^E = 5$.

T3: Let $x$ be a node such that $x \neq u, x \neq v, x \in l_\alpha$ with $G_x^u = G_x^v$ in $S_t$ (before transformation). Let $c'$ be the size of the longest common postfix between $m(u)$ and $m(x)$ in $S_t$ and $c''$ be the longest common postfix between $m(u)$ and $m(x)$ in $S_{t+1}$. If $c' - 1 > c'' + 1$, each node $x$ sets $T_i^x = T_c^x$ for all $i = c' - 1, c' - 2, \cdots, c'' + 1$. Similarly, each $x, x \neq u, x \neq v, x \in l_\alpha$ with $G_x^u = G_x^v$ at time $t$ (before transformation), updates their timestamps w.r.t. node $v$. For example, for node $E$ in Figure 3.1, $c' = 3$ and $c'' = 2$. Hence T3 does not apply for node $E$.

T4: Each node $x$ that initialized or received $G_{lower}$ finds the lowest level $d$ such that $T_{d+1}^x = 0$. If such a $d$ exists and if $d > B_x$, node $x$ sets $T_i^x = T_{d+1}^x$ for $i = d, d - 1, \cdots, B_y$.

T5: Let $x$ be a node, $x \in l_\alpha$ and $x$ belongs to a group $g$ at level $d$ in $S_t$, $d \geq \alpha$, such that $g$ splits into two subgroups at level $d$ in $S_{t+1}$. Each node $x$ sets $T_{d-1}^x = T_d^x$ only if $T_{d-1}^x = 0$.

T6: A group-base of a node is the highest level at which the node belongs to its biggest group. For example, in the skip graph $S_8$ in Figure 3.1(b), the group-base for node $B$ is 1, as 1 is the highest level at which node $B$ is a member of its biggest group (B,G,D,U). Appendix A.2 presents details about how nodes maintain their group-
base in a distributed manner. Let $B_x$ denote the group-base of a node $x$. Each node $x \in l_\alpha$ sets $T^x_d = 0$ for all $d < B_x$. For an example, check the timestamps of nodes $F$ and $I$ in $S_{t+1}$ (Figure 3.1(c)) at level 1 and 0. Note that, $B_F = B_I = 2$ in $S_{t+1}$.

After reassigning the timestamps, all nodes $x \in l_\alpha$ independently set themselves free for the next communication or transformation. A summary of the algorithm DSG is presented in Algorithm 1.

### 3.1.6 Maintaining the a-balance Property

When a node $x$ moves to a new subgraph at level $d$, node $x$ checks if the a-balance property is being violated at level $d$ by the rearrangement of nodes. The balanced skip list created by AMF is reused to check if any consecutive $a$ nodes at level $d − 1$ have moved to the same subgraph at level $d$. All nodes in the skip list at level $d$ check the new membership vector bit of $a$ nearby nodes at level $d − 1$ in both sides and share this information with both neighbors of the skip list at level $d$ to detect chains. If a chain of size $a$ or longer is detected, a dummy node is placed in the sibling subgraph at level $d$ to break the chain.

A dummy node is a logical node which requires $O(\log n)$ links and an identifier. A dummy node does not hold any data and only used for routing purpose. To implement dummy nodes, all regular nodes need to have the ability to handle extra $O(\log n)$ links.

When a dummy node is placed to break a chain, the identifier is picked by checking the identifier of a neighbor of the dummy node at level $d$ to ensure that all identifiers remain sorted at the base level of $S_{t+1}$. A dummy node does not participate in transformation and destroys itself when a transformation notification is received. While being destroyed, a
Algorithm 1: Dynamic Skip Graph DSG (summary)

1. Upon request \((u, v)\) in \(S_t\), establish the communication by using standard skip graph routing and find \(\alpha\). Broadcast the membership vector, timestamps \(T^u_d, T^v_d\), group-ids \(G^u_d, G^v_d\), and group-bases \(B_u, B_v\), where \(d = \alpha, \alpha + 1, ..., H_t\), to all nodes in \(l_\alpha\).
2. Each node \(x \in l_\alpha\) computes their priority \(P(x)\) using the using priority-rules P1, P2 and P3.
3. Node \(u\)'s group at level \(\alpha\) merges with node \(v\)'s group at level \(\alpha\) by setting \(G^x_\alpha = u\) for each group member \(x\).
4. Let \(d = \alpha + 1\) and linked list \(l_{d-1} = l_\alpha\). Compute the approximate median priority \(M\) by using the algorithm AMF.
5. Compute \(|L_{low}|, |L_{high}|, |g_s|\) using the balanced skip list formed by AMF if the condition in Equation 3.1 is true.
6. Determine the membership vector bit \(V^x_d\) by using \(P(x), M, |L_{low}|, |L_{high}|, |g_s|\) and update is-dominating-group \(D^x_d\).
7. Reuse the balanced skip list formed by AMF to check if a-balance property is being violated by the rearrangement. If yes, put a dummy node to break the chain violating the a-balance property.
8. If a group at level \(d\) is split into two subgraphs (because of step 6), find new (level \(d\)) group-id for the split group that moves to the 1-subgraph, and broadcast the new group-id by using the balanced skip list formed by AMF. Each \(x\) in that group updates its \(G^x_d\) with the new group-id. However, if the linked list formed by nodes that moved to 0-subgraph contains nodes \(u\) and \(v\), nodes \(x\) set \(G^x_d = u\). Each node \(x\) also updates their priorities \((P(x))\) using priority-rule P4 if its linked list does not contain nodes \(u\) and \(v\). Destroy the balanced skip list formed by AMF.
9. Repeat steps 2 to 8 recursively and parallely for all newly formed linked lists \((l_{d-1})\) that contains at least 2 nodes.
10. Update group-ids and group-bases for involved nodes.
12. Independently set nodes \((x \in l_\alpha)\) free for next communication.
dummy node simply links its left and right neighbors at all levels and deletes itself. The sole purpose of dummy nodes is to ensure that a-balance property is preserved after a transformation. Note that, the maximum number of dummy nodes possible is \( n/a \).

### 3.1.7 Node Addition/Removal

Nodes can be added or removed by using standard node addition or removal procedures for skip graphs. When a node is added, the new node needs to initialize its variables with the default (initial) values. Following a node addition, the new node checks if the a-balance property is violated due to the node addition. Following a node deletion, a neighbor of the deleted node from each level checks if the a-balance property is violated due to node deletion. In case of a violation, a dummy node is placed to protect the a-balance property, as described in Section 3.1.6.

### 3.2 Approximate Median Finding for Skip Graphs (AMF)

Given a linked list of size \( n \) with each node holding a value, AMF is a distributed algorithm that finds an approximate median of the values in expected \( O(\log n) \) rounds. Given that the size of the linked list is bigger than a constant \( a \), we first construct a probabilistic skip list where the left-most node steps up to the next level with probability 1, and all other nodes step up to the next level with a probability \( 1/a \). After stepping up to the next level, nodes find their neighbors linearly from the level it stepped up. We write two consecutive nodes are supported by \( k \) nodes if they have \( k - 1 \) nodes in between at the immediate lower level. When a linked list at some level is built, nodes locally check if two consecutive nodes are supported by at least \( a/2 \) and at most \( 2a \) nodes. If two or more consecutive nodes are
supported by less than $a/2$ nodes, they select the node with the highest identifier as a leader, and leader asks some nodes to step down to make sure each consecutive nodes in the list is supported by at least $a/2$ modes. Similarly, if two consecutive nodes are supported by more than $2a$ nodes, the node with the higher identifier asks some nodes in between to step up so that no two consecutive nodes in the list are supported by more than $2a$ nodes. The construction ends when the left-most node become the member of a singleton list (i.e. the root of the skip list) at some level. Let $l_i$ denote the linked list at level $i$ of the skip list. We refer the base level as level 0. Let $h$ be the height of the skip list, then level $h$ is the only level where the left-most node is singleton. The left-most node (i.e. root) broadcasts the value $h$ to all nodes of the skip list. Let $h = \log_b n$, then $2a \geq b \geq a/2$.

Median finding algorithm is a recursive algorithm that works in rounds. At the first round each node $x \in l_0, x \notin l_1$ forwards their values to their left neighbors, and any value received from the right neighbor is also forwarded to the left neighbor. This way values hold by all nodes that did not step up to level 1 are gathered to their nearest left neighbor that stepped up to level 1 (nodes in $l_1$). For implementation, while forwarding the value to the left neighbor, a node adds a “last-node” tag with its value if it has an immediate right neighbor that has lifted to the upper level. When a node in linked list $l_1$ receives such tag, it can move to the next step knowing that it will not receive any more value from level 0.

Each node $x \in l_1$ is expected to have $a$ values including its own. All nodes $x \in l_1$ but $x \notin l_2$ forward all values they have to the nearest left neighbor that stepped up at level 2. Therefore, all nodes $x \in l_3$ are expected to have $a^2$ values. This process continues until nodes at level $\lceil \log_{a/2} h \rceil + 2$ (note that $\log_{a/2} h = \log_{a/2} \log_b n$) receives all their values.
Clearly each node $x \in l_{\lceil \log_{a/2} h \rceil +2}$ must receive at least $(a/2)^{\lceil \log_{a/2} h \rceil +2} = \frac{a^2 h}{4}$ values.

These values are sorted locally by the nodes $x \in l_{\lceil \log_{a/2} h \rceil +2}$, and each node uniformly samples $ah$ values from the sorted list. Nodes keep only the sampled values for the next round and discard all other values they have. Nodes that did not step up to any further level forward their sampled values to the nearest left-neighbor that stepped up to the next level. A similar tagging mechanism explained earlier can be used for the implementation purpose. It is important to note that this algorithm satisfies the $CONGEST$ model since all messages are $O(\log n)$ in size. The process of gathering values in the nearest left neighbor at upper level and sampling them continues recursively until the left-most node of the skip list receives all the (expected $a^2 h$) values at level $h$ (the top level).

Each value forwarded by any node is attached with a left rank and a right rank. The left (right) rank attached with a value is the number of nodes in $l_a$ that are guaranteed to have a larger (smaller) value than the value. Before every sampling, each node computes their left rank and right rank. Initially (at the base level) each node at the base level set both the left and right ranks attached with their value to zero. When a list of values are locally sorted by any node, the node computes the new left and right ranks for all the sampled values. The new left rank of a value is computed by adding the left ranks attached with all larger values in the sorted list. Similarly the new right rank of a value is computed by adding all the attached right ranks attached with the smaller values in the sorted list. Nodes forward their sampled values with the computed left and right ranks to the nearest left neighbor at the current level of the skip list.

When the left-most node at the top level of the skip list receives values from level
$h - 1$ (the second highest level), it computes the median based on the left and right rank attached with the values and then broadcasts the value to all nodes in $l_0$ as the approximate median.

The algorithm AMF is summarized in Algorithm 2.

**Algorithm 2: Approximate Median Finding AMF**

1. Construct a probabilistic skip list where the left-most node steps up to next level with probability 1 and all other nodes step up to next level with probability $1/a$, where $a$ is a constant and parameter for a-balance property. While the linked list at any level is being built, nodes locally ensure that no two consecutive nodes are supported by less than $a/2$ or more than $2a$ nodes.

2. All nodes $x \in l_0$ but $x \not\in l_1$ forward their value and any value received from right neighbor (at base level of skip list) to the left neighbor. A tagging mechanism explained in section 3.2 can be used for the synchronization purpose.

3. **forall levels** $d = 1, 2, ..., h - 2$ **in the probabilistic skip list, sequentially do**

4. all nodes $x \in l_d$, $x \not\in l_{d+1}$ (except the left-most node) forward (using level $d$ links) the values they have to the nearest right neighbor that stepped up to the level $d + 1$.

5. if $d \geq \lceil \log_{a/2} h \rceil + 1$ then

6. all nodes in $l_{d+1}$ locally sort all the values they received, uniformly sample $ah$ values from the sorted list, and compute new left and right ranks for all the sampled values. Nodes keep only the sampled values for the next level and discard all other values.

7. All nodes at level $h - 1$ (except the left-most node) forward their sampled $ah$ values to the left-most node.

8. The left-most node (and also the only node in level $h$) sort all the values it receives, computes new left and right ranks for all values, and finds the approximate median from the sorted list based on the left and right ranks. The approximate median is then broadcasted to all nodes of the base level.

**Lemma 1.** Given a linked list of $n$ nodes (each with a value), the algorithm AMF outputs a value within the range of ranks $\frac{n}{2} \pm \frac{n}{2a}$. 
Proof. Let \( m \) be the actual median value, and \( m_l \) and \( m_r \) be two consecutive values in the sorted list of values received by the left-most node of the skip list at the level \( h \), such that \( m_l \geq m \geq m_r \). Clearly either \( m_l \) or \( m_r \) is picked as the approximate median, determined by their final right and left ranks. To prove this lemma, we shall quantify the maximum possible number of values discarded from range \((m_l, m_r)\) at any level.

Let \( S_d^x \) denote the set of values that node \( x \) receives at level \( d \) from nodes at level \( d-1 \) (including own values of node \( x \)). From the construction of the skip list, it is ensured that no two consecutive nodes are supported by less than \( a/2 \) or more than \( 2a \) nodes. Therefore, size \( |S_d^x| \) for any node \( x \) at any level \( d > \lceil \log_{a/2} h \rceil + 2 \) must be in between \( a^2 h/2 \) and \( 2a^2 h \).

Let \( I_d^x \) denote the sampling interval for node \( x \) at level \( d \). Obviously, \( I_d^x = \frac{|S_d^x|}{ah} - 1 \).

Each node \( x \) from level \( h-1 \) (i.e. \( x \in l_{h-1} \)) contributes \( ah \) values to the sorted list processed by the left-most node at level \( h \). Therefore, any node \( x \in l_{h-1} \) can discard at most \( I_{h-1}^x \) values between \( m_l \) and \( m_r \). To maximize the number of values discarded between \( m_l \) and \( m_r \), each of \( I_{h-1}^x \) values between \( m_l \) and \( m_r \) in \( S_{h-1}^x \) must come from different nodes at the lower level \( h-2 \). Let \( s \in S_{h-1}^x \), thus \( s \) is one of the sampled values from node \( y \in l_{h-2} \). if \( m_l < s < m_r \), then there can be at most \( 2I_{h-2}^y \) values from range \((m_l, m_r)\) in \( S_{h-2}^y \) (considering two sampling intervals from both sides). Hence, \( S_{h-2}^y \) can have at most \( 2I_{h-2}^y + 1 \) values at level \( h-2 \) from range \((m_l, m_r)\).

Again to maximize the number of discarded values from range \((m_l, m_r)\), all these \( 2I_{h-2}^y + 1 \) values must come from all possible \( I_{h-2}^y + 1 \) nodes at level \( h-3 \); only one node from level \( h-3 \) contributes one (sampled) value, and each of rest of the \( I_{h-2}^y \) nodes contributes 2 values each. Therefore, for any node \( x \in l_{h-3} \), there can be at most \( 3I_{h-2}^x + 2 \)
values from range \((m_l, m_r)\) in \(S_{h-2}^x\).

Similar reasoning can show that there can be at most \((h - k - 1)I_{k}^x + k\) values from range \((m_l, m_r)\) in \(S_{k}^x\) for any node \(x \in l_k\), where \(k = \lceil \log_{a/2} h \rceil + 2\). However, \(S_{k}^x\) has \((ah + 1)I_{k}^x\) nodes and \(k\) is the lowest level where values were discarded through sampling.

Since \(\frac{(h-k-1)I_{k}^x + k}{(ah+1)I_{k}^x} < \frac{1}{a}\), there are less than \(n/a\) values in between \(m_l\) and \(m_r\). Thus at least one of the values between \(m_l\) and \(m_r\) must fall in the range of ranks \(\frac{n}{2} \pm \frac{n}{2a}\).  

### 3.3 Analysis

Table 3.1 presents a list of frequently used notations used in this section.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m(x))</td>
<td>membership vector of node (x)</td>
</tr>
<tr>
<td>(V_{j}^{i})</td>
<td>membership vector bit of node (i) for level (j)</td>
</tr>
<tr>
<td>(T_{j}^{i})</td>
<td>timestamp for node (i) at level (j)</td>
</tr>
<tr>
<td>(G_{j}^{i})</td>
<td>group-id for node (i) at level (j)</td>
</tr>
<tr>
<td>(D_{j}^{i})</td>
<td>is-dominating-group for node (i) at level (j)</td>
</tr>
<tr>
<td>(B_{x})</td>
<td>group-base of node (x)</td>
</tr>
<tr>
<td>(l_{a})</td>
<td>a linked list at level (a)</td>
</tr>
<tr>
<td>(S_{t})</td>
<td>skip graph at time (t)</td>
</tr>
<tr>
<td>(M)</td>
<td>approximate median priority</td>
</tr>
<tr>
<td>(T_{i}(x, y))</td>
<td>The working set number for node pair ((x, y)) at time (i)</td>
</tr>
<tr>
<td>(WS(\sigma))</td>
<td>(\sum_{i=1}^{m} \log(T_{i}(\sigma_{i}))), where (\sigma = \sigma_{1}, \sigma_{2}, \ldots, \sigma_{m})</td>
</tr>
<tr>
<td>(\sigma_{i})</td>
<td>Communication request ((u_{i}, v_{i})) at time (i)</td>
</tr>
<tr>
<td>(d_{S_{i}}(x, y))</td>
<td>Distance between nodes (x) and (y) in skip graph (S_{i})</td>
</tr>
</tbody>
</table>

Table 3.1: Notations
Lemma 2. Let $g_d$ be a group at level $d$ and $x$ be a node in $g_d$ in skip graph $S$. Let $G_x(V, E)$ be a communication graph where $V$ is the set of all nodes in $S$ and $E$ represents only communications during the time period starting from time $T_{d}^{x}$, and ending at time $t$ (inclusive). All nodes $y \in g_d$ with $T_{d}^{y} > T_{d}^{x}$ are connected in $G_x$.

Proof. We present a proof by induction for this lemma. Let us assume that the lemma holds for all groups in $S_t$. We show that the lemma holds for $S_{t+1}$ as well. Let nodes $u_t$ and $v_t$ communicate at time $t$. By assumption, the lemma holds for all groups of nodes $u_t$ and $v_t$ in $S_t$. Now, since nodes $u_t$ and $v_t$ get connected in the communication graph at time $t$, priority rule P2 and timestamp rule T2 ensure that the lemma holds for any newly formed group containing nodes $u_t$ and $v_t$ in $S_{t+1}$.

Now, for any newly formed group in $S_{t+1}$ that does not contain nodes $u_t$ and $v_t$, there are two possibilities. The first possibility is that all nodes of such a new group had a positive priority during the transformation from $S_t$ to $S_{t+1}$ while receiving $M$ for level $d$. For this case, the lemma holds because (1) all these nodes were in at least one group of either $u_t$ or $v_t$ in $S_t$, and (2) timestamp rule T3 ensures that the new timestamps for the corresponding level are consistent with the lemma. The second possibility is that all node of the new group had a negative priority for level $d$ during the transformation. Such a new group can only be created by a split of a group of size greater than two-third of the number of nodes in the corresponding linked list. Because of the use of boolean variable is-dominating-group, these groups are identical of one of the groups in $S_t$ at level $d + 1$. Hence the lemma holds.

Now we analyze the base case for induction. Clearly, the lemma holds for $S_1$ since
all groups are the only member of their groups. Now, from the construction of the algorithm, it is easy to see that the lemma holds for $S_2$ as the groups with size $> 1$ are the groups that contain nodes $u_1$ and $u_2$ and timestamp rules T1 ensures that the timestamps in $S_2$ are consistent.

Lemma 3. Let $g_d$ be a group at level $d$ in $S_t$ such that for all pair of nodes $(x, y) \in g_d$, distance $d_{S_t}(x, y) = O(\log T_t(x, y))$, where $T_t(x, y)$ is the working set number for the node pair $(x, y)$ at time $t$. If $g_d$ is split into 2 new (sub)groups at level $d$ in $S_{t+1}$ due to a negative $M$, for all pair of nodes $(x, y) \in g_d$, $d_{S_{t+1}}(x, y) = O(\log T_{t+1}(x, y))$.

Proof. Let us consider a pair $(x, y)$ such that the distance between nodes $x$ and $y$ increased due to the split (i.e. $d_{S_{t+1}}(x, y) > d_{S_t}(x, y)$). Then one of the nodes from pair $(x, y)$ must move to the 0-subgraph and the other node must move to the 1-subgraph at level $d$ in $S_{t+1}$. Let us assume that, node $x$ moves to the 0-subgraph and node $y$ moves to the 1-subgraph.

A group can split due to a negative $M$ only if the size of the group is bigger than two-third of the size of the corresponding linked list. From Section 3.1.3, clearly nodes moving to the 1-subgroup (due to a split resulted by negative $M$) have $D^x_d = True$. Now, nodes set is-dominating-group as $True$ only on formation of a group (due to a positive $M$) that contains the communicating nodes. Hence, according to the timestamp rule T2, all nodes $x$ with $D^x_d = True$ have $T^x_d \geq M_p$, where $M_p$ is the positive priority used (in past) to set $D^x_d = True$. This implies that, after time $M_p$, the node $x$ did not communicate with any of the nodes in the group moving to 1-subgraph. According to the definition of
working set number, this implies that at least $T_t(x, y)$ nodes move to the 1-subgraph due to the split resulted by a negative $M$.

We construct a communication graph $G_M$ with communications during the time period starting from time $M_p$ and ending at the current time $t$ (inclusive). According to Lemma 2, all nodes moved to the 1-subgraph are connected in $G_M$. Lemma 1 implies that the number of nodes moved to the 1-subgraph is at most $n' + n'/2a$, where $n'$ is the size of the corresponding linked list. This follows:

$$d_{S_{t+1}}(x, y) \leq a \log_{3/2} \left( \frac{n'}{2} + \frac{n'}{2a} \right) + a = O(\log T_{t+1}(x, y))$$  \hspace{1cm} (3.3)

\hfill \Box

**Lemma 4.** Given that nodes $u$ and $v$ communicate at time $t$, there exists a direct link between nodes $u$ and $v$ in $S_{t+1}$ at a level no higher than $\log_{2a+1} n$.

**Proof.** As described in the case 1 in Section 3.1.3, any group containing communicating nodes $u$ and $v$ at any level $d$ can split at level $d + 1$ only if the nodes in the group receive a positive $M$. Now, according to Lemma 1, a positive $M$ can split a subgraph of size $n$ into two subgraphs where size of any of the new subgraphs is at most $n/2 + n/2a$. Since communicating nodes always receive a positive $M$, the maximum possible height in $S_{t+1}$ at which nodes $u$ and $v$ move to a subgraph of size 2 is $\log_{2+2a} n = \log_{2a+1} n$. \hfill \Box

**Lemma 5.** The maximum possible height after a transformation by $DSG$ is $\log_2 n$.

**Proof.** Algorithm DSG split a subgraphs into two smaller subgraphs at the immediate upper level. From Section 3.1.3, it is easy to see that a subgraph of size $n$ can split into
two subgraphs where the size of any of the new subgraphs is at most $\frac{2n}{3}$. Therefore, the maximum possible height of the skip graph after a transformation is $\log_2 n$.

\[ \text{Theorem 2.} \text{ At any time} \ t, \text{ given that any two nodes} \ u \text{ and} \ v \text{ communicated earlier, the distance between} \ u \text{ and} \ v \text{ in skip graph} \ S_t \text{ is} \ O(\log T_t(u, v)), \text{ where} \ T_t(u, v) \text{ is the working set number for the node pair} \ (u, v) \text{ at time} \ t. \]

\text{Proof.} \text{ Let} \ t' \text{ be a time between} \ t \text{ and the last time} \ u \text{ and} \ v \text{ communicated. Let} \ k \text{ be the longest common postfix between} \ m(u) \text{ and} \ m(v) \text{ in skip graph} \ S_{t'}, \text{ then the distance between} \ u \text{ and} \ v \text{ in} \ S_{t'} \text{ is at most} \ ak. \text{ Suppose node} \ u_t \text{ communicates with node} \ v_t \text{ at time} \ t'. \text{ Let} \ z = H_t - k, \text{ then there exists a linked list} \ l_z \text{ at level} \ z \text{ such that} \ u \in l_z, v \in l_z. \text{ Now one of the following four cases must occur:}

\text{Case 1:} \ u_t \in l_z, v_t \in l_z, \text{ and} \ u_t \text{ and} \ v_t \text{ are in the same linked lists at level} \ z + 1. \text{ Clearly, the distance between} \ u \text{ and} \ v \text{ remains unchanged in skip graph} \ S_{t+1}.

\text{Case 2:} \ u_t \in l_z, v_t \in l_z, \text{ and} \ u_t \text{ and} \ v_t \text{ are in two different linked lists at level} \ z + 1. \text{ Clearly, the distance between} \ u \text{ and} \ v \text{ cannot be increased in skip graph} \ S_{t+1}.

\text{Case 3:} \ u_t \in l_z, v_t \notin l_z \text{ or vice versa. We analyze only the case when} \ u_t \in l_z, v_t \notin l_z, \text{ and analysis for the opposite case} \ u_t \notin l_z, v_t \in l_z \text{ is similar. Since} \ u \text{ and} \ v \text{ communicated earlier, they must hold the same group-id for level} \ z, \text{ i.e.} \ G^u_z = G^v_z. \text{ Let} \ g^u_z \text{ be the group at level} \ z \text{ that contains both nodes} \ u \text{ and} \ v. \text{ Let} \ \alpha \text{ be the highest common level in} \ S_{t'} \text{ for communicating nodes} \ u_t \text{ and} \ v_t. \text{ Then if} \ u_t \notin g^u_z, \text{ the rearrangement will not increase the distance between} \ u \text{ and} \ v \text{ unless the group} \ g^u_z \text{ splits at level} \ z \text{ due to a negative} \ M \text{ (case 2 in Section 3.1.3). According to Lemma 3, the distance between} \ u \text{ and} \ v \text{ in skip} \]
graph $S_{t'}$ remains $O(\log T_{t'+1}(u, v))$, even if group $g_z^u$ splits at level $z$ due to a negative $M$.

Now, if $u_t \in g_z^u$, let $X = \{x \in l_{\alpha}| P(x) > \min(P(u), P(v))\}$. we argue that $|X| \leq T_{t'+1}(u, v)$, which proves the lemma for this case since $d_{S_{t'+1}}(u, v) = O(\log |X|)$.

Let $l_{\alpha+1}^u$ and $l_{\alpha+1}^v$ be the linked lists in $S_{t'}$ at level $\alpha + 1$ such that $u_t \in l_{\alpha+1}^u$ and $v_t \in l_{\alpha+1}^v$. We construct a communication graph $G'$ for communications during the time period between the time $\min(P(u), P(v))$ and the current time $t$ (inclusive). According to Lemma 2, all nodes $x \in l_{\alpha+1}^u$ with $P(x) > \min(P(u), P(v))$ are connected in the communication graph $G'$. Similarly, all nodes $x \in l_{\alpha+1}^v$ with $P(x) > \min(P(u), P(v))$ are connected in communication graph $G'$ as well. Now since nodes $u_t$ and $v_t$ communicate at time $t'$, all nodes $x \in X$ must be connected in $G'$. Based on our definition of working set number, $|X| \leq T_{t'+1}(u, v)$ follows.

Case 4: $u_t \notin l_z, v_t \notin l_z$. Two possible scenarios under this case: (1) neither node $u_t$ nor node $v_t$ belongs to the group of nodes $u$ and $v$ at any level; and (2) node $u_t$, or node $v_t$, or both nodes $u_t$ and $v_t$ belong to the group of nodes $u$ and $v$ at a level lower than $z$.

The first scenario is equivalent to the scenario $G_z^{u_t} \neq G_z^u$ described in case 3. To analyze the second scenario, let $z'$ be the level such that $G_{z'}^{u_t} = G_{z'}^u = G_{z'}^v$. Now as transformation takes place recursively at different levels, the scenario is equivalent to the scenario $u_t \in g_z^u$ described in case 3 as long as both nodes $u$ and $v$ move to the 0-subgraph. However, if both nodes $u$ and $v$ move to the 1-subgraph at some level, the scenario becomes equivalent to the scenario $G_z^{u_t} \neq G_z^u$ described in case 3.

\[\square\]
**Theorem 3.** Given a communication sequence $\sigma$, the expected running time of algorithm $\text{DSG}$ is $(WS(\sigma))^2$ rounds.

The proof relies on the fact that the most expensive operation performed by $\text{DSG}$ is to find the approximate median priority for all involved levels for communication request $\sigma_t$. The most expensive operation performed by a single call of algorithm $\text{AMF}$ is to construct the balanced skip list. Due to the randomization involved in construction, the expected time to construct a balanced skip list is $O(h)$, where $h$ is the height of the skip list.

**Theorem 4.** The routing cost for algorithm $\text{DSG}$ is at most a constant factor more than the amortized routing cost of the optimal algorithm.

*Proof.* Follows from Theorems 1 and 2.

**Theorem 5.** The cost for algorithm $\text{DSG}$ is at most logarithmic factor more than the amortized cost of the optimal algorithm.

*Proof.* Follows from Theorems 1 and 3.

### 3.4 Conclusion

We present a self-adjusting algorithm for skip graphs that relies on the idea of grouping frequently communicating nodes at different levels and using timestamps to determine nodes’ attachments with their groups. We believe this study will lead to a general framework for distributed data structures with overlapping tree-like structures.

Our algorithm $\text{DSG}$ can be useful in networks where multiples levels are involved. For example, VM migration problem in data centers with levels such as rack-level, intra-
and inter-data-center level, inter-continental level etc. Moreover, while the amortized routing time for most self-adjusting data structures is $O(\log n)$, our algorithm DSG guarantees $O(\log n)$ routing time for each of the individual communication requests. Thus, compared to most other self-adjusting networks, DSG is better suited for the cases where there is a time limit associated with each of the communications.
CHAPTER 4
LOCALLY SELF-ADJUSTING HYPERCUBIC NETWORKS

4.1 Introduction

In this chapter we present a self-adjusting algorithm DYHYPES for hypercubic networks. The self-adjusting model we use for hypercubic networks is presented in Chapter 2. In summary, we use CONGEST model for communications, $O(\log n)$ bit memory for each node, and after any communication $(u, v)$, communicating nodes $u$ and $v$ get attached with a direct link between each other in the transformed network.

Upon a communication request, algorithm DYHYPES performs routing using the standard routing algorithm of hypercubic networks and then partially transforms the network conforming to our self-adjusting model. Both the routing and transformation costs of DYHYPES are at most a constant factor more than that of the optimal algorithm, which implies, algorithm DYHYPES improves the transformation cost by a logarithmic factor over algorithm DSG. A comparison between DSG and DYHYPES is presented in Table 4.1.

The structure of hypercubic networks is more rigid compared to that of the skip graphs as nodes are split into two exact halves as they are placed in the 0-networks and 1-networks in their equivalent tree model. This structural rigidity is a major challenge for self-adjustment in hypercubic networks.
### Table 4.1: Results summary

<table>
<thead>
<tr>
<th></th>
<th>DSG</th>
<th>DyHYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing cost to serve a communication sequence $\sigma$</td>
<td>$O(WS(\sigma))$</td>
<td>$O(WS(\sigma))$</td>
</tr>
<tr>
<td>Transformation cost to serve a communication sequence $\sigma$</td>
<td>$O((WS(\sigma))^2)$</td>
<td>$O(WS(\sigma))$</td>
</tr>
<tr>
<td>Total cost to serve a communication sequence $\sigma$</td>
<td>$O((WS(\sigma))^2)$</td>
<td>$O(WS(\sigma))$</td>
</tr>
<tr>
<td>Routing cost factor to the working set bound</td>
<td>constant</td>
<td>constant</td>
</tr>
<tr>
<td>Routing cost factor to the optimal cost</td>
<td>constant</td>
<td>$O(\log \log n)$</td>
</tr>
<tr>
<td>Total cost factor to the working set bound</td>
<td>logarithmic</td>
<td>constant</td>
</tr>
<tr>
<td>Total cost factor to the optimal algorithm</td>
<td>$O(\log n)$</td>
<td>$O(\log \log n)$</td>
</tr>
<tr>
<td>Memory per node</td>
<td>$O(\log n)$ bits</td>
<td>$O(\log n)$ bits</td>
</tr>
<tr>
<td>Communication model</td>
<td>$CONGEST$</td>
<td>$CONGEST$</td>
</tr>
</tbody>
</table>

#### 4.2 Proposed Algorithm

##### 4.2.1 Algorithm Overview and Notations

We propose a randomized and distributed self-adjusting algorithm DyHYPES for hypercubic networks to perform topological adaptation to unknown communication patterns. Upon a communication request, our algorithm DyHYPES first establishes communication using the standard routing algorithm of the network, and then partially transforms the topology according to our self-adjusting model.

Let $H$ be the height of the tree modeling of the network at time $t$. Similar to DSG, DyHYPES allows nodes to form groups at different levels. The purpose of a group is to keep frequently communicating nodes together by placing them together in the network. At any time $t$, we require every node to store a group-id and a timestamp for each of the $H$ levels. Let $G_d^x$ and $T_d^x$ denote the group-id and the timestamp, respectively, of node $x$. 
for level \( d \). Each node of a group at some level holds the same group-id for that level, and the timestamp of a node for some level indicates the node’s attachment to its group at that level.

In this chapter, we refer to a subgraph (or a subnetwork) as a \textit{subtree}. For any node \( x \), it’s subtree at level \( d \) is denoted as \( s^x_d \). The network at time \( t \) is denoted as \( N_t \).

For any time \( t \) and any node \( x \), \( G_x(A, t) \) denotes node \( x \)’s connected component in the communication graph drawn until time \( t \) (inclusive), starting from time \( A \). Also, if \( t \) is referred to as the current time, we often write \( G_x(A) \) instead of \( G_x(A, t) \).

**Definition (Group).** At any time \( t \), a set of nodes \( S \) are considered to be in a group at a level \( d \) if the following are true:

1. There exist a start membership vector \( S_d \) and an end membership vector \( E_d \) such that \( E_d > S_d \), and the number of nodes in \( S \) is exactly \( E_d - S_d \). All the nodes in \( S \) are positioned together in the positions ranged by the start and end membership vectors in the network \( N_t \).

2. Let \( d' \) be the highest level such that \( T^x_{d'} \geq 0 \) and \( d' \geq d \). All nodes \( x \in S \) are positioned in subtree \( s^x_{d'} \), and all nodes of the 0-subtree at level \( d' + 1 \) in subtree \( s^x_{d'} \) belong to \( S \).

3. \( S \subseteq G_x(T^x_d) \) for any node \( x \in S \).

It is possible that nodes of a group at a level are split into multiple groups at upper levels. Figure 4.1 shows an example of groups at different levels in a hypercubic network.
Figure 4.1: Tree representation of a hypercubic networks of 8 nodes. Green boxes indicate groups at different levels. Nodes are placed in the incremental order of their membership vectors from left to right.

Our algorithm DYHYPES maps the groups linearly in the network. In other words, we consider the node with the lowest membership vector as the left most node of the network, and the node with the highest membership vector as the right most node of the network, and all other nodes are ordered from left to right in the ascending order of their membership vectors. As the membership vectors of nodes change during transformation when nodes change their position, our algorithm ensures that all nodes of a group are placed in the network adjacently. This implies that each group has a start and an end membership vectors (the membership vectors of the left-most and right-most nodes of the group). For a group at level $d$, the start and end membership vectors are denoted as $S_d^x$ and $E_d^x$, respectively, where $x$ is any node in the group.

We say groups $g_1, g_2, \cdots, g_k$ at any level $d$ are relatives to each other if all nodes of
these groups are in $G_x(\mathcal{T})$, where $\mathcal{T} = \min(T_1, T_2, \ldots, T_k)$, where $T_i = T_{xD}^i$ such that (1) $T_{xD}^i \geq 0$, (2) $D$ is the lowest level with $D \geq d$, and (3) $x$ is any node in group $g_i$. Which implies, the relative groups should form one single group to ensure that the working set property for the nodes of the groups are not violated.

4.2.2 Major Algorithmic Challenges

There are few algorithmic challenges that we need to address. First, when two nodes from two different groups communicate, we merge the communicating groups. While merging, we need to move other groups to make room for the merging groups. This may result in violation of working set property for any non-communicating groups moved due to the transformation.

Second, it is possible that nodes of multiple groups belong to the same connected component from the communication graph drawn for all communications until the time of current communication. This is especially true when multiple groups are split from the same group at a lower level, as shown in Figure 4.1. Thus, while moving one of such groups to a new position in the network, a challenge is not to violate the working set property for the nodes across the groups.

Third, in hypercubic networks, nodes of a subtree at any level are split into two equally sized subtrees at the immediate upper level. While merging two groups, it is possible that the merged group is too big to fit in a subtree. In that case we need to split the merged group, which may result in violation of the working set property of the nodes of the split groups.
4.2.3 Our Key Idea to Address the Challenges

Our algorithm DYHYPES focuses on minimizing the number of “relative” groups in the network at any point of time. Note that, due to the structural rigidity of hypercubic networks, it is impossible to entirely eliminate the existence of the relative groups. To this end, DYHYPES picks randomly chosen groups and split them into relative groups and at the same time keeps the number of relative groups in the network low. This forces the “adversary” to try communications within different non-split groups to increase the likelihood of finding a communication between separated relative groups. We refer to the communications across the relative groups as “bad communications” and any other communications as “good communications”.

Our algorithm DYHYPES charges any communication more than their working set numbers. Thus, “good communications” are overcharged to pay for the “bad communications”. In other words, algorithm DYHYPES ensures that, the expected number of “good communications” are large compared to that of the “bad communications”.

To keep number of relative groups low, we maintain the following invariant at all time:

**Invariant I:** At any time \( t \), for any subtree \( s_d \) at any level \( d \), there are at most two groups at level \( d \) that are relatives to each other such that one of the two groups are in the 0-subtree and the other is in 1-subtree at level \( d + 1 \) in \( s_d \).

We say a pair of relative groups \( d \)-relatives if one of the groups is positioned in some subtree \( s_{d+1} \) at level \( d + 1 \), and the other is positioned in the complementary subtree \( \sim s_{d+1} \). To keep track of relative groups, each node holds information about the pair of
relative groups for each level. More specifically, for any level \( d \), a node \( x \) needs to hold four variables: the membership vectors of the start (left most) and end (right most) nodes of the group from \( d \)-relative groups (if exists) in the subtree \( s_{d+1}^x \), and the membership vectors of the start and end nodes of the other \( d \)-relative group in the subtree \( \sim s_{d+1}^x \). We denote these variables as \( S_d^x \), \( E_d^x \), \( \sim S_d^x \), and \( \sim E_d^x \), respectively. These variables may set to NULL if the corresponding pair of relative groups do not exist.

To address the third challenge listed in Section 4.2.2, algorithm DYHYPES uses a randomized approach. When there is a need to evict a subgroup (i.e., a group inside a communicating group) from a subtree to make room for a communicating group, DYHYPES chooses a random subgroup for eviction and ensures the distance between the subgroup with any of its relative increases by no more than one caused by the transformation following a communication. We show that, with this approach, the expected distance between any two nodes remain at most constant factor more than the logarithm of the working set number of the nodes.

We summarize all the notations introduced so far in Table 4.2.

4.2.4 The Algorithm: Dynamic Hypercubic Networks (DYHYPES)

Upon a communication request and routing, communicating nodes \((u, v)\) record the level of the smallest common subtree that contains both nodes \( u \) and \( v \). Let the recorded level be \( \alpha \). Our algorithm DYHYPES performs transformation conforming to the self-adjusting model to move the communicating nodes in a subtree of size 2. Transformation may take up to three steps depending upon the existing network topology. These three steps
<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m(x)$</td>
<td>membership vector of node x</td>
</tr>
<tr>
<td>$s_d^x$</td>
<td>Node x’s subtree at level d</td>
</tr>
<tr>
<td>$\sim s_d^x$</td>
<td>Node x’s complementary subtree at level d</td>
</tr>
<tr>
<td>$s_{d \text{rand}}^x \subset s_d^y$</td>
<td>A subtree at level d randomly chosen from subtree $s_d^y$, where $y &lt; d$.</td>
</tr>
<tr>
<td>$N_t$</td>
<td>Network at time $t$</td>
</tr>
<tr>
<td>$T_i(x, y)$</td>
<td>The working set number for node pair $(x, y)$ at time $i$</td>
</tr>
<tr>
<td>$WS(\sigma)$</td>
<td>$\sum_{i=1}^{m} \log(T_i(\sigma_i))$, where $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_m$</td>
</tr>
<tr>
<td>$\sigma_i$</td>
<td>Communication request $(u_i, v_i)$ at time $i$</td>
</tr>
<tr>
<td>$dN_i(x, y)$</td>
<td>Distance between nodes $x$ and $y$ in network $N_t$</td>
</tr>
<tr>
<td>$G_{x}(a)$</td>
<td>Node $x$’s connected component in the communication graph drawn for the time interval $[a, t]$ where $t$ is the current time</td>
</tr>
<tr>
<td>$G_d^x$</td>
<td>group-id of node $x$ for level $d$</td>
</tr>
<tr>
<td>$T_d^x$</td>
<td>timestamp for node $x$ at level $d$</td>
</tr>
<tr>
<td>$C_d^x$</td>
<td>counter of node $x$ at level $d$</td>
</tr>
<tr>
<td>$S_d^x$</td>
<td>membership vector for the start (i.e. left most) node of the group</td>
</tr>
<tr>
<td>$\sim S_d^x$</td>
<td>membership vector of the start node of the $d$-relative group in subtree $s_d^x$</td>
</tr>
<tr>
<td>$\sim S_d^x$</td>
<td>membership vector of the start node of the $d$-relative group in subtree $s_d^x$</td>
</tr>
<tr>
<td>$E_d^x$</td>
<td>membership vector of the end node of the $d$-relative group in subtree $s_d^x$</td>
</tr>
<tr>
<td>$\sim E_d^x$</td>
<td>membership vector of the end node of the $d$-relative group in subtree $s_d^x$</td>
</tr>
</tbody>
</table>

Table 4.2: Notations
are described in the following subsections.

4.2.4.1 Subtree Rotation

Let \( l(u) \) be the lowest level such that a pair of \( l(u) \)-relative groups exist in subtree \( s^u_{l(u)} \). Similarly, let \( l(v) \) be the lowest level such that a pair of \( l(v) \)-relative groups exist in subtree \( s^v_{l(v)} \). If \( \alpha \) happens to be lower than both \( l(u) \) and \( l(v) \), we transform the network to make subtrees \( s^u_{\min(l(u),l(v))} \) and \( s^v_{\min(l(u),l(v))} \) complements of each other. Note that, this will not move any group away from its relative groups.

This transformation is done by recursive subtree rotations. At the beginning, the subtree \( s^u_{\alpha+2} \) is swapped with the subtree \( \sim s^v_{\alpha+2} \). We refer this rotation as the level \( \alpha+1 \) rotation. Note that, after this rotation, the distance between nodes \( u \) and \( v \) is reduced by 1. Similar rotations take place recursively at levels \( \alpha+3, \alpha+4, \cdots, \min(l(u),l(v)) \). This step is also summarized in Algorithm 5.

As an example, Figure 4.2 shows nodes A and E moves next to each other using subtree rotations.

---

**Algorithm 3: ROTATE\( (s^u_x, s^v_y) \)**

1. Let \( \beta \) be the highest level such that \( s^u_\beta = s^v_\beta \).
2. while \( \beta < \min(x, y) + 1 \) do
   3. swap \( s^u_{\beta+1} \) with \( \sim s^v_{\beta+1} \).
   4. \( \beta \leftarrow \beta + 1 \).
3. End result: \( s^u_x \subset s^u_{\min(x,y)+1} \) and \( s^v_y \subset s^v_{\min(x,y)+1} \) (i.e., \( s^u_{\min(x,y)+1} = s^v_{\min(x,y)+1} \)).
(a) Network when nodes A and E communicate.

(b) After subtree rotation at level 1.

(c) After subtree rotation at level 2.

Figure 4.2: Subtree rotation to move nodes A and E in a subtree of size 2.
4.2.4.2 Inter-Group Transformation

In this step we bring the communicating groups adjacent to each other. This step is necessary only if the communicating nodes are from two different groups at level $\alpha$. Let, $g(u)$ be the highest level such that the node $u$’s group at level $g(u)$ is not a strict subset of any other group at any of the levels lower than $g(u)$. Let $g(v)$ is the similar highest level for node $v$. If the communicating nodes belong to different groups at level $\alpha$ (meaning, $G_u^\alpha \neq G_v^\alpha$) and either $g(u)$ or $g(v)$ is greater than $\alpha$, then as a result of this step, node $u$’s and $v$’s groups move next to each other to merge together.

Our algorithm DYHYPES ensures that the Invariant I is preserved after this transformation. If there is any $\alpha$-relative groups in the current network, both nodes $u$ and $v$ are aware of their existence as they hold the information in variables $S_u^\alpha, \sim S_u^\alpha$ etc. Suppose $A$ and $B$ are the groups of nodes $u$ and $v$ at level $\alpha$, and $C_1$ and $C_2$ are the $\alpha$-relative groups, as shown in Figures 4.3(a) and 4.4(a).

Let group $A$ be larger than group $B$ in size (this can be computed using the variables $S_u^\alpha, E_u^\alpha, S_v^\alpha$ and $E_v^\alpha$), so we move group $B$ next to group $A$ as shown in Figures 4.3(c) and 4.4(c). If the size of group $B$ is larger than the size of group $C_1$, $B$ can be swapped with a portion of $C_1$ as shown in Figure 4.3(b). However, if the size of $B$ is smaller than the size of $C_1$, we randomly choose a membership vector $m_r$ in subtree $s_{\alpha+1}^u$ excluding the range of membership vectors assigned for the groups $A$ and $C_1$, as shown in Figure 4.4(a). We take a portion of the network $R$ at the position $m_r$ such that the sum of the size of group $A$ and the size of portion $R$ is greater than the size of group $B$. Then we move group $B$ next to group $A$ as shown in Figure 4.4(c).
In order to preserve the Invariant I, this procedure requires following cautions:

1. All the groups involved, including groups $A$, $B$, $C_1$, $C_2$, and groups in portion $R$, cannot be $\alpha + 1$-relative. If any of these groups are $\alpha + 1$-relative, algorithm DyHyPES moves all the $\alpha + 1$-relative groups next to each other recursively.

2. As shown in Figures 4.3 and 4.4, moving groups $A$ and $B$ next to each other, may require moving other groups (for example $A$ and $C_1$) next to each other in subtrees $s_{\alpha+1}^u$ and $s_{\alpha+1}^v$. Similarly, these moves may require move moves at the subtrees of upper levels. All these moves take place recursively and parallelly before groups $A$ and $B$ moves next to each other.

4.2.4.3 Intra-Group Transformation

The inter-group transformation brings communicating groups next to each other only when the communicating groups are not subgroups of other groups at any level lower than $\alpha$. However, when these communicating groups are next to each other (by using inter-group transformation when necessary), we need further transformation to move the communicating nodes in a subtree of size 2. Note that, the groups (at level $\alpha$) that brought next to each other by inter-group transformation may be divided into multiple subgroups at upper levels. Intra-group transformation rearranges these subgroups at different levels using the timestamps stored by the communicating nodes.

Let $d$ be the highest level such that both communicating nodes $u$ and $v$ are in subtree $s_d^u$. We compare the level $d + 1$ timestamps of the communicating nodes. Suppose, $T_{d+1}^u > T_{d+1}^v$ (the algorithm is symmetrical otherwise), then we find the lowest level $d'$, $d' > d + 1$,
(a) Groups $A$ and $B$ are communicating and groups $C_1$ and $C_2$ are $\alpha$-relative groups.

(b) Groups $A$ and $C_1$ move next to each other without violating Invariant I. Similarly groups $B$ and $C_2$ move next to each other in the 1-subtree at level $\alpha + 1$.

(c) Groups $A$ and $B$ move next to each other without violating Invariant I.

Figure 4.3: Intergroup transformation example: group $B$ moves next to group $A$, and the size of group $B$ is larger than the size of the $\alpha$-relative group $C_1$. 
(a) Groups $A$ and $B$ are communicating and groups $C_1$ and $C_2$ are $\alpha$-relative groups. $R$ is the randomly chosen portion of the network.

(b) Groups $A$, $C_1$ and $R$ move next to each other without violating Invariant I. Similarly groups $B$ and $C_2$ move next to each other in the 1-subtree at level $\alpha + 1$.

(c) Groups $A$ and $B$ move next to each other without violating Invariant I.

Figure 4.4: Intergroup transformation example: group $B$ moves next to group $A$, and the size of group $B$ is smaller than the size of the $\alpha$-relative group $C_1$. 
such that $T^v_d > T^u_{d+1}$. It is important to note that we set $T^x_H = \infty$ for every node $x$, where $H$ is the highest possible level (i.e. the height) in the network.

At this point, we find a random node $r$ in the subtree $\sim s^u_{d+2}$ and check the start and end membership vectors of the node $r$’s subgroup at level $d + 2$. Our goal is to swap node $r$’s subgroup at level $d + 2$ with node $v$’s subgroup at level $d'$. If the size of the both groups are exactly the same (it might be possible that both subgroups has only one node or exactly the same number of nodes), then we swap the subgroups. Otherwise, we move node $v$’s subgroup in subtree $s^u_{d+2}$ using the technique similar to inter-group transformation to ensure that the Invariant I is preserved.

The transformation described above for level $d$ is repeated for all levels $d + 1, d + 2, \cdots, H$ until nodes $u$ and $v$ moves to a subtree of size 2. Figure 4.5 shows an example of intragroup transformation.

**Timestamp update:** Each node $x$ has a counter $C^x_d$ for each level $d$. All the counters are initialized to zero at the beginning. Whenever one or more nodes are swapped in any subtree $\sim s^u_{d+1}$, all nodes $x \in s^u_d$ update their timestamp as $T^x_d \leftarrow T^x_{d+1}$ and update their counter as $C^x_d \leftarrow C^x_d \mod |s^u_{d+1}|$. In other words, the timestamp shifts one level down when the algorithm believes there are sufficient number of nodes in subtree $s^u_d$ that are in the connected component $G_u(T^u_{d+1})$.

Any node $x$ moved away from either of the communicating nodes due to a swap at level $d$ sets $T^x_{d+1} \leftarrow 0$. To see an example, in Figure 4.5(b), node $D$ moved away from communicating node $A$ as it was swapped by node $E$. Node $D$ sets $T^D_1 \leftarrow 0$. 
(a) Upon communication \((A, E)\), intergroup transformation brings node \(A\)'s group and node \(E\)'s group next to each other in subtree \(s^A_0\).

(b) Node \(E\) is swapped with a randomly chosen node \((D)\) in \(\sim s^A_2\) as \(T^E_3 > T^A_1\). Note that \(T^E_3 = \infty\) and \(T^A_1 = 6\).

(c) Node \(E\) is swapped with the only node \((B)\) in \(\sim s^A_3\) as \(T^E_3 > T^A_2\). Note that \(T^E_3 = \infty\) and \(T^A_1 = 6\).

Figure 4.5: Intragroup transformation example for communication \((A, E)\). The green boxes indicate groups at different levels and the numbers below the nodes indicate the node’s timestamp for the corresponding level.
4.3 Analysis

Lemma 6. **(One-Node Timestamp Lemma)** Let us choose an arbitrary node $u$ and perform transforms only for the communications in which node $u$ takes part as a communicating node. For any time $t$ and any level $d$, let $G_d(V, E)$ be a communication graph drawn for all the communications of node $u$ starting from time $T_d^u$. At any time $t$, the expected number of nodes $x \in (s_d^u \cap V)$ is at least $0.72 \cdot |s_d^u|$.

Proof. Let, $v_1, v_2, \cdots$, be the nodes that communicated with node $u$ since time $T_d^u$, and node $v_i$ communicated with node $u$ at time $t_i$, and $t_i < t_j$ if $i < j$.

For time $t$ and levels $d$ and $d'$, let $S(t, d') = (s_{d'}^u \cap V)$ and $\tilde{S}(t, d') = (\sim s_{d'}^u \cap V)$, where $V$ is the set of nodes in the communication graph $G_d(V, E)$ defined in the lemma statement.

We use proof by induction. The induction hypotheses are:

[H1] $E[S(t_i, d+3)] > 0.999 \cdot |s_{d+3}^u|$

[H2] $E[\tilde{S}(t_i, d+3)] > 0.99 \cdot |s_{d+3}^u|$

[H3] $E[\tilde{S}(t_i, d+2)] > 0.86 \cdot |\sim s_{d+2}^u|$

[H4] $E[\tilde{S}(t_i, d+1)] > 0.52 \cdot |\sim s_{d+1}^u|$

[H5] $E[\tilde{S}(t_i, d)] > 0.125 \cdot |\sim s_d^u|$

Note that, the lemma holds automatically if the induction hypotheses are true, since the summation of the inequalities in hypotheses H0-H4 yields $E[S(t_i, d)] \geq (3/4)|s_d^u|$. 
**Base Case.** We first show that the lemma holds for any \( d > H_t - 3 \). Since DYHYPES places communicating nodes in a subtree of size 2, clearly for \( d = H_t - 1 \), the number of nodes \( x \in S_{(t,d)} \) is always 2. Also, for \( d = H_t - 2 \), the number of nodes \( x \in S_{(t,d,d)} \) is at least 3 with probability 1. Thus the lemma holds for \( d > H_t - 3 \).

Now we show that the induction hypotheses hold for \( d = H_t - 3 \). According to DYHYPES, \( T_d^u \) becomes nonzero at time \( t_7 \). Therefore, for \( t < t_7 \), the hypotheses hold trivially for level \( d = H_t - 3 \), as \( T_d^u = 0 \). Also, if the hypotheses hold for \( t = t_7 \), then they will also hold for \( t > t_7 \) because any node that moves in \( s_d^u \) at time \( t_i, \ i > 7 \), is the communicating node \( v_i \), and \( v_i \in (s_d^u \cap V) \). That leaves us in the necessity to show that the hypotheses hold for \( t = t_7 \).

Since a subtree at level \( d \) is split into two subtrees at level \( d + 1 \), we can write the following:

\[
E[|S_{(t,d)}|] = E[|S_{(t,d+1)}|] + E[|\tilde{S}_{(t,d)}|] = E[|S_{(t,H_t-1)}|] + \sum_{j=H_t-1}^{d+1} E[|\tilde{S}_{(t,j)}|] \quad (4.1)
\]

Also, for any subtree \( s \), the expected number of nodes in \( (s \cap V) \) at time \( t_i \) depends on the expected number of nodes in \( (s \cap V) \) at \( t_{i-1} \), the expected number of nodes in \( (s \cap V) \) that move in subtree \( s \) at time \( t_i \), and the expected number of nodes that leave the subtree \( s \) at time \( t_i \). We get:

\[
E[|\tilde{S}_{(t,d)}|] = E[|\tilde{S}_{(t-1,d)}|] + \frac{E[|\tilde{S}_{(t-1,d+1)}|]}{|s_{d+1}^u|} - \frac{E[|\tilde{S}_{(t-1,d)}|]}{|s_{d+1}^u|} \quad (4.2)
\]

\[
= E[|\tilde{S}_{(t-1,d)}|] + \frac{E[|\tilde{S}_{(t-1,d+1)}|]}{2^{H_t-d+2}} - \frac{E[|\tilde{S}_{(t-1,d)}|]}{2^{H_t-d-1}}
\]
As we know, $E[|S(t_i, H_t-1)|] = 2$ for $i \geq 1$, combining Equations 4.1 and 4.2, we get:

$$E[|S(t_i,d)|] = 2 + \sum_{j=H_t-1}^{d+1} \left( E[|\tilde{S}(t_{i-1},j)|] + \frac{E[|\tilde{S}(t_{i-1},j-1)|]}{2^{H_t-j-2}} - \frac{E[|\tilde{S}(t_{i-1},d)|]}{2^{H_t-j-1}} \right)$$  \hspace{1cm} (4.3)

We use Equation 4.3 to calculate $E(S(t_7, H_t-1))$, $E(S(t_7, H_t-2))$ and $E(S(t_7, H_t-3))$ in Appendix A.3, and show that the hypotheses hold for time $t_7$ and $d = H_t - 3$.

**Induction Steps.** We prove each of the hypotheses with following assumptions:

[A1] $E[S(t_i,d+4)] > 0.999 \cdot |s_{d+4}^u|$

[A2] $E[\tilde{S}(t_i,d+4)] > 0.99 \cdot |s_{d+4}^u|$

[A3] $E[\tilde{S}(t_i,d+3)] > 0.86 \cdot |\sim s_{d+3}^u|$

[A4] $E[\tilde{S}(t_i,d+2)] > 0.52 \cdot |\sim s_{d+2}^u|$

[A5] $E[\tilde{S}(t_i,d+1)] > 0.125 \cdot |\sim s_{d+1}^u|$

If $T_d^u = 0$, all the hypotheses (and the lemma) hold trivially. So let us assume $T_d^u > 0$ while proving the hypotheses. According to the inter-group transformation of DYHYPES, in order to have a positive $T_d^u$, there must be at least $|s_d^u| - 1$ nodes communicated with node $u$ since time $T_d^u$. Also, as discussed above, the hypotheses hold for time $t_i$, $i > |s_d^u| - 1$, if they hold for time $t_i$, $i = |s_d^u| - 1$. Hence, it suffices to show that the hypotheses hold for time $t_i$, $i = |s_d^u| - 1$.

We define the $i^{th}$ position of a subtree as the position associated with the $i^{th}$ membership vector (or bit string) in the subtree. Let $A(x, y, t_s, t_e)$ be the event that the node
in the $x^{th}$ position in $\sim s^u_y$ is placed in $\sim s^u_{y-1}$ of the transformed network as a result of a swap performed by an intra-group transformation between times $t_s$ and $t_e$ (inclusive). This means, a node from $\sim s^u_{y+1}$ is also placed in the $x$th position in $\sim s^u_y$ of the transformed network.

Clearly, events $A(x, y, t_s, t_e)$ take place exactly $t_e - t_s$ times. Recall that each position in a subtree is equally likely to be chosen when an event $A(x, y)$ takes place.

Let $x^{d}_{(j,i)}$ be an indicator random variable such that $x^{d}_{(j,i)} = 1$ if the $i^{th}$ position in subtree $\sim s^u_y$ is chosen $j$ times by an event $A(x, d, k', k)$ after time $k$, where $k' = |s^u_{d-1}| - 1$ and $k = |s^u_{d-1}| - 1$; $x^{d}_{(j,i)} = 0$ otherwise. Let $X^d_j = x^{d}_{(j,1)} + x^{d}_{(j,2)} + \cdots + x^{d}_{(j,|\sim s^u_y|)}$.

**Proof of H1.** This hypothesis holds trivially if $d \leq H_t - 4$, as $|s^u_{d+3}| = 2$ and $E[S(t_i,d+3)] = |s^u_{d+3}|$ for $d = H_t - 4$ and $i \geq 1$. Thus, we prove this hypothesis for $d > H_t - 4$ and it suffices to show the following:

[H1A] $E[S(t_i,d)] \geq E[\tilde{S}(t_i,d)]$ for any $d$

[H1B] $E[\tilde{S}(t_i,d+4)] > 0.999 \cdot |s^u_{d+4}|$

Since $(u, v_1)$ is the first communication of node $u$, $E[S(t_1,d)] = 2$ and $E[\tilde{S}(t_1,d)] = 0$ for any $d \geq H_t - 1$. Thus H1B holds for time $t_1$. Now for any communication onward, $E[S(t_1,d)]$ increases exactly by one, as a new node joins $s^u_d \cap V$. However, $E[S(t_1,d)]$ increases by less than one for $d \leq H_t - 2$, because there is always some node $x \notin (\sim s^u_d \cap V)$ with some probability for $d \leq H_t - 2$. Therefore, H1A holds.

---

1This is equivalent to throwing $k - k'$ balls in $| \sim s^u_{d+2} |$ bins, where each ball is thrown into a uniformly random bin, independent of other balls
Now we prove H1B. We compute the expected number of times a position in subtree $\sim s_{d+4}^u$ is chosen at least once during a transformation between time $k'$ and $k$.

$$E[X_{0}^{d+4}] = \sum_{i=1}^{|\sim s_{d+4}^u|} E[x_{(j,i)}^{d+4}] = |\sim s_{d+4}^u| \cdot \left( \frac{|\sim s_{d+4}^u| - 1}{|\sim s_{d+4}^u|} \right)^{k-k'}$$

$$= |\sim s_{d+4}^u| \cdot \left( \frac{|\sim s_{d+4}^u| - 1}{|\sim s_{d+4}^u|} \right)^{8|\sim s_{d+4}^u|} \leq \left( \frac{1}{e} \right)^8 \cdot |\sim s_{d+4}^u|$$

Thus,

$$E[X_{\geq 1}^{d+4}] = |\sim s_{d+4}^u| - E[X_{0}^{d+4}] > \left( 1 - \left( \frac{1}{e} \right)^8 \right) \cdot |\sim s_{d+4}^u| = 0.9996 \cdot |\sim s_{d+4}^u|$$

Each position $i$ with $x_{\geq 1, d+4}^{d+4}$ will be occupied by a node from $s_{d+4}^u$. We get:

$$E[\tilde{S}_{(i, d+4)}] > 0.999 \times E[X_{\geq 1}^{d+4}] + 0.99 \times E[X_{0}^{d+4}] \approx 0.999 \cdot |s_{d+4}^u|$$

**Proof of H2.**

$$E[X_{1}^{d+4}] = \sum_{i=1}^{|\sim s_{d+4}^u|} E[x_{(j,i)}^{d+4}] = |\sim s_{d+3}^u| \cdot \left( \frac{|\sim s_{d+4}^u| - 1}{|\sim s_{d+4}^u|} \right)^{k-k'-1} \leq \left( \frac{1}{e} \right)^8 \cdot |\sim s_{d+4}^u|$$

$$E[X_{\geq 2}^{d+4}] = |\sim s_{d+4}^u| - E[X_{0}^{d+3}] - E[X_{1}^{d+3}] \geq \left( 1 - 2 \cdot \left( \frac{1}{e} \right)^8 \right) \cdot |\sim s_{d+3}^u|$$

$$E[X_{0}^{d+3}] = \sum_{x=1}^{\sim s_{d+3}^u} E[A(x, d + 3)] \leq \left( \frac{1}{e} \right)^4 \cdot |\sim s_{d+3}^u|$$
Each position \( i \) with \( x_{d+3}^{(\geq 1,i)} \) will be occupied by a node from \( s_{d+4}^u \). We get:

\[
E[\tilde{S}_{(t,d+3)}] > 0.86 \times E[X_0^{d+3}] + 0.99 \times E[X_{d+1}^{d+3}] + 0.999 \times (E[X_{d+1}^{d+3}] - E[X_{d+1}^{d+4}])
\]
\[
> 0.87 \times \left( \frac{1}{e} \right)^4 \cdot |s_{d+3}^u| + 0.99 \times \left( 0.9996 \cdot \frac{|s_{d+3}^u|}{2} \right)
\]
\[
+ 0.999 \times \left( |s_{d+3}^u| - E[X_{d+3}^{d+3}] - E[X_{d+1}^{d+4}] \right)
\]
\[
= \left( 0.86 \times 0.0183 + 0.99 \times 0.4998 + 0.999 \times (1 - 0.0183 - 0.4998) \right) \cdot |s_{d+3}^u|
\]
\[
> 0.99 \cdot |s_{d+3}^u|
\]

**Proof of H3.** Similar to the proof of H2, we get,

\[
E[\tilde{S}_{(t,d+2)}] > 0.52 \times E[X_0^{d+2}] + 0.86 \times E[X_{d+1}^{d+3}] + 0.99 \times (E[X_{d+1}^{d+2}] - E[X_{d+1}^{d+3}])
\]
\[
> 0.52 \times \left( \frac{1}{e} \right)^2 \cdot |s_{d+2}^u| + 0.86 \times \left( \left( 1 - \frac{1}{e^t} \right) \cdot \frac{|s_{d+2}^u|}{2} \right)
\]
\[
+ 0.99 \times \left( |s_{d+2}^u| - E[X_{d+2}^{d+2}] - E[X_{d+1}^{d+3}] \right)
\]
\[
= \left( 0.52 \times 0.135 + 0.86 \times 0.4908 + 0.99 \times (1 - 0.135 - 0.4908) \right) \cdot |s_{d+2}^u|
\]
\[
> 0.86 \times |s_{d+3}^u|
\]

**Proof of H4.** We prove this hypothesis in four steps. We divide the time interval \((k', k)\) into four equal subintervals, and calculate expectations for each subinterval using the expectations calculated for the previous subinterval.

Let \( y^d(j, i, t_s, t_e) \) be an indicator random variable such that \( y^d(j, i, t_s, t_e) = 1 \) if the \( i^{th} \) position in subtree \( \sim s_d^u \) is chosen \( j \) times by an event \( A(x, d, t_s, t_e) \) after time \( t_e \);

\( y^d(j, i, t_s, t_e) = 0 \) otherwise. Let \( Y^d(j, t_s, t_e) = y^d(j, 1, t_s, t_e) + y^d(j, 2, t_s, t_e) + \cdots + y^d(j, |s_d^u|, t_s, t_e) \).
\[
E[\tilde{S}(k' + \frac{k - k'}{4}, d + 2)] > 0.52 \times E[Y^{d+2}(0, k', k' + \frac{k - k'}{4})] + 0.87 \times E[Y^{d+2}(\geq 1, k', k' + \frac{k - k'}{4})]
\]
\[
> 0.52 \times \frac{1}{e^{0.5}} |s_{d+2}^{u}| + 0.87 \times \left( 1 - \frac{1}{e^{0.5}} \right) |s_{d+2}^{u}|
\]
\[
= 0.65 \cdot |s_{d+2}^{u}|
\]

Similarly,

\[
E[\tilde{S}(k' + \frac{k - k'}{4}, d + 2)] > 0.65 \times \frac{1}{e^{0.5}} |s_{d+2}^{u}| + 0.87 \times \left( 1 - \frac{1}{e^{0.5}} \right) |s_{d+2}^{u}| = 0.73 \cdot |s_{d+2}^{u}|
\]

\[
E[\tilde{S}(k' + \frac{3(k - k')}{4}, d + 2)] > 0.73 \times \frac{1}{e^{0.5}} |s_{d+2}^{u}| + 0.87 \times \left( 1 - \frac{1}{e^{0.5}} \right) |s_{d+2}^{u}| = 0.78 \cdot |s_{d+2}^{u}|
\]

Now, we use the above expectations to calculate \( E[\tilde{S}(t_i, d+1)] \) in four steps:

\[
E[\tilde{S}(k' + \frac{k - k'}{4}, d + 1)] > 0.14 \times E[Y^{d+1}(0, k', k' + \frac{k - k'}{4})] + 0.53 \times E[Y^{d+2}(1, k', k' + \frac{k - k'}{4})]
\]
\[
+ 0.86 \times (E[Y^{d+1}(\geq 1, k', k' + \frac{k - k'}{4})] - E[Y^{d+2}(\geq 1, k', k' + \frac{k - k'}{4})])
\]
\[
> 0.125 \times \frac{1}{e^{0.25}} \cdot | \sim s_{d+1}^{u} | + 0.53 \times \left( 1 - \frac{1}{e^{0.5}} \right) \cdot \left( \frac{| \sim s_{d+1}^{u} |}{2} \right)
\]
\[
+ 0.86 \times \left( 1 - \frac{1}{e^{0.25}} - \frac{1}{2} \left( 1 - \frac{1}{e^{0.5}} \right) \right) \cdot | \sim s_{d+1}^{u} |
\]
\[
= \left( 0.125 \times 0.778 + 0.52 \times 0.15 + 0.86 \times \left( 1 - 0.778 - 0.15 \right) \right) \cdot | \sim s_{d+1}^{u} |
\]
\[
> 0.24 \cdot | s_{d+1}^{u} |
\]
$E[\tilde{S}_{(k' + \frac{k-k'}{2},d+1)}] > E[\tilde{S}_{(k' + \frac{k-k'}{2},d+1)}] \times E[Y^{d+1}(0, k' + \frac{k-k'}{4}, k' + \frac{k-k'}{2})]$

$+ E[\tilde{S}_{(k' + \frac{k-k'}{2},d+2)}] \times E[Y^{d+2}(1, k', k' + \frac{k-k'}{4})]$

$+ 0.86 \times \left( E[Y^{d+1}(\geq 1, k', k' + \frac{k-k'}{4})] - E[Y^{d+2}(\geq 1, k', k' + \frac{k-k'}{4})] \right)$

$> \left( 0.24 \times 0.778 + 0.66 \times 0.15 + 0.86 \times (1 - 0.778 - 0.15) \right) \cdot |\sim s_{d+2}^u|$

$> 0.34 \cdot |s_{d+1}^u|$

Similarly,

$E[\tilde{S}_{(k' + \frac{3(k-k')}{4},d+1)}] > \left( 0.34 \times 0.778 + 0.73 \times 0.15 + 0.86 \times (1 - 0.778 - 0.15) \right) \cdot |\sim s_{d+2}^u|$

$> 0.44 \cdot |s_{d+1}^u|$

$E[\tilde{S}_{t_d,d+1}] > \left( 0.44 \times 0.778 + 0.78 \times 0.15 + 0.86 \times (1 - 0.778 - 0.15) \right) \cdot |\sim s_{d+2}^u|$

$> 0.52 \cdot |s_{d+1}^u|$

**Proof of H5.** To prove this hypothesis, we use the same four-step approach that we used to prove H4. The expectations calculated for different subintervals in the proof of H4 are used to prove H5. We briefly show the calculations below:

$E[\tilde{S}_{(k' + \frac{k-k'}{4},d)}] > \left( 0 \times \frac{1}{e^{1/8}} + 0.125 \times \frac{1}{2} \times (1 - \frac{1}{e^{1/4}}) + 0.52 \times \left( 1 - \frac{1}{e^{1/8}} - \frac{1}{2} \times (1 - \frac{1}{e^{1/4}}) \right) \right) \cdot |\sim s_{d}^u|$

$= \left( 0 \times 0.88 + 0.125 \times 0.11 + 0.52 \times (1 - 0.88 - 0.11) \right) \cdot |\sim s_{d}^u|$

$> 0.018 \cdot |s_{d}^u|$
\[ E[\tilde{S}_{(k' + \frac{k-k'}{2},d)}] > \left(0.018 \times 0.88 + 0.24 \times 0.11 + 0.66 \times (1 - 0.88 - 0.11)\right) \cdot |s^u_d| \]
\[ > 0.04 \cdot |s^u_d| \]

\[ E[\tilde{S}_{(k' + \frac{3(k-k')}{4},d)}] > \left(0.05 \times 0.88 + 0.34 \times 0.11 + 0.74 \times (1 - 0.88 - 0.11)\right) \cdot |s^u_d| \]
\[ > 0.08 \cdot |s^u_d| \]

\[ E[\tilde{S}_{(t_i,d)}] > \left(0.08 \times 0.88 + 0.24 \times 0.11 + 0.796 \times (1 - 0.88 - 0.11)\right) \cdot |s^u_d| \]
\[ > 0.125 \cdot |s^u_d| \]

\[
\text{Lemma 7. (Timestamp Lemma)} \text{ For any node } u, \text{ any level } d \text{ such that } T^u_d > 0 \text{ at time } t, \text{ the expected number of nodes } x \in (s^u_d \cap G_u(T^u_d, t)) \text{ is at least } 0.72 \cdot |s^u_d|. \]

\textbf{Proof.} Starting from time } T^u_d, \text{ if every swap operation performed by an intra-group trans-formation involves exactly two nodes (i.e. replacing exactly one node with another), then the lemma is identical to Lemma 6, and holds thereby.

Let us consider the case when more than two nodes get involved in a swap operation. Suppose due to the intra-group transformation as a result of communication } (u, v) \text{ at time } t_i \text{ (} t_i > T^u_d\text{), a group } g \text{ at a level above } d \text{ is swapped by another group } g_v, \text{ such that } g_v \text{ contains one of the communicating nodes } v. \text{ Suppose, after the transformation at time } t_i, \text{ the distance between any communicating node and any of the nodes in group } g \text{ is } d'. \text{ According to } \text{DyHypes, this can only happen if } T^v_{\mathcal{V}} > T^x_{d'+1}, \text{ at time } t_i, \text{ where } x \text{ is a node in } g \text{ and } \mathcal{V} \text{ is the depth of group } g_v.\]
Starting from time $T^u_V$, if every swap operation performed by an intra-group transformation in group $g_v$ involves exactly two nodes, then Lemma 6 applies for group $g_v$.

Note that, due the communication $(u, v)$, an edge between nodes $u$ and $v$ is established at time $t_i$ in the communication graph. This implies, the expected number of nodes $x \in (s^u_d \cap G(u(T^u_d, t)))$ moving in subtree $s^u_{d'+1}$ is at least $0.72 \cdot |g_v|$. Now, since group $g$ is chosen uniformly at random in subtree $\sim s^u_{d+1}$, clearly the expected number of nodes $x \in (s^u_d \cap G_u(T^u_d, t))$ is at least $0.72 \cdot |s^u_{d+1}|$ at time $t_i$, as the expected number of such nodes in group $|g_v| - |g_v|$ at time $t_{i-1}$.

On the other hand, if some swap operations in group $g_v$ since time $T^u_V$ involves more than two nodes, we show that the expected number of nodes $x \in (s^u_d \cap G_u(T^u_d, t))$ moving in subtree $s^u_{d'+1}$ is still at least $0.72 \cdot |g_v|$. We argue using the that this is an inductive instance of the lemma, where the base case is a pair of nodes in subtree $s^u_{d+1}$ communicated for the first time in the time interval $[T^u_V, t_{i-1})$. Clearly Lemma 6 is applicable for the base case and hence the lemma follows.

\[ \square \]

**Lemma 8. (Intra-Group Distance Lemma)** Let a node pair $(u, v)$ communicated at time $t'$ for the last time. Let $\alpha$ be the highest level such that $s^u_\alpha$ is a common subtree for both nodes $u$ and $v$. If nodes $u$ and $v$ belong to the same group at level $\alpha$ at time $t$, the expected distance between nodes $u$ and $v$ at time $t$ is at most $\log T(u, v) + 1$.

Proof. Let $g_\alpha$ be the node $u$’s group at level $\alpha$ at time $t$. Let $t_i$ denote the time at which the $i^{th}$ communication of any nodes $x \in g_\alpha$ took place after time $t'$. Let there has been $k$ such communications between time $t'$ and time $t$. Obviously, $t' < t_1 < t_2 < \cdots < t_k < t$. 

Let \( \alpha_i \) refer to the highest level such that \( s^u_{\alpha_i} \) is a common subtree for both nodes \( u \) and \( v \) at time \( t_i \) (i.e., \( d_{N_{t_i}}(u, v) = H - \alpha_i \)). Let \( p_i \) be the probability that the distance between nodes \( u \) and \( v \) increases (by one) at the transformation that takes place at time \( t_i \). Note that, the distance between nodes \( u \) and \( v \) can increase only if node \( v \) happens to be a member of the set of nodes in \( \sim s^u_{\alpha_i+1} \), chosen randomly by intra-group transformation at time \( t_i \). Let \( g^{rand}_{\alpha_i} \subseteq \sim s^u_{\alpha_i+1} \) be the set of nodes randomly chosen at time \( t_i \). Clearly,

\[
p_i \leq \frac{|g^{rand}_{\alpha_i}|}{|\sim s^u_{\alpha_i+1}|} \tag{4.4}
\]

Since the random selections of subtree are independent of each other, we model the distance between nodes \( u \) and \( v \) at time \( t \) using Poisson Binomial Distribution [61]. Let \( D_i = d_{N_{t_i}}(u, v) = H - \alpha_i \). We get:

\[
E[D_i] = \sum_{i=1}^{i} p_t
\]

Let \( X_i \) be the set of nodes that have a path from node \( u \) in the communication graph drawn for communications during the time interval \([t', t_i]\). Using Lemma 7, we get:

\[
E[X_k] > \sum_{j=1}^{k} 0.72 \cdot p_i \cdot \sim s^u_{E[\alpha_i+1]}
\]

\[
= \sum_{j=1}^{k} 0.72 \cdot p_i \cdot \sim s^u_{\sum_{i=1}^{1} p_i} \quad (\text{Using Equality 4.5})
\]

\[
= 0.72 \cdot \left| s^u_{\sum_{i=1}^{k} p_i} \right| = 0.72 \cdot \left| s^u_{E[\alpha_i]} \right|
\]

Thus, the lemma follows.
Theorem 6. (Routing Theorem) For any communication sequence \( \sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m) \), the expected routing cost for DYHYPES is at most \( 2(WS(\sigma) + m) \).

According to Invariant I, for any network of size \( n \), there can be at most \( 2^i (\log n - i) \)-relative groups. Since relative groups are chosen randomly, the adversary needs to try communications in different groups to find a communication between a relative groups. Since we charge any communication twice more than its corresponding working set number, the extra charge is used to pay for communications between relative group pairs.

Also, according to the construction of the algorithm, the transformation cost is exactly same as the communication cost.

Theorem 7. (Hypercube Optimality Theorem) For an unknown communication sequence \( \sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m) \), if the routing cost of the optimal self-adjusting algorithm conforming to our model in skip graph is \( C \) rounds, then the routing cost of the optimal self-adjusting algorithm for hypercube networks conforming to our model is at least \( \frac{C}{O(\log \log n)} \) rounds.

Proof. for a \( \log n \)-dimensional hypercube, from any node, number of nodes with distance \( k \) is \( \log n \) choose \( k \). Using the upper bound of binomial coefficients, we get:

\[
\binom{\log n}{k} \leq \left( \frac{e \cdot \log n}{k} \right)^k
\]

On the other hand, in skip graphs, number of nodes at distance \( k \) from any node is \( 2^{k-1} \). Clearly, \( \left( \frac{e \cdot \log n}{k} \right)^k < 2^{O(\log \log n) \cdot k} \). This means, the number of nodes a node can reach with a routing cost of at most \( k \) in a hypercubic network, will require at most \( O(\log \log n) \cdot k \) routing cost to be reached by any node in a skip graph of same size.
$WS(\sigma)$ is the lower bound of the optimal cost for skip graphs, the theorem follows.

\[\square\]

**Theorem 8. Optimality Theorem.** For any communication sequence $\sigma = (\sigma_1, \sigma_2, \cdots, \sigma_m)$, the expected total cost (i.e. routing and transformation cost) for DYHYPES is at most $O(\log \log n)$ factor more than that of the optimal algorithm.

\textit{Proof.} This directly follows from Theorems 6 and 7. \[\square\]

4.4 Conclusion

We present a self-adjusting algorithm for hypercubic networks that relies on randomization and grouping of frequently communicating nodes at different levels. The transformation cost of algorithm DYHYPES is less than that of algorithm DSG. Although skip graphs have a different network structure, it might be possible to use a randomization technique (similar to the one used in DYHYPES) to improve the transformation cost for skip graphs. Despite higher average distance between nodes, the structural flexibility of skip graphs may be utilized in designing a faster self-adjusting algorithm that uses a similar randomization technique used in this chapter. We leave this idea for future work.
CHAPTER 5
DYNAMIC HASH SPACE ADJUSTMENT IN NETWORKED CACHES

5.1 Introduction

Figure 5.1: Front-end web server requests for a data objects to the appropriate cache server. The cache server replies with object if the object is present in cache. Otherwise, the object is copied from the origin storage.

Problem Statement. Figure 5.1 shows a typical network of cache servers. Front-end web servers forward requests to cache servers by mapping the requests to a circular hash space using a random hash function. When a request maps to a specific position in the circular hash space, the nearest clockwise server becomes responsible for responding to that request. If the requested object is available at the cache server, it sends the object to the front-end web server. Otherwise, the cache server responds with a cache miss, following
which, the web server fetches the object from backend storage and fills the cache. Our goal is to design a protocol that minimizes load imbalance among cache servers without requiring centralized coordination. To this end, cache servers can replicate objects in other cache servers, redistribute the hash space amongst themselves, and add or remove cache servers.

**Proposed Approach.** In this chapter, we propose Meezan\(^1\), a novel distributed protocol for optimizing load balancing in large-scale networked key-value caches. Meezan optimizes load balancing by addressing the skewed and dynamic workload using fully distributed techniques. Meezan addresses the skewness of workload by ensuring that the load associated with a distinct key is sufficiently small. To this end, Meezan replicates popular objects at multiple cache servers. More specifically, Meezan identifies popular objects and then adds a salt\(^2\) to their object identifiers before computing the hash function. This ensures that the load associated with popular objects is divided among multiple cache servers.

Meezan addresses the dynamic nature of workload by enabling cache servers to efficiently adjust their hash space boundaries. To this end, Meezan forms random localities to divide cache servers into smaller groups. Each locality contains adjacent servers; which means each locality is a random chain cut from the ring. After localities are formed, each server shares its load with other servers in its locality. Once a server knows the load of all other servers in its locality, it conducts one of the following actions: (1) removes itself from the system and appropriately divides the hash space between the continuing surviv-

\(^1\)Meezan means *balance* in Arabic language.

\(^2\)A small suffix to create a distinct identity.
ing neighbors; (2) adds one or more new server(s) as neighbor(s) and migrate appropriate hash space to the continuing surviving or new neighbors; (3) migrates appropriate hash space to the continuing surviving or new neighbors. The procedure of locality formation, server addition/removal, and hash space migration is repeated after a fixed time interval. Note that locality formation and locality computation require only current local knowledge. The distributed cooperation of randomly chosen and independent localities to implement system-wide load balance is the cornerstone of our proposed approach.

Results. Our theoretical analysis shows that Meezan’s replication policy mitigates the skewness in key distribution by limiting the expected number of requests mapped to a distinct key. We also derive an upper bound for load imbalance and show that Meezan achieves near perfect load balancing for a reasonable choice of parameters. We further conduct trace-driven simulations using content access logs from five geographically distributed data centers of a commercial Content Delivery Network (CDN). We compare Meezan with three prior approaches (CH [38], SPORE [29], APAC [35]) in terms of several performance metrics. In addition to load balancing, the performance metrics include number of servers, number of server additions/removals, data migration, and replication overhead. Our results demonstrate that Meezan significantly outperforms prior approaches. As compared to the prior approaches, Meezan reduces load imbalance by up to 52%, average number of servers up to 28%, number of server additions/removals up to 91%, data migration up to 81%, and replication overhead up to 40%.
5.2 Preliminaries

5.2.1 Design Goal

We consider a key-value networked cache architecture, illustrated in Figure 5.1, where web servers forward user requests to cache servers by using a hash function. If the cache server does not already have the requested object, it notifies the web server about cache miss. The web server then fetches the object from a backend storage to respond to user request, and the object may also be stored in the cache server depending on the cache replacement policy. Our goal is to design a distributed protocol for optimally balancing load among cache servers. Since cache servers can be added or removed (i.e., elastic resource allocation) in response to varying load, in addition to load balancing, we also aim to maximize utilization of cache servers. In sum, we desire all servers to be as evenly loaded as possible, and at the same time as much loaded as possible without exceeding the capacity of any server.

5.2.2 Background & Motivation

5.2.2.1 Consistent Hashing

Consistent hashing [38] is widely used for load balancing in many distributed systems. As illustrated in Figure 5.2, consistent hashing uses a circular hash space, where the position of a server in the hash space is determined by computing the hash of the server’s unique identifier. Note that a server can have multiple presence (i.e., virtual nodes) in the hash space. The positions of virtual nodes in the hash space are determined by adding incremental numbers to the server’s unique identifier before hash computation. When a re-
quest arrives, the hash value of the requested object’s identifier is computed and the server with closest clockwise virtual node becomes responsible for serving the request.

Consistent hashing can lead to severe load imbalance for skewed and dynamic workloads because it solely relies on randomization for resource allocation and is non-adaptive to change in access patterns. To illustrate this, we evaluate load balancing provided by consistent hashing using real-world content access logs from a commercial CDN. Our analysis suggests that some objects become very popular and cause load imbalance among cache servers. Figure 5.4(a) shows the skewed popularity distribution of objects in the CDN workload. The straight line on the log-log popularity distribution indicates that some objects are extremely popular whereas others are much less popular. Figure 5.4(b) shows the number of total requests and the number of distinct requests served by each cache server. We note that some servers are almost $5 \times$ more loaded than the average load ($\text{max/avg} = 4.9$).
5.2.2.2 Data Replication

The use of data replication in consistent hashing can improve load balancing by replicating popular objects across multiple cache servers. To replicate an object, a salt is concatenated with the object’s identifier before passing to the hash function. For example, if an object’s frequency exceeds a predefined replication threshold $r$, we replicate the object by adding a random salt picked from the range $[1, \lceil f/r \rceil]$, where $f$ is the request frequency of the object. Figure 5.3 shows an example where an object is replicated in two additional servers (C and E) by using salts.

![Diagram of data replication](image)

Figure 5.3: An example of data replication. Incremental numbers are added to the object’s identifier to distribute the load associated with popular objects across multiple servers.

We evaluate load balancing provided by consistent hashing and data replication [29] using CDN access logs. Figures 5.4(c) and (d) show the number of total and distinct requests served by each server for $r = 200$ and $r = 25$, respectively. Comparing Figures 5.4(b)
and (c), we observe that data replication improves load balancing in terms of $max/avg$ from 4.9 to 3.9. Comparing Figures 5.4(c) and (d), we observe that reducing the replication threshold from 200 to 25 improves load balancing in terms of $max/avg$ from 3.9 to 3.3.

To analyze the impact of data replication on load balancing, we compare key popularity distributions in Figure 5.4(a). We observe that data replication reduces the skewness for keys with request frequency greater than the replication threshold. Data replication improves load balancing because the skewness of key popularity distribution is reduced. However, load imbalance persists because the skewness is not completely eliminated. In Figure 5.4(e), we show that more aggressive data replication does not result in considerably reducing load imbalance. Further, in Figure 5.4(f), we show that reduction of replication threshold results in increased storage overhead. Thus, as also reported in prior work [31], we conclude that data replication alone is not sufficient for load balancing.

5.2.2.3 Hash Space Adjustment

Transferring a portion of hash space from a highly loaded server to a server with relatively lower load can reduce load imbalance. Figure 5.5 shows an example of hash space adjustment, where parts of hash space from servers E and C are transferred to server A. We implement and evaluate load balancing provided by consistent hashing and hash space adjustment [35]. We adjust the hash space by shifting the hash space boundaries between top five most load disparity neighboring server pairs. Figure 5.4(g) shows that hash space adjustment performs worse ($max/avg = 4.7$) than data replication in terms of load balancing. The reason is that some objects are extremely popular and any server
Figure 5.4: Performance of different consistent hashing based approaches for an one hour long real-world access log.
caching them becomes much more loaded than other servers. In fact, the load associated with a viral object may even exceed a server’s capacity. Hence, we conclude that hash space adjustment alone is not sufficient for load balancing.

We observe a substantial improvement in load balancing when we combine data replication and hash space adjustment. Figures 5.4(g), (h), and (i) show that hash space adjustment with data replication significantly improves load balancing over the approaches that use only data replication or only hash space adjustment. We observe that hash space adjustment with data replication for $r = 25$ improves load balancing in terms of $max/avg$ to 1.4. Figure 5.4(e) shows that hash space adjustment with data replication leads to substantially better load balancing over data replication alone for a wide range of replication thresholds. As shown in Figure 5.4(f), hash space adjustment with data replication incurs at most 0.2% more storage overhead as compared to data replication without hash space
adjustment for any \( r \geq 4 \). Comparing Figures 5.4(e) and (f), we conclude that hash space adjustment reduces storage overhead by allowing us to choose a relatively higher replication threshold, yet achieving a significant improvement in load balancing.

![Diagram showing server addition/removal](image)

(a) A ring based hash space for 5 servers before addition/removal
(b) Addition of 2 servers (E, F) and removal of server C in the hash space shown in (a)

Figure 5.6: An example of server addition/removal.

### 5.2.2.4 Dynamic server addition/removal

Most web applications deploy elastic cache systems [58] to address large variations in workload. While under-provisioning causes performance bottlenecks, over-provisioning results in waste of resources. A practical approach to address this issue is to dynamically add or remove servers on an on-demand basis. When the overall system load approaches the overall capacity, we need to add more servers to ensure that the load on any server does not exceed its capacity. Similarly, to optimize resource utilization, we need to remove servers when the overall system load decreases sufficiently. Figure 5.6 shows an example
where two servers are added and one server is removed from the hash space.

5.3 Proposed Approach

5.3.1 Architectural Overview

Cache servers are placed in a ring overlay in the order of their hash space placement. The position of a server in the hash space is determined by computing the hash of the server’s unique identifier. Each server has a clockwise neighbor and an anticlockwise neighbor in the ring overlay which hold clockwise and anticlockwise adjacent portions in hash space, respectively. Web servers use consistent hashing to send a request to the appropriate cache server. Specifically, a request’s identifier is hashed to compute its key and the request is sent to the cache server responsible for that portion of the hash space. The cache server that receives a request is assigned as the home server\(^3\) for the request. Upon receiving a request, the home server performs one of the following actions. (1) It responds with a cache miss if the requested object is not present in the cache; (2) it responds with the requested object if the requested object is present in the cache; or (3) it forwards the request to another cache server that contains a replica of the requested object. When a request is forwarded to another server by the home server, the recipient server can either respond with the requested object or a cache miss. In the latter case, the home server sends the requested object to the client (from its cache or backend storage) and also replicates the requested object at the server that reported the cache miss.

\(^3\)The term home server is borrowed from [29].
5.3.2 Solution Overview

Our solution for load balancing relies on two techniques: (1) identification and replication of popular objects, and (2) adjustment of hash space boundaries. Home servers are responsible for identifying and replicating popular objects which are mapped to them through consistent hashing. Hash space adjustment takes place at the beginning of each time interval. In hash space adjustment, servers form random localities and broadcast their load counts to other servers in their localities. Based on this information, servers adjust their hash space boundaries within the locality to minimize load imbalance. One or multiple servers may be added or removed from the locality depending upon the overall load of a locality. Servers transfer keys to appropriate servers so that web servers can continue using consistent hashing.

(a) Cache servers; their positions in the ring based hash space; and their loads before adjustment

(b) Random localities are formed.

(c) servers are repositioned in the hash space within their locality; one server is added and one server is removed.

Figure 5.7: Illustration of hash space adjustment.
5.3.3 Popular Object Replication

We propose a replication policy to limit the maximum number of requests within a time interval associated with a key. To identify popular objects, home servers maintain an exponentially weighted moving average (EWMA) of request counts for each object that is mapped to them by consistent hashing. Since the request for an object always goes to the same home server, only one cache server is responsible for maintaining EWMA for the object.

Upon a request for an object \(a\), a popularity weight \(w = \frac{\max(C, M)}{r}\) is computed, where \(C\) is the request count of an object so far in the current time interval, \(M\) is the EWMA of request count for the object, and \(r\) is the replication threshold. If \(w < 1\), the original request is hashed without adding any salt. Otherwise, if \(w \geq 1\), the salt is chosen as follows:

\[
salt = \begin{cases} 
\text{random integer from } [1, \lceil w \rceil] & \text{if } C \leq M \\
\lceil w \rceil + 1 & \text{if } C > M
\end{cases}
\]  

(5.1)

We use random salt to ensure that roughly the same number of requests are mapped to each distinct key. However, the expected number of requests mapped to a distinct key is greater than \(r\) when \(C > M\). Thus, to limit the severity of skewness in key distribution, we use a deterministic salt when \(C > M\) to ensure that the expected number of requests mapped to a distinct key in a time interval is at most \(r\).

**Theorem 9. (Replication Theorem)** Considering the hash function to be perfect (i.e., no collisions), for a given time interval, the expected number of requests mapped to a distinct
key is at most r.

Proof. From Equation 5.1, salt = $\left\lceil \frac{C}{r} \right\rceil$ when $C > M$. Thus, a unique salt is used for each $r$ requests of object $a$ after $C$ exceeds $M$. Now, consider the case when $C \leq M$. According to Equation 5.1, salt is an integer that is chosen randomly from $[1, \left\lceil \frac{M}{r} \right\rceil]$. Let $j$ be an integer s.t. $1 \leq j \leq \left\lceil \frac{M}{r} \right\rceil$, and $X_i$ be an indicator random variable s.t. for the given time interval:

$$X_i = \begin{cases} 
0 & \text{if salt} \neq j \text{ for } i^{th} \text{ request of } a \\
1 & \text{if salt} = j \text{ for } i^{th} \text{ request of } a 
\end{cases}$$

Let $c$ denotes the number of times a random salt was used for object $a$ in the previous time interval. Obviously $c \leq M$. Let object $a$ map to key $k$ when $salt = j$, and the number of requests that map to key $k$ is $X$. Clearly, $X = \sum_{i=1}^{c} X_i$. Using linearity of expectation, the expectation of $X$:

$$E[X] = \sum_{i=1}^{f} E[X_i] = \sum_{i=1}^{c} Pr[X_i = 1] = c \left\lceil \frac{r}{M} \right\rceil \leq r.$$

\[\square\]

5.3.4 Hash Space Adjustment

We propose a novel distributed implementation of hash space adjustment policy where servers rely on local knowledge to adjust their position in the hash space. Each cache server maintains request counts for all distinct keys requested in a time interval. Each server also maintains a load counter to keep track of the total number of requests received in the current time interval. As we discuss next, hash space adjustment is performed in three steps: (1) form random localities and share load counters; (2) add or remove servers, if necessary; and (3) adjust hash space boundaries. Figure 5.7 illustrates these three steps.
5.3.4.1 Formation of Random Localities

To divide the hash space into random localities, each server is declared a locality separator with probability $1/p$, where $p$ is the locality threshold, and $p > 1$. Servers become part of the locality of their clockwise nearest locality separator. We refer the number of servers in a locality as the size of the locality. Note that the expected size of a locality is $p$. After formation of random localities, all servers broadcast their load counters (one count per server) to other servers within their localities.

5.3.4.2 Addition or Removal of Servers

Each server computes the total load of the locality by adding the load counters it receives from other servers in the locality. The server then divides the total load by the load threshold (also called server capacity) to compute the adjusted number of servers for the locality. Let $n_0$ and $n$ be the number of servers in a locality before and after adjustment, respectively. If $n_0 = n$, there is no need for addition or removal of servers. Otherwise, we add (or remove) $|n - n_0|$ servers if $n > n_0$ (or $n < n_0$), respectively. We remove the least loaded servers and add servers neighboring to the most loaded servers.

5.3.4.3 Server Positioning and Key Transfer

After addition or removal of servers in each locality, we determine the adjusted server positions in the hash space as follows. Each server computes the average load ($A$) by dividing the total load by the adjusted number of servers. Consider the locality of server $s$ as a linked list consisting of the servers of the locality, where servers are placed in the linked list from left to right in their clockwise order in hash space after server addi-
tion/removal. Let $Load_{left}$ be the sum of load counts of all the servers positioned left to $s$ in the linked list. Each server maintains a list of keys in its cache sorted \(^4\) in clockwise order. Let $k_1, k_2, \ldots, k_m$ be the sorted list of keys maintained by server $s$. Let $\text{count}(k)$ be request count of key $k$ in the previous time interval. Then we define the rank of a key $k_i$ as $\text{rank}(k_i) = Load_{left} + \sum_{j=1}^{i} \text{count}(k_j)$. Now server $s$ transfers any key $k_i$ to the $y$-th server (from left in the linked list) if $(y - 1)A < \text{rank}(k_i) \leq yA$. To do so, servers need to check the first $m'$ keys such that $Load_{left} + \sum_{j=1}^{m'} \text{count}(k_j) \leq (y - 1)A$ and the last $m''$ keys such that $Load_{left} + A - \sum_{j=m''}^{m} \text{count}(k_j) > yA$. Note that the right most server in the linked list does not change its position in the hash space. All other servers adjust their positions in the hash space as a result of key transfer.

**Definitions.** We define *adjusted load* of a server as the number of requests that map to the adjusted hash space of the server during the previous time interval. The *adjusted max/avg* is defined as the ratio of the maximum adjusted load to the average adjusted load of servers. Finally, we define *average load* as the average load of servers within a locality after server addition/removal.

**Theorem 10. (Local Load Balance Theorem)** Let $H$ be the hash space covered by the servers of a locality. If the average load for the locality is $A$ and the maximum number of requests mapped in the previous time interval to a distinct key $k \in H$ is $R$, the adjusted max/avg for the locality is at most $1 + \frac{R-1}{A}$.

**Proof.** Let $k_1, k_2, \ldots, k_m$ be the keys, sorted in the clockwise order based on their positions in the hash space $H$, requested to all servers of a locality in the previous time interval. We

\(^4\)Meezan does not sort keys, it simply maintains a sorted list.
rank each key $k_i$ as:

$$\text{rank}(k_i) = \sum_{j=1}^{i} \text{count}(k_j)$$  \hspace{1cm} (5.2)
Theorem 11. *(Global Load Balance Theorem)* The adjusted \( \frac{\text{max}}{\text{avg}} \) for the entire system is upper bounded by \( \frac{n}{n-1} \), where \( n \) is the size of the smallest locality after server addition/removal.

**Proof.** Using the argument we used to prove Theorem 10, the maximum adjusted load to any server is \( l + R - 1 \), where \( R \) is the maximum number of requests mapped to a distinct key. Using Lemma 9, we get:

\[
\frac{\text{max}}{\text{avg}} \leq \frac{l + R - 1}{l(n-1)} < \frac{n(l + R - 1)}{l(n-1)} = \frac{n}{n-1} \left( 1 + \frac{R - 1}{l} \right)
\]

\[
\approx \frac{n}{n-1} [R << l, \text{when } r \text{ small enough}]
\]

Thus, \( \frac{\text{max}}{\text{avg}} \to 1 \) when \( n \) is sufficiently large. \( \square \)

5.3.5 **Implementation Details**

**Key transfer.** Servers transfer keys in the sorted order so that the recipient servers do not need to sort them again. Since all servers use the same deterministic protocol to compute adjusted load and positions of new servers, servers can transfer keys in parallel without ambiguity. While transferring keys, servers also transfer the corresponding objects and their EWMA values (if available). Key transfer should include objects because the recipient servers would otherwise need to fetch objects from the backend storage. Since all servers in a locality are in geographical proximity (perhaps the same rack in a data center), object transfer to other servers in a locality is more efficient than fetching from backend storage.
Further note that multiple keys, which are to be transferred, may map to the same object due to replication. To avoid redundant data transfer, servers transfer such objects only once.

**Distributed implementation.** Our proposed load balancing approach is distributed and does not require any central coordination or global knowledge. Servers form localities only by talking to their neighbors in the ring overlay. Locality separators forward their load counters to their anti-clockwise neighbors, while other servers forward their load counters to both the clockwise and anti-clockwise neighbors. Any server that is not a locality separator also forward load counters received from the anti-clockwise neighbor to the clockwise neighbor. It is noteworthy that we do not require any inter-locality communication since key transfers take place only within localities.

**Bookkeeping overhead.** We require servers to maintain a counter for all distinct keys that are requested in the current time interval. Servers delete these counters after hash space adjustment. Although the number of distinct keys in a time interval is manageable, we can further reduce the bookkeeping overhead by using sampling. Since there are typically only a few popular objects, a reasonable sampling rate will ensure that we maintain counters for most popular objects. We can also use sampling to reduce the bookkeeping overhead of EWMA counters that are maintained by home servers.

**Heterogeneous servers.** Note that servers may have different computational and storage abilities. We can generalize our approach by assigning different load thresholds to servers and use weighted averages to compute their adjusted load. Other modifications in our approach for adaptation to heterogenous servers are trivial.

**Fixed number of servers.** Our approach can also be used when the number of servers
is fixed. To this end, we can bypass the server addition/removal step during hash space adjustment. While we can still guarantee load balancing in this case, we cannot provide any guarantees for resource utilization.

### 5.4 Experimental Results

#### 5.4.1 Data

For empirical evaluation, we use content access logs from five different geographically distributed data centers (Los Angeles, Singapore, Sydney, Amsterdam, and Atlanta) of a commercial CDN. Content access logs contain detailed information for each request, including timestamp, URL containing object’s name, size of the object, and client information (e.g., hashed IP address). Overall, content access logs contain more than 212 million requests for 5.6 million distinct objects over the duration of one week. We use these content access logs to conduct trace-driven simulations for different load balancing approaches.

#### 5.4.2 Performance Metrics

We use the following performance metrics.

1. \( \text{max/avg} \): The \( \text{max/avg} \) metric quantifies the load imbalance of a networked cache system. It is defined as the ratio of the number of requests received by the most loaded server (\( \text{max} \)) to the average load of all servers (\( \text{avg} \)). The minimum value of \( \text{max/avg} \) is 1, which indicates perfect load balancing. The values of \( \text{max/avg} \) larger than 1 represent increasing load imbalance.

2. **Number of servers**: We measure the average number of servers to quantify the resource consumption of a networked cache system. If \( N_i \) is the number of servers used in the time
interval $i$, the average number of servers for $T$ time intervals is $\frac{\sum_{i=1}^{T} N_i}{T}$.

4. **Data Migration:** We quantify data migration as the amount of data transferred among servers due to server addition/removal, hash space adjustment and data replication. If $D_i$ is the amount of data transferred in the time interval $i$, the average data migration for $T$ time intervals is $\frac{\sum_{i=1}^{T} D_i}{T}$.

5. **Replication Overhead:** Replication overhead quantifies the number of replicas (copies) of objects used to serve requests. Let $X$ be the number of distinct objects requested in a time interval, and $Y$ be the number of total replicas used to serve the $X$ requested objects, replication overhead is defined as $\frac{Y-X}{X}$.

5.4.3 Baselines for Comparison

We compare our approach with the following baseline approaches for a range of operating parameters. We briefly summarize the baselines as follows.

1. **Consistent Hashing (CH).** Our first baseline is a dynamic version of consistent hashing [38] that adds or removes servers on demand in every time interval. To dynamically add or remove servers, we use two thresholds referred to as *load threshold* and *low threshold*. A server is removed if its load falls below the low threshold and a server is added if the load of an existing server exceeds the load threshold. The servers are mapped to the hash space by hashing server identifiers by using a random hash function.

2. **Self-adaptive POpularity based REplication (SPORE):** SPORE aims to identify and replicate popular objects for minimizing load imbalance [29]. We use a dynamic version of SPORE as our baseline. To dynamically add or remove servers, we again use the *load*
threshold and low threshold.

3. **Adaptive Performance-Aware Caching (APAC):** APAC performs hash space adjustment and dynamic server addition/removal to reduce load imbalance [35]. This approach requires all servers to have multiple replicas (positions) in the hash space, where each replica is called a *virtual node*. As we do for other baselines, we remove a server when its load falls below the *low threshold*. However, unlike other baselines, APAC always adds new servers to support the overloaded servers. Hash space adjustment is performed in each time interval by finding the most load disparity pair of neighboring servers and adjusting the hash space boundary between them. We used the data replication policy from SPORE and the hash space adjustment technique from APAC in this baseline.

5.4.4 Experimental Setup

We conduct trace-driven simulations to compare and contrast our approach and the aforementioned baselines. We use Least Recently Used (LRU) as the underlying cache eviction policy in our experiments. Our simulations for CH and SPORE start with 25 servers each with 10 replicas (or *virtual nodes*) in the hash space. We initialize our experiments for APAC with 25 servers each with 100 (the system default in [35]) virtual nodes. APAC requires two more parameters, \( \alpha \) and \( \beta \), to control adjustments in the hash space boundaries. We set \( \alpha = 0 \) to ensure that the protocol intends to only optimize load balancing and does not consider hit rate as a metric of cost. After pilot analysis, we set \( \beta = 0.95 \), which means the load balancer works aggressively by moving up to 95% of the hash space during hash space adjustment. Same as all the baselines, we initialize simulation
for Meezan with 25 servers. We do not use virtual nodes for our approach as we require every server to have a single position in the hash space. We use 1 minute time intervals for all our simulations. To evaluate Meezan and baselines, we set the replication threshold $r = 25$ and we set the locality threshold $p = 15$ for Meezan.

![Figure 5.9: Average $max/avg$.](image)

5.4.5 Performance Comparison

Figures 5.9, 5.10, 5.11, and 5.12 show the performance of baselines for different combinations of load and low thresholds, and the performance of Meezan for different load thresholds.

**Load balancing.** As shown in Figure 5.9, Meezan outperforms all baselines in terms of load balancing ($max/avg$). Meezan outperforms CH and SPORE because new servers are placed at the random positions in the hash space by CH and SPORE. Thus, overloaded servers are not guaranteed to be supported by the new servers by these approaches. As a consequence, overloaded servers may continue with their high load for an extended period of time. Meezan outperforms APAC due to the following reasons. First, Meezan conducts
more aggressive hash space adjustment as compared to APAC. APAC adjusts the hash space boundaries only between the neighboring virtual nodes at the most load disparity server pairs. In contrast, Meezan allows all servers to adjust their hash space boundaries simultaneously. Second, APAC transfers exactly half of the hash space from an overloaded server to a new server. This approach does not guarantee optimal hash space adjustment. In
contrast, Meezan identifies optimal positions and adjusts hash space for all servers when adding/removing servers.

**Average number of servers.** Figure 5.10 shows that Meezan uses fewer servers on average as compared to baselines. The reason is that Meezan aggressively adjusts hash space boundaries and adds/removes servers on demand to ensure that the average load is as close as possible to the load threshold. Furthermore, we note that, for a wide range of threshold combinations, Meezan reduces average number of server additions/removals by 91% as compared to baselines.

**Data migration.** Figure 5.11 reports less average data migration for Meezan as compared to the baselines. The reason is that Meezan adds/removes fewer servers as compared to baselines. We note that data migrated due to server addition/removal is much more than that due to hash space adjustment. Thus, aggressive hash space adjustment conducted by Meezan has a little impact on data migration.

**Replication overhead.** Figure 5.12 shows that Meezan incurs 40% less replication overhead compared to SPORE and APAC. The reason is Meezan minimizes the average number of servers 5.10. Since a server can have at most one replica of an object, Meezan requires less replicas to serve popular objects compared to SPORE and APAC. Moreover, SPORE and APAC add/remove more servers to respond to load dynamics as compared to Meezan, resulting in creation of more replicas for popular objects.
5.4.6 Discussions

**Bookkeeping Overhead.** Figure 5.13 shows average number of counters maintained and transferred by a server in a time interval (1 minute time interval is used). We observe from Figure 5.13 that the average number of total counters maintained by a server is roughly equal to the load threshold, and the percentage of counters that need to be migrated due to load balancing decreases as we increase the load threshold.

**Parameter Selection.** To understand parameter selection for Meezan, we conduct a series of experiments for varying locality thresholds $p$, load thresholds $\{400, 1600, 4000, 8000\}$, and replication thresholds $\{5, 15, 25, 100\}$. Our experiments show that load balancing improves as we decrease replication threshold for a fixed load and locality threshold. For a fixed replication and load threshold, increasing the locality threshold has no impact on load balancing given that we chose a reasonable value (e.g., at least 5) for locality threshold. Replication and locality thresholds have a little impact in load balancing when we choose a sufficiently large load threshold. For large load thresholds, other parameters have almost no impact on load balancing and we achieve near perfect load balancing. We conclude that Meezan achieves near perfect load balancing when the load threshold is sufficiently larger than the replication threshold, which corroborates our analysis presented in Theorems 10 and 11.

5.4.7 Summary

Meezan outperforms all baselines in terms of different performance metrics for a wide range of parameters. We find that Meezan reduces load imbalance by 52%, average
number of servers by 12%, and average data migration by 43%, on average, as compared to CH. We find that Meezan reduces load imbalance by 38%, average number of servers by 28%, average data migration by 81%, and replication overhead by 40%, on average, as compared to SPORE. We find that Meezan reduces load imbalance by 0.47%, average number of servers by 18%, average data migration by 81%, and replication overhead by 39%, on average, as compared to APAC.

5.5 Conclusion

Load imbalance in large-scale networked cache systems can substantially degrade overall system performance. Achieving optimal load balancing is challenging due to the dynamic and skewed nature of real-world workloads. To this end, we presented a distributed load balancing protocol (Meezan) for key-value networked caches. Meezan addresses workload skewness by ensuring that the load associated with popular objects is...
divided among multiple cache servers. Meezan addresses the dynamic nature of workload by enabling cache servers to efficiently adjust their hash space boundaries. Our theoretical analysis shows that Meezan achieves nearly perfect load balancing for a wide range of operating parameters. Our empirical results show that, as compared to baselines, Meezan reduces load imbalance by up to 52%, average number of servers up to 28%, number of server additions/removals up to 91%, data migration up to 81% and replication overhead up to 40%.
6.1 Distributed Online Balanced Repartitioning

Balanced RePartitioning (BRP) problem was coined by Avin et al. [21]. Given a pairwise communication sequence, the objective of the problem is to dynamically partition \( n \) nodes into \( l \) clusters, each of size \( k \). The problem assumes that a communication between two nodes within the same cluster costs 0, a communication between two nodes from different clusters costs 1, and migration of a node from one cluster to another costs \( \alpha \). The goal is to minimize the total cost of serving all communication requests by migrating nodes between clusters in an online manner.

BRP features interesting connections with many well-known graph and online problems. For example, BRP generalizes a novel online version of the maximum matching problem when \( k = 2 \), and the online paging problem when \( l = 2 \). Moreover, BRP commonly arises in the context of server virtualization in data centers.

The following two settings are considered for the BRP problem.

1. **Without augmentation**: The nodes fit perfectly into the clusters (i.e. \( n = k \cdot l \)).

2. **With augmentation**: The algorithm has access to additional space in each cluster. The algorithm is said to be \( \delta \)-augmented if the size of each cluster is \( k' = \delta \cdot k \), whereas the total number of nodes \( n = k \cdot l < k' \cdot l \). Note that, in the augmented online algorithm is compared to the optimal offline algorithm without augmentation in the competitive analysis.
The subproblem of finding a good trade-off between the communication and the migration cost is essentially a ski rental or rent-or-buy problem [41, 46]. Online page and server migration problems [14, 15] are also known for dealing with a similar trade-off. However, BRP is different from most online migration problems as both endpoints of a communication request are subject to optimization in BRP. Moreover, a large number of possible node-cluster configurations renders BRP challenging in an online matrical task system [17] scenario.

Avin et al. [21] presented a non-trivial $O(k \log k)$-competitive algorithm for the setting with 4-augmentation. However, their algorithm is not decentralized and requires global knowledge. Designing a distributed algorithm to solve BRP is an open problem.

**Open Problem:** Design a distributed algorithm to solve the BRP problem, where the algorithm has access to $O(\log n)$ memory on each node and no node is aware of the global distribution of nodes among clusters.

### 6.2 Topological Self-Adjustment in the Internet of Things (IoT)

RPL [62] is the IETF standard for multi-hop routing in Low Power Lossy Networks (LLN), and considered to be the *de facto* routing protocol [36] for the Internet of Things (IoT). RPL forms one or more Destination Oriented DAGs (DODAG), where sink nodes become the roots of the DODAGs. Data packets associated with any communication between two arbitrary nodes must route through the links of DODAGs. Such point-to-point routing functionality is inefficient for many applications due to the following reasons.

1. A DAG is built to optimize routing costs for communications destined to the root
node only. DAGs formed by standard RPL does not optimize routing for arbitrary point-to-point communications.

2. Nodes must identify themselves as potential destination nodes before sources reach them.

A new IETF standard P2P-RPL [47] has been proposed to address these shortcomings. P2P-RPL is an on demand point-to-point route discovery mechanism and an extension to the core RPL. In P2P-RPL, each source node initiates a temporary DAG rooted at itself to find the routes to destination nodes. However, forming a temporary DAG for each source node results in inefficiency in terms of energy consumption. Moreover, discovered routes by P2P-RPL might be vulnerable to node failures and mobility.

**Open Problem 4.** Design an online self-adjusting protocol (SA-RPL) that dynamically transforms the topology formed by RPL based on communication patterns, in order to maximize communication efficiency in terms of given metrics (e.g. energy consumption, number of routing hops etc.)

This problem is significantly different from the problems related to classical self-adjusting tree networks, due to the resource constraints imposed by LLNs. Unlike general tree networks, a node in LLN can directly connect only with the nodes that are placed within its communication range. Moreover, short-distance communications are preferable in LLNs as they are typically more energy efficient. Furthermore, unlike a general tree network, heterogeneity is an important factor to consider while designing a practical solution in the context of IoT. A typical LLN contains nodes with different communication ranges,
computational resources, and energy levels.

**A Possible Direction.** We propose to study the feasibility of a self-adjusting bidirectional routing tree based solution. Let each node store a membership vector which represents the nodes relative position in the current routing tree. All nodes are aware of membership vectors and communication costs associated with other nodes (not necessarily tree neighbors) in their transmission range. When the membership vector of the destination node is known, nodes perform routing based on the destination nodes membership vector. However, when the membership vector of the destination node is unknown by the source node, routing goes towards and possibly through the root, assuming that the root(sink) node knows the membership vector of the destination node or can directly reach the destination node. Upon routing, all nodes taking part in routing happen to know the membership vectors of both the source and destination nodes. Transformation follows as one or more of these nodes update their neighboring relationships to perform topological adaptation to the communication pattern.

Below are some possible technical challenges regarding this approach.

1. How to adjust transmission windows dynamically? Does SA-RPL require modification of the standard Trickle algorithm [43]?
2. How to adjust the signal strengths of nodes dynamically to optimize SA-RPL?
3. Mobility support is a fundamental issue with RPL. How to handle mobility in SA-RPL?
4. What happens when the membership vector of a node changes due to transformation
or mobility? How to decide when should some nodes have multiple parents/children?

Finally, we propose to run extensive simulations to evaluate SA-RPL and compare it with P2P-RPL and other state of the art protocols in this domain.
APPENDIX A

A.1 Standard Skip Graph Routing

The standard skip graph routing [7] works as follows. Routing starts at the top level from the source node and traverses through the skip graph structure. If the identifier of the destination node is greater than that of the source node, then at each level, routing moves to the next right node until the identifier of the next node is greater then the identifier of the destination node. When a node with an identifier greater than the destination node is found, the routing drops to the next lower level, continuing until the destination node is found. If the identifier of the destination nodes is smaller than that of the source node, routing takes place in the similar manner expect it moves to the next left node instead of right, and drops to the lower level when a node with smaller (instead of greater) identifier is found.

A.2 Updating Group-ids for Levels Below $\alpha$

DSG requires each node to store a number (between 0 and $H_t$), referred to as group-base. The group-base for a node is the highest level at which the node belongs to its biggest group. For example, in the skip graph $S_8$ in Figure 3.1(b), the group-bases for nodes H,F,B and G are 3,2,1 and 1, respectively. We use the notation $B_x$ to denote the group-base of any node $x$. Initially, before any communication, group-base for each node is set to the lowest level at which the node is singleton. Also, the notification message sent after routing includes the group-bases for both nodes $u$ and $v$, along with group-ids, timestamps, and membership vectors.
Each node \( x \in l_\alpha \) with \( G_\alpha^x = u \) initializes a vector \( G_{\text{lower}} \) as follows:

\[
G_{\text{lower}} = \begin{cases} 
[G_u^0, G_u^1, \ldots, G_u^{\alpha-1}], & \text{if } B_u \leq B_v \\
[G_v^0, G_v^1, \ldots, G_v^{\alpha-1}], & \text{otherwise}
\end{cases}
\]

Node \( u \) broadcasts a message \( \langle G_{\text{lower}}, \min(B_u, B_v), G_u^{\min(B_u, B_v)}, G_v^{\max(B_u, B_v)} \rangle \) to all nodes \( y \in l_{\max(B_u, B_v)} \) such that \( u, v \in l_{\max(B_u, B_v)} \). Each such node \( y \) with \( G_y^{\max(B_u, B_v)} = G_u^{\max(B_u, B_v)} \) or \( G_y^{\max(B_u, B_v)} = G_v^{\max(B_u, B_v)} \) updates their group-base by setting \( B_y = \min(B_u, B_v) \) and updates group-ids \( G_y^i = G_{\text{lower}}^i \) for \( i = 0, 1, \ldots, \alpha - 1 \).

Regardless of the outcome of the comparison \( \min(B_u, B_v) < \alpha \), each node \( x \in l_\alpha \) with \( G_\alpha^x = u \) sets group-ids \( G_i^x = G_{\text{lower}}^i \) for \( i = 0, 1, \ldots, \alpha - 1 \). Moreover, each node \( x \in l_\alpha \) updates its group-base \( B_x \) as follows:

- If \( x \)'s group at any level \( d \) (\( d \geq \alpha \)) splits into 2 subgroups due to transformation, and if \( B_x = d \), \( x \) sets \( B_x = B_x - 1 \).

- Let \( d \) be the lowest level at which \( x \)'s group splits due to transformation. if \( B_x = \alpha \) and \( d > \alpha + 1 \), \( x \) sets \( B_x = d - 1 \).

It is important to understand that if \( G_{\alpha-1}^u \neq G_{\alpha-1}^v \), then the working set number for the node pair \( (u, v) \) is greater than the routing distance for \( (u, v) \).

### A.2.1 Distributed Sum Using a Skip List

Each node holds a number and we want to compute the sum of the numbers held by all the nodes. Each node of the base level of the skip list forwards their number to the nearest neighbor that steps up to the upper level of the skip list. Any node receiving numbers from the neighbors from lower level computes the sum of the numbers and forwards
the sum to the nearest neighbor stepping up to the upper level. As this happens recursively at each level, the head node of the skip list computes the final sum in $O(\log n)$ rounds and then broadcasts the sum to all the nodes.

### A.3 Base Case Calculations for Lemma 6

For the time $t_4$ and $d = H_t - 2$,

$$E[\left| S_{(t_4, H_t-2)} \right|] = 2 + \left( E[\left| \tilde{S}_{(t_3, H_t-1)} \right|] + \left( E[\left| \tilde{S}_{(t_3, H_t-1)} \right|] \cdot \frac{1}{2} \right) - \left( E[\left| \tilde{S}_{(t_3, H_t-2)} \right|] \cdot \frac{1}{2} \right) \right)$$

Clearly, $E[\left| \tilde{S}_{(t_3, H_t-1)} \right|] = 1$, $E[\left| \tilde{S}_{(t_3, H_t-1)} \right|] = 1$, and $E[\left| \tilde{S}_{(t_3, H_t-2)} \right|] = 1$. Thus,

$$E[\left| S_{(t_4, H_t-2)} \right|] = 2 + \left( 1 + \left( 1 \times 1 \right) - \left( 1 \times \frac{1}{2} \right) \right) = 3.5$$

And,

$$E[\left| \tilde{S}_{(t_4, H_t-1)} \right|] = E[\left| S_{(t_4, H_t-2)} \right|] - 2 = 3.5 - 2 = 1.5$$

Thus,

$$E[\left| \tilde{S}_{(t_4, H_t-2)} \right|] = E[\left| \tilde{S}_{(t_3, H_t-2)} \right|] + \left( E[\left| \tilde{S}_{(t_3, H_t-2)} \right|] \cdot \frac{1}{2} \right) - \left( E[\left| \tilde{S}_{(t_3, H_t-1)} \right|] \cdot \frac{1}{2} \right)$$

$$\implies E[\left| \tilde{S}_{(t_4, H_t-2)} \right|] = 0 + \frac{1}{2} - 0 \times \frac{1}{4} = \frac{1}{2}$$

Thus, we get:

$$E[\left| S_{(t_5, H_t-2)} \right|] = 2 + 1.5 + \left( 1 \times \frac{1}{1} \right) - \left( 1.5 \times \frac{1}{2} \right) = 3.75$$
\[ E[|S_{(t_6,H_t-2)}|] = 2 + 1.75 + \left( 1 \times \frac{1}{1} \right) - \left( 1.75 \times \frac{1}{2} \right) = 3.88 \]

\[ E[|S_{(t_7,H_t-2)}|] = 2 + 1.88 + \left( 1 \times \frac{1}{1} \right) - \left( 1.88 \times \frac{1}{2} \right) = 3.94 \]

\[ E[|S_{(t_8,H_t-2)}|] = 2 + 3.94 + \left( 1 \times \frac{1}{1} \right) - \left( 3.94 \times \frac{1}{2} \right) = 3.97 \]

Similarly,

\[ E[|S_{(t_5,H_t-3)}|] = E[|S_{(t_5,H_t-2)}|] + \left( E[|\tilde{S}_{(t_4,H_t-2)}|] - (E[|\tilde{S}_{(t_4,H_t-3)}|] \cdot \frac{1}{4}) \right) \]

\[ \Rightarrow E[|S_{(t_5,H_t-3)}|] = 3.75 + \frac{1}{2} + \left( 1.75 \times \frac{1}{2} \right) - \left( \frac{1}{2} \times \frac{1}{4} \right) = 5.0 \]

Similarly,

\[ E[|S_{(t_6,H_t-3)}|] = 3.88 + (5.0 - 3.88) + \left( 1.88 \times \frac{1}{2} \right) - \left( (5.0 - 3.88) \times \frac{1}{4} \right) = 5.66 \]

\[ E[|S_{(t_7,H_t-3)}|] = 3.94 + (5.66 - 3.94) + \left( 1.94 \times \frac{1}{2} \right) - \left( (5.66 - 3.94) \times \frac{1}{4} \right) = 6.20 \]

\[ E[|S_{(t_8,H_t-3)}|] = 3.97 + (6.20 - 3.97) + \left( 1.97 \times \frac{1}{2} \right) - \left( (6.20 - 3.97) \times \frac{1}{4} \right) = 6.62 \]
We can also calculate $E[|\tilde{S}_{(t_8, H_t-3)}|]$ as the following:

$$E[|\tilde{S}_{(t_5, H_t-3)}|] \leq \frac{1}{2} \times \frac{1}{4} - \frac{1}{2} \times \frac{1}{4} \times 18 = 0.11$$

$$E[|\tilde{S}_{(t_6, H_t-3)}|] \leq 0.11 + \left( (5.0 - 3.88) \times \frac{1}{4} \right) - \left( 0.11 \times 18 \right) = 0.38$$

$$E[|\tilde{S}_{(t_7, H_t-3)}|] \leq 0.38 + \left( (5.66 - 3.94) \times \frac{1}{4} \right) - \left( 0.38 \times 18 \right) = 0.76$$

$$E[|\tilde{S}_{(t_8, H_t-3)}|] \leq 0.76 + \left( (6.20 - 3.97) \times \frac{1}{4} \right) - \left( 0.76 \times 18 \right) = 1.22$$
REFERENCES


